Thesis for the Degree of Doctor of Philosophy

General Recursion in Type Theory

Ana Bove

CHALMERS | GÖTEBORG UNIVERSITY

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

Göteborg, November 2002

General Recursion in Type Theory
Ana Bove
ISBN 91-7291-207-3

Doktorsavhandlingar vid Chalmers Tekniska Högskola
Ny serie nr 1889
ISSN 0346-718X

Cover: Whirlpool Galaxy * M51 * NGC 5194
Distance: About 31 million light-years
NASA and The Hubble Heritage Team,
Space Telescope Science Institute,
Association of Universities for Research in Astronomy

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

Printed at Chalmers
Göteborg, Sweden, November 2002

Abstract

This thesis deals with the use of constructive type theory as a programming language. In particular, it presents a method to translate general recursive functional programs into their type-theoretic equivalents.

A key notion in functional programming is recursion, which allows that the object being defined refers to itself. Associated with recursion is the problem of termination since, in general, there is no guarantee that a recursive function will always terminate. It is hence very important to have mechanisms to prove that a certain function terminates or even to ensure the termination of the function by the form of its definition. Examples of the latter are structurally recursive definitions, where every recursive call is performed on a structurally smaller argument.

Standard functional programming languages impose no restrictions on recursive programs and thus, they allow the definition of any general recursive function. There exist, though, less conventional functional programming languages where only terminating recursive definitions are allowed. Constructive type theory can be seen as one of them. In addition, logic can also be represented in constructive type theory by identifying propositions with types and proofs with programs of the corresponding type. Therefore, a type can encode a complete specification, requiring also logical properties from an algorithm. As a consequence, algorithms are correct by construction or can be proved correct by using the expressive power of constructive type theory. This is clearly an advantage of constructive type theory over standard programming languages. A computational limitation of type theory is that, to keep the logic consistent and type-checking decidable, only structurally recursive definitions are allowed.

Many important algorithms are general recursive. Although many such algorithms can be proved to terminate, there is no syntactic condition that guarantees their termination and thus, general recursive algorithms have no direct translation into type theory.

In this thesis, a method to formalise general recursive algorithms in type theory that separates the computational and logical parts of the definitions is presented. As a consequence, the resulting type-theoretic algorithms are clear, compact and easy to understand. Given a general recursive algorithm, the method consists in defining an inductive special-purpose accessibility predicate that characterises the inputs on which the algorithm terminates. The type-theoretic version of the algorithm can then be defined by structural recursion on the proof that the input values satisfy this predicate. When formalising nested algorithms, the special-purpose accessibility predicate and the type-theoretic version of the algorithm much be defined simultaneously because they depend on each other. Since the method separates the computational part from the logical part of a definition, formalising partial functions becomes also possible.

Acknowledgements

First of all, I would like to thank my supervisor Björn von Sydow for being such an extraordinary person. Work related, he has been a huge source of knowledge for me during these years. Besides reading and commenting every piece of work I have written in the past six years, he has always been willing to spend time discussing my ideas and answering my usually silly questions. I am very glad that our relation went beyond work because he is one of the most generous and good-hearted person I have ever meet. He and Verónica Gaspes have helped me a lot through some difficult periods and they have welcomed my daughter and I into their family. For this, we will always be grateful.

I am very thankful to the other members of my committee, namely Bengt Nordström and Mary Sheeran, for their constant support and encouragement during all these years and for reading and commenting on several drafts.

I am also very grateful to Venanzio Capretta, who is the co-author of a couple of the papers in this thesis. Not only is he very bright and hard-working but also he has been very patient with me, always accepting my needs and schedule.

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

I cannot continue without expressing my immense gratitude to Jörgen Gustavsson. He has been an unconditional friend in the last few years. He has been there for me when I wanted to laugh, when I needed a hand to hold or a shoulder to cry, or simply when I wanted to share the moment with a friend. I hope he knows well how much he helped me and how much his friendship means to me. Finally, I hope that, wherever we decide to settle down in life, distance can never break our friendship apart.

I am also very grateful to Thomas Hallgren, among other things, for his constant help with computer-related matters, which I can honestly say, were a lot!

I would like to take this opportunity to thank everybody at the Computing Science Department at Chalmers University of Technology for providing such a friendly environment. Special thanks go to Jeanette Träff for her help with several administrative matters.

I also want to thank the people at the Department of Computer Science of the University of the Republic in Uruguay. In particular, I would like to thank the members of the Formal Method group where I first started my academic life, and the members of the Computer Support group for their help every time I visit Uruguay.

I am very thankful to my sister Patricia, my brother Italo, my "aunt" Martha and to all my friends, both here and there, for their constant help and support during all this time. Special thanks go Steve Berkhuizen, Maria Mercedes

Couchet, Verónica Gaspes, Jacqueline Gómez, Birgit Grohe, Gianella Ratto, Nora Szasz and Virginia Pérez.

Finally, I would like to thank my mother, even though she is not among us anymore. She always helped me and supported me in my decisions, even if they would imply that I would live in a far, cold and dark country where people speak a strange language. Most of what I am and what I am not I own to her, for which I am enormously thankful. I only wish she could have been present here with me at this important moment. I am sure, though, she will be with me in some other way.

This thesis contains the following five papers:

I *Simple General Recursion in Type Theory.* In Nordic Journal of Computing, volume 8, number 1, pages 22–42, Spring 2001.

II *Nested General Recursion and Partiality in Type Theory.* Joint work with Venanzio Capretta. In proceedings of Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001, volume 2152 of Springer-Verlag LNCS, pages 121–135, hold in Edinburgh, Scotland, September 2001.

The method here is based on the method described in the previous paper by Ana Bove. Venanzio Capretta's main contribution here was the idea to use Dybjer's schema for simultaneous inductive-recursive definition.

III *Mutual General Recursion in Type Theory.* Technical report of the Department of Computing Science, Chalmers University of Technology, May 2002.

A variant of this paper with the name *Generalised Simultaneous Inductive-Recursive Definitions and their Application to Programming in Type Theory* is currently under consideration for publication in the proceedings of TYPES workshop, hold in Nijmegen, The Netherlands, April 2002.

IV *Modelling General Recursion in Type Theory.* Joint work with Venanzio Capretta. Technical report of the Department of Computing Science, Chalmers University of Technology, October 2002.

The method formalised here is based on the method described in the previous three papers. This said, the contribution of each of the authors is difficult to distinguish except that section 5, on Turing completeness, was mainly done by Venanzio Capretta.

V *Programming in Martin-Löf Type Theory: Unification - A non-trivial Example.* Licentiate Thesis of the Department of Computing Science, Chalmers University of Technology, November 1999.

Contents

General Introduction 1
Paper I: Simple General Recursion in Type Theory 11
Paper II: Nested General Recursion and Partiality in Type Theory . . 35
Paper III: Mutual General Recursion in Type Theory 53
Paper IV: Modelling General Recursion in Type Theory 75
Paper V: Programming in Martin-Löf Type Theory: Unification - A non-trivial Example 121

General Recursion in Type Theory

Ana Bove

September 2002

General Introduction

Functional programming is one of the different programming styles and it has become widely used in the last decade because of, among other reasons, the simplicity and elegance of its programs.

In functional programming, a program consists of several functions. The main program is a function which, in turn, is defined using other functions. These functions are very similar to mathematical functions and they are usually defined by equations. To compute a functional program, the main function in the program receives the program's input as its arguments and produces the program's output as its result.

A very simple functional program is the program that doubles its input, which can be defined as:

```
double x = 2 * x
```

Then, when we apply this function to the value `3` we obtain the value `6`. A problem arises though if we want to apply the function `double` to the boolean value `True`, since the operation `2 * True` has no associated meaning. In this kind of problems types become of great help.

A *type* is a collection of values, such as numbers or booleans. If we look at the function `double`, it makes sense to double numbers but not boolean values. Functions and types can be connected by defining the functions to operate only over particular types. Hence, the function above can be instead be defined as:

```
double :: Integer -> Integer
double x = 2 * x
```

Types have been broadly accepted in programming because of, among other reasons, their help in the elimination of errors in the programs. Hence, most of the more important programming languages are typed. Functional programming languages are not an exception and then, also most of the more important functional programming languages are typed. Examples of typed functional languages are Haskell [JHe+99] and Standard ML [MTHM97]. Throughout this

thesis we mainly use Haskell when we introduce examples in a functional programming language.

Typed programming languages usually provide a few basic types and a mechanism that allows the definition of *inductive* data types. When defining a data type, we have to introduce the constructors that generate the elements of the new type together with the types of its arguments. An example of an inductive type is the following, provided that `Number`, `Origin` and `Destination` are types.

```
data Transport = Taxi | Bus Number Origin Destination
```

Here, the inductive type `Transport` has two ways of constructing its elements. The first constructor, `Taxi`, takes no arguments while the second constructor, `Bus`, takes the number of the bus, and its origin and destination as arguments.

A key notion in functional programming is the notion of *recursion*, which allows that the object being defined refers to itself. The notion of recursion applies both to types and to functions. A very well known inductive data type which is recursive is the type of natural numbers, which can be defined as:

```
data N = O | S N
```

Some functional programming languages have also the possibility of defining data types with infinite objects, such as the type of infinite lists of natural numbers defined as:

```
data Stream = SCons N Stream
```

However, we disregard such possibility in this introduction and hence, every type definition contains only finite elements.

Functions defined over inductive types are recursive functions by nature. The factorial function over natural numbers can be defined as below, where we use the *pattern matching* facility of Haskell.

```
fac :: N -> N
fac O = S O
fac (S n) = (S n) * fac n
```

Associated with recursion, we have the crucial problem of *termination*. In general, there is no guarantee that a recursive function will always terminate. For example, a small change in the factorial function above will make that the function fails termination for positive inputs.

```
fac' :: N -> N
fac' O = S O
fac' (S n) = (S n) * fac' (S n)
```

It is hence very important to have mechanisms that allow us to prove that a certain function terminates or even to ensure the termination of the function by the form of its definition. An example of the former are the so called *proofs by induction* and of the latter are the *structurally recursive* definitions. A structurally recursive definition is such that every recursive call is performed on a

structurally smaller argument. In this way we can be sure that the recursion terminates. The definition of the function `fac` is an example of such a definition. Hence, the function `fac` terminates for any input. Another example of a structurally recursive definition is the *insertion sort* algorithm defined as:

```
sort :: [N] -> [N]
sort [] = []
sort (x:xs) = insert x (sort xs)

insert :: N -> [N] -> [N]
insert n [] = [n]
insert n (x:xs) = if n < x then n:x:xs
                           else x: insert n xs
```

Definitions where the recursive calls are not required to be on structurally smaller arguments, that is, where the recursive calls can be performed on any argument are called *general recursive* definitions.

Standard functional programming languages impose no restrictions on recursive programs. Thus, they allow the definition of general recursive algorithms and it is the responsibility of the programmer to write terminating programs. There exist though less conventional functional programming languages where only structurally recursive definitions are allowed. Constructive type theory can be seen as an example of such a functional programming language and hence, in type theory, all functions are guaranteed to terminate.

Except for a change in notation, the insertion sort algorithm can be written in type theory in the same way as above:

$$
\begin{array}{l}
\mathsf{sort} \in (\mathsf{List}(\mathsf{Nat}))\mathsf{List}(\mathsf{Nat}) \\
\quad \mathsf{sort}(\mathsf{nil}) = \mathsf{nil} \\
\quad \mathsf{sort}(\mathsf{cons}(x, xs)) = \mathsf{insert}(x, (\mathsf{sort}(xs))) \\
\\
\mathsf{insert} \in (\mathsf{Nat}; \mathsf{List}(\mathsf{Nat}))\mathsf{List}(\mathsf{Nat}) \\
\quad \mathsf{insert}(n, \mathsf{nil}) = \mathsf{cons}(n, \mathsf{nil}) \\
\quad \mathsf{insert}(n, \mathsf{cons}(x, xs)) = \\
\qquad \mathsf{if\_then\_else}(n < x, \mathsf{cons}(n, \mathsf{cons}(x, xs)), \mathsf{cons}(x, \mathsf{insert}(n, xs)))
\end{array}
$$

where if_then_else is a function that selects the second or the third argument depending or whether the first argument evaluates to true or to false, respectively.

However, constructive type theory is more expressive than standard programming languages thanks to the possibility of defining *dependent* types. A clear example where dependent data types in programming are very useful is the definition of the type of vectors of certain length. If Nat is the type of the elements in the vector, the definition of such a type would be as follows in type theory:

$$
\begin{array}{l}
\mathsf{Vector} \in (\mathsf{Nat})\mathsf{Set} \\
\quad \mathsf{nil_v} \in \mathsf{Vector}(0) \\
\quad \mathsf{cons_v} \in (n \in \mathsf{Nat}; m \in \mathsf{Nat}; v \in \mathsf{Vector}(n))\mathsf{Vector}(\mathsf{s}(n))
\end{array}
$$

According to the Curry-Howard isomorphism [How80], logic can also be represented in type theory by identifying propositions with types and proofs with programs of the corresponding types. Predicates and relations are then seen in type theory as functions yielding propositions as output. As an example of an inductively defined predicate in type theory we present the definition of the Sorted predicate over lists of natural numbers. Given a list of natural numbers ls, $\mathsf{Sorted}(ls)$ is true if ls is sorted, that is, we can find a proof of $\mathsf{Sorted}(ls)$ if ls is sorted.

$$
\begin{array}{l}
\mathsf{Sorted} \in (\mathsf{List}(\mathsf{Nat}))\mathsf{Set} \\
\quad \mathsf{sorted}_{\mathsf{nil}} \in \mathsf{Sorted}(\mathsf{nil}) \\
\quad \mathsf{sorted}_{\mathsf{cons}} \in (n \in \mathsf{Nat}; l \in \mathsf{List}(\mathsf{Nat}); p \in \mathsf{Minimum}(n, l); h \in \mathsf{Sorted}(l) \\
\qquad\qquad)\mathsf{Sorted}(\mathsf{cons}(n, l))
\end{array}
$$

where $\mathsf{Minimum}(n, l)$ is a proposition that is true if the number n is less than or equal to any element in the list l.

We can now use the expressive power of constructive type theory to prove the correctness of our insertion sort algorithm. Let $\mathsf{Perm}(l, l')$ be a proposition which is true if the list l' is a permutation of the list l. The correctness theorem for our sorting example can be stated as follows:

$$\forall\, l \in \mathsf{List}(\mathsf{Nat})\,.\,\mathsf{Sorted}(\mathsf{sort}(l)) \wedge \mathsf{Perm}(l, \mathsf{sort}(l))$$

where $\wedge$ represent the conjunction of two propositions in type theory.

Given a list of natural numbers ls and any proof p of the above statement, $p(ls)$ is a proof that the list $\mathsf{sort}(ls)$ is sorted and that it is a permutation of the original list ls. In this way, we have defined a sorting algorithm and then we have proved that it produces the expected result. This approach is known in the literature as the *external approach*.

An alternative way of obtaining our sorting algorithm is by the *internal* or *integrated approach*. In the internal approach, we integrate the algorithm itself with its specification. In this way, we encode in a type a complete specification, requiring also logical properties from an algorithm. A program satisfying such specification is simply an object of the type in question. In other words, since theorems are represented as types in type theory then, a proof of a theorem is a function that, given proofs of the hypotheses of the theorem, computes the proof of the thesis. In particular, when a theorem states the existence of an object with certain properties, the proof of the theorem computes such an object from the given proofs of the hypotheses of the theorem. As a consequence, when using the internal approach, algorithms are correct by construction.

For our sorting example, the type of the integrated approach is:

$$\forall\, l \in \mathsf{List}(\mathsf{Nat})\,.\,\exists\, l' \in \mathsf{List}(\mathsf{Nat})\,.\,\mathsf{Sort}(l') \wedge \mathsf{Perm}(l, l')$$

Here, given a list of natural numbers ls and any proof p of the above statement, $p(ls)$ is a pair of the form $\langle l', h\rangle$ where l' is a list of natural numbers and h is a proof that the list l' is sorted and that it is a permutation of the original

list l. Thus, l' is the output list and h is a term containing the algorithm and a proof of its correctness. A technique called *program extraction* allows us to separate the algorithm itself from its correctness proof and hence, we can use the extracted algorithm as in any functional programming language.

It is worth mentioning here that the above statement is not particularly associated to the insertion sort algorithm but to any sorting algorithm on list of natural numbers. Different proofs of the statement would correspond to the different sorting algorithms.

The fact that algorithms are correct by construction or can be proved correct by using the expressive power of constructive type theory is clearly an advantage of constructive type theory over standard programming languages. A computational limitation of type theory is that, to keep the logic consistent and type-checking decidable, only structurally recursive definitions are allowed, as we have already mentioned.

Many important and well known algorithms are not structurally recursive but general recursive. Although many general recursive algorithms can be proved to terminate, there is no syntactic condition that guarantees their termination and thus, general recursive algorithms have no direct formalisation in type theory.

Coming back to our sorting example, we might decide to implement a more efficient sorting algorithm known as *quicksort*. Its Haskell definition is as follows:

```
quicksort :: [N] -> [N]
quicksort [] = []
quicksort (x:xs) = quicksort (filter (< x) xs) ++
                   x : quicksort (filter (>= x) xs)
```

Although the recursive calls are made on non-structurally smaller argument, it is easy to see that the sizes of the lists on which we perform the recursive calls are strictly smaller than the size of the original list. In this way, we can convince ourselves that the algorithm terminates for any input.

The standard way of handling general recursion in constructive type theory uses a well-founded recursion principle derived from the accessibility predicate Acc (see [Acz77, Nor88]). The idea behind the accessibility predicate is that an element a is *accessible* by a relation $\prec$ if there exists no infinite decreasing sequence starting from a. A set A is said to be *well-founded* with respect to $\prec$ if all its elements are accessible by $\prec$. Hence, to guarantee that a general recursive algorithm that performs the recursive calls on elements of type A terminates, we have to prove that A is well-founded and that the arguments supplied to the recursive calls are smaller than the input.

If we want to use the accessibility predicate for the formalisation of the quicksort algorithm, we have to mix the algorithm itself with proofs that the sizes of the lists `filter (< x) xs` and `filter (>= x) xs` are strictly smaller than the size of the list `(x:xs)`. However, these proofs have no computational content and their only purpose is to provide the necessary conditions to allow the recursive calls.

Since Acc is a general predicate, it gives no information that can help us in the formalisation of a specific recursive algorithm. As a consequence, its use in the type-theoretic formalisation of general recursive algorithms often results in unnecessarily long and complicated codes. Moreover, its use adds a considerable amount of code with no computational content, that distracts our attention from the computational part of the algorithm.

To bridge the gap between programming in type theory and programming in a functional language, we developed a method to formalise general recursive algorithms in type theory that separates the computational and logical parts of the definitions. As a consequence, the resulting type-theoretic algorithms are clear, compact and easy to understand. They are as simple as their Haskell-like versions, where there is no restriction on the recursive calls. Given a general recursive algorithm, our method consists in defining an inductive special-purpose accessibility predicate that characterises the inputs on which the algorithm terminates. The type-theoretic version of the algorithm can then be defined by structural recursion on the proof that the input values satisfy this predicate. Since our method separates the computational part from the logical part of a definition, formalising partial functions becomes possible. Proving that a certain function is total amounts to proving that the corresponding accessibility predicate is satisfied by every input.

If we want to use our method in the type-theoretic formalisation of the quicksort algorithm, we first have to construct the special-purpose accessibility predicate associated with the algorithm. To construct this predicate, we analyse the Haskell code and characterise the inputs on which the algorithm terminates. Thus, we distinguish the following two cases:

- The algorithm `quicksort` terminates on the input `[]`;
- Given a natural number `x` and a list `xs` of natural numbers, the algorithm `quicksort` terminates on the input `(x:xs)` if it terminates on the inputs `(filter (< x) xs)` and `(filter (>= x) xs)`.

From this description, we define the type-theoretic inductive predicate qsAcc over lists of natural numbers by the introduction rules we give below.

$$\frac{}{\mathsf{qsAcc}(\mathsf{nil})} \qquad \frac{\mathsf{qsAcc}(\mathsf{filter}((<x),xs)) \qquad \mathsf{qsAcc}(\mathsf{filter}((\geqslant x),xs))}{\mathsf{qsAcc}(\mathsf{cons}(x,xs))}$$

We formalise this predicate in type theory as follows:

$$\begin{array}{l}\mathsf{qsAcc} \in (zs \in \mathsf{List}(\mathsf{Nat}))\mathsf{Set}\\ \quad \mathsf{qs\_acc}_{\mathsf{nil}} \in \mathsf{qsAcc}(\mathsf{nil})\\ \quad \mathsf{qs\_acc}_{\mathsf{cons}} \in (x \in \mathsf{Nat}; xs \in \mathsf{List}(\mathsf{Nat}); h_1 \in \mathsf{qsAcc}(\mathsf{filter}((<x),xs));\\ \qquad\qquad h_2 \in \mathsf{qsAcc}(\mathsf{filter}((\geqslant x),xs))\\ \qquad\qquad)\mathsf{qsAcc}(\mathsf{cons}(x,xs))\end{array}$$

We define the quicksort algorithm by structural recursion on the proof that

the input list of natural numbers satisfies the predicate qsAcc.

$$\begin{array}{l}\mathsf{quicksort} \in (zs \in \mathsf{List}(\mathsf{Nat}); \mathsf{qsAcc}(zs))\mathsf{List}(\mathsf{Nat}) \\ \quad \mathsf{quicksort}(\mathsf{nil}, \mathsf{qs\_acc}_{\mathsf{nil}}) = \mathsf{nil} \\ \quad \mathsf{quicksort}(\mathsf{cons}(x, xs), \mathsf{qs\_acc}_{\mathsf{cons}}(x, xs, h_1, h_2)) = \\ \qquad \mathsf{quicksort}(\mathsf{filter}((< x), xs), h_1) +\!\!+ \mathsf{cons}(x, \mathsf{quicksort}(\mathsf{filter}((\geqslant x), xs), h_2))\end{array}$$

Finally, as the quicksort algorithm is total, we can prove

$$\mathsf{allQsAcc} \in (zs \in \mathsf{List}(\mathsf{Nat}))\mathsf{qsAcc}(zs)$$

and use that proof to define the type-theoretic function QuickSort.

$$\begin{array}{l}\mathsf{QuickSort} \in (zs \in \mathsf{List}(\mathsf{Nat}))\mathsf{List}(\mathsf{Nat}) \\ \quad \mathsf{QuickSort}(zs) = \mathsf{quicksort}(zs, \mathsf{allQsAcc}(zs))\end{array}$$

At this point, we would like to draw the reader's attention to the simplicity of this translation. The accessibility predicate can be automatically generated from the recursive equations of the functional program and the type-theoretic version of the algorithm looks very similar to the original program, except for the extra proof argument. If we suppress the proofs of the accessibility predicate, we get almost exactly the original algorithm.

This method was first introduced in *Simple General Recursion in Type Theory* (paper I). Here, we illustrate the method to translate simple general recursive algorithms into type theory. By simple here we mean non-nested and non-mutually recursive algorithms. In *Programming in Martin-Löf Type Theory: Unification - A non-trivial Example* (paper V), we use the method for simple algorithms to formalise a unification algorithm over lists of pairs of terms. There, we present both the internal and the external approach of the example. In addition, we present the formalisation of the unification algorithm using the standard accessibility predicate Acc and we compare the resulting algorithms.

The method has been extended to treat nested recursion in *Nested General Recursion and Partiality in Type Theory* (paper II), which was a joint work with Venanzio Capretta. When we want to formalise nested algorithms, the special-purpose accessibility predicate and the type-theoretic version of the algorithm depend on each other and thus, they must be defined simultaneously. This kind of definitions is not allowed in ordinary type theory, but it is provided in type theories extended with Dybjer's schema for simultaneous inductive-recursive definitions [Dyb00]. In addition, in paper II, we show that partial functions can also be formalised in type theory using our method.

Later, in *Mutual General Recursion in Type Theory* (paper III), the method has been extended to treat mutually recursive functions, nested or not. When we have mutually recursive algorithms, the termination of one function depends on the termination of the others and hence, the accessibility predicates are also mutually recursive. If, in addition to mutual recursion, we have nested calls, we again need to define the predicates simultaneously with the algorithms. In order to do so, we need to extend Dybjer's schema for cases where we have several mutually recursive predicates defined simultaneously with several functions. Such a generalisation is also presented in this paper.

In the papers I, II and III, we have presented our method by means of examples. The purpose of paper IV, *Modelling General Recursion in Type Theory* which is also a joint work with Venanzio Capretta, is to give a general presentation of the method. We start by giving a characterisation of the class of recursive definitions that we consider, which is a subclass of commonly used functional programming languages like Haskell. This class, which we prove to be Turing complete, consists of functions defined by recursive equations that are not necessarily well-founded. Then, we formally describe how we can translate any function in that class into type theory using our special-purpose accessibility predicates.

To summarise, this thesis deals with the use of constructive type theory as a programming language. In particular, it presents a method to write general recursive programs in type theory or more precisely, to translate general recursive functional programs into their type-theoretic equivalents. Throughout this thesis, by constructive type theory we primarily understand Martin-Löf type theory [ML84]. However, some of the ideas presented here can also be applied in other type theories as for example the Calculus of Constructions [CH88].

As a final remark, we would like to add that writing programs in constructive type theory amounts to writing completely formal proofs of often complex theorems. This makes the practical applicability of type theory strongly dependent on the availability of adequate programming environments, usually known as proof assistants. For the examples we have presented throughout this thesis, we have used the proof assistant ALF [Mag92, AGNvS94, MN94] and some times also the proof assistant Coq [coq99, coq01].

References

[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`.

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

[coq99] Coq homepage. `http://pauillac.inria.fr/coq/`, 1999.

[coq01] The Coq development team. Logical project. The Coq proof assistant reference manual. Version 7.2, 2001. INRIA.

[Dyb00] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.

[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.

[JHe+99] S. Peyton Jones, J. Hughes, (editors), L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.

[Mag92] L. Magnusson. The new Implementation of ALF. In *The informal proceeding from the logical framework workshop at Båstad, June 1992*, 1992.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[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*. MIT Press, 1997.

[Nor88] B. Nordström. Terminating General Recursion. *BIT*, 28(3):605–619, October 1988.

Paper I

Simple General Recursion in Type Theory

Simple General Recursion in Type Theory

Ana Bove
Department of Computing Science
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
bove@cs.chalmers.se

January 2001

Abstract

General recursive algorithms are such that the recursive calls are performed on arguments satisfying no condition that guarantees termination. Hence, there is no direct way of formalising them in type theory.

The standard way of handling general recursion in type theory uses a well-founded recursion principle. Unfortunately, this way of formalising general recursive algorithms often produces unnecessarily long and complicated codes. On the other hand, functional programming languages like Haskell impose no restrictions on recursive programs, and then writing general recursive algorithms is straightforward. In addition, functional programs are usually short and self-explanatory. However, the existing frameworks for reasoning about the correctness of Haskell-like programs are weaker than the framework provided by type theory.

The goal of this work is to present a method that combines the advantages of both programming styles when writing simple general recursive algorithms. The method introduced here separates the computational and logical parts of the definition of an algorithm, which has several advantages. First, the resulting type-theoretic algorithms are compact and easy to understand; they are as simple as their Haskell versions. Second, totality is now a separate task and hence, this method can also be used in the formalisation of partial functions. Third, the method presented here also simplifies the task of formal verification. Finally, it can easily be extended to treat nested and mutual recursion.

The main feature of the method is the introduction of an inductive predicate, specially defined for the algorithm to be formalised. This predicate can be thought of as characterising the set of inputs for which the algorithm terminates. It contains an introduction rule for each of the cases that need to be considered and provides an easy syntactic condition that guarantees the termination of the algorithm.

1 Introduction

Following the Curry-Howard isomorphism [How80], constructive type theory (see for example [ML84, CH88]) can 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. In addition, 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.

Simple general recursive algorithms (by simple we mean non-nested and non-mutually recursive) are defined by cases where the recursive calls are on non-structurally smaller arguments. In other words, the recursive calls are performed on objects satisfying no condition that guarantees termination. As a consequence, there is no direct way of formalising them in type theory.

The standard way of handling general recursion in type theory uses a well-founded recursion principle derived from the accessibility predicate Acc (see [Acz77, Nor88]). The predicate Acc captures the idea that an element a of type A is accessible by a relation $\prec$ if there exists no infinite decreasing sequence starting from a. A set A is said to be well-founded with respect to $\prec$ if all its elements are accessible by $\prec$.

When using this predicate to write the type-theoretic version of a general recursive algorithm that performs the recursive calls on elements of type A, we guarantee that the algorithm terminates if A is well-founded and if we provide proofs showing that the arguments on which we perform the recursive calls are smaller than the initial input argument. These proofs, although essential to guarantee the termination of the algorithm, add a considerable amount of code and distract our attention since they do not have any computational content. As the standard accessibility predicate is a general predicate, it contains no information that can help us in the formalisation of a particular algorithm. Consequently, this way of formalising general recursive algorithms in type theory often results in unnecessarily long and complicated codes.

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. Therefore, writing general recursive algorithms in Haskell is straightforward. In addition, functional programs are usually short and self-explanatory. However, the existing frameworks for reasoning 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 take a step towards combining the advantages of both programming styles when writing general recursive algorithms. To this purpose, we present a method to formalise simple general recursive algorithms in type theory that results in short and self-explanatory codes. Afterwards, these codes can be proven correct by using the expressive power of the theory.

Given the Haskell version of a general recursive algorithm f_alg, the main feature of our method is the introduction of an inductive predicate called fAcc, specially defined for f_alg. We construct this predicate directly from f_alg and it can be thought of as characterising the set of inputs on which the algorithm terminates. The predicate has an introduction rule for each of the cases that need to be considered in the algorithm and provides an easy syntactic condition that guarantees termination. In this way, we can formalise f_alg in type theory by structural recursion on the proof that the input of f_alg satisfies fAcc, obtaining a compact and readable formalisation of the algorithm.

Here, we use Martin-Löf's type theory [ML84, NPS90] as our constructive type theory. In addition, we use the proof assistant ALF [AGNvS94, MN94] for presenting our examples. However, our method can also be used in other type theories such as the Calculus of Constructions [CH88], and then we could use the proof assistant Coq [DFH+91] for presenting the examples. In what follows, sometimes we use "type theory" as an abbreviation for "Martin-Löf's type theory". This paper summarises the work presented in [Bov99].

The rest of the paper is organised as follows:

Although this paper is intended for readers who have some basic knowledge of Martin-Löf's type theory, in Section 2 we give a brief introduction to this theory and to its interactive proof assistant ALF.

In Section 3, we illustrate our method by formalising two general recursive algorithms: the modulo algorithm over natural numbers and a unification algorithm over lists of pairs of terms.

Finally, in Section 4 we present some conclusions and related work.

2 Martin-Löf's type theory and ALF

2.1 Brief introduction to Martin-Löf's type theory

The basic notion in Martin-Löf's type theory (see [ML84, NPS90]) is that of *type*. A type is explained by saying what its objects are and what it means for two of its objects to be equal. We write $a \in \alpha$ for "a is an object of type α".

Martin-Löf's type theory has a basic type and two type formers.

The basic type is the type of sets and we call it **Set**. Sets are inductively defined. In conformity with the explanation of what it means to be a type, we know that A is an object of **Set** if we know how to form its canonical elements and when two canonical elements are equal.

The first type former constructs the type of the elements of a set: for each set A, the elements of A form a type called $\mathbf{El}(A)$. However, for simplicity, if a is an element of A, we say that a has type A, and thus we simply write $a \in A$ instead of $a \in \mathbf{El}(A)$. Since every set is inductively defined, we know how to build its elements.

The second type former constructs the types of dependent functions. Let α be a type and β be a family of types over α, that is, for every element a in α, $\beta(a)$ is a type. We write $(a \in \alpha)\beta(a)$ for the type of dependent functions from

α to β. If f has type $(a \in \alpha)\beta(a)$, then, when we apply f to an object a of type α, we obtain an object $f(a)$ of type $\beta(a)$.

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.

In type theory, predicates and relations are seen as functions yielding propositions as output. As well as sets, propositions are inductively defined. A proposition is interpreted as a set whose elements represent 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. The way propositions are introduced allows us to identify propositions and sets, and then we write **Set** to also refer to the type of propositions.

2.2 Brief introduction to ALF

ALF (see [AGNvS94, MN94]) 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. 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.

In ALF, any inductive definition is introduced as a constant A of type $(x_1 \in \alpha_1; \ldots; x_n \in \alpha_n)$**Set**, for $\alpha_1, \ldots, \alpha_n$ types. Afterwards, we introduce the constructors that generate the elements of $A(a_1, \ldots, a_n)$ by giving their type, 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 $(x \in \alpha)$. Whenever $(x_1 \in \alpha;\, x_2 \in \alpha; \ldots; x_n \in \alpha)$ occurs, ALF displays instead $(x_1, x_2, \ldots, x_n \in \alpha)$.

A function can be defined by performing pattern matching over one (or more) of its arguments. The user should then select the desired argument over which to perform the pattern matching and run the pattern matching option. 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 here termination is guaranteed because we always apply the recursion on a structurally smaller argument. In this way, checking well-foundedness in our proofs is easy—although 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.

In particular, if we do case analysis on a proof a of absurdity (that is, $a \in \bot$) or on a proof that $a \in A$ when A is isomorphic to absurdity, we do not obtain any case to study since there does not exist any proof of absurdity.

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.

We now present some general set formers and constructors in Martin-Löf's type theory. See figure 1 for the ALF definitions of the sets presented in the following sections, together with the type of the main operators, functions and properties for those sets.

2.3.1 Logical constants

In figure 1, $\bot$ represents absurdity, $\wedge$ conjunction, $\vee$ disjunction, $\Rightarrow$ implication and $\neg$ negation. The universal quantifier is represented by $\forall$ and the existential quantifier by $\exists$.

2.3.2 Some useful general predicates

$=$: Represents propositional equality. Its only constructor states that an object is equal to itself.

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

Acc : 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 $\mathsf{Acc}(A, \prec, a)$. 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 there is no x in A such that $x \prec a$, then a is accessible. This idea can be expressed by the following rule:

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

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:

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

and its computation rule is the following:

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

If all the elements in A are accessible by $\prec$, we say that the set A is *well-founded* by $\prec$.

Dec : 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.

$\bot \in$ **Set**
$\wedge \in (A, B \in$ **Set**$)$ **Set**
  $\wedge_{\mathrm{I}} \in (a \in A;\ b \in B)\ \wedge(A, B)$
fst $\in (\wedge(A, B))\ A$
snd $\in (\wedge(A, B))\ B$
$\vee \in (A, B \in$ **Set**$)$ **Set**
  $\vee_{\mathrm{L}} \in (a \in A)\ \vee(A, B)$
  $\vee_{\mathrm{R}} \in (b \in B)\ \vee(A, B)$
$\Rightarrow \in (A, B \in$ **Set**$)$ **Set**
  $\Rightarrow_{\mathrm{I}} \in (f \in (A)\,B)\ \Rightarrow(A, B)$
$\Rightarrow_{\mathrm{E}} \in (f \in \Rightarrow(A, B);\ a \in A)\ B$
$\Rightarrow_{\mathrm{trans}} \in (\Rightarrow(A, B);\ \Rightarrow(B, C))\ \Rightarrow(A, C)$
$\neg \in (A \in$ **Set**$)$ **Set**
  $\neg \equiv [A]\Rightarrow(A, \bot)$
$\forall \in (A \in$ **Set**$;\ B \in (A)$**Set**$)$ **Set**
  $\forall_{\mathrm{I}} \in (f \in (a \in A)\,B(a))\ \forall(A, B)$
$\forall_{\mathrm{E}} \in (\forall(A, B);\ a \in A)\ B(a)$
$\exists \in (A \in$ **Set**$;\ B \in (A)$**Set**$)$ **Set**
  $\exists_{\mathrm{I}} \in (a \in A;\ b \in B(a))\ \exists(A, B)$
witness $\in (\exists(A, B))\ A$
proof $\in (h \in \exists(A, B))\ B(\mathrm{witness}(h))$
$= \in (a, b \in A)$ **Set**
  refl $\in (a \in A)\ =(a, a)$
$=_{\mathrm{symm}} \in (=(a, b))\ =(b, a)$
$=_{\mathrm{trans}} \in (=(a, b);\ =(b, c))\ =(a, c)$
$=_{\mathrm{cong1}} \in (f \in (A)\,B;\ =(a_1, a_2))\ =(f(a_1), f(a_2))$
$=_{\mathrm{subst1}} \in (=(a, b);\ P(a))\ P(b)$

Acc $\in (A \in$ **Set**$;\ less \in (A; A)$**Set**$;\ a \in A)$ **Set**
  acc $\in (a \in A;$
    $(x \in A;\ less(x, a))\ \mathrm{Acc}(A, less, x)$
    $)\ \mathrm{Acc}(A, less, a)$
WF $\in (A \in$ **Set**$;\ less \in (A; A)$**Set**$)$ **Set**
wfrec $\in (a \in A;$
    $\mathrm{Acc}(A, less, a);$
    $e \in (x \in A;$
      $\mathrm{Acc}(A, less, x);$
      $(y \in A; less(y, x))P(y)$
    $)\,P(x)$
  $)\,P(a)$
Dec $\in$ (**Set**) **Set**
  yes $\in (h \in P)\ \mathrm{Dec}(P)$
  no $\in (h \in \neg(P))\ \mathrm{Dec}(P)$
List $\in (A \in$ **Set**$)$ **Set**
  [] $\in$ List(A)
  : $\in (l \in \mathrm{List}(A);\ a \in A)\ \mathrm{List}(A)$
++ $\in (l_1, l_2 \in \mathrm{List}(A))\ \mathrm{List}(A)$
$\in_{\mathrm{L}} \in (a \in A;\ l \in \mathrm{List}(A))$ **Set**
$\notin_{\mathrm{L}} \in (a \in A;\ l \in \mathrm{List}(A))$ **Set**
Disjoint $\in (l, l_1 \in \mathrm{List}(A))$ **Set**
$\subseteq \in (l_1, l_2 \in \mathrm{List}(A))$ **Set**
N $\in$ **Set**
  0 $\in$ N
  s $\in (n \in \mathrm{N})$ N
+ $\in (n, m \in \mathrm{N})$ N
$< \in (n, m \in \mathrm{N})$ **Set**
$\leq \in (n, m \in \mathrm{N})$ **Set**
Pair $\in (A, B \in$ **Set**$)$ **Set**
  . $\in (a \in A;\ b \in B)\ \mathrm{Pair}(A, B)$

Figure 1: ALF definition of the logical constants, and general predicates and data types.

2.3.3 Some useful general data types

In figure 1, List(A) represents the set of lists over a set A, N the set of natural numbers and Pair(A, B) the set of pairs over the sets A and B.

3 Formalising general recursive algorithms

Here, we illustrate our method by formalising two general recursive algorithms: the modulo algorithm over natural numbers and a unification algorithm over lists of pairs of terms. In both examples we follow the same approach. First, we introduce the Haskell version of the algorithm. If necessary, we also give an informal explanation of its termination. Afterwards, we describe how we can write the algorithm in Martin-Löf's type theory by using the standard accessibility predicate to handle the recursive calls, and we point out the problems with this formalisation. Finally, we present a special-purpose accessibility predicate for the example we are considering and we show how we can use this predicate to write our type-theoretic version of algorithm.

3.1 First example: The modulo algorithm

3.1.1 The Haskell version of the algorithm

In the Haskell definition of the modulo algorithm we use the set `Nat` of natural numbers, the subtraction operation `<->` and the less-than relation `<<` over `Nat`, defined in Haskell in the usual way. We also use Haskell's data type `Maybe A`, whose elements are `Nothing` and `Just a`, for any `a` of type `A`.

Now, the Haskell version of the modulo algorithm can be defined as follows (we ignore efficiency aspects such as the fact that the expression `n << m` is computed twice):

```
mod :: Nat -> Nat -> Maybe Nat
mod n 0 = Nothing
mod n m | n << m      = Just n
        | not(n << m) = mod (n <-> m) m
```

It is easy to see that this algorithm terminates on all inputs. However, the recursive call is made on the argument $n - m$ that is not structurally smaller than the argument n, although the value of $n - m$ is less than n.

3.1.2 Using the standard accessibility predicate for the formalisation

Before introducing the type-theoretic version of the algorithm that uses the standard accessibility predicate, we give the ALF types of the subtraction operation and the less-than relation over natural numbers, and the ALF types of two lemmas that we use later on:

$- \in (n, m \in \mathrm{N})\ \mathrm{N}$ $\qquad$ $<_{dec} \in (n, m \in \mathrm{N})\ \mathrm{Dec}(<(n, m))$

$< \in (n, m \in \mathrm{N})\ \mathbf{Set}$ $\qquad$ $\text{-}< \in (n, m \in \mathrm{N};\ \neg(<(n, s(m)))) <(\text{-}(n, s(m)), n)$

The first lemma states that it can be decided whether a natural number is less than another. The second lemma establishes that if the natural number n is not less than the natural number $s(m)$, then the result of subtracting $s(m)$ from n is less than n.

Instead of the `Maybe` type of Haskell, we use the logic connective $\vee$ and a singleton set Error whose only element is error.

Now, we present the type-theoretic version of the modulo algorithm that uses the standard accessibility predicate Acc to handle the recursive call.

$\mathsf{mod}_{\mathsf{acc}} \in (n, m \in \mathsf{N};\ \mathsf{Acc}(\mathsf{N}, <, n)) \vee (\mathsf{N}, \mathsf{Error})$
$\quad \mathsf{mod}_{\mathsf{acc}}(n, 0, h) \equiv \vee_{\mathsf{R}}(\mathsf{error})$
$\quad \mathsf{mod}_{\mathsf{acc}}(n, \mathsf{s}(m_1), \mathsf{acc}(\_, h_1)) \equiv$
$\quad\quad \mathbf{case}\ <_{\mathsf{dec}}(n, \mathsf{s}(m_1)) \in \mathsf{Dec}(<(n, \mathsf{s}(m_1)))\ \mathbf{of}$
$\quad\quad\quad \mathsf{yes}(h) \Rightarrow \vee_{\mathsf{L}}(n)$
$\quad\quad\quad \mathsf{no}(h) \Rightarrow \mathsf{mod}_{\mathsf{acc}}(\text{-}(n, \mathsf{s}(m_1)), \mathsf{s}(m_1), h_1(\text{-}(n, \mathsf{s}(m_1)), \text{-}<(n, m_1, h)))$
$\quad\quad \mathbf{end}$

This algorithm is defined by recursion on the proof that the first argument of the modulo operator is accessible by $<$. To define the algorithm, we first consider cases on m. If m is zero, we return an error since the modulo zero operation is not defined. If m is equal to $s(m_1)$ for some natural number m_1, we consider cases depending on whether or not n is less than $s(m_1)$. If so, we return the value n. Otherwise, we subtract $s(m_1)$ from n and we call the modulo algorithm recursively with the values $n - s(m_1)$ and $s(m_1)$. To the recursive call, we have to supply a proof that the value $n - s(m_1)$ is accessible, which is given by the expression $h_1(-(n, s(m_1)), -<(n, m_1, h))$.

Finally, if the function $\mathsf{allacc}_{\mathsf{N}}$ takes a natural numbers and returns a proof that the natural number is accessible by $<$, we can define the following function:

$\mathsf{Mod}_{\mathsf{acc}} \in (n, m \in \mathsf{N}) \vee (\mathsf{N}, \mathsf{Error})$
$\quad \mathsf{Mod}_{\mathsf{acc}}(n, m) \equiv \mathsf{mod}_{\mathsf{acc}}(n, m, \mathsf{allacc}_{\mathsf{N}}(n))$

The main disadvantage of this formalisation of the modulo algorithm is that we have to supply a proof that $n - s(m_1)$ is accessible by $<$ to the recursive call. This proof has no computational meaning and its only purpose is to serve as a structurally smaller argument on which to perform the recursion and, in this way, guarantee the termination of the modulo algorithm. Notice that, even for such a small example, this accessibility proof distracts our attention and enlarges the code of the algorithm.

3.1.3 Using a special-purpose accessibility predicate for the formalisation

To overcome the problem described above, we define a special-purpose accessibility predicate for the modulo algorithm, that we call ModAcc. This predicate contains useful information that can help us to write a new version of the algorithm in type theory.

To construct our special-purpose accessibility predicate we ask ourselves the following question: on which inputs does the modulo algorithm terminate? To

find the answer to this question, we inspect the Haskell version of the modulo algorithm, putting special attention on the input values, the conditions that should be satisfied in order to give a basic result or to perform a recursive call, and the values on which we perform the recursive call. We distinguish three cases:

- if the input numbers are n and 0, then the algorithm terminates;
- if the input number n is less than the input number m, then the algorithm terminates;
- if the number n is not less than the number m and m is not zero (observe that this condition is not needed in the Haskell version of the algorithm due to the way Haskell processes the equations that define an algorithm), then the algorithm can only terminate on the inputs n and m if it terminates on the inputs $n - m$ and m.

Following this description, the ALF definition of the inductive predicate ModAcc over pairs of natural numbers is the following:

```
ModAcc ∈ (n, m ∈ N) Set
    modacc0 ∈ (n ∈ N) ModAcc(n, 0)
    modacc< ∈ (<(n, m)) ModAcc(n, m)
    modacc≤ ∈ (¬(=(m, 0)); ¬(<(n, m)); ModAcc(- (n, m), m)) ModAcc(n, m)
```

We can now use this predicate to formalise the modulo algorithm in type theory as follows:

```
mod ∈ (n, m ∈ N; ModAcc(n, m)) ∨(N, Error)
    mod(n, _, modacc0(_)) ≡ ∨R(error)
    mod(n, m, modacc<(h1)) ≡ ∨L(n)
    mod(n, m, modacc≤(h, h1, h2)) ≡ mod(- (n, m), m, h2)
```

This function is defined by structural recursion on the proof that the input pair of numbers satisfies the predicate ModAcc. To write the algorithm, we first perform pattern matching over the proof that the input pair of numbers satisfies the predicate ModAcc. As a result of the pattern matching we obtain three equations, one for each of the introduction rules of the predicate. The first equation considers the case where m is zero, and then we should return an error. The second equation considers the case where n is less than m being h_1 a proof of it, and we return the value n. The last equation considers the case where n is not less than m. Here, h is a proof that m is different from zero, h_1 is a proof that n is not less than m and h_2 is a proof that the pair $(n - m, m)$ satisfies the predicate ModAcc. Then, we call the algorithm recursively with the values $n - m$ and m. To the recursive call we have to supply a proof that the pair $(n - m, m)$ satisfies the predicate ModAcc which is given by h_2.

Now, we present the proof that the modulo algorithm terminates on all possible inputs, that is, the proof that all pairs of natural numbers satisfy our special-purpose accessibility predicate ModAcc.

$\mathrm{modacc}_{\mathrm{aux}} \in (m, p \in \mathrm{N};\ \mathrm{Acc}(\mathrm{N}, <, p);\ f \in (q \in \mathrm{N}; <(q, p))\,\mathrm{ModAcc}(q, m))\ \mathrm{ModAcc}(p, m)$

$\mathrm{modacc}_{\mathrm{aux}}(0, p, h, f) \equiv \mathrm{modacc0}(p)$

$\mathrm{modacc}_{\mathrm{aux}}(\mathrm{s}(m_1), p, h, f) \equiv$

case $<_{\mathrm{dec}}(p, \mathrm{s}(m_1)) \in \mathrm{Dec}(<(p, \mathrm{s}(m_1)))$ **of**

$\mathrm{yes}(h_1) \Rightarrow \mathrm{modacc}_{<}(h_1)$

$\mathrm{no}(h_1) \Rightarrow \mathrm{modacc}_{\leq}(\neg \mathrm{s}{=}0(m_1), h_1, f(\text{-}(p, \mathrm{s}(m_1)), \text{-}{<}(p, m_1, h_1)))$

end

$\mathrm{P_n} \in (n \in \mathrm{N})\ \forall(\mathrm{N}, [m]\mathrm{ModAcc}(n, m))$

$\mathrm{P_n}(n) \equiv \forall_{\mathrm{I}}([m]\mathrm{wfrec}(n, \mathrm{allacc}_{\mathrm{N}}(n), \mathrm{modacc}_{\mathrm{aux}}(m)))$

$\mathrm{allModAcc} \in (n, m \in \mathrm{N})\ \mathrm{ModAcc}(n, m)$

$\mathrm{allModAcc}(n, m) \equiv \forall_{\mathrm{E}}(\mathrm{P_n}(n), m)$

Notice that the skeleton of the proof of the function $\mathrm{modacc}_{\mathrm{aux}}$ is very similar to the skeleton of the algorithm $\mathrm{mod}_{\mathrm{acc}}$.

We can use the previous function to write our final algorithm.

$\mathrm{Mod} \in (n, m \in \mathrm{N})\ \vee(\mathrm{N}, \mathrm{Error})$

$\mathrm{Mod}(n, m) \equiv \mathrm{mod}(n, m, \mathrm{allModAcc}(n, m))$

Notice that we were able to move the non-computational parts from the code of the algorithm $\mathrm{mod}_{\mathrm{acc}}$ into the proof that the predicate ModAcc holds for all possible inputs. Thus, we are able to separate the algorithmic part of the definition from the proof of its termination. In this way, the code of the algorithm is not clouded with logical information, which is not interesting from a programming point of view. This fact makes the version of the algorithm that uses our special predicate quite compact and readable.

3.2 Second example: A unification algorithm

3.2.1 The Haskell version of the unification algorithm

In order to present the Haskell version of the unification algorithm, we need 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 function symbols, 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 function symbols.

A term is either a variable or a function applied to a (possibly empty) list of terms. We use t and lt (possibly primed or subscripted) to range over terms and lists of terms, respectively.

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). Observe that, as part of the definition of a substitution, we impose no restrictions on the variables that occur in the left hand sides of the pairs. 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 $[(x_1, t_1), \ldots, (x_n, t_n)]$ and a term t, the result of *applying* sb to t is denoted by $sb(t)$ and it is defined as the par-

allel 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 $sb(t_1) = sb(t_2)$. We say that 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 term t.

The unification algorithm we consider here is a deterministic version of the first (non-deterministic) algorithm presented by Martelli and Montanari [MM82]. Its Haskell version (once again, we ignore efficiency aspects) is presented in figure 2. 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.

The functions `length`, `zip`, `elem`, `++`, `==` and `&&` are predefined functions in Haskell: `length` takes a list and returns its length, `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.

```
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 2: Haskell version of the unification algorithm.

The function `varsT` returns the list of variables in a term, and the functions `substLPT` and `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 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 [], the substitution that results from the execution of `unify_H` *lp* will be the most general unifier of the input list *lp*. To define the algorithm `unify_h` we distinguish whether or not the input list is empty. If the input list is not empty, we consider four cases depending on the form of the terms in the first pair in the list. These cases are exhaustive and mutually disjoint.

3.2.2 Termination of the unification algorithm

As one can see from the definition given in figure 2, 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. To show that our unification algorithm always terminates we define a function that maps lists of pairs of terms into triples of natural numbers. This mapping, which we call $\mathrm{LPT_{to}N3}$, is such 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 input list of pairs of terms. This function $\mathrm{LPT_{to}N3}$ is a simplification of the function F presented in [MM82] to show the termination of their (non-deterministic) algorithm.

Consider the set N of natural numbers and the inequality relation $<$ over N with its usual meaning. Consider now the set N^3 of triples of natural numbers and the standard lexicographic order $<_{\mathrm{N3}}$ over triples. As the set of natural numbers is well-founded by $<$, it can be proven that the set N^3 of triples of natural numbers is well-founded by $<_{\mathrm{N3}}$.

To define the function $\mathrm{LPT_{to}N3}$, we use three auxiliary functions that take a list of pairs of terms and return a natural number. The first function, called $\#\mathrm{vars_{LPT}}$, takes a list of pairs of terms and returns the number of different variables that occur in the list. The second function, called $\#\mathrm{funs_{LPT}}$, takes a list of pairs of terms and returns the number of function applications that occur in the list. The last function, called $\#\mathrm{eqs_{LPT}}$, 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. Thus, we define the function $\mathrm{LPT_{to}N3}$ as follows:

$$\mathrm{LPT_{to}N3} :: \mathrm{ListPT} \to \mathrm{N}^3$$
$$\mathrm{LPT_{to}N3}(lp) = (\#\mathrm{vars_{LPT}}(lp), \#\mathrm{funs_{LPT}}(lp), \#\mathrm{eqs_{LPT}}(lp))$$

The termination of the unification algorithm is guaranteed since the following inequalities (whose proofs are straightforward) hold:

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

where $lp\,[x{:=}t]$ denotes the function that substitutes the term t for the variable x in the list of pairs of terms lp, $\mathrm{vars_T}$ is the function that returns the set of variables in a term and $\notin_{\mathrm{L}}$ 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 2).

3.2.3 Terms, lists of pairs of terms and substitutions in ALF

In order to formalise the set Term of terms in ALF, we use the notion of vectors of a certain length instead of the lists used in the Haskell version of the algorithm.

A vector is either empty and has length 0, or it has length $n+1$ and is formed by adding an element to a vector of length n. The type of vectors in type theory is the following:

Vector ∈ (n ∈ N; A ∈ **Set**) **Set**

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))

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)

We now define the set ListPT of lists of pairs of terms and the set Subst of substitutions:

PairT ∈ **Set**
  PairT ≡ Pair(Term, Term)
ListPT ∈ **Set**
  ListPT ≡ List(PairT)

PairS ∈ **Set**
  PairS ≡ Pair(Var, Term)
Subst ∈ **Set**
  Subst ≡ List(PairS)

In ALF, we write $:=_{\mathrm{T}}$, $:=_{\mathrm{LPT}}$ and $:=_{\mathrm{S}}$ for the functions that substitute a term for a variable in a term, in a list of pairs of terms and in a substitution respectively; we write $\mathrm{vars_T}$, $\mathrm{vars_{LPT}}$ and $\mathrm{vars_S}$ for the functions that return the list of

variables that occur in a term, in a list of pairs of terms and in a substitution respectively.

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

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

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

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

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

3.2.4 Using the standard accessibility predicate for the formalisation

In order to write the algorithm $\text{Unify}_{\text{acc}}$, which is the type-theoretic version of the unification algorithm that uses the standard accessibility predicate to handle the recursive calls, we first define a binary relation $<_{\text{LPT}}$ over lists of pairs of terms as follows:

$<_{\text{LPT}} \in (lp_1, lp_2 \in \text{ListPT})\ \textbf{Set}$

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

As the set N^3 is well-founded by $<_{\text{N3}}$, it is easy to prove that the set ListPT is well-founded by $<_{\text{LPT}}$. Hence, we define a function $\text{allacc}_{\text{LPT}}$ that given a list of pairs of terms returns a proof that the list is accessible by $<_{\text{LPT}}$.

Given a list of pairs of terms lp, the unification algorithm $\text{Unify}_{\text{acc}}$ 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 $\text{Unify}_{\text{acc}}$ calls the auxiliary algorithm $\text{unify}_{\text{acc}}$ with the list lp and the empty substitution, but now it also supplies a proof that the list lp is accessible by $<_{\text{LPT}}$, for which we use the function $\text{allacc}_{\text{LPT}}$.

$\text{Unify}_{\text{acc}} \in (lp \in \text{ListPT}) \vee(\text{Subst}, \text{Error})$

$\text{Unify}_{\text{acc}}(lp) \equiv \text{unify}_{\text{acc}}(lp, [], \text{allacc}_{\text{LPT}}(lp))$

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

$\text{unify}_{\text{acc}} \in (lp \in \text{ListPT};\ sb \in \text{Subst};\ \text{Acc}(\text{ListPT}, <_{\text{LPT}}, lp)) \vee(\text{Subst}, \text{Error})$

$\text{unify}_{\text{acc}}(lp, sb, \text{acc}(\_, h_1)) \equiv ?_{\text{unify\_acc.0.0.E}}$

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 $<_{\text{LPT}}$.

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.

After filling in the basic results, 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 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 the list to be unified and a proof that this list is smaller than the original list. We use the ALF lemmas presented in the previous section for the proofs of the inequalities that we supply to the function h_1.

In figure 3, we present the complete formalisation of the algorithm $\mathsf{unify}_{\mathsf{acc}}$.

Problems of this formalisation

If we compare the algorithms in figures 2 and 3, it is easy to see that the latter is much longer than the former, which makes the reading and the understanding of the algorithm in figure 3 more difficult. 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-theoretic version needs extra information in order to handle the recursive calls. In the type-theoretic version of the algorithm, the proofs that the lists to be unified are accessible serve as a structurally smaller argument on which to perform the recursion but they are computationally irrelevant. Moreover, these proofs are usually long, which contribute to distract our attention from the actual algorithmic part.

3.2.5 Using a special-purpose accessibility predicate for the formalisation

To overcome the problems described above, we define a special-purpose accessibility predicate, which we call UniAcc. This predicate contains useful information that allows us to perform the recursive calls of the unification algorithm in a simple way.

Intuitively, we can think of the predicate UniAcc as characterising 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'.

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;
- 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 $x \in \mathsf{vars}_{\mathrm{T}}(t)$ and $x \neq t$ (observe that this condition is not necessary in the Haskell version of the algorithm due to the way Haskell processes the equations that define an algorithm), 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 $x \notin_{\mathrm{L}} \mathsf{vars}_{\mathrm{T}}(t)$, then the al-

```
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(lp1, sb, h1(lp1, <_LPT(<_LPTvar_var(x1, lp1))))
            no(h) ⇒
                unify_acc(:=_LPT(x, var(x1), lp1),
                          :(:=_S(x, var(x1), sb), .(x, var(x1))),
                          h1(:=_LPT(x, var(x1), lp1), <_LPT(<_LPT:=var_term(lp1, ∉:(∉[](x), h)))))
        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(
                    :=_LPT(x, fun(f, lt), lp1),
                    :(:=_S(x, fun(f, lt), sb), .(x, fun(f, lt))),
                    h1(:=_LPT(x, fun(f, lt), lp1), <_LPT(<_LPT:=var_term(lp1, ¬∈_to∉(vars_T(fun(f, lt)), h)))))
        end
    unify_acc(:(lp1, .(fun(f, lt), var(x))), sb, acc(–, h1)) ≡
        unify_acc(:(lp1, .(var(x), fun(f, lt))),
                  sb,
                  h1(:(lp1, .(var(x), fun(f, lt))), <_LPT(<_LPTvar_fun(f, x, lt, lp1))))
    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(++(zip(lt1, lt2), lp1),
                                  sb,
                                  h1(++(zip(lt1, lt2), lp1), <_LPT(<_LPTzip_fun_fun(f2, f2, lt1, lt2, lp1))))
                    no(h2) ⇒ ∨_R(error)
                end
            no(h) ⇒ ∨_R(error)
        end
```

Figure 3: ALF definition of the algorithm $\mathsf{unify}_{\mathsf{acc}}$.

gorithm 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 has the form $(f(lt_1),\, g(lt_2)) : lp$ with $\mathsf{length}(lt_1) \neq \mathsf{length}(lt_2)$ or $f \neq g$, then the algorithm terminates since there does not exist a unifier for the input list;

- if the input has the form $(f(lt_1),\, g(lt_2)) : lp$ with $\mathsf{length}(lt_1) = \mathsf{length}(lt_2)$ and $f = g$, 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.

Following this description, the ALF definition of the inductive predicate UniAcc is the following:

```
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))))
```

Observe that 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. Notice that the Haskell function `zip` is not exactly the same function as the ALF 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 constructor, we directly use the function symbol f twice instead of using both function symbols f and g and having $f = g$ as part of the constructor.

Given the definition of the predicate UniAcc, it is possible to define a function $\mathsf{allUniAcc}_{\mathrm{LPT}}$ showing that all lists of pairs of terms satisfy the predicate. This proof is based on the facts that the set N^3 is well-founded and that the inequalities presented in Section 3.2.2 hold. This proof, discussed in [Bov99], has the same skeleton as the formalisation of the unification algorithm that uses the predicate Acc to handle the recursive calls.

We now describe how we can write the algorithm Unify in type theory using the UniAcc predicate to handle the recursive calls of the unification algorithm.

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 list to be unified satisfies the predicate UniAcc. Once we have performed pattern matching over the proof that the input list satisfies the predicate UniAcc, we obtain an incomplete ALF code with seven equations, one equation for each of the constructors of the predicate UniAcc. 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 "_". 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 thus we have to supply the new list to 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.

Below, we present the type-theoretic formalisation of the function unify.

$$\begin{array}{l}\mathsf{unify} \in (lp \in \mathsf{ListPT};\ sb \in \mathsf{Subst};\ \mathsf{UniAcc}(lp)) \vee (\mathsf{Subst}, \mathsf{Error}) \\ \quad \mathsf{unify}(\_, sb, \mathsf{uniacc}_{[]}) \equiv \vee_{\mathrm{L}}(sb) \\ \quad \mathsf{unify}(\_, sb, \mathsf{uniacc}_{\mathsf{var\_var}}(x, h_1)) \equiv \mathsf{unify}(lp_1, sb, h_1) \\ \quad \mathsf{unify}(\_, sb, \mathsf{uniacc}_{\mathsf{var\_term}}(lp_1, h_1, h_2)) \equiv \vee_{\mathrm{R}}(\mathsf{error}) \\ \quad \mathsf{unify}(\_, sb, \mathsf{uniacc}_{:=\mathsf{var\_term}}(h_1, h_2)) \equiv \mathsf{unify}(:=_{\mathrm{LPT}}(x, t, lp_1), :(:=_{\mathrm{S}}(x, t, sb), .(x, t)), h_2) \\ \quad \mathsf{unify}(\_, sb, \mathsf{uniacc}_{\mathsf{var\_fun}}(h_1)) \equiv \mathsf{unify}(:(lp_1, .(\mathsf{var}(x), \mathsf{fun}(f, lt))), sb, h_1) \\ \quad \mathsf{unify}(\_, sb, \mathsf{uniacc}_{\mathsf{fun\_fun}}(lt_1, lt_2, lp_1, h_1)) \equiv \vee_{\mathrm{R}}(\mathsf{error}) \\ \quad \mathsf{unify}(\_, sb, \mathsf{uniacc}_{\mathsf{zip\_fun\_fun}}(f, h_1)) \equiv \mathsf{unify}(++(\mathsf{zip}(lt_1, lt_2), lp_1), sb, h_1)\end{array}$$

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 algorithm.

In [Bov99], we have also proven the partial correctness of this formalisation. In our experience, this proof has also benefitted from the way we defined the predicate UniAcc and the unification algorithm. Most of the theorems we need in order to prove the partial correctness of the unification algorithm are defined by recursion on the proof that the input list satisfies the predicate UniAcc, and they are short and concise.

4 Conclusions

Here, we present some conclusions and related work.

Our work is heavily based on inductive families, for which ALF is very suitable. However, the formalisation of general recursive algorithm following our method 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].

We describe a method to formalise simple general recursive algorithms in type theory that separates the computational and logical parts of the definition or, in other words, the computational part and the proof of its total correctness. This fact has several advantages. First, the resulting type-theoretic algorithms are compact and easy to understand. They are as simple as their Haskell versions, where there exists no restriction on the recursive calls. Second, as totality is now a separate task, we can also use this method to formalise partial functions as is shown in [BC01]. Finally, our method also simplifies the task of formal verification. Often, in the process of verifying complex algorithms, the formalisation of the algorithm is so complicated and clouded with logical information, that the formal verification of its properties becomes very difficult. If the algorithms are formalised as we propose, the simplicity of its definition would make the task of formal verification dramatically easier.

Although we have defined our special-purpose accessibility predicate manually, we have done so in a systematic way. Hence, we see no reason why this process cannot be made automatically. This mechanisation would require some further work since some minor decisions should be taken in order to define the special predicates from the Haskell algorithms.

The method we present here can be used with no problems in the formalisation of mutually general recursive algorithms. The extension of our method for nested recursive algorithms is, though, not that straightforward. If we would follow the method we describe here to formalise a nested algorithm f_alg, we would obtain a special-purpose predicate fAcc where the result of f_alg would appear in the premises of the rules of fAcc. This would at first seem to be not possible since fAcc would need f_alg to be defined and we would use fAcc to define f_alg. However, Dybjer's schema on simultaneous inductive-recursive definitions [Dyb00] gives us the means to define fAcc and f_alg in a simultaneous way. In [BC01], we show how we can formalise nested recursive algorithms using Dybjer's schema. The method for formalising nested algorithm is actually very similar to the one presented here for formalising simple general recursive algorithms.

4.1 Related work

There are not many studies on formalising general recursion in type theory, as far as we know. Nordström [Nor88] uses the predicate Acc for that purpose. Balaa and Bertot [BB00] use fix-point equations to obtain the desired equalities for the recursive definitions in Coq, but one still has to mix the actual algorithm

with proofs concerning the well-foundedness of the recursive calls.

Slind [Sli96] has studied the problem of defining general recursive functions in classical higher order logic. As his framework is quite different from ours, he does not face the same kind of problem as we do. In [Sli96], Slind develops a recursive induction principle for general recursive definitions. The power given by this induction principle is equivalent to structural induction over our special-purpose predicate.

Unification algorithms have been the centre of several studies. Here we briefly discuss only those that are more relevant. See [Bov99] for a more complete discussion of the related work on unification.

Paulson [Pau85] closely follows the work by Manna and Waldinger [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. Rouyer [Rou92] has presented a verification of a first-order unification algorithm using the Coq proof assistant. A special attention should be paid to McBride's formalisation (see [McB99]) of a unification algorithm in Lego [LP92], where he exploits the use of dependent types in programming. The way in which McBride defines terms and substitutions permits a reformulation of the unification problem in a structural way with a lexicographic recursive structure. In this way, McBride does not need to impose either an external termination ordering or an accessibility argument.

Acknowledgements. We want to thank two anonymous referees for carefully reading and commenting on a previous version of this paper.

References

[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`.

[BB00] A. Balaa and Y. Bertot. Fix-point equations for well-founded recursion in type theory. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2000.

[BC01] A. Bove and V. Capretta. Nested general recursion and partiality in type theory. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science, Springer-Verlag*, pages 121–135, September 2001.

[Bov99] A. Bove. Programming in Martin-Löf type theory: Unification - A non-trivial example, November 1999. Licentiate Thesis of the Department of Computer Science, Chalmers University of Technology. Available on the WWW `http://cs.chalmers.se/~bove/Papers/lic_thesis.ps.gz`.

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

[Coq92] T. Coquand. Pattern matching with dependent types. In *Proceeding from the logical framework workshop at Båstad*, June 1992.

[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.

[Dyb00] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.

[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.

[JHe+99] S. Peyton Jones, J. Hughes, (editors), L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. 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.

[McB99] C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, Department of Computer Science, University of Edinburgh, October 1999.

[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.

[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.

[Rou92] J. Rouyer. Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs. Technical Report 1795, INRIA-Lorraine, November 1992.

[Sli96] K. Slind. Function definition in higher-order logic. In *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, Turku, Finland, August 1996.

Paper II

Nested General Recursion and Partiality in Type Theory

Nested General Recursion and Partiality in Type Theory

Ana Bove
Department of Computing Science
Chalmers University of Technology
412 96 Göteborg, Sweden
e-mail: bove@cs.chalmers.se
telephone: +46-31-7721020, fax: +46-31-165655

Venanzio Capretta
Computing Science Institute, University of Nijmegen
Postbus 9010, 6500 GL Nijmegen, The Netherlands
e-mail: venanzio@cs.kun.nl
telephone: +31+24+3652647, fax: +31+24+3553450

September 2001

Abstract

We extend Bove's technique for formalising simple general recursive algorithms in constructive type theory to nested recursive algorithms. The method consists in defining an inductive special-purpose accessibility predicate, that characterises the inputs on which the algorithm terminates. As a result, the type-theoretic version of the algorithm can be defined by structural recursion on the proof that the input values satisfy this predicate. This technique results in definitions in which the computational and logical parts are clearly separated; hence, the type-theoretic version of the algorithm is given by its purely functional content, similarly to the corresponding program in a functional programming language. In the case of nested recursion, the special predicate and the type-theoretic algorithm must be defined simultaneously, because they depend on each other. This kind of definitions is not allowed in ordinary type theory, but it is provided in type theories extended with Dybjer's schema for simultaneous inductive-recursive definitions. The technique applies also to the formalisation of partial functions as proper type-theoretic functions, rather than relations representing their graphs.

1 Introduction

Constructive type theory (see for example [ML84, CH88]) can be seen as a programming language where specifications are represented as types and programs as elements of types. Therefore, algorithms are correct by construction or can be proved correct by using the expressive power of constructive type theory.

Although this paper is intended mainly for those who already have some knowledge of type theory, we recall the basic ideas that we use here. The basic notion in type theory is that of *type*. A type is explained by saying what its objects are and what it means for two of its objects to be equal. We write $a \in \alpha$ for "a is an object of type α".

We consider a basic type and two type formers.

The basic type comprises sets and propositions and we call it Set. Both sets and propositions are inductively defined. A proposition is interpreted as a set whose elements represent its proofs. In conformity with the explanation of what it means to be a type, we know that A is an object of Set if we know how to form its canonical elements and when two canonical elements are equal.

The first type former constructs the type of the elements of a set: for each set A, the elements of A form a type. If a is an element of A, we say that a has type A. Since every set is inductively defined, we know how to build its elements.

The second type former constructs the types of dependent functions. Let α be a type and β be a family of types over α, that is, for every element a in α, $\beta(a)$ is a type. We write $(x \in \alpha)\beta(x)$ for the type of dependent functions from α to β. If f has type $(x \in \alpha)\beta(x)$, then, when we apply f to an object a of type α, we obtain an object $f(a)$ of type $\beta(a)$.

A set former or, in general, any inductive definition is introduced as a constant A of type $(x_1 \in \alpha_1; \ldots; x_n \in \alpha_n)\mathsf{Set}$, for $\alpha_1, \ldots, \alpha_n$ types. We must specify the constructors that generate the elements of $A(a_1, \ldots, a_n)$ by giving their types, for $a_1 \in \alpha_1, \ldots, a_n \in \alpha_n$.

Abstractions are written 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. We write $(x_1, x_2, \ldots, x_n \in \alpha)$ instead of $(x_1 \in \alpha; x_2 \in \alpha; \ldots; x_n \in \alpha)$.

General recursive algorithms are defined by cases where the recursive calls are performed on objects that satisfy no syntactic condition guaranteeing termination. As a consequence, there is no direct way of formalising them in type theory. The standard way of handling general recursion in type theory uses a well-founded recursion principle derived from the accessibility predicate Acc (see [Acz77, Nor88, BB00]). The idea behind the accessibility predicate is that an element a is accessible by a relation $\prec$ if there exists no infinite decreasing sequence starting from a. A set A is said to be well-founded with respect to $\prec$ if all its elements are accessible by $\prec$. Formally, given a set A, a binary relation $\prec$ on A and an element a in A, we can form the set $\mathsf{Acc}(A, \prec, a)$. The only

introduction rule for the accessibility predicate is

$$\frac{a \in A \quad p \in (x \in A;\, h \in x \prec a)\mathsf{Acc}(A, \prec, x)}{\mathsf{acc}(a, p) \in \mathsf{Acc}(A, \prec, a)}$$

The corresponding elimination rule, also known as the rule of well-founded recursion, is

$$\frac{\begin{array}{c} a \in A \\ h \in \mathsf{Acc}(A, \prec, a) \\ e \in (x \in A;\, h_x \in \mathsf{Acc}(A, \prec, x);\, p_x \in (y \in A;\, q \in y \prec x)P(y))P(x) \end{array}}{\mathsf{wfrec}(a, h, e) \in P(a)}$$

and its computation rule is

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

Hence, to guarantee that a general recursive algorithm that performs the recursive calls on elements of type A terminates, we have to prove that A is well-founded and that the arguments supplied to the recursive calls are smaller than the input.

Since Acc is a general predicate, it gives no information that can help us in the formalisation of a specific recursive algorithm. As a consequence, its use in the formalisation of general recursive algorithms often results in long and complicated code. On the other hand, functional programming languages like Haskell [JHe+99] impose no restrictions on recursive programs; therefore, writing general recursive algorithms in Haskell is straightforward. In addition, functional programs are usually short and self-explanatory. However, there is no powerful framework to reason about the correctness of Haskell-like programs.

Bove [Bov01] introduces a method to formalise simple general recursive algorithms in type theory (by simple we mean non-nested and non-mutually recursive) in a clear and compact way. We believe that this technique helps to close the gap between programming in a functional language and programming in type theory.

This work is similar to that of Paulson in [Pau86]. He defines an ordering associated with the recursive steps of an algorithm, such that the inputs on which the algorithm terminates are the objects accessible by the order. Then he defines the algorithm by induction on the order. The proof of termination for the algorithm reduces to a proof that the order is well-founded. Bove's idea is a translation of this in the framework of type theory in a way more convenient than the straightforward translation. Given the Haskell version of an algorithm f_alg, the method in [Bov01] uses an inductive special-purpose accessibility predicate called fAcc. We construct this predicate directly from f_alg, and we regard it as a characterisation of the collection of inputs on which f_alg terminates. It has an introduction rule for each case in the algorithm and provides a syntactic condition that guarantees termination. In this way, we can formalise f_alg in type theory by structural recursion on the proof that the input of f_alg satisfies fAcc, obtaining a compact and readable formalisation of the algorithm.

However, the technique in [Bov01] cannot be immediately applied to nested recursive algorithms. Here, we present a method for formalising nested recursive algorithms in type theory in a similar way to the one used in [Bov01]. Thus, we obtain short and clear formalisations of nested recursive algorithms in type theory. This technique uses the schema for simultaneous inductive-recursive definitions presented by Dybjer in [Dyb00]; hence, it can be used only in type theories extended with such schema.

The rest of the paper is organised as follows. In section 2, we illustrate the method used in [Bov01] on a simple example. In addition, we point out the advantages of this technique over the standard way of defining general recursive algorithms in type theory by using the predicate Acc. In section 3, we adapt the method to nested recursive algorithms, using Dybjer's schema. In section 4, we show how the method can be put to use also in the formalisation of partial functions. Finally, in section 5, we present some conclusions and related work.

2 Simple General Recursion in Type Theory

Here, we illustrate the technique used in [Bov01] on a simple example: the modulo algorithm on natural numbers. In addition, we point out the advantages of this technique over the standard way of defining general recursive algorithms in type theory by using the accessibility predicate Acc.

First, we give the Haskell version of the modulo algorithm. Second, we define the type-theoretic version of it that uses the standard accessibility predicate Acc to handle the recursive call, and we point out the problems of this formalisation. Third, we introduce a special-purpose accessibility predicate, ModAcc, specifically defined for this case study. Intuitively, this predicate defines the collection of pairs of natural numbers on which the modulo algorithm terminates. Fourth, we present a formalisation of the modulo algorithm in type theory by structural recursion on the proof that the input pair of natural numbers satisfies the predicate ModAcc. Finally, we show that all pairs of natural numbers satisfy ModAcc, which implies that the modulo algorithm terminates on all inputs.

In the Haskell definition of the modulo algorithm we use the set `N` of natural numbers, the subtraction operation `<->` and the less-than relation `<<` over `N`, defined in Haskell in the usual way. We also use Haskell's data type `Maybe A`, whose elements are `Nothing` and `Just a`, for any `a` of type `A`. Here is the Haskell code for the modulo algorithm[1]:

```
mod :: N -> N -> Maybe N
mod n 0 = Nothing
mod n m | n << m = Just n
        | not(n << m) = mod (n <-> m) m
```

It is evident that this algorithm terminates on all inputs. However, the recursive call is made on the argument $n - m$, which is not structurally smaller than the

[1] For the sake of simplicity, we ignore efficiency aspects such as the fact that the expression `n << m` is computed twice.

argument n, although the value of $n - m$ is smaller than n.

Before introducing the type-theoretic version of the algorithm that uses the standard accessibility predicate, we give the types of two operators and two lemmas[2]:

$$\begin{array}{lcllcl} - & \in & (n, m \in \mathsf{N})\mathsf{N} & \text{less-dec} & \in & (n, m \in \mathsf{N})\mathsf{Dec}(n < m) \\ < & \in & (n, m \in \mathsf{N})\mathsf{Set} & \text{min-less} & \in & (n, m \in \mathsf{N}; \neg(n < \mathsf{s}(m)))(n - \mathsf{s}(m) < n) \end{array}$$

On the left side we have the types of the subtraction operation and the less-than relation over natural numbers. On the right side we have the types of two lemmas that we use later on. The first lemma states that it is decidable whether a natural number is less than another. The second lemma establishes that if the natural number n is not less than the natural number $\mathsf{s}(m)$, then the result of subtracting $\mathsf{s}(m)$ from n is less than n [3].

In place of Haskell's `Maybe` type, we use the type-theoretic disjunction of the set N of natural numbers and the singleton set Error whose only element is error. The type-theoretic version of the modulo algorithm that uses the standard accessibility predicate Acc to handle the recursive call is[4]

$$\begin{array}{l} \mathsf{mod_{acc}} \in (n, m \in \mathsf{N}; \mathsf{Acc}(\mathsf{N}, <, n))\mathsf{N} \vee \mathsf{Error} \\ \quad \mathsf{mod_{acc}}(n, 0, \mathsf{acc}(n, p)) = \mathsf{inr}(\mathsf{error}) \\ \quad \mathsf{mod_{acc}}(n, \mathsf{s}(m_1), \mathsf{acc}(n, p)) = \\ \quad\quad \textbf{case } \mathsf{less\text{-}dec}(n, \mathsf{s}(m_1)) \in \mathsf{Dec}(n < \mathsf{s}(m_1)) \textbf{ of} \\ \quad\quad\quad \mathsf{inl}(q_1) \;\Rightarrow\; \mathsf{inl}(n) \\ \quad\quad\quad \mathsf{inr}(q_2) \;\Rightarrow\; \mathsf{mod_{acc}}(n - \mathsf{s}(m_1), \mathsf{s}(m_1), p(n - \mathsf{s}(m_1), \mathsf{min\text{-}less}(n, m_1, q_2))) \\ \quad\quad \textbf{end} \end{array}$$

This algorithm is defined by recursion on the proof that the first argument of the modulo operator is accessible by $<$. We first distinguish cases on m. If m is zero, we return an error, because the modulo zero operation is not defined. If m is equal to $\mathsf{s}(m_1)$ for some natural number m_1, we distinguish cases on whether n is smaller than $\mathsf{s}(m_1)$. If so, we return the value n. Otherwise, we subtract $\mathsf{s}(m_1)$ from n and we call the modulo algorithm recursively on the values $n - \mathsf{s}(m_1)$ and $\mathsf{s}(m_1)$. The recursive call needs a proof that the value $n - \mathsf{s}(m_1)$ is accessible. This proof is given by the expression $p(n - \mathsf{s}(m_1), \mathsf{min\text{-}less}(n, m_1, q_2))$, which is structurally smaller than $\mathsf{acc}(n, p)$.

We can easily define a function $\mathsf{allacc_N}$ that, applied to a natural number n, returns a proof that n is accessible by $<$. We use this function to define the desired modulo algorithm:

$$\begin{array}{l} \mathsf{Mod_{acc}} \in (n, m \in \mathsf{N})\mathsf{N} \vee \mathsf{Error} \\ \quad \mathsf{Mod_{acc}}(n, m) = \mathsf{mod_{acc}}(n, m, \mathsf{allacc_N}(n)) \end{array}$$

[2] Dec is the decidability predicate: given a proposition P, $\mathsf{Dec}(P) \equiv P \vee \neg P$.

[3] The hypothesis $(n - \mathsf{s}(m) < n)$ is necessary because the subtraction of a larger number from a smaller one is set to be 0 by default.

[4] The set former $\vee$ represents the disjunction of two sets, and inl and inr the two constructors of the set.

The main disadvantage of this formalisation of the modulo algorithm is that we have to supply a proof that $n - \mathsf{s}(m_1)$ is accessible by $<$ to the recursive call. This proof has no computational content and its only purpose is to serve as a structurally smaller argument on which to perform the recursion. Notice that, even for such a small example, this accessibility proof distracts our attention and enlarges the code of the algorithm.

To overcome this problem, we define a special-purpose accessibility predicate, ModAcc, containing information that helps us to write a new type-theoretic version of the algorithm. To construct this predicate, we ask ourselves the following question: on which inputs does the modulo algorithm terminate? To find the answer, we inspect closely the Haskell version of the modulo algorithm. We can directly extract from its structure the conditions that the input values should satisfy to produce a basic (that is, non recursive) result or to perform a terminating recursive call. In other words, we formulate the property that an input value must satisfy for the computation to terminate: either the algorithm does not perform any recursive call, or the values on which the recursive calls are performed have themselves the property. We distinguish three cases:

- if the input numbers are n and zero, then the algorithm terminates;
- if the input number n is less than the input number m, then the algorithm terminates;
- if the number n is not less than the number m and m is not zero[5], then the algorithm terminates on the inputs n and m if it terminates on the inputs $n - m$ and m.

Following this description, we define the inductive predicate ModAcc over pairs of natural numbers by the introduction rules (for n and m natural numbers)

$$\frac{}{\mathsf{ModAcc}(n,0)} \qquad \frac{n < m}{\mathsf{ModAcc}(n,m)} \qquad \frac{\neg(m=0) \quad \neg(n<m) \quad \mathsf{ModAcc}(n-m,m)}{\mathsf{ModAcc}(n,m)}$$

This predicate can easily be formalised in type theory:

$$\begin{array}{l}
\mathsf{ModAcc} \in (n,m \in \mathsf{N})\mathsf{Set} \\
\quad \begin{array}{lcl}
\mathsf{modacc}_0 & \in & (n \in \mathsf{N})\mathsf{ModAcc}(n,0) \\
\mathsf{modacc}_< & \in & (n,m \in \mathsf{N}; n<m)\mathsf{ModAcc}(n,m) \\
\mathsf{modacc}_\geq & \in & (n,m \in \mathsf{N}; \neg(m=0); \neg(n<m); \mathsf{ModAcc}(n-m,m)) \\
 & & \mathsf{ModAcc}(n,m)
\end{array}
\end{array}$$

We now use this predicate to formalise the modulo algorithm in type theory:

$$\begin{array}{l}
\mathsf{mod} \in (n,m \in \mathsf{N}; \mathsf{ModAcc}(n,m))\mathsf{N} \vee \mathsf{Error} \\
\quad \mathsf{mod}(n,0,\mathsf{modacc}_0(n)) = \mathsf{inr}(\mathsf{error}) \\
\quad \mathsf{mod}(n,m,\mathsf{modacc}_<(n,m,q)) = \mathsf{inl}(n) \\
\quad \mathsf{mod}(n,m,\mathsf{modacc}_\geq(n,m,q_1,q_2,h)) = \mathsf{mod}(n-m,m,h)
\end{array}$$

[5]Observe that this condition is not needed in the Haskell version of the algorithm due to the order in which Haskell processes the equations that define an algorithm.

This algorithm is defined by structural recursion on the proof that the input pair of numbers satisfies the predicate ModAcc. The first two equations are straightforward. The last equation considers the case where n is not less than m; here q_1 is a proof that m is different from zero, q_2 is a proof that n is not less than m and h is a proof that the pair $(n-m,\, m)$ satisfies the predicate ModAcc. In this case, we call the algorithm recursively on the values $n-m$ and m. We have to supply a proof that the pair $(n-m,\, m)$ satisfies the predicate ModAcc to the recursive call, which is given by the argument h.

To prove that the modulo algorithm terminates on all inputs, we use the auxiliary lemma $\mathsf{modacc_{aux}}$. Given a natural number m, this lemma proves $\mathsf{ModAcc}(i, m)$, for i an accessible natural number, from the assumption that $\mathsf{ModAcc}(j, m)$ holds for every natural number j smaller than i. The proof proceeds by case analysis on m and, when m is equal to $\mathsf{s}(m_1)$ for some natural number m_1, by cases on whether i is smaller than $\mathsf{s}(m_1)$. The term $\mathsf{nots0}(m_1)$ is a proof that $\mathsf{s}(m_1)$ is different from 0.

$$
\begin{array}{l}
\mathsf{modacc_{aux}} \;\in\; (m, i \in \mathsf{N}; \mathsf{Acc}(\mathsf{N}, <, i); f \in (j \in \mathsf{N}; j < i)\mathsf{ModAcc}(j, m)) \\
\qquad\qquad\quad \mathsf{ModAcc}(i, m) \\
\quad \mathsf{modacc_{aux}}(0, i, h, f) = \mathsf{modacc_0}(i) \\
\quad \mathsf{modacc_{aux}}(\mathsf{s}(m_1), i, h, f) = \\
\qquad \textbf{case}\ \mathsf{less\text{-}dec}(i, \mathsf{s}(m_1)) \in \mathsf{Dec}(i < \mathsf{s}(m_1))\ \textbf{of} \\
\qquad\quad \mathsf{inl}(q_1) \;\Rightarrow\; \mathsf{modacc_<}(i, \mathsf{s}(m_1), q_1) \\
\qquad\quad \mathsf{inr}(q_2) \;\Rightarrow\; \mathsf{modacc_\geq}(i, \mathsf{s}(m_1), \mathsf{nots0}(m_1), q_2, \\
\qquad\qquad\qquad\qquad f(i - \mathsf{s}(m_1), \mathsf{min\text{-}less}(i, m_1, q_2))) \\
\qquad \textbf{end}
\end{array}
$$

Now, we prove that the modulo algorithm terminates on all inputs, that is, we prove that all pairs of natural numbers satisfy ModAcc[6]:

$$
\begin{array}{l}
\mathsf{allModAcc} \in (n, m \in \mathsf{N})\mathsf{ModAcc}(n, m) \\
\quad \mathsf{allModAcc}(n, m) = \mathsf{wfrec}(n, \mathsf{allacc_N}(n), \mathsf{modacc_{aux}}(m))
\end{array}
$$

Notice that the skeleton of the proof of the function $\mathsf{modacc_{aux}}$ is very similar to the skeleton of the algorithm $\mathsf{mod_{acc}}$.

Finally, we can use the previous function to write the final modulo algorithm:

$$
\begin{array}{l}
\mathsf{Mod} \in (n, m \in \mathsf{N})\mathsf{N} \vee \mathsf{Error} \\
\quad \mathsf{Mod}(n, m) = \mathsf{mod}(n, m, \mathsf{allModAcc}(n, m))
\end{array}
$$

Observe that, even for such a small example, the version of the algorithm that uses our special predicate is slightly shorter and more readable than the type-theoretic version of the algorithm that is defined by using the predicate Acc. Notice also that we were able to move the non-computational parts from the code of $\mathsf{mod_{acc}}$ into the proof that the predicate ModAcc holds for all possible inputs, thus separating the actual algorithm from the proof of its termination.

We hope that, by now, the reader is quite familiar with our notation. So, in the following sections, we will not explain the type-theoretic codes in detail.

[6] Here, we use the general recursor wfrec with the elimination predicate $P(n) \equiv \mathsf{ModAcc}(n, m)$.

3 Nested Recursion in Type Theory

The technique we have just described to formalise simple general recursion cannot be applied to nested general recursive algorithms in a straightforward way. We illustrate the problem on a simple nested recursive algorithm over natural numbers. Its Haskell definition is

```
nest :: N -> N
nest 0 = 0
nest (S n) = nest(nest n)
```

Clearly, this is a total algorithm returning 0 on every input.

If we want to use the technique described in the previous section to formalise this algorithm, we need to define an inductive special-purpose accessibility predicate NestAcc over the natural numbers. To construct NestAcc, we ask ourselves the following question: on which inputs does the nest algorithm terminate? By inspecting the Haskell version of the nest algorithm, we distinguish two cases:

- if the input number is 0, then the algorithm terminates;
- if the input number is $\mathsf{s}(n)$ for some natural number n, then the algorithm terminates if it terminates on the inputs n and $\mathsf{nest}(n)$.

Following this description, we define the inductive predicate NestAcc over natural numbers by the introduction rules (for n natural number)

$$\frac{}{\mathsf{NestAcc}(0)} \qquad \frac{\mathsf{NestAcc}(n) \quad \mathsf{NestAcc}(\mathsf{nest}(n))}{\mathsf{NestAcc}(\mathsf{s}(n))}$$

Unfortunately, this definition is not correct since nest is not yet defined. Moreover, the purpose of defining the predicate NestAcc is to be able to define the algorithm nest by structural recursion on the proof that its input value satisfies NestAcc. Hence, the definitions of NestAcc and nest are locked in a vicious circle.

However, there is an extension of type theory that gives us the means to define the predicate NestAcc inductively generated by two constructors corresponding to the two introduction rules of the previous paragraph. This extension has been introduced by Dybjer in [Dyb00] and it allows the simultaneous definition of an inductive predicate P and a function f, where f has the predicate P as part of its domain and is defined by recursion on P. In our case, given the input value n, nest requires an argument of type $\mathsf{NestAcc}(n)$. Using Dybjer's schema, we can simultaneously define NestAcc and nest:

$$\begin{array}{lcl} \mathsf{NestAcc} & \in & (n \in \mathsf{N})\mathsf{Set} \\ \mathsf{nest} & \in & (n \in \mathsf{N}; \mathsf{NestAcc}(n))\mathsf{N} \end{array}$$

$$\begin{array}{lcl} \mathsf{nestacc0} & \in & \mathsf{NestAcc}(0) \\ \mathsf{nestaccs} & \in & (n \in \mathsf{N}; h_1 \in \mathsf{NestAcc}(n); h_2 \in \mathsf{NestAcc}(\mathsf{nest}(n, h_1))) \\ & & \mathsf{NestAcc}(\mathsf{s}(n)) \end{array}$$

$$\begin{array}{l} \mathsf{nest}(0, \mathsf{nestacc0}) = 0 \\ \mathsf{nest}(\mathsf{s}(n), \mathsf{nestaccs}(n, h_1, h_2)) = \mathsf{nest}(\mathsf{nest}(n, h_1), h_2) \end{array}$$

This definition may at first look circular: the type of nest requires that the predicate NestAcc is defined, while the type of the constructor nestaccs of the predicate NestAcc requires that nest is defined. However, we can see that it is not so by analysing how the elements in NestAcc and the values of nest are generated. First of all, NestAcc(0) is well defined because it does not depend on any assumption and its only element is nestacc0. Once NestAcc(0) is defined, the result of nest on the inputs 0 and nestacc0 becomes defined and its value is 0. Now, we can apply the constructor nestaccs to the arguments $n = 0$, $h_1 = \mathsf{nestacc0}$ and $h_2 = \mathsf{nestacc0}$. This application is well typed since h_2 must be an element in $\mathsf{NestAcc}(\mathsf{nest}(0, \mathsf{nestacc0}))$, that is, NestAcc(0). At this point, we can compute the value of $\mathsf{nest}(\mathsf{s}(0), \mathsf{nestaccs}(0, \mathsf{nestacc0}, \mathsf{nestacc0}))$ and obtain the value zero[7], and so on. Circularity is avoided because the values of nest can be computed at the moment a new proof of the predicate NestAcc is generated; in turn, each constructor of NestAcc calls nest only on those arguments that appear previously in its assumptions, for which we can assume that nest has already been computed.

The next step consists in proving that the predicate NestAcc is satisfied by all natural numbers:

$$\mathsf{allNestAcc} \in (n \in \mathsf{N})\mathsf{NestAcc}(n)$$

This can be done by first proving that, given a natural number n and a proof h of $\mathsf{NestAcc}(n)$, $\mathsf{nest}(n, h) \leq n$ (by structural recursion on h), and then using well-founded recursion on the set of natural numbers.

Now, we define Nest as a function from natural numbers to natural numbers:

$$\begin{array}{l}\mathsf{Nest} \in (n \in \mathsf{N})\mathsf{N} \\ \quad \mathsf{Nest}(n) = \mathsf{nest}(n, \mathsf{allNestAcc}(n))\end{array}$$

Notice that by making the simultaneous definition of NestAcc and nest we can treat nested recursion similarly to how we treat simple recursion. In this way, we obtain a short and clear formalisation of the nest algorithm.

To illustrate our technique for nested general recursive algorithms in more interesting situations, we present a slightly more complicated example: Paulson's normalisation function for conditional expressions [Pau86]. Its Haskell definition is

```
data CExp = At | If CExp CExp CExp

nm :: CExp -> CExp
nm At = At
nm (If At y z) = If At (nm y) (nm z)
nm (If (If u v w) y z) = nm (If u (nm (If v y z))
                                  (nm (If w y z)))
```

[7] Since $\mathsf{nest}(\mathsf{s}(0), \mathsf{nestaccs}(0, \mathsf{nestacc0}, \mathsf{nestacc0})) = \mathsf{nest}(\mathsf{nest}(0, \mathsf{nestacc0}), \mathsf{nestacc0}) = \mathsf{nest}(0, \mathsf{nestacc0}) = 0$

To define the special-purpose accessibility predicate, we study the different equations in the Haskell version of the algorithm, putting the emphasis on the input expressions and the expressions on which the recursive calls are performed. We obtain the following introduction rules for the inductive predicate nmAcc (for y, z, u, v and w conditional expressions):

$$\frac{}{\mathsf{nmAcc}(\mathsf{At})} \qquad \frac{\mathsf{nmAcc}(y) \quad \mathsf{nmAcc}(z)}{\mathsf{nmAcc}(\mathsf{If}(\mathsf{At}, y, z))}$$

$$\frac{\begin{array}{c}\mathsf{nmAcc}(\mathsf{If}(v, y, z)) \\ \mathsf{nmAcc}(\mathsf{If}(w, y, z)) \\ \mathsf{nmAcc}(\mathsf{If}(u, \mathsf{nm}(\mathsf{If}(v, y, z)), \mathsf{nm}(\mathsf{If}(w, y, z))))\end{array}}{\mathsf{nmAcc}(\mathsf{If}(\mathsf{If}(u, v, w), y, z))}$$

In type theory, we define the inductive predicate nmAcc simultaneously with the function nm, recursively defined on nmAcc:

$$\begin{array}{lcl} \mathsf{nmAcc} & \in & (e \in \mathsf{CExp})\mathsf{Set} \\ \mathsf{nm} & \in & (e \in \mathsf{CExp}; \mathsf{nmAcc}(e))\mathsf{CExp} \end{array}$$

$$\begin{array}{lcl} \mathsf{nmacc}_1 & \in & \mathsf{nmAcc}(\mathsf{At}) \\ \mathsf{nmacc}_2 & \in & (y, z \in \mathsf{CExp}; \mathsf{nmAcc}(y); \mathsf{nmAcc}(z))\mathsf{nmAcc}(\mathsf{If}(\mathsf{At}, y, z)) \\ \mathsf{nmacc}_3 & \in & (u, v, w, y, z \in \mathsf{CExp}; \\ & & h_1 \in \mathsf{nmAcc}(\mathsf{If}(v, y, z)); h_2 \in \mathsf{nmAcc}(\mathsf{If}(w, y, z)); \\ & & h_3 \in \mathsf{nmAcc}(\mathsf{If}(u, \mathsf{nm}(\mathsf{If}(v, y, z), h_1), \mathsf{nm}(\mathsf{If}(w, y, z), h_2)))) \\ & & \mathsf{nmAcc}(\mathsf{If}(\mathsf{If}(u, v, w), y, z)) \end{array}$$

$$\begin{array}{l} \mathsf{nm}(\mathsf{At}, \mathsf{nmacc}_1) = \mathsf{At} \\ \mathsf{nm}(\mathsf{If}(\mathsf{At}, y, z), \mathsf{nmacc}_2(y, z, h_1, h_2)) = \mathsf{If}(\mathsf{At}, \mathsf{nm}(y, h_1), \mathsf{nm}(z, h_2)) \\ \mathsf{nm}(\mathsf{If}(\mathsf{If}(u, v, w), y, z), \mathsf{nmacc}_3(u, v, w, y, z, h_1, h_2, h_3)) = \\ \quad \mathsf{nm}(\mathsf{If}(u, \mathsf{nm}(\mathsf{If}(v, y, z), h_1), \mathsf{nm}(\mathsf{If}(w, y, z), h_2)), h_3) \end{array}$$

We can justify this definition as we did for the nest algorithm, reasoning about the well-foundedness of the recursive calls: the function nm takes a proof that the input expression satisfies the predicate nmAcc as an extra argument and it is defined by structural recursion on that proof, and each constructor of nmAcc calls nm only on those proofs that appear previously in its assumptions, for which we can assume that nm has already been computed.

Once again, the next step consists in proving that the predicate nmAcc is satisfied by all conditional expressions:

$$\mathsf{allnmAcc} \in (e \in \mathsf{CExp})\mathsf{nmAcc}(e)$$

To do this, we first show that the constructors of the predicate nmAcc use inductive assumptions on smaller arguments, though not necessarily structurally smaller ones. To that end, we define a measure that assigns a natural number to each conditional expression:

$$|\mathsf{At}| = 1 \quad \text{and} \quad |\mathsf{If}(x, y, z)| = |x| * (1 + |y| + |z|)$$

With this measure, it is easy to prove that

$$|\mathsf{If}(v,y,z)| < |\mathsf{If}(\mathsf{If}(u,v,w),y,z)|, \quad |\mathsf{If}(w,y,z)| < |\mathsf{If}(\mathsf{If}(u,v,w),y,z)|$$
$$\text{and} \quad |\mathsf{If}(u,v',w')| < |\mathsf{If}(\mathsf{If}(u,v,w),y,z)|$$

for every v', w' such that $|v'| \leq |\mathsf{If}(v,y,z)|$ and $|w'| \leq |\mathsf{If}(w,y,z)|$. Therefore, to prove that the predicate nmAcc holds for a certain $e \in \mathsf{CExp}$, we need to call nm only on those arguments that have smaller measure than e[8].

Now, we can prove that every conditional expression satisfies nmAcc by first proving that, given a conditional expression e and a proof h of $\mathsf{nmAcc}(e)$, $|\mathsf{nm}(e,h)| \leq |e|$ (by structural recursion on h), and then using well-founded recursion on the set of natural numbers.

We can then define NM as a function from conditional expressions to conditional expressions:

$$\mathsf{NM} \in (e \in \mathsf{CExp})\mathsf{CExp}$$
$$\mathsf{NM}(e) = \mathsf{nm}(e, \mathsf{allnmAcc}(e))$$

4 Partial Functions in Type Theory

Until now we have applied our technique to total functions for which totality could not be proven easily by structural recursion. However, it can also be put to use in the formalisation of partial functions. A standard way to formalise partial functions in type theory is to define them as relations rather than objects of a function type. For example, the minimisation operator for natural numbers, which takes a function $f \in (\mathsf{N})\mathsf{N}$ as input and gives the least $n \in \mathsf{N}$ such that $f(n) = 0$ as output, cannot be represented as an object of type $((\mathsf{N})\mathsf{N})\mathsf{N}$ because it does not terminate on all inputs. A standard representation of this operator in type theory is the inductive relation

$$\begin{array}{l} \mu \in (f \in (\mathsf{N})\mathsf{N}; n \in \mathsf{N})\mathsf{Set} \\ \quad \mu_0 \quad \in \quad (f \in (\mathsf{N})\mathsf{N}; f(0) = 0)\mu(f,0) \\ \quad \mu_1 \quad \in \quad (f \in (\mathsf{N})\mathsf{N}; f(0) \neq 0; n \in \mathsf{N}; \mu([m]f(\mathsf{s}(m)),n))\mu(f,\mathsf{s}(n)) \end{array}$$

The relation μ represents the graph of the minimisation operator. If we indicate the minimisation function by min, then $\mu(f,n)$ is inhabited if and only if $\mathsf{min}(f) = n$. The fact that min may be undefined on some function f is expressed by $\mu(f,n)$ being empty for every natural number n.

There are reasons to be unhappy with this approach. First, for a relation to really define a partial function, we must prove that it is univocal: in our case, that for all $n, m \in \mathsf{N}$, if $\mu(f,n)$ and $\mu(f,m)$ are both nonempty then $n = m$. Second, there is no computational content in this representation, that is, we cannot actually compute the value of $\mathsf{min}(f)$ for any f.

Let us try to apply our technique to this example and start with the Haskell definition of min:

[8] We could have done something similar in the case of the algorithm nest by defining the measure $|x| = x$ and proving the inequality $y < \mathsf{s}(x)$ for every $y \leq x$

```
min :: (N -> N) -> N
min f | f 0 == 0 = 0
      | f 0 /= 0 = s (min (\m -> f (s m)))
```

We observe that the computation of min on the input f terminates if $f(0) = 0$ or if $f(0) \neq 0$ and min terminates on the input $[m]f(\mathsf{s}(m))$. This leads to the inductive definition of the special predicate minAcc on functions defined by the introduction rules (for f a function from natural numbers to natural numbers and m a natural number)

$$\frac{f(0) = 0}{\mathsf{minAcc}(f)} \qquad \frac{f(0) \neq 0 \quad \mathsf{minAcc}([m]f(\mathsf{s}(m)))}{\mathsf{minAcc}(f)}$$

We can directly translate these rules into type theory:

$$\begin{array}{l} \mathsf{minAcc} \in (f \in (\mathsf{N})\mathsf{N})\mathsf{Set} \\ \quad \mathsf{minacc0} \quad \in \quad (f \in (\mathsf{N})\mathsf{N}; f(0) = 0)\mathsf{minAcc}(f) \\ \quad \mathsf{minacc1} \quad \in \quad (f \in (\mathsf{N})\mathsf{N}; f(0) \neq 0; \mathsf{minAcc}([m]f(\mathsf{s}(m))))\mathsf{minAcc}(f) \end{array}$$

Now, we define min for those inputs that satisfy minAcc:

$$\begin{array}{l} \mathsf{min} \in (f \in (\mathsf{N})\mathsf{N}; \mathsf{minAcc}(f))\mathsf{N} \\ \quad \mathsf{min}(f, \mathsf{minacc0}(f, q)) = 0 \\ \quad \mathsf{min}(f, \mathsf{minacc1}(f, q, h)) = \mathsf{s}(\mathsf{min}([m]f(\mathsf{s}(m)), h)) \end{array}$$

In this case, it is not possible to prove that all elements in (N)N satisfy the special predicate, simply because it is not true. However, given a function f, we may first prove $\mathsf{minAcc}(f)$ (that is, that the recursive calls in the definition of min are well-founded and, thus, that the function min terminates for the input f) and then use min to actually compute the value of the minimisation of f.

Partial functions can also be defined by occurrences of nested recursive calls, in which case we need to use simultaneous inductive-recursive definitions. We show how this works on the example of the normal-form function for terms of the untyped λ-calculus. The Haskell program that normalises λ-terms is

```
data Lambda = Var N | Abst N Lambda | App Lambda Lambda

sub :: Lambda -> N -> Lambda -> Lambda

nf :: Lambda -> Lambda
nf (Var i) = Var i
nf (Abst i a) = Abst i (nf a)
nf (App a b) = case (nf a) of
                 Var i -> App (Var i) (nf b)
                 Abst i a' -> nf (sub a' i b)
                 App a' a'' -> App (App a' a'') (nf b)
```

The elements of `Lambda` denote λ-terms: `Var i`, `Abst i a` and `App a b` denote the variable x_i, the term $(\lambda x_i.a)$ and the term $a(b)$, respectively. We assume

that a substitution algorithm `sub` is given, such that `(sub a i b)` computes the term $a[x_i := b]$.

Notice that the algorithm contains a hidden nested recursion: in the second sub-case of the case expression, the term `a'`, produced by the call `(nf a)`, appears inside the call `nf (sub a' i b)`. This sub-case could be written in the following way, where we abuse notation to make the nested calls explicit:

```
nf (App a b) = nf (let (Abst i a') = nf a in (sub a' i b))
```

Let Λ be the type-theoretic definition of `Lambda`. To formalise the algorithm, we use the method described in the previous section with simultaneous induction-recursion definitions. The introduction rules for the special predicate nfAcc, some of which use nf in their premises, are (for i natural number, and a, a', a'' and b λ-terms)

$$\frac{}{\mathsf{nfAcc}(\mathsf{Var}(i))} \qquad \frac{\mathsf{nfAcc}(a) \quad \mathsf{nfAcc}(b) \quad \mathsf{nf}(a) = \mathsf{Var}(i)}{\mathsf{nfAcc}(\mathsf{App}(a,b))}$$

$$\frac{\mathsf{nfAcc}(a)}{\mathsf{nfAcc}(\mathsf{Abst}(i,a))} \qquad \frac{\mathsf{nfAcc}(a) \quad \mathsf{nf}(a) = \mathsf{Abst}(i,a') \quad \mathsf{nfAcc}(\mathsf{sub}(a',i,b))}{\mathsf{nfAcc}(\mathsf{App}(a,b))}$$

$$\frac{\mathsf{nfAcc}(a) \quad \mathsf{nfAcc}(b) \quad \mathsf{nf}(a) = \mathsf{App}(a',a'')}{\mathsf{nfAcc}(\mathsf{App}(a,b))}$$

To write a correct type-theoretic definition, we must define the inductive predicate nfAcc simultaneously with the function nf, recursively defined on nfAcc:

$$\begin{array}{lll} \mathsf{nfAcc} & \in & (x \in \Lambda)\mathsf{Set} \\ \mathsf{nf} & \in & (x \in \Lambda; \mathsf{nfAcc}(x))\Lambda \end{array}$$

$$\begin{array}{lll}
\mathsf{nfacc}_1 & \in & (i \in \mathsf{N})\mathsf{nfAcc}(\mathsf{Var}(i)) \\
\mathsf{nfacc}_2 & \in & (i \in \mathsf{N}; a \in \Lambda; h_a \in \mathsf{nfAcc}(a))\mathsf{nfAcc}(\mathsf{Abst}(i,a)) \\
\mathsf{nfacc}_3 & \in & (a,b \in \Lambda; h_a \in \mathsf{nfAcc}(a); h_b \in \mathsf{nfAcc}(b); i \in \mathsf{N}; \mathsf{nf}(a,h_a) = \mathsf{Var}(i)) \\
 & & \mathsf{nfAcc}(\mathsf{App}(a,b)) \\
\mathsf{nfacc}_4 & \in & (a,b \in \Lambda; h_a \in \mathsf{nfAcc}(a); i \in \mathsf{N}; a' \in \Lambda; \\
 & & \mathsf{nf}(a,h_a) = \mathsf{Abst}(i,a'); \mathsf{nfAcc}(\mathsf{sub}(a',i,b))) \\
 & & \mathsf{nfAcc}(\mathsf{App}(a,b)) \\
\mathsf{nfacc}_5 & \in & (a,b \in \Lambda; h_a \in \mathsf{nfAcc}(a); h_b \in \mathsf{nfAcc}(b); \\
 & & a',a'' \in \Lambda; \mathsf{nf}(a,h_a) = \mathsf{App}(a',a'')) \\
 & & \mathsf{nfAcc}(\mathsf{App}(a,b))
\end{array}$$

$$\begin{array}{l}
\mathsf{nf}(\mathsf{Var}(i), \mathsf{nfacc}_1(i)) = \mathsf{Var}(i) \\
\mathsf{nf}(\mathsf{Abst}(i,a), \mathsf{nfacc}_2(i,a,h_a)) = \mathsf{Abst}(i, \mathsf{nf}(a,h_a)) \\
\mathsf{nf}(\mathsf{App}(a,b), \mathsf{nfacc}_3(a,b,h_a,h_b,i,q)) = \mathsf{App}(\mathsf{Var}(i), \mathsf{nf}(b,h_b)) \\
\mathsf{nf}(\mathsf{App}(a,b), \mathsf{nfacc}_4(a,b,h_a,i,a',q,h)) = \mathsf{nf}(\mathsf{sub}(a',i,b),h) \\
\mathsf{nf}(\mathsf{App}(a,b), \mathsf{nfacc}_5(a,b,h_a,h_b,a',a'',q)) = \mathsf{App}(\mathsf{App}(a',a''), \mathsf{nf}(b,h_b))
\end{array}$$

5 Conclusions and Related Work

We describe a technique to formalise algorithms in type theory that separates the computational and logical parts of the definition. As a consequence, the resulting type-theoretic algorithms are compact and easy to understand. They are as simple as their Haskell versions, where there is no restriction on the recursive calls. The technique was originally developed by Bove for simple general recursive algorithms. Here, we extend it to nested recursion using Dybjer's schema for simultaneous inductive-recursive definitions. We also show how we can use this technique to formalise partial functions. Notice that the proof of the special predicate for a particular input is a trace of the computation of the original algorithm, therefore its structural complexity is proportional to the number of steps of the algorithm.

We believe that our technique simplifies the task of formal verification. Often, in the process of verifying complex algorithms, the formalisation of the algorithm is so complicated and clouded with logical information, that the formal verification of its properties becomes very difficult. If the algorithm is formalised as we propose, the simplicity of its definition would make the task of formal verification dramatically easier.

The examples we presented have been formally checked using the proof assistant ALF (see [AGNvS94, MN94]), which supports Dybjer's schema.

There are not many studies on formalising general recursion in type theory, as far as we know. In [Nor88], Nordström uses the predicate Acc for that purpose. Balaa and Bertot [BB00] use fix-point equations to obtain the desired equalities for the recursive definitions, but one still has to mix the actual algorithm with proofs concerning the well-foundedness of the recursive calls. In any case, their methods do not provide simple definitions for nested recursive algorithms. Both Giesl [Gie97], from where we took some of our examples, and Slind [Sli00] have methods to define nested recursive algorithms independently of their proofs of termination. However, neither of them works in the framework of constructive type theory. Giesl works in first order logic and his main concern is to prove termination of nested recursive algorithms automatically. Slind works in classical higher order logic. He uses an inductive principle not available in type theory but closely similar to structural induction over our special purpose accessibility predicate.

Some work has been done in the area of formalising partial functions. Usually type theory is extended with partial objects or nonterminating computations. This is different from our method, in which partiality is realized by adding a new argument that restricts the domain of the original input; the function is still total in the two arguments. In [Con83], Constable associates a domain to every partial function. This domain is automatically generated from the function definition and contains basically the same information as our special-purpose predicates. However, the definition of the function does not depend on its domain as in our case. Based on this work, Constable and Mendler [CM85] introduce the type of partial functions as a new type constructor. In [CS87], Constable and Smith develop a partial type theory in which every type has a

twin containing diverging objects. Inspired by the work in [CS87], Audebaud [Aud91] introduces fix-points to the Calculus of Constructions [CH88], obtaining a conservative extension of it where the desired properties still hold.

Acknowledgement. We want to thank Herman Geuvers for carefully reading and commenting on a previous version of this paper.

References

[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`.

[Aud91] P. Audebaud. Partial Objects in the Calculus of Constructions. In *6th Annual IEEE Symposium on Logic in Computer Science, Amsterdam*, pages 86–95, July 1991.

[BB00] A. Balaa and Y. Bertot. Fix-point equations for well-founded recursion in type theory. In Harrison and Aagaard [HA00], pages 1–16.

[Bov01] A. Bove. Simple general recursion in type theory. *Nordic Journal of Computing*, 8(1):22–42, Spring 2001.

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

[CM85] R. L. Constable and N. P. Mendler. Recursive Definitions in Type Theory. In *Logic of Programs, Brooklyn*, volume 193 of *Lecture Notes in Computer Science*, pages 61–78. Springer-Verlag, June 1985.

[Con83] R. L. Constable. Partial Functions in Constructive Type Theory. In *Theoretical Computer Science, 6th GI-Conference, Dortmund*, volume 145 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, January 1983.

[CS87] R. L. Constable and S. F. Smith. Partial Objects in Constructive Type Theory. In *Logic in Computer Science, Ithaca, New York*, pages 183–193, June 1987.

[Dyb00] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.

[Gie97] J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19:1–29, 1997.

[HA00] J. Harrison and M. Aagaard, editors. *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[JHe+99] S. Peyton Jones, J. Hughes, (editors), L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[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.

[Nor88] B. Nordström. Terminating General Recursion. *BIT*, 28(3):605–619, October 1988.

[Pau86] L. C. Paulson. Proving Termination of Normalization Functions for Conditional Expressions. *Journal of Automated Reasoning*, 2:63–74, 1986.

[Sli00] K. Slind. Another look at nested recursion. In Harrison and Aagaard [HA00], pages 498–518.

Paper III

Mutual General Recursion in Type Theory

Mutual General Recursion in Type Theory

Ana Bove
Department of Computing Science
Chalmers University of Technology
412 96 Göteborg, Sweden
e-mail: bove@cs.chalmers.se
telephone: +46-31-7721020, fax: +46-31-165655

May 2002

Abstract

We show how the methodology presented by Bove for the formalisation of simple general recursive algorithms and extended by Bove and Capretta to treat nested recursion can also be used in the formalisation of mutual general recursive algorithms. The methodology consists of defining special-purpose accessibility predicates that characterise the inputs on which the algorithms terminate. Each algorithm is then formalised in type theory by structural recursion on the proof that its input satisfies the corresponding special-purpose accessibility predicate. When the mutually recursive algorithms are also nested, we make use of a generalisation of Dybjer's schema for simultaneous inductive-recursive definitions, which we also present in this work. Hence, some of the formalisations we present in this work are not allowed in ordinary type theory, but they can be carried out in type theories extended with such a schema. Similarly to what happens for simple and nested recursive algorithms, this methodology results in definitions in which the computational and logical parts are clearly separated also when the algorithms are mutually recursive. Hence, the type-theoretic version of the algorithms is given by its purely functional content, similarly to the corresponding program in a functional programming language.

1 Introduction

Following the Curry-Howard isomorphism [How80], constructive type theory (see [ML84, CH88]) can be seen as a programming language where specifications are represented as types and programs as objects of those types. Therefore, algorithms are correct by construction or can be proved correct by using the expressive power of constructive type theory. This is clearly an advantage of constructive type theory over standard programming languages.

General recursive algorithms are defined by cases where the recursive calls are on non-structurally smaller arguments. In other words, the recursive calls are performed on objects satisfying no syntactic condition that guarantees termination. As a consequence, there is no direct way of formalising this kind of algorithms in type theory.

The standard way of handling general recursion in type theory uses a well-founded recursion principle derived from the accessibility predicate Acc (see [Acz77, Nor88]). However, the use of this predicate in the type-theoretic formalisation of general recursive algorithms often results in unnecessarily long and complicated codes. Moreover, its use adds a considerable amount of code with no computational content that distracts our attention from the actual computational part of the algorithm (see for example [Bov99], where we present a formalisation of a unification algorithm over pairs of terms using the standard accessibility predicate Acc).

On the other hand, 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. Therefore, writing general recursive algorithms in Haskell is straightforward. In addition, functional programs are usually short and self-explanatory. However, the existing frameworks for reasoning 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 order to give a step towards closing the existing gap between programming in type theory and programming in a functional language, we have developed a methodology to formalise general recursive algorithms in type theory that separates the computational and logical parts of the definitions. As a consequence, the resulting type-theoretic algorithms are clear, compact and easy to understand. They are as simple as their Haskell versions, where there is no restriction on the recursive calls. Given a general recursive algorithm, the methodology consists of defining an inductive special-purpose accessibility predicate that characterises the inputs on which the algorithm terminates. The type-theoretic version of the algorithm can then be defined by structural recursion on the proof that the input values satisfy this predicate. If the algorithm has nested recursive calls, the special predicate and the type-theoretic algorithm must be defined simultaneously, because they depend on each other. This kind of definitions is not allowed in ordinary type theory, but it is provided in type theories extended with Dybjer's schema for simultaneous inductive-recursive definitions (see [Dyb00]).

Originally, this methodology was introduced in [Bov01] to formalise simple general recursive algorithms in type theory (by simple we mean non-nested and non-mutually recursive) and extended in [BC01] to treat nested recursion. In order to be able to formalise any general recursive algorithm in type theory using the same methodology, it remains to study how to formalise mutually recursive algorithms (nested and not nested). This is the purpose of this work.

In what follows we assume that the reader is familiar with the main concepts of constructive type theory. A short description of the main concepts of

type theory is given both in [Bov01] and [BC01]. For a more complete presentation of constructive type theory the reader is referred to [ML84, CH88, NPS90, CNSvS94]. In addition, we assume that the reader fully understands the methodology introduced in [Bov01, BC01] to formalise simple and nested general recursion respectively.

The rest of the paper is organised as follows. In section 2, we show, with the help of very simple examples, how to formalise mutually recursive algorithms using the methodology presented in [Bov01, BC01]. In section 3, we present the formalisation of more interesting nested and mutually recursive algorithms. In section 4, we discuss some conclusions. Finally, in appendix A, we introduce a generalisation of Dybjer's schema of simultaneous inductive-recursive definition for the cases where we have several mutually recursive predicates defined simultaneously with several functions which, in turn, are defined by recursion on those predicates.

2 Mutual Recursion

In [Bov01] and [BC01], we present how simple and nested general recursive algorithms, respectively, can be formalised in type theory in an easy way. With the help of some simple examples, we show here how to formalise mutually recursive algorithms using the methodology presented in [Bov01, BC01].

We start by presenting a non-structurally smaller version of the algorithms that determine whether a natural number is even or odd[1]. Following the same approach as in [Bov01, BC01], we start by introducing the Haskell version of the algorithms[2].

```
even :: N -> Bool
even Z = True
even (S n) = odd n

odd :: N -> Bool
odd Z = False
odd (S n) = not (even (S n))
```

Following the methodology presented in [Bov01, BC01] and in order to write the type-theoretic version of these algorithms, we first construct the special-purpose accessibility predicates associated with the algorithms. To construct those predicates, we study the Haskell code in order to characterise the inputs for which the algorithms terminate. Therefore, we distinguish the following cases:

- if the input is 0, the algorithm even terminates;

[1] Usually, the structurally smaller version of these algorithms is used. However, that version is of no interest for us in this work. Thus, we have modified it slightly in order to consider it as a simple case example.

[2] Here, we consider the set of natural numbers defined as `data N = Z | S N` in Haskell.

- if the input is $\mathsf{s}(n)$, for some natural number n, the algorithm even terminates if the algorithm odd terminates for the input n;
- if the input is 0, the algorithm odd terminates;
- if the input is $\mathsf{s}(n)$, for some natural number n, the algorithm odd terminates if the algorithm even terminates for the same input, that is, if it terminates for $\mathsf{s}(n)$.

Given this description, we can easily define the inductive predicates evenAcc and oddAcc over natural numbers by the following introduction rules (for n a natural number):

$$\frac{}{\mathsf{evenAcc}(0)} \qquad \frac{\mathsf{oddAcc}(n)}{\mathsf{evenAcc}(\mathsf{s}(n))} \qquad \frac{}{\mathsf{oddAcc}(0)} \qquad \frac{\mathsf{evenAcc}(\mathsf{s}(n))}{\mathsf{oddAcc}(\mathsf{s}(n))}$$

Observe that, whenever we have mutually recursive algorithms, the termination of one algorithm depends on the termination of the other(s). Hence, the special-purpose accessibility predicates associated with those algorithms are also mutually recursive.

Now, we can easily formalise the predicates evenAcc and oddAcc in type theory.

$$\begin{array}{l}
\mathsf{evenAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{evenacc0} \in \mathsf{evenAcc}(0) \\
\quad \mathsf{evenaccs} \in (n \in \mathsf{N}; h \in \mathsf{oddAcc}(n))\mathsf{evenAcc}(\mathsf{s}(n))
\end{array}$$

$$\begin{array}{l}
\mathsf{oddAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{oddacc0} \in \mathsf{oddAcc}(0) \\
\quad \mathsf{oddaccs} \in (n \in \mathsf{N}; h \in \mathsf{evenAcc}(\mathsf{s}(n)))\mathsf{oddAcc}(\mathsf{s}(n))
\end{array}$$

The algorithms can then be easily defined in the theory by structural recursion on the special-purpose predicates evenAcc and oddAcc as follows:

$$\begin{array}{l}
\mathsf{even} \in (m \in \mathsf{N}; \mathsf{evenAcc}(m))\mathsf{Bool} \\
\quad \mathsf{even}(0, \mathsf{evenacc0}) = \mathsf{True} \\
\quad \mathsf{even}(\mathsf{s}(n), \mathsf{evenaccs}(n, h)) = \mathsf{odd}(n, h)
\end{array}$$

$$\begin{array}{l}
\mathsf{odd} \in (m \in \mathsf{N}; \mathsf{oddAcc}(m))\mathsf{Bool} \\
\quad \mathsf{odd}(0, \mathsf{oddacc0}) = \mathsf{False} \\
\quad \mathsf{odd}(\mathsf{s}(n), \mathsf{oddaccs}(n, h)) = \mathsf{not}(\mathsf{even}(\mathsf{s}(n), h))
\end{array}$$

Observe the simplicity of this type-theoretic version of the algorithms and its similarity with the Haskell presentation of the algorithms that we introduced above. The reader is encouraged to write the type-theoretic version of the algorithms that uses the standard accessibility predicate Acc and compare afterwards the two type-theoretic versions.

Let us consider another simple example. Below, we have a mutually recursive version of the algorithm nest presented in [BC01].

```
f :: N -> N
f Z = Z
f (S n) = f (g n)

g :: N -> N
g Z = Z
g (S n) = g (f n)
```

Notice that we can easily prove that both f and g are equivalent to the function that returns 0 for any input n, for n a natural number. That is, $\forall n \in \mathsf{N}. f(n) = 0 \wedge g(n) = 0$.

Observe the nested calls of the algorithms f and g. Thus, if we want to define the special-purpose accessibility predicates fAcc and gAcc we face the same problems as we faced in [BC01] when we wanted to formalise the algorithm nest; namely, we need to know about the algorithms f and g in order to be able to define the special-purpose predicates which, in turn, allow us to define the algorithms f and g. As it is explained in [BC01], when we have nested recursive algorithms we need to define the special-purpose predicates and the functions simultaneously. In order to do so, in [BC01] we make use of Dybjer's schema for simultaneous inductive-recursive definitions and thus, we formalise the algorithms in type theories extended with such a schema. Since our algorithms are also mutually recursive, we have to define several predicates simultaneously with several functions (two in the example we present above). If we look at Dybjer's schema of simultaneously inductive-recursive definitions, we see that Dybjer only considers the case of one predicate and one function. Hence, in order to be able to formalise nested and mutually recursive algorithms, we need to extend Dybjer's schema so it can consider several inductive predicates defined simultaneously with several functions defined by recursion on those predicates. We present such a generalisation of Dybjer's schema in appendix A. In the rest of this work, we use this generalisation to formalise our nested and mutually recursive algorithms.

We now return to the example introduced above where we have a mutual and nested definition of the algorithms f and g. To define the special-purpose accessibility predicates we study the equations in the Haskell version of the algorithms, putting emphasis on the input expressions and the expressions on which we perform the recursive calls. We obtain then the following introduction rules for the inductive predicates fAcc and gAcc (for n a natural number):

$$\frac{}{\mathsf{fAcc}(0)} \qquad \frac{\mathsf{gAcc}(n) \quad \mathsf{fAcc}(\mathsf{g}(n))}{\mathsf{fAcc}(\mathsf{s}(n))} \qquad \frac{}{\mathsf{gAcc}(0)} \qquad \frac{\mathsf{fAcc}(n) \quad \mathsf{gAcc}(\mathsf{f}(n))}{\mathsf{gAcc}(\mathsf{s}(n))}$$

Formally, in type theory we define the inductive predicates fAcc and gAcc simultaneously with the algorithms f and g, recursively defined on the predicates.

$$
\begin{array}{l}
\mathsf{fAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{facc}_0 \in \mathsf{fAcc}(0) \\
\quad \mathsf{facc}_\mathsf{s} \in (n \in \mathsf{N}; h_1 \in \mathsf{gAcc}(n); h_2 \in \mathsf{fAcc}(\mathsf{g}(n, h_1)))\mathsf{fAcc}(\mathsf{s}(n))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{gAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{gacc}_0 \in \mathsf{gAcc}(0) \\
\quad \mathsf{gacc}_\mathsf{s} \in (n \in \mathsf{N}; h_1 \in \mathsf{fAcc}(n); h_2 \in \mathsf{gAcc}(\mathsf{f}(n, h_1)))\mathsf{gAcc}(\mathsf{s}(n))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{f} \in (m \in \mathsf{N}; \mathsf{fAcc}(m))\mathsf{N} \\
\quad \mathsf{f}(0, \mathsf{facc}_0) = 0 \\
\quad \mathsf{f}(\mathsf{s}(n), \mathsf{facc}_\mathsf{s}(n, h_1, h_2)) = \mathsf{f}(\mathsf{g}(n, h_1), h_2)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{g} \in (m \in \mathsf{N}; \mathsf{gAcc}(m))\mathsf{N} \\
\quad \mathsf{g}(0, \mathsf{gacc}_0) = 0 \\
\quad \mathsf{g}(\mathsf{s}(n), \mathsf{gacc}_\mathsf{s}(n, h_1, h_2)) = \mathsf{g}(\mathsf{f}(n, h_1), h_2)
\end{array}
$$

We can easily prove now that

$$\forall n \in \mathsf{N}.\forall h_1 \in \mathsf{fAcc}(n).\forall h_2 \in \mathsf{gAcc}(n).\mathsf{f}(n, h_1) = 0 \wedge \mathsf{g}(n, h_2) = 0$$

3 Two Other Examples

We present in this section the formalisation of more interesting nested and mutually recursive algorithms. The reader can check that the type-theoretic formalisations follow the schema presented in appendix A. Once again, the reader is encouraged to write the type-theoretic version of the algorithms that uses the standard accessibility predicate Acc and compare afterwards the two type-theoretic versions.

3.1 Terms Unification

The first example is a very well known and useful algorithm: a unification algorithm over terms, where a term is either a variable or a function applied to a list of terms. We assume that the set of variables and the set of functions are both infinite sets and that equality is decidable over them. Let us start by introducing some definitions in Haskell[3].

[3]Here, we define both the set of variables and the set of functions as the set of natural numbers. Any other definition that ensures the assumptions made over those sets is also possible.

```
type Var = N
type Fun = N
data Term = Var Var | Fun Fun [Term]
type ListPT = [(Term, Term)]
type Subst = [(Var, Term)]

vars :: Term -> [Var]
vars t = ....

appSb :: Subst -> ListPT -> ListPT
appSb sb lpt = ....
```

where `vars` is the function that returns the list of variables in a term t and `appSb` is the function that applies a substitution sb to each of the terms of a list of pair of terms lpt. Now, the algorithm that unifies a pair of terms can be defined as follows:

```
unifyPT :: (Term, Term) -> Maybe Subst
unifyPT (Var x, t) = if x `elem` vars t
                        then Nothing
                        else Just [(x,t)]
unifyPT (t, Var x) = if x `elem` vars t
                        then Nothing
                        else Just [(x,t)]
unifyPT (Fun f lt1, Fun g lt2) = if f /= g ||
                                    length lt1 /= length lt2
                                 then Nothing
                                 else unifyLPT (zip lt1 lt2)

unifyLPT :: ListPT -> Maybe Subst
unifyLPT [] = []
unifyLPT (p:lpt) = case unifyPT p of
                   Nothing -> Nothing
                   Just sb -> case unifyLPT (appSb sb lpt) of
                              Nothing -> Nothing
                              Just sb' -> Just (sb ++ sb')
```

where `/=` is the inequality operator in Haskell, `||` is the boolean disjunction, `length` computes the length of a list and `zip` takes two lists and returns a list of corresponding pairs.

The algorithm `unifyPT` returns a substitution that unifies a pair of terms pt if such a substitution exists or the value `Nothing` otherwise. Observe that the algorithms `unifyPT` and `unifyLPT` are mutually recursive algorithms. Notice the indirect nested recursive call in the definition of the algorithm `unifyLPT` since the expression `unifyLPT (appSb sb lpt)` is equivalent to the expression `unifyLPT (appSb (fromJust (unifyPT p)) lpt)` when we know that `unifyPT p` results in a substitution.

For clarity sake and in order to make our point more explicit, we actually consider a simplification of the unification algorithms presented above. Let us assume we already know that the algorithm `unifyPT` returns a substitution, in other words, let us assume we already know that the input pair of terms is unifiable. Then, we can present the algorithms introduced above in the following way[4]:

```
unifyPT :: (Term, Term) -> Subst
unifyPT (Var x, t) = [(x,t)]
unifyPT (t, Var x) = [(x,t)]
unifyPT (Fun f lt1, Fun g lt2) = unifyLPT (zip lt1 lt2)

unifyLPT :: ListPT -> Subst
unifyLPT [] = []
unifyLPT (p:lpt) = unifyPT p ++
                   unifyLPT (appSb (unifyPT p) lpt)
```

To define the special-purpose accessibility predicates associated with the algorithms we follow the same methodology as before. We inspect the Haskell version of the algorithms in order to characterise the set of inputs for which the algorithms terminate, putting emphasis on the relation between the input expressions and the expressions on which we perform the recursive calls. Thus, the introduction rules for the inductive predicates unifyPTAcc and unifyLPTAcc are as follows:

$$\frac{}{\mathsf{unifyPTAcc}(\mathsf{pair}(\mathsf{var}(x), t))} \qquad \frac{}{\mathsf{unifyPTAcc}(\mathsf{pair}(\mathsf{fun}(f, lt), \mathsf{var}(x)))}$$

$$\frac{\mathsf{unifyLPTAcc}(\mathsf{zip}(lt_1, lt_2))}{\mathsf{unifyPTAcc}(\mathsf{pair}(\mathsf{fun}(f, lt_1), \mathsf{fun}(g, lt_2)))} \qquad \frac{}{\mathsf{unifyLPTAcc}(\mathsf{nil})}$$

$$\frac{h \in \mathsf{unifyPTAcc}(p) \quad \mathsf{unifyLPTAcc}(\mathsf{appSb}(\mathsf{unifyPT}(p, h), lpt))}{\mathsf{unifyLPTAcc}(\mathsf{cons}(p, lpt))}$$

where x is a variable, f and g are functions, t is a term, lt, lt_1 and lt_2 are lists of terms, p is a pair of terms and lpt is a list of pairs of terms. Observe that in the second rule, the shape of the left term differs from the corresponding term in the Haskell equations. In Haskell, the equations are considered in the order in which they are presented. Thus, the second equation of the algorithm `unifyPT` will never be executed when the left term of the input pair is a variable. Hence, it is not necessary to specify that as a condition. However, this is not the case in type theory and then, we need to be explicit about which kind of term we are considering in each case. As the left term in the second rule cannot be a variable, it has to be a function applied to a list of terms.

Before introducing the type-theoretic formalisation of our unification algorithms, let us first translate same Haskell definitions and functions into their

[4]We ignore here efficiency aspects such as the fact that some expressions are computed twice.

type-theoretic equivalents. We start by presenting the definition of pairs and lists in type theory.

$$
\begin{array}{l}
\mathsf{Pair} \in (A, B \in \mathsf{Set})\mathsf{Set} \\
\quad \mathsf{pair} \in (\downarrow A, \downarrow B \in \mathsf{Set}; a \in A; b \in B)\mathsf{Pair}(A, B) \\
\mathsf{List} \in (A \in \mathsf{Set})\mathsf{Set} \\
\quad \mathsf{nil} \in (\downarrow A \in \mathsf{Set})\mathsf{List}(A) \\
\quad \mathsf{cons} \in (\downarrow A \in \mathsf{Set}; a \in A; l \in \mathsf{List}(A))\mathsf{List}(A)
\end{array}
$$

The down arrow in front of a set A within a definition, as in $\downarrow A$, indicates that we have hidden the set A in the definition. This is a layout facility provided by some proof assistants and it is usually exploited by the user when the hidden argument can be easily deduced from the context. In the rest of this section, we make full use of this facility and we hide some arguments. In this way, we hope to simplify the reading of the code.

$$
\begin{array}{l}
\mathsf{length} \in (\mathsf{List}(A))\mathsf{N} \\
\quad \mathsf{length}(\mathsf{nil}) \equiv \mathsf{0} \\
\quad \mathsf{length}(\mathsf{cons}(a, l)) \equiv \mathsf{s}(\mathsf{length}(l)) \\
\mathsf{zip} \in (l_1 \in \mathsf{List}(A); l_2 \in \mathsf{List}(B))\mathsf{List}(\mathsf{Pair}(A, B)) \\
\quad \mathsf{zip}(\mathsf{nil}, l_2) \equiv \mathsf{nil} \\
\quad \mathsf{zip}(\mathsf{cons}(a, l), \mathsf{nil}) \equiv \mathsf{nil} \\
\quad \mathsf{zip}(\mathsf{cons}(a, l), \mathsf{cons}(a', l')) \equiv \mathsf{cons}(\mathsf{pair}(a, a'), \mathsf{zip}(l, l')) \\
+\!+ \in (l_1, l_2 \in \mathsf{List}(A))\mathsf{List}(A) \\
\quad +\!+ (\mathsf{nil}, l_2) \equiv l_2 \\
\quad +\!+ (\mathsf{cons}(a, l), l_2) \equiv \mathsf{cons}(a, +\!+ (l, l_2))
\end{array}
$$

After presenting the general definitions, we introduce some definitions that are particular to our case example.

$$
\begin{array}{l}
\mathsf{Var} \in \mathsf{Set} \\
\quad \mathsf{Var} \equiv \mathsf{N} \\
\mathsf{Fun} \in \mathsf{Set} \\
\quad \mathsf{Fun} \equiv \mathsf{N} \\
\mathsf{Term} \in \mathsf{Set} \\
\quad \mathsf{var} \in (x \in \mathsf{Var})\mathsf{Term} \\
\quad \mathsf{fun} \in (f \in \mathsf{Fun}; lt \in \mathsf{List}(\mathsf{Term}))\mathsf{Term} \\
\mathsf{ListPT} \in \mathsf{Set} \\
\quad \mathsf{ListPT} \equiv \mathsf{List}(\mathsf{Pair}(\mathsf{Term}, \mathsf{Term})) \\
\mathsf{Subst} \in \mathsf{Set} \\
\quad \mathsf{Subst} \equiv \mathsf{List}(\mathsf{Pair}(\mathsf{Var}, \mathsf{Term})) \\
\mathsf{appSb} \in (\mathsf{Subst}; \mathsf{ListPT})\mathsf{ListPT} \\
\quad \mathsf{appSb}(sb, lpt) \equiv ....
\end{array}
$$

We can now present the type-theoretic version of our simplified unification algorithm over pair of terms.

$\mathsf{unifyPTAcc} \in (p \in \mathsf{Pair}(\mathsf{Term}, \mathsf{Term}))\mathsf{Set}$
  $\mathsf{uptacc1} \in (x \in \mathsf{Var}; t \in \mathsf{Term})\mathsf{unifyPTAcc}(\mathsf{pair}(\mathsf{var}(x), t))$
  $\mathsf{uptacc2} \in (x \in \mathsf{Var}; f \in \mathsf{Fun}; lt \in \mathsf{List}(\mathsf{Term})$
    $)\mathsf{unifyPTAcc}(\mathsf{pair}(\mathsf{fun}(f, lt), \mathsf{var}(x)))$
  $\mathsf{uptacc3} \in (f, g \in \mathsf{Fun}; lt_1, lt_2 \in \mathsf{List}(\mathsf{Term}); \mathsf{unifyLPTAcc}(\mathsf{zip}(lt_1, lt_2))$
    $)\mathsf{unifyPTAcc}(\mathsf{pair}(\mathsf{fun}(f, lt_1), \mathsf{fun}(g, lt_2)))$

$\mathsf{unifyLPTAcc} \in (lpt \in \mathsf{ListPT})\mathsf{Set}$
  $\mathsf{ulptacc1} \in \mathsf{unifyLPTAcc}(\mathsf{nil})$
  $\mathsf{ulptacc2} \in (p \in \mathsf{Pair}(\mathsf{Term}, \mathsf{Term}); lp \in \mathsf{ListPT}; h_1 \in \mathsf{unifyPTAcc}(p);$
    $h_2 \in \mathsf{unifyLPTAcc}(\mathsf{appSb}(\mathsf{unifyPT}(p, h_1), lp))$
    $)\mathsf{unifyLPTAcc}(\mathsf{cons}(p, lp))$

$\mathsf{unifyPT} \in (p \in \mathsf{Pair}(\mathsf{Term}, \mathsf{Term}); \mathsf{unifyPTAcc}(p))\mathsf{Subst}$
  $\mathsf{unifyPT}(\mathsf{pair}(\mathsf{var}(x), t), \mathsf{uptacc1}(x, t)) \equiv \mathsf{cons}(\mathsf{pair}(x, t), \mathsf{nil})$
  $\mathsf{unifyPT}(\mathsf{pair}(\mathsf{fun}(f, lt), \mathsf{var}(x)), \mathsf{uptacc2}(x, f, lt)) \equiv$
    $\mathsf{cons}(\mathsf{pair}(x, \mathsf{fun}(f, lt)), \mathsf{nil})$
  $\mathsf{unifyPT}(\mathsf{pair}(\mathsf{fun}(f, lt_1), \mathsf{fun}(g, lt_2)), \mathsf{uptacc3}(f, g, lt_1, lt_2, h)) \equiv$
    $\mathsf{unifyLPT}(\mathsf{zip}(lt_1, lt_2), h)$

$\mathsf{unifyLPT} \in (lpt \in \mathsf{ListPT}; \mathsf{unifyLPTAcc}(lpt))\mathsf{Subst}$
  $\mathsf{unifyLPT}(\mathsf{nil}, \mathsf{ulptacc1}) \equiv \mathsf{nil}$
  $\mathsf{unifyLPT}(\mathsf{cons}(p, lp), \mathsf{ulptacc2}(p, lp, h_1, h_2)) \equiv$
    $\mathsf{unifyPT}(p, h_1) +\!+ \mathsf{unifyLPT}(\mathsf{appSb}(\mathsf{unifyPT}(p, h_1), lp), h_2)$

3.2 List Reversal

Our second example is an algorithm to reverse the order of the elements in a list and it has been taken from [Gie97]. Although this is a very well known and common task, the approach we introduce here is not the standard one. Furthermore, it is a very awkward and inefficient approach. However, it is an interesting example if we just take the recursive calls into account.

```
rev :: [a] -> [a]
rev [] = []
rev (x:xs) = last x xs : rev2 x xs

rev2 :: a -> [a] -> [a]
rev2 y [] = []
rev2 y (x:xs) = rev (y : rev (rev2 x xs))

last :: a -> [a] -> a
last y [] = y
last y (x:xs) = last x xs
```

In this example, the algorithm `rev` reverses a list with the help of the algorithms `last` and `rev2`. The algorithm `last` is a structurally smaller recursive algorithm and its formalisation in type theory is straightforward. The algorithms `rev` and `rev2` are nested and mutually recursive. In the rest of this section, we just pay attention to the two general recursive algorithms `rev` and `rev2` and we assume that we already have a type-theoretic translation of the algorithm `last`.

As usual, we first present the introduction rules for the special-purpose inductive predicates revAcc and rev2Acc. Notice that, since the algorithms rev and rev2 are nested, the two predicates need to know about the two algorithms.

$$\frac{}{\mathsf{revAcc}([])} \qquad \frac{\mathsf{rev2Acc}(x, xs)}{\mathsf{revAcc}(x : xs)}$$

$$\frac{}{\mathsf{rev2Acc}(y, [])} \qquad \frac{\begin{array}{c}\mathsf{rev2Acc}(x, xs)\\ \mathsf{revAcc}(\mathsf{rev2}(x, xs))\\ \mathsf{revAcc}(y : \mathsf{rev}(\mathsf{rev2}(x, xs)))\end{array}}{\mathsf{rev2Acc}(y, (x : xs))}$$

Finally, in type theory we formalise the inductive predicates revAcc and rev2Acc simultaneously with the algorithms rev and rev2, recursively defined on the predicates. We again make use of the layout facility that allows us to hide arguments. In addition, we use the type-theoretic formalisation of lists presented in the previous section.

$$\begin{array}{l}
\mathsf{revAcc} \in (zs \in \mathsf{List(A)})\mathsf{Set}\\
\quad \mathsf{revacc1} \in \mathsf{revAcc}(\mathsf{nil})\\
\quad \mathsf{revacc2} \in (x \in A; xs \in \mathsf{List}(A); h \in \mathsf{rev2Acc}(x, xs))\mathsf{revAcc}(\mathsf{cons}(x, xs))\\
\\
\mathsf{rev2Acc} \in (y \in A; zs \in \mathsf{List(A)})\mathsf{Set}\\
\quad \mathsf{rev2acc1} \in (y \in A)\mathsf{rev2Acc}(y, \mathsf{nil})\\
\quad \mathsf{rev2acc2} \in (y, x \in A; xs \in \mathsf{List}(A); h_1 \in \mathsf{rev2Acc}(x, xs);\\
\qquad\qquad h_2 \in \mathsf{revAcc}(\mathsf{rev2}(x, xs, h_1));\\
\qquad\qquad h_3 \in \mathsf{revAcc}(\mathsf{cons}(y, \mathsf{rev}(\mathsf{rev2}(x, xs, h_1), h_2)))\\
\qquad\qquad)\mathsf{rev2Acc}(y, \mathsf{cons}(x, xs))\\
\\
\mathsf{rev} \in (zs \in \mathsf{List}(A); \mathsf{revAcc}(zs))\mathsf{List}(A)\\
\quad \mathsf{rev}(\mathsf{nil}, \mathsf{revacc1}) \equiv \mathsf{nil}\\
\quad \mathsf{rev}(\mathsf{cons}(x, xs), \mathsf{revacc2}(x, xs, h)) \equiv \mathsf{cons}(\mathsf{last}(x, xs), \mathsf{rev2}(x, xs, h))\\
\\
\mathsf{rev2} \in (y \in A; zs \in \mathsf{List}(A); \mathsf{rev2Acc}(y, zs))\mathsf{List}(A)\\
\quad \mathsf{rev2}(y, \mathsf{nil}, \mathsf{rev2acc1}(y)) \equiv \mathsf{nil}\\
\quad \mathsf{rev2}(y, \mathsf{cons}(x, xs), \mathsf{rev2acc2}(y, x, xs, h_1, h_2, h_3)) \equiv\\
\qquad\qquad\qquad\qquad \mathsf{rev}(\mathsf{cons}(y, \mathsf{rev}(\mathsf{rev2}(x, xs, h_1), h_2)), h_3)
\end{array}$$

4 Conclusions

We show here how the methodology presented in [Bov01] for the formalisation of simple general recursive algorithms and extended in [BC01] to treat nested general recursion can also be used in the formalisation of mutual general recursive algorithms. This methodology consists of defining special-purpose accessibility predicates that characterise the inputs on which the algorithms terminate. Each algorithm is then formalised in type theory by structural recursion on the proof that its input satisfies the corresponding special-purpose accessibility predicate. As the algorithms we consider in this work are mutually recursive, the termination of one algorithm depends on the termination of the others and hence, the special-purpose accessibility predicates are also mutually recursive.

When the mutually recursive algorithms are also nested, we need to define the special-purpose accessibility predicates and the type-theoretic version of the algorithms simultaneously, because they depend on each other. This kind of definitions is not allowed in ordinary type theory, but it can be carried out in type theories extended with the general schema for simultaneous inductive-recursive definitions that we present in appendix A. This schema is a generalisation of Dybjer's schema for simultaneous inductive-recursive definitions introduced in [Dyb00]. While Dybjer considers the case with only one predicate and one function in his work, we present here a generalisation of the schema for the cases where we have several mutually recursive predicates defined simultaneously with several functions which, in turn, are defined by recursion on those predicates.

Similarly to what happens for simple and nested recursive algorithms, this methodology results in definitions in which the computational and logical parts are clearly separated also when the algorithms are mutually recursive. Hence, the type-theoretic version of the algorithms is given by its purely functional content, similarly to the corresponding program in a functional programming language. As a consequence, the resulting type-theoretic algorithms are compact and easy to understand. We firmly believe that our methodology also helps in the process of formal verification since the simplicity of the definitions of the type-theoretic algorithms usually simplifies the task of their formal verification.

The examples we presented in this work have been formally checked using the proof assistant ALF (see [AGNvS94, MN94]), which supports the schema in appendix A.

A Generalisation of Dybjer's Schema for Simultaneous Inductive-Recursive Definitions

A.1 Preliminary Comments

In [Dyb00], Dybjer defines an schema for simultaneous inductive-recursive definitions in type theory. In the schema, Dybjer considers the case with only one predicate and one function. We generalise here Dybjer's schema for the cases where we have several mutually recursive predicates defined simultane-

ously with several functions, which in turn are defined by recursion on those predicates. The presentation we introduce here is by no means the most general one. However, it gives us the necessary theoretical strength in order to formalise nested and mutually recursive algorithms with the methodology we described in previous sections of this work.

Here, we assume that a definition is always relative to a theory containing the rules for previously defined concepts. Thus, the requirements on the different parts of the definitions are always judgements with respect to that theory.

In order to make the reading easier, we use Dybjer's notation as much as possible. Then, $(a :: \alpha)$ is an abbreviation of $(a_1 : \alpha_1) \cdots (a_o : \alpha_o)$ and a *small* type is a type that does not contain occurrences of Set. In addition, to help with the understanding of our generalisation, we follow closely the formalisation of the mutually recursive algorithms f and g introduced in section 2 through the different sections of this appendix.

A.2 Formation Rules

We describe here the formation rules for the simultaneous definition of m inductive predicates and n functions defined by recursion over those predicates.

In order to present the formation rules for predicates and functions, let

- $1 \leqslant k \leqslant m$, $1 \leqslant w \leqslant m$ and $m+1 \leqslant l \leqslant m+n$;
- σ be a sequence of types;
- $\alpha_k[A]$ and $\alpha_w[A]$ be sequences of small types under the assumption $(\mathrm{A} :: \sigma)$;
- $\psi_l[A, a]$ be a type under the assumptions $(A :: \sigma; a :: \alpha_w[A])$.

Thus, if f_l is defined by recursion over a certain predicate P_w, the formation rules for predicates and functions are of the form:

$$P_k : (A :: \sigma)(a :: \alpha_k[A])\mathsf{Set}$$
$$f_l : (A :: \sigma)(a :: \alpha_w[A])(c : P_w(A, a))\psi_l[A, a]$$

Note that each function f_l actually determines which is the predicate P_w needed as part of the domain of its formation rule. If we want to be totally formal here, we should indicate this by indexing the w's with l's as in P_{w_l}. However, for the sake of simplicity we will not do so. The reader should keep this dependence in mind when reading the rest of this appendix.

Observe also that, in the formation rules stated above, we have assumed that all predicates and functions have a common set of parameters $(A :: \sigma)$. In case each predicate and function has its own set of parameters $(A_h :: \sigma_h)$, we take $(A :: \sigma)$ as the union of the $(A_h :: \sigma_h)$, for $1 \leqslant h \leqslant m+n$.

If we carefully analyse the assumptions stated above, we see that none of our inductive predicates or recursive functions is known when we construct the sequences of small types α's and the types ψ's. Hence, no one of our predicates or functions can be mentioned in those sequences or types, since they are not yet

defined. As a consequence, no one of our predicates can have any of the other predicates or functions as part of its formation rule. On the other hand, each function is defined by recursion on one of our inductive predicates and thus, this predicate must be part of the domain of the function. However, no other of our predicates or functions can be part of the formation rule of the function.

In our example, the formation rules of the predicates fAcc and gAcc (P_1 and P_2 respectively) and of the functions f and g (f_3 and f_4 respectively) are as follows:

$$\mathsf{fAcc} \in (m \in \mathsf{N})\mathsf{Set}$$
$$\mathsf{gAcc} \in (m \in \mathsf{N})\mathsf{Set}$$

$$\mathsf{f} \in (m \in \mathsf{N}; \mathsf{fAcc}(m))\mathsf{N}$$
$$\mathsf{g} \in (m \in \mathsf{N}; \mathsf{gAcc}(m))\mathsf{N}$$

Here, the sequence σ is the empty sequence, the sequences of small types α's consist of the sequence $(m \in \mathsf{N})$ and the types ψ's are the set of natural numbers N.

A.3 Introduction Rules

Before presenting the schema for the introduction rules of the predicates, we recall the notions of the different premises presented in [Dyb00]. Then, a premise of an introduction rule is either *non-recursive* or *recursive*.

A non-recursive premise has the form $(b : \beta[A])$, where $\beta[A]$ is a small type depending on the assumption $(A :: \sigma)$ and previous premises of the rule.

A recursive premise has the form $u : (x :: \xi[A])P_h(A, p[A, x])$, where $\xi[A]$ is a sequence of small types under the assumption $(A :: \sigma)$ and previous premises of the rule, $p[A, x] :: \alpha_h[A]$ under the assumptions $(A :: \sigma; x :: \xi[A])$ and previous premises of the rule and $1 \leqslant h \leqslant m$. If $\xi[A]$ is empty, the premise is called *ordinary* and otherwise it is called *generalised*.

Now, the schema for the jth introduction rule of the kth predicate is the following:

$$\mathsf{intro}_{kj} : (A :: \sigma) \ldots (b : \beta[A]) \ldots (u : (x :: \xi[A])P_i(A, p[A, x])) \ldots P_k(A, q_{kj}[A])$$

where

- $1 \leqslant k \leqslant m$, $1 \leqslant j$ and $1 \leqslant i \leqslant m$;
- The b's and the u's can occur in any order. The b's and/or the u's can also be omitted;
- Each recursive premise might refer to several predicates P_i. Observe that each P_i can occur in several recursive premises of the introduction rule;
- $q_{kj}[A] :: \alpha_k[A]$ under the assumption $(A :: \sigma)$ and previous premises of the rule.

Note that each pair kj actually determines the β's, ξ's, P_i's and p's that occur in the introduction rule intro_{kj}. If we want to be more formal about this dependence as well as about the fact that there might be several b's and several u's, we should give the following more precise schema for the jth introduction rule of the kth predicate:

$$\mathsf{intro}_{kj} : (A :: \sigma) \ldots (b_d : \beta_{kjd}[A]) \ldots (u_r : (x :: \xi_{kjr}[A])P_{i_{kjr}}(A, p_{kjr}[A, x])) \ldots \\ P_k(A, q_{kj}[A])$$

where d indicates the dth non-recursive premise and r indicates the rth recursive premise of the introduction rule, with $0 \leqslant d$ and $0 \leqslant r$. However, for the sake of simplicity we will not do so and hence, in the rest of this appendix we will not write extra indices. The reader should keep this in mind when reading the rest of the section.

In our example, the introduction rules for the predicates fAcc and gAcc are as follows:

$$\mathsf{facc}_0 \in \mathsf{fAcc}(0)$$
$$\mathsf{facc}_\mathsf{s} \in (n \in \mathsf{N}; h_1 \in \mathsf{gAcc}(n); h_2 \in \mathsf{fAcc}(\mathsf{g}(n, h_1)))\mathsf{fAcc}(\mathsf{s}(n))$$

$$\mathsf{gacc}_0 \in \mathsf{gAcc}(0)$$
$$\mathsf{gacc}_\mathsf{s} \in (n \in \mathsf{N}; h_1 \in \mathsf{fAcc}(n); h_2 \in \mathsf{gAcc}(\mathsf{f}(n, h_1)))\mathsf{gAcc}(\mathsf{s}(n))$$

Here, facc_0 is an introduction rule with no premises. The premises of the introduction rule facc_s are as follows: $(n \in \mathsf{N})$ is a non-recursive premise, $(h_1 \in \mathsf{gAcc}(n))$ is an ordinary recursive premise (that is, the corresponding ξ is empty) which depends on the previous non-recursive premise and finally, $(h_2 \in \mathsf{fAcc}(\mathsf{g}(n, h_1)))$ is also an ordinary recursive premise which depends on the previous two non-recursive and recursive premises, respectively. The introduction rules of the predicate gAcc are similar to those of the predicate fAcc.

A.4 Possible Dependencies

We now spell out the typing criteria for $\beta[A]$ in the schema above. The criteria for $\xi[A]$, $p[A, x]$ and $q_{kj}[A]$ are analogous.

We write $\beta[A] = \beta[A, \ldots, b', \ldots, u', \ldots]$ to explicitly indicate the dependence on previous non-recursive premises $b' : \beta'[A]$ and recursive premises of the form $u' : (x :: \xi'[A])P_g(A, p'[A, x])$, for $1 \leqslant g \leqslant m$. The dependence on a previous recursive premise can only occur through an application of one of the simultaneously defined functions f_t, for $m + 1 \leqslant t \leqslant m + n$. Formally, we have:

$$\beta[A, \ldots, b', \ldots, u', \ldots] = \hat{\beta}[A, \ldots, b', \ldots, (x)f_t(A, p'[A, x], u'(x)), \ldots]$$

where $\hat{\beta}[A, \ldots, b', \ldots, v', \ldots]$ is a small type in the context

$$(A :: \sigma; \ldots; b' : \beta'[A]; \ldots; v' : (x :: \xi'[A])\psi_t[A, p'[A, x]]; \ldots)^5.$$

[5] Note that this context is obtained from the context of β by replacing each recursive premise of the form $u' : (x :: \xi'[A])P_g(A, p'[A, x])$ by $v : (x :: \xi'[A])\psi_t[A, p'[A, x]]$.

In our example, the recursive premise ($h_2 \in \mathsf{fAcc}(\mathsf{g}(n, h_1))$) of the predicate fAcc depends on the recursive premise ($h_1 \in \mathsf{gAcc}(n)$). This dependence occurs through the application of the simultaneously defined function g. Similarly, if we study the dependence on previous recursive premises in the introduction rules of the predicate gAcc, we observe that they occur through the application of the function f.

That the dependence on previous recursive premises can only occur through applications of the simultaneous defined functions ensures the correctness of the inductive-recursive definitions. In this way, whenever we apply a predicate to the result of one of the simultaneously defined functions, we make sure that such argument has been previously constructed. In addition, observe that as the simultaneous definition of the predicates and the functions is not yet complete, the application of any previously defined predicate or function to one of our recursive premises would be incorrect. We come back to this matter after we have presented the equality rules for our example, that is, at the end of next section.

A.5 Equality Rules

If f_y is defined by recursion on P_k, the schema for the equality rule for f_y and intro_{kj} is as follows, for $m+1 \leqslant y \leqslant m+n$ and $m+1 \leqslant z \leqslant m+n$:

$$f_y(A, q_{kj}[A], \mathsf{intro}_{kj}(A, \ldots, b, \ldots, u, \ldots)) = \\ e_{yj}(A, \ldots, b, \ldots, (x)f_z(A, p[A, x], u(x)), \ldots) : \psi_y[A, q_{kj}[A]]$$

in the context

$$(A :: \sigma; \ldots; b : \beta[A]; \ldots; u : (x :: \xi[A])P_i(A, p[A, x]); \ldots)$$

where $e_{yj}(A, \ldots, b, \ldots, v, \ldots) : \psi_y[A, q_{kj}[A]]$ in the context

$$(A :: \sigma; \ldots; b : \beta[A]; \ldots; v : (x :: \xi[A])\psi_z[A, p[A, x]]; \ldots).$$

In our example, the equality rules for the functions f and g are as follows:

$$\mathsf{f}(0, \mathsf{facc}_0) = 0$$
$$\mathsf{f}(\mathsf{s}(n), \mathsf{facc}_\mathsf{s}(n, h_1, h_2)) = \mathsf{f}(\mathsf{g}(n, h_1), h_2)$$

$$\mathsf{g}(0, \mathsf{gacc}_0) = 0$$
$$\mathsf{g}(\mathsf{s}(n), \mathsf{gacc}_\mathsf{s}(n, h_1, h_2)) = \mathsf{g}(\mathsf{f}(n, h_1), h_2)$$

Here, if we analyse the equality rules for f, we have that the function e_{31} is the constant function 0 and the function e_{32} is the algorithm f itself. The occurrence of g in the right hand side of the second equality rule for f corresponds to the occurrence of f_z as one of the arguments of the function e_{yj} in the schema above. Similarly, we can analyse the equality rules for the function g.

We now go back to the dependence matter. We show the correctness of our simultaneous definition by analysing the way the proofs of fAcc and gAcc are

constructed and the way the results of f and g are defined. First, we construct the proofs facc_0 and gacc_0 of $\mathsf{fAcc}(0)$ and $\mathsf{gAcc}(0)$, respectively. Now, we define both the result of $\mathsf{f}(0, \mathsf{facc}_0)$ and the result of $\mathsf{g}(0, \mathsf{gacc}_0)$ as the constant 0. Then, we construct the proofs $\mathsf{facc}_\mathsf{s}(0, \mathsf{gacc}_0, \mathsf{facc}_0)$ and $\mathsf{gacc}_\mathsf{s}(0, \mathsf{facc}_0, \mathsf{gacc}_0)$ of $\mathsf{fAcc}(\mathsf{s}(0))$ and $\mathsf{gAcc}(\mathsf{s}(0))$, respectively. Now, we define both the result of $\mathsf{f}(\mathsf{s}(0), \mathsf{facc}_\mathsf{s}(0, \mathsf{gacc}_0, \mathsf{facc}_0))$ and the result of $\mathsf{g}(\mathsf{s}(0), \mathsf{gacc}_\mathsf{s}(0, \mathsf{facc}_0, \mathsf{gacc}_0))$ as the constant 0. Recall that $\mathsf{facc}_0 \in \mathsf{fAcc}(0)$, $\mathsf{gacc}_0 \in \mathsf{gAcc}(0)$, $\mathsf{f}(0, \mathsf{facc}_0) = 0$ and $\mathsf{g}(0, \mathsf{gacc}_0) = 0$. We can now continue by constructing the proofs of $\mathsf{fAcc}(\mathsf{s}(\mathsf{s}(0)))$ and of $\mathsf{gAcc}(\mathsf{s}(\mathsf{s}(0)))$ and subsequently, use those proofs in order to define the result of the functions f and g on the natural number $\mathsf{s}(\mathsf{s}(0))$. In general, we first construct the proof h_1 of $\mathsf{fAcc}(n)$ and the proof h_2 of $\mathsf{gAcc}(n)$, and we then use those proofs to define the result of $\mathsf{f}(n, h_1)$ and of $\mathsf{g}(n, h_1)$. Thereafter, we construct the proofs of $\mathsf{fAcc}(\mathsf{s}(n))$ and of $\mathsf{gAcc}(\mathsf{s}(n))$, which in turn will be used to define the result of the functions f and g on the natural number $\mathsf{s}(n)$, and so on.

A.6 Recursive Definitions

In general, after the simultaneous definition of the m predicates and the n functions has been done, we may define new functions of the form:

$$f'_y : (A :: \sigma)(A' :: \sigma')(a :: \alpha_k[A])(c : P_k(A, a))\psi'_y[A, A', a, c]$$

by recursion on P_k, where

- $0 \leqslant y$;
- σ' is a sequence of types;
- $\psi'_y[A, A', a, c]$ is a type under $(A :: \sigma; A' :: \sigma'; a :: \alpha_k[A]; c : P_k(A, a))$.

Observe that f'_y might have a different set of parameters than those needed for the definitions of the m inductive predicates and the n recursive functions[6]. Note also that both the inductive predicates and the recursive functions are known when we define the function f'_y and hence, they can be mentioned as part of the type ψ'_y (compare ψ' here with the type ψ introduced in section A.2; there ψ must be already known when stating the types of the predicates and functions we are about to define).

Now, the equality rules for the new functions are as follows:

$$f'_y(A, A', q_{kj}[A], \mathsf{intro}_{kj}(A, \ldots, b, \ldots, u, \ldots)) = \\ e'_{yj}(A, A', \ldots, b, \ldots, u, (x)f'_z(A, A', p[A, x], u(x)), \ldots)$$

in the context

$$(A :: \sigma; A' :: \sigma'; \ldots; b : \beta[A]; \ldots; u : (x :: \xi[A])P_i(A, p[A, x]); \ldots)$$

[6] Let us assume here that all the recursive functions we define afterwards have the same set of parameters σ'. If this is not the case, we let σ' be the union of the sets of parameters needed in order to define the new functions (see section A.2 for a similar and more detail explanation of how to construct σ as the union of the different sets of parameters).

where

$$e'_{yj}(A, A', \ldots, b, \ldots, u, v, \ldots) : \psi'_y[A, A', q_{kj}[A], \mathsf{intro}_{kj}(A, \ldots, b, \ldots, u, \ldots)]$$

in the context

$$(A :: \sigma; A' :: \sigma'; \ldots; b : \beta[A]; \ldots; u : (x :: \xi[A])P_i(A, p[A, x]);$$
$$v : (x :: \xi[A])\psi'_z[A, A', p[A, x], u(x)]; \ldots).$$

Note that the criteria are identical for a simultaneously defined function f_l and a function f'_y defined afterwards, except that the type ψ'_y may depend on c as well as on a. In addition, the right hand side of a recursion equation e'_{yj} for f'_y may depend on u as well as on v. This is simply because these new dependencies can occur only after the inductive predicates have been defined.

In our example, after the definition of the predicates fAcc and gAcc and of the functions f and g has been completed, we can define the auxiliary functions f' and g' (f'_1 and f'_2 respectively). These functions count the number of steps the functions f and g need in order to compute their result when applied to a certain natural number n. In addition, they are defined by recursion on the proof that the input natural number satisfies the corresponding special accessibility predicate and have the following definitions:

$$\begin{array}{l}
\mathsf{f}' \in (m \in \mathsf{N}; \mathsf{fAcc}(m))\mathsf{N} \\
\quad \mathsf{f}'(0, \mathsf{facc}_0) = 0 \\
\quad \mathsf{f}'(\mathsf{s}(n), \mathsf{facc}_\mathsf{s}(n, h_1, h_2)) = \mathsf{s}(\mathsf{g}'(n, h_1))
\end{array}$$

$$\begin{array}{l}
\mathsf{g}' \in (m \in \mathsf{N}; \mathsf{gAcc}(m))\mathsf{N} \\
\quad \mathsf{g}'(0, \mathsf{gacc}_0) = 0 \\
\quad \mathsf{g}'(\mathsf{s}(n), \mathsf{gacc}_\mathsf{s}(n, h_1, h_2)) = \mathsf{s}(\mathsf{f}'(n, h_1))
\end{array}$$

Here, both e'_{11} and e'_{21} are the constant function 0 and both e'_{12} and e'_{22} are the successor function over natural numbers.

Acknowledgements. We want to thank Björn von Sydow for carefully reading and commenting on previous versions of this paper. We are grateful to Peter Dybjer for useful discussions on the generalisation of his schema and for his comments on the formalisation of the general schema that we present in appendix A.

References

[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`.

[BC01] A. Bove and V. Capretta. Nested general recursion and partiality in type theory. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science, Springer-Verlag*, pages 121–135, September 2001.

[Bov99] A. Bove. Programming in Martin-Löf type theory: Unification - A non-trivial example, November 1999. Licentiate Thesis of the Department of Computer Science, Chalmers University of Technology. Available on the WWW `http://cs.chalmers.se/~bove/Papers/lic_thesis.ps.gz`.

[Bov01] A. Bove. Simple general recursion in type theory. *Nordic Journal of Computing*, 8(1):22–42, Spring 2001.

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

[CNSvS94] T. Coquand, B. Nordström, J. M. Smith, and B. von Sydow. Type theory and programming. *EATCS*, 52, February 1994.

[Dyb00] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.

[Gie97] J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19:1–29, 1997.

[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.

[JHe+99] S. Peyton Jones, J. Hughes, (editors), L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[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.

[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.

Paper IV

Modelling General Recursion in Type Theory

Modelling General Recursion in Type Theory

Ana Bove* Venanzio Capretta†

September 2002

Abstract

Constructive type theory is a very expressive programming language. However, general recursive algorithms have no direct formalisation in type theory since they contain recursive calls that do not satisfy any syntactic condition that guarantees termination. We present a method to formalise general recursive algorithms in type theory that uses an inductive predicate to characterise termination and that separates the computational and logical parts of the definitions. As a consequence, the resulting type-theoretic algorithms are clear, compact and easy to understand. They are as simple as their equivalents in a functional programming language, where there is no restriction on the recursive calls. Given a general recursive algorithm, our method consists in defining an inductive special-purpose accessibility predicate that characterises the inputs on which the algorithm terminates. The type-theoretic version of the algorithm is then defined by structural recursion on the proof that the input values satisfy this predicate. We give a formal definition of the method and discuss its power and its limitations.

1 Introduction

Constructive type theory (see for example [ML84, CH88]) is a very expressive programming language with dependent types. According to the Curry-Howard isomorphism [How80, SU98], logic can also be represented in it by identifying propositions with types and proofs with terms of the corresponding type. Therefore, we can encode in a type a complete specification, requiring also logical properties from an algorithm. As a consequence, algorithms are correct by construction or can be proved correct by using the expressive power of constructive type theory. This is clearly an advantage of constructive type theory over standard programming languages. A computational limitation of type theory is that, to keep the logic consistent and type-checking decidable, only structural

*Department of Computing Science, Chalmers University of Technology, 412 96 Göteborg, Sweden, e-mail: bove@cs.chalmers.se, telephone: +46-31-7721020, fax: +46-31-165655

†INRIA Sophia Antipolis, Project LEMME, e-mail: Venanzio.Capretta@sophia.inria.fr, telephone: +33+4+92385051, fax: +33+4+92385060

recursive definitions are allowed, that is, definitions in which the recursive calls must have structurally smaller arguments.

On the other hand, functional programming languages like Haskell [JHe+99], Standard ML [MTHM97] or Clean [dMJB+01] are less expressive in the sense that they do not have dependent types and they cannot represent logic. Moreover, the existing frameworks for reasoning about the correctness of Haskell-like programs are weaker than the framework provided by type theory, and it is the responsibility of the programmer to write correct programs. However, functional programming languages are computationally stronger because they impose no restriction on recursive programs and thus, they allow the definition of general recursive algorithms. In addition, functional programs are usually short and self-explanatory.

General recursive algorithms are defined by cases where the recursive calls are not required to have structurally smaller arguments. In other words, the recursive calls are performed on objects that satisfy no syntactic condition that guarantees termination. As we have already mentioned, there is no direct way of formalising this kind of algorithms in type theory.

The standard way of handling general recursion in constructive type theory uses a well-founded recursion principle derived from the accessibility predicate Acc (see [Acz77, Nor88, BB00]). However, the use of this predicate in the type-theoretic formalisation of general recursive algorithms often results in unnecessarily long and complicated codes. Moreover, its use adds a considerable amount of code with no computational content, that distracts our attention from the computational part of the algorithm (see for example [Bov99], where we present the formalisation of a unification algorithm over lists of pairs of terms using the standard accessibility predicate Acc).

To bridge the gap between programming in type theory and programming in a functional language, we developed a method to formalise general recursive algorithms in type theory that separates the computational and logical parts of the definitions. As a consequence, the resulting type-theoretic algorithms are clear, compact and easy to understand. They are as simple as their Haskell-like versions, where there is no restriction on the recursive calls. Given a general recursive algorithm, our method consists in defining an inductive special-purpose accessibility predicate that characterises the inputs on which the algorithm terminates. The type-theoretic version of the algorithm can then be defined by structural recursion on the proof that the input values satisfy this predicate. If the algorithm has nested recursive calls, the accessibility predicate and the type-theoretic algorithm must be defined simultaneously, because they depend on each other. This kind of definitions is not allowed in ordinary type theory, but it is provided in type theories extended with Dybjer's schema for simultaneous inductive-recursive definitions [Dyb00].

This method was introduced by Bove [Bov01] to formalise simple general recursive algorithms in constructive type theory (by simple we mean non-nested and non-mutually recursive). It was extended by Bove and Capretta [BC01] to treat nested recursion and by Bove [Bov02] to treat mutually recursive algorithms, nested or not. Since our method separates the computational part from

the logical part of a definition, formalising partial functions becomes possible [BC01]. Proving that a certain function is total amounts to proving that the corresponding special-purpose accessibility predicate is satisfied by every input.

So far, we have just presented our method by means of examples in [Bov01, BC01, Bov02]. The purpose of this work is to give a general presentation of the method. We start by giving a characterisation of the class of recursive definitions that we consider, which is a subclass of commonly used functional programming languages like Haskell, ML or Clean. This class consists of functions defined by recursive equations that are not necessarily well-founded. Then, we show how we can translate any function in that class into type theory using our special-purpose accessibility predicates.

When talking about functional programming, we use the terms *algorithm*, *function* and *program* as synonymous.

The rest of the paper is organised as follows. In section 2, we present a brief introduction to constructive type theory. In section 3, we illustrate our method by formalising a few examples of general recursive algorithms in type theory. In section 4, we define the class $\mathcal{FP}$ of recursive definitions that can be translated into type theory by applying our method. In section 5, we prove that this class is large enough to allow the definition of any recursive function. In section 6, we formally describe our method to translate general recursive functions into type theory. In section 7, we discuss the semantics of functional programs and we show that our method is sound with respect to a strict semantics for $\mathcal{FP}$. Finally, in section 8, we present some conclusions and related work.

2 Constructive Type Theory

Although this paper is intended mainly for those who already have some knowledge of type theory, we recall the basic ideas and notions that we use. For a complete presentation of constructive type theory, see [ML84, NPS90, CNSvS94]. For impredicative type theory, that we do not use but only mention in section 3, see [CH88]. A general formulation of type systems and their use in formal verification can be found in [Bar92] and [BG01].

Constructive type theory comprises a basic type called Set and two type formers, that is, two ways of constructing new types.

The first type former constructs the type of the elements of a set. Every element of Set is an inductively defined type. It is usual to call the elements of Set *small* types, and the types that are not elements of Set, like Set itself, *large* types. According to the Curry-Howard isomorphism [How80, SU98], propositions are also objects in Set and their elements are proofs of the corresponding proposition.

A set former or, in general, any inductive definition is introduced as a constant A of type $(x_1 \in \alpha_1; \ldots; x_n \in \alpha_n)\mathsf{Set}$, for $\alpha_1, \ldots, \alpha_n$ sets. For each set former, we must specify the constructors that generate the elements of $A(a_1, \ldots, a_n)$ by giving their types, for $a_1 \in \alpha_1, \ldots, a_n \in \alpha_n$.

The second type former allows the construction of dependent function types.

Given a type α and a family of types indexed on α, that is a type β depending on a variable $x \in \alpha$, we can form the dependent function type $(x \in \alpha)\beta$. The elements of function types are λ-abstractions. If b is an element of β depending on a variable $x \in \alpha$, then $[x \in \alpha]b$ is an element of $(x \in \alpha)\beta$. If the type of the abstracted variable is clear from the context, we write $[x]b$. If f is an element of $(x \in \alpha)\beta$ and a an element of α, the application of f to a, $f(a)$, is an element of $\beta[x := a]$ (β where every free occurrence of x is substituted by a). In the case where β does not depend on x, we can omit the reference to the variable and simply write $(\alpha)\beta$ for $(x \in \alpha)\beta$. We usually write sequential function types as $(x_1 \in \alpha_1; \ldots; x_n \in \alpha_n)\beta$ and sequential λ-abstractions as $[x_1, \ldots, x_n]b$. We write $(x_1, x_2, \ldots, x_n \in \alpha)$ instead of $(x_1 \in \alpha; x_2 \in \alpha; \ldots; x_n \in \alpha)$.

A particularly important Set is the set for propositional equality, also called intensional equality. Given a set α and two elements a and b in α, $\mathsf{Id}(\alpha, a, b)$ is the set that expresses that a and b are equal elements of type α. As the type α can usually be inferred from the context, we just write $a = b$ to refer to the set that expresses the propositional equality of a and b. The only way to introduce elements in this set is through the constructor refl. If $a \in \alpha$, then $\mathsf{refl}(a)$ is a proof that a is equal to itself. Hence, $\mathsf{refl}(a) \in a = a$.

Beside propositional equality, we also use dependent product and sum sets, and disjoint unions.

The elements of dependent products sets are dependent functions. Formally, if α is a set and β a family of sets over α, that is, $\beta(x)$ is a set provided $x \in \alpha$, we can form the dependent product set $\Pi(\alpha, \beta)$. If f is a dependent function from α to β then $\lambda(f)$ is a canonical element in $\Pi(\alpha, \beta)$. Given an element h in $\Pi(\alpha, \beta)$ and an element a in α, $\mathsf{apply}(h, a)$ yields an element in $\beta(a)$. If the value of h is $\lambda(f)$ then the value of $\mathsf{apply}(h, a)$ is $f(a)$. In order to reduce the notation in the type-theoretics programs, in what follows we simply write the usual function notation for the dependent product sets, their elements and the application of their functional elements. Therefore, we write $(\alpha)\beta$ for the product set $\Pi(\alpha, \beta)$, λ-abstractions for its elements and the usual function application instead of the non-canonical constant apply.

If α is a set and β a family of sets depending on a variable $x \in \alpha$, we can form the dependent sum set $\Sigma x \in \alpha.\beta$. The canonical terms of the Σ-sets are pairs $\langle a, b\rangle$, where $a \in \alpha$ and $b \in \beta[x := a]$. In the case that β does not depend on x, $\Sigma x \in \alpha.\beta$ is called the Cartesian product of α and β and it is simply denoted by $\alpha \times \beta$.

If α and β are sets, the disjoint union of α and β is denoted by $\alpha + \beta$. If $a \in \alpha$ and $b \in \beta$, then $\mathsf{in}_\mathsf{l}(a)$ and $\mathsf{in}_\mathsf{r}(b)$ are canonical elements in $\alpha + \beta$.

Open terms, that is, terms in which not all the variable occurrences are abstracted, are valid in a *context* in which types are assigned to variables. We use the capital Greek letters Γ, Δ, Φ and Θ to range over contexts. A context Γ is a sequence of variable assumptions: $\Gamma \equiv x_1 \in \alpha_1; \ldots; x_n \in \alpha_n$, where the variable names $x_1, \ldots, x_n$ are pairwise distinct and each type α_i, for $1 \leqslant i \leqslant n$, may contain the variables with indices smaller that i. If Γ is a context, a sequence of variable assumptions Δ is called a *context extension* of Γ if $\Gamma; \Delta$ is a context. If there is no place for confusion, we might refer to context extensions

simply as contexts or as extensions. In addition, we might simply say that Δ is an extension whenever the context Γ of which Δ is an extension can be easily deduced.

We extend product and sum sets, and disjoint unions to more than two sets.

If Γ is a context and β a set whose free variables are among the ones assumed in Γ, we write $(\Gamma)\beta$ for the dependent product of all the sets in Γ over β. Formally it is defined by recursion on the length of Γ. If Γ is empty, then $()\beta \equiv \beta$. If $\Gamma \equiv x \in \alpha; \Gamma'$, then $(x \in \alpha; \Gamma')\beta \equiv (x \in \alpha)((\Gamma')\beta)$. Abusing notation once more, we write consecutive dependent products as $(x_1 \in \alpha_1; \ldots; x_n \in \alpha_n)\beta$, $(x_1, x_2, \ldots, x_n \in \alpha)\beta$ instead of $(x_1 \in \alpha; x_2 \in \alpha; \ldots; x_n \in \alpha)\beta$ and their functional elements as $[x_1, \ldots, x_n]b$.

Similarly, $\Sigma(\Gamma)$ is the sum of all the sets in Γ, for a non-empty context Γ. Formally, it is defined by recursion on the length of Γ. If $\Gamma \equiv x \in \alpha$, then $\Sigma(x \in \alpha) \equiv \alpha$. If $\Gamma \equiv x \in \alpha; \Gamma'$, then we have that $\Sigma(x \in \alpha; \Gamma') \equiv \Sigma x \in \alpha.\Sigma(\Gamma')$. If Γ has n assumptions, that is, if $\Gamma \equiv x_1 \in \alpha_1; \ldots; x_n \in \alpha_n$, we use n-tuple notation for its canonical elements and then we write $\langle a_1, a_2, \ldots, a_{n-1}, a_n\rangle$ for $\langle a_1, \langle a_2, \ldots, \langle a_{n-1}, a_n\rangle \cdots \rangle\rangle$.

The disjoint union of n sets $\alpha_1, \ldots, \alpha_n$ is denoted by $\alpha_1 + \ldots + \alpha_n$ and defined as $(\cdots(\alpha_1 + \alpha_2) + \ldots + \alpha_n)$. For the sake of simplicity, we call the corresponding constructors $\mathsf{in}_1, \mathsf{in}_2, \ldots, \mathsf{in}_n$.

3 Some Examples

We illustrate our method for formalising general recursive algorithms in type theory by describing the formalisation of a few easy examples. More detailed descriptions and more examples can be found in [Bov01] (for simple recursive algorithms), [BC01] (for nested algorithms and partial functions) and [Bov02] (for mutually recursive algorithms).

All the auxiliary functions that we use in the examples below are well-known structurally recursive functions defined in the usual way. That is, the recursive calls in those functions are on structurally smaller argument. Therefore, they can be straightforwardly translated in type theory and we can use their translation in the formalisation of the corresponding example. Unless we state the contrary, we assume that the type-theoretic translation of an auxiliary functions has the same name as in the functional program.

The first example is a simple general recursive algorithm: the `quicksort` algorithm over lists of natural numbers. We start by introducing its Haskell definition. Here, we use the set `N` of natural numbers, the inequalities `<` and `>=` over `N` defined in Haskell in a structurally recursive way, and the functions `filter` and `++` defined in the Haskell prelude.

```
quicksort :: [N] -> [N]
quicksort [] = []
quicksort (x:xs) = quicksort (filter (< x) xs) ++
                   x : quicksort (filter (>= x) xs)
```

The first step in the definition of the type-theoretic version of `quicksort` is the construction of the special-purpose accessibility predicate associated with the algorithm. To construct this predicate, we analyse the Haskell code and characterise the inputs on which the algorithm terminates. Thus, we distinguish the following two cases:

- The algorithm `quicksort` terminates on the input `[]`;
- Given a natural number `x` and a list `xs` of natural numbers, the algorithm `quicksort` terminates on the input `(x:xs)` if it terminates on the inputs `(filter (< x) xs)` and `(filter (>= x) xs)`.

From this description, we define the inductive predicate qsAcc over lists of natural numbers by the introduction rules we give below. To avoid confusion in what follows, the type-theoretic translation of the boolean functions < and >= are called $\prec$ and $\succcurlyeq$, respectively. We do not use the symbols $<$ and $\geqslant$ for the formalisation of those functions because, later on, we use the symbols $>$ and $\leqslant$ to denote relations in type theory, that is, terms of type $(\mathsf{N},\mathsf{N})\mathsf{Set}$, while in this example we need terms of type $(\mathsf{N},\mathsf{N})\mathsf{Bool}$. The definition of qsAcc is then

$$\frac{}{\mathsf{qsAcc}(\mathsf{nil})} \qquad \frac{\mathsf{qsAcc}(\mathsf{filter}((\prec x), xs)) \qquad \mathsf{qsAcc}(\mathsf{filter}((\succcurlyeq x), xs))}{\mathsf{qsAcc}(\mathsf{cons}(x, xs))}$$

where $(\prec x)$ denotes the function $[y](y \prec x)$ as in functional programming, similarly for $\succcurlyeq$. We formalise this predicate in type theory as follows:

$$\begin{array}{l}
\mathsf{qsAcc} \in (zs \in \mathsf{List}(\mathsf{N}))\mathsf{Set} \\
\quad \mathsf{qs_acc}_{\mathsf{nil}} \in \mathsf{qsAcc}(\mathsf{nil}) \\
\quad \mathsf{qs_acc}_{\mathsf{cons}} \in (x \in \mathsf{N}; xs \in \mathsf{List}(\mathsf{N}); h_1 \in \mathsf{qsAcc}(\mathsf{filter}((\prec x), xs)); \\
\qquad\qquad h_2 \in \mathsf{qsAcc}(\mathsf{filter}((\succcurlyeq x), xs)) \\
\qquad\qquad)\mathsf{qsAcc}(\mathsf{cons}(x, xs))
\end{array}$$

We define the quicksort algorithm by structural recursion on the proof that the input list of natural numbers satisfies the predicate qsAcc.

$$\begin{array}{l}
\mathsf{quicksort} \in (zs \in \mathsf{List}(\mathsf{N}); \mathsf{qsAcc}(zs))\mathsf{List}(\mathsf{N}) \\
\quad \mathsf{quicksort}(\mathsf{nil}, \mathsf{qs_acc}_{\mathsf{nil}}) = \mathsf{nil} \\
\quad \mathsf{quicksort}(\mathsf{cons}(x, xs), \mathsf{qs_acc}_{\mathsf{cons}}(x, xs, h_1, h_2)) = \\
\qquad \mathsf{quicksort}(\mathsf{filter}((\prec x), xs), h_1) +\!\!+ \mathsf{cons}(x, \mathsf{quicksort}(\mathsf{filter}((\succcurlyeq x), xs), h_2))
\end{array}$$

Finally, as the algorithm `quicksort` is total, we can prove

$$\mathsf{allQsAcc} \in (zs \in \mathsf{List}(\mathsf{N}))\mathsf{qsAcc}(zs)$$

and use that proof to define the type-theoretic function QuickSort.

$$\begin{array}{l}
\mathsf{QuickSort} \in (zs \in \mathsf{List}(\mathsf{N}))\mathsf{List}(\mathsf{N}) \\
\quad \mathsf{QuickSort}(zs) = \mathsf{quicksort}(zs, \mathsf{allQsAcc}(zs))
\end{array}$$

In the next example, we consider the simple partial function given by the following Haskell definition:

```
f :: N -> N
f 0 = 0
f (S n)
   | even n = f(div2 n) + 1
   | odd n  = f(n + 4)
```

where `even` and `odd` are mutually recursive functions that tell whether a natural number is even or odd, respectively, `+` is the addition operation and `div2` is division by two over natural numbers. All of these functions are defined in a structurally recursive way.

Following the description given above, we define the special-purpose accessibility predicate fAcc that characterises the inputs on which the algorithm f terminates. In this example, we have conditional recursive equations depending on the boolean expressions[1] `(even n)` and `(odd n)`. These expressions are translated using the predicates Even and Odd in type theory and they are added as arguments of the corresponding constructor of the accessibility predicate. Here is the type-theoretical definition of the accessibility predicate:

$$
\begin{array}{l}
\mathsf{fAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{f\_acc}_0 \in \mathsf{fAcc}(0) \\
\quad \mathsf{f\_acc}_{\mathsf{s}1} \in (n \in \mathsf{N}; q \in \mathsf{Even}(n); h \in \mathsf{fAcc}(\mathsf{div2}(n)))\mathsf{fAcc}(\mathsf{s}(n)) \\
\quad \mathsf{f\_acc}_{\mathsf{s}2} \in (n \in \mathsf{N}; q \in \mathsf{Odd}(n); h \in \mathsf{fAcc}(n+4))\mathsf{fAcc}(\mathsf{s}(n))
\end{array}
$$

We use this predicate to define the type-theoretical version of `f` by structural recursion on the proof that the input natural number satisfies the predicate fAcc.

$$
\begin{array}{l}
\mathsf{f} \in (m \in \mathsf{N}; \mathsf{fAcc}(m))\mathsf{N} \\
\quad \mathsf{f}(0, \mathsf{f\_acc}_0) = 0 \\
\quad \mathsf{f}(\mathsf{s}(n), \mathsf{f\_acc}_{\mathsf{s}1}(n, q, h)) = \mathsf{f}(\mathsf{div2}(n), h) + 1 \\
\quad \mathsf{f}(\mathsf{s}(n), \mathsf{f\_acc}_{\mathsf{s}2}(n, q, h)) = \mathsf{f}(n + 4, h)
\end{array}
$$

In this example we cannot prove $\forall m \in \mathsf{N}.\mathsf{fAcc}(m)$, simply because it is not true. However, for those $m \in \mathsf{N}$ that have a proof $h \in \mathsf{fAcc}(m)$, we can compute $\mathsf{f}(m, h)$. This example shows that the formalisation of partial recursive functions in type theory is not a problem.

Our method applies also to the formalisation of nested recursive algorithms. Here is the Haskell code of McCarthy's f_{91} function [MM70].

```
f_91 :: N -> N
f_91 n
   | n > 100  = n - 10
   | n <= 100 = f_91 (f_91 (n + 11))
```

where `-` is the subtraction operation over natural numbers, and `<=` and `>` are inequalities over `N` defined in the usual way. The function `f_91` computes the number 91 for inputs that are less than or equal to 101 and for other inputs n, it computes the value $n - 10$.

[1] Conditional expressions are usually called *guards* in Haskell literature.

Following our method, we would construct the predicate $\mathsf{f_{91}Acc}$ defined by the following introduction rules (for n a natural number):

$$\frac{n > 100}{\mathsf{f_{91}Acc}(n)} \qquad \frac{n \leqslant 100 \qquad \mathsf{f_{91}Acc}(n+11) \qquad \mathsf{f_{91}Acc}(\mathsf{f_{91}}(n+11))}{\mathsf{f_{91}Acc}(n)}$$

Unfortunately, this definition is not correct in ordinary type theory, since the algorithm $\mathsf{f_{91}}$ is not defined yet and, therefore, cannot be used in the definition of the predicate. Moreover, the purpose of defining the predicate $\mathsf{f_{91}Acc}$ is to be able to define the algorithm $\mathsf{f_{91}}$ by structural recursion on the proof that its input value satisfies $\mathsf{f_{91}Acc}$, so we need $\mathsf{f_{91}Acc}$ to define $\mathsf{f_{91}}$. However, there is an extension of type theory that gives us the means to define the predicate $\mathsf{f_{91}Acc}$ and the function $\mathsf{f_{91}}$ at the same time. This extension has been introduced by Dybjer in [Dyb00] and it allows the simultaneous definition of a predicate P and a function f, where f has P as part of its domain and is defined by recursion on P. Using Dybjer's schema, we can define $\mathsf{f_{91}Acc}$ and $\mathsf{f_{91}}$ simultaneously as follows:

$$\begin{array}{l}
\mathsf{f_{91}Acc} \in (n \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{f_{91}acc}_{>100} \in (n \in \mathsf{N}; q \in (n > 100))\mathsf{f_{91}Acc}(n) \\
\quad \mathsf{f_{91}acc}_{\leqslant 100} \in (n \in \mathsf{N}; q \in (n \leqslant 100); h_1 \in \mathsf{f_{91}Acc}(n+11); \\
\qquad\qquad\qquad h_2 \in \mathsf{f_{91}Acc}(\mathsf{f_{91}}(n+11, h_1)) \\
\qquad\qquad\qquad)\mathsf{f_{91}Acc}(n)
\end{array}$$

$$\begin{array}{l}
\mathsf{f_{91}} \in (n \in \mathsf{N}; \mathsf{f_{91}Acc}(n))\mathsf{N} \\
\quad \mathsf{f_{91}}(n, \mathsf{f_{91}acc}_{>100}(n, q)) = n - 10 \\
\quad \mathsf{f_{91}}(n, \mathsf{f_{91}acc}_{\leqslant 100}(n, q, h_1, h_2)) = \mathsf{f_{91}}(\mathsf{f_{91}}(n+11, h_1), h_2)
\end{array}$$

Mutually recursive algorithms, with or without nested recursive calls, can also be formalised with our method. If the mutually recursive algorithms are not nested, their formalisation is similar to the formalisation of the quicksort algorithm in the sense that we first define the accessibility predicate for each function and then, we formalise the algorithms by structural recursion on the proofs that the input values satisfy the corresponding predicate. When we have mutually recursive algorithms, the termination of one function depends on the termination of the others and hence, the accessibility predicates are also mutually recursive. If, in addition to mutual recursion, we have nested calls, we again need to define the predicates simultaneously with the algorithms. In order to do so, we need to extend Dybjer's schema for cases where we have several mutually recursive predicates defined simultaneously with several functions (originally, Dybjer's schema considers one predicate and one function). This extension and its application to the formalisation of mutually recursive functions in type theory was given in [Bov02]. Let us consider a simple example where we define two mutually recursive algorithms. In Haskell we write them as follows[2]:

[2]We ignore efficiency aspects such that the fact that some expressions are computed more than once.

```
f :: N -> N
f 0 = 0
f (S n)
   | g n <= n = f(g n) + n
   | g n >  n = 0

g :: N -> N
g 0 = 1
g (S n)
   | f n <= n = g(f n) + f n
   | f n >  n = f n + n
```

In the type-theoretic translation, we need to define two mutually recursive predicates fAcc and gAcc simultaneously with two mutually recursive algorithms f and g that, in turn, are defined by structural recursion on the corresponding accessibility predicate. Here is the type-theoretical version of our example:

$$
\begin{array}{l}
\mathsf{fAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{f\_acc}_0 \in \mathsf{fAcc}(0) \\
\quad \mathsf{f\_acc}_{\mathsf{s}1} \in (n \in \mathsf{N}; h_1 \in \mathsf{gAcc}(n); q \in (\mathsf{g}(n, h_1) \leqslant n); h_2 \in \mathsf{fAcc}(\mathsf{g}(n, h_1)) \\
\qquad\qquad)\mathsf{fAcc}(\mathsf{s}(n)) \\
\quad \mathsf{f\_acc}_{\mathsf{s}2} \in (n \in \mathsf{N}; h_1 \in \mathsf{gAcc}(n); q \in (\mathsf{g}(n, h_1) > n))\mathsf{fAcc}(\mathsf{s}(n)) \\
\\
\mathsf{gAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{g\_acc}_0 \in \mathsf{gAcc}(0) \\
\quad \mathsf{g\_acc}_{\mathsf{s}1} \in (n \in \mathsf{N}; h_1 \in \mathsf{fAcc}(n); q \in (\mathsf{f}(n, h_1) \leqslant n); h_2 \in \mathsf{gAcc}(\mathsf{f}(n, h_1)) \\
\qquad\qquad)\mathsf{gAcc}(\mathsf{s}(n)) \\
\quad \mathsf{g\_acc}_{\mathsf{s}2} \in (n \in \mathsf{N}; h_1 \in \mathsf{fAcc}(n); q \in (\mathsf{f}(n, h_1) > n))\mathsf{gAcc}(\mathsf{s}(n)) \\
\\
\mathsf{f} \in (m \in \mathsf{N}; \mathsf{fAcc}(m))\mathsf{N} \\
\quad \mathsf{f}(0, \mathsf{f\_acc}_0) = 0 \\
\quad \mathsf{f}(\mathsf{s}(n), \mathsf{f\_acc}_{\mathsf{s}1}(n, h_1, q, h_2)) = \mathsf{f}(\mathsf{g}(n, h_1), h_2) + n \\
\quad \mathsf{f}(\mathsf{s}(n), \mathsf{f\_acc}_{\mathsf{s}2}(n, h_1, q)) = 0 \\
\\
\mathsf{g} \in (m \in \mathsf{N}; \mathsf{gAcc}(m))\mathsf{N} \\
\quad \mathsf{g}(0, \mathsf{g\_acc}_0) = 1 \\
\quad \mathsf{g}(\mathsf{s}(n), \mathsf{g\_acc}_{\mathsf{s}1}(n, h_1, q, h_2)) = \mathsf{g}(\mathsf{f}(n, h_1), h_2) + \mathsf{f}(n, h_1) \\
\quad \mathsf{g}(\mathsf{s}(n), \mathsf{g\_acc}_{\mathsf{s}2}(n, h_1, q)) = \mathsf{f}(n, h_1) + n
\end{array}
$$

Partial functions may also be defined by occurrences of nested and/or mutually recursive calls. This fact is irrelevant to our method and hence, their formalisations present no problem.

As a final remark, we draw the reader's attention to the simplicity of the translations. The accessibility predicates can be automatically generated from the recursive equations and the type-theoretic versions of the algorithms look very similar to the original programs except for the extra proof argument. If

we suppress the proofs of the accessibility predicate we get almost exactly the original algorithms.

3.1 Necessary Restrictions

In the following sections, we show that our method is of general applicability. Specifically, we define a large class of functional programs to which it can be applied.

However, we need to impose some restrictions on that class. Here, we illustrate the need of those restrictions by showing a few functional programs that cannot be translated using our method.

The first restriction is that, in the definition of a function `f`, any occurrence of `f` should always be fully applied. Let us consider the following definition

```
f :: N -> N
f 0 = 0
f (S n) = (iter f n n) + 1
```

where `iter` is an iteration function such that when applied to a function f and a number n gives f^n as a result. Here, the defined function `f` appears in the right-hand side of the second equation without being applied to any argument. Although it is easy to see that `f` computes the identity, we do not know at the moment how to translate this definition in type theory using our special-purpose accessibility predicates. Hence, in what follows, we do not allow this kind of definitions.

The reason why we need to impose this restriction becomes clear when we try to apply our method to the function above. For the formalisation of this function, we have to define a predicate fAcc and a function f with types:

$$\mathsf{fAcc} \in (m \in \mathsf{N})\mathsf{Set}$$
$$\mathsf{f} \in (m \in \mathsf{N}; \mathsf{fAcc}(m))\mathsf{N}$$

What should the constructors of fAcc look like? Our method requires that every argument to which the function `f` is applied satisfies the predicate fAcc. But the occurrence of `f` in the right-hand side of the second equation in the definition of `f` is not directly applied to an argument, so we do not know how to formulate the type of the corresponding constructor of fAcc. For this reason, we require every occurrence of `f` in the right-hand side of a recursive equation and in the conditional expression corresponding to the equation (if any) to be fully applied. If the function `f` is one of the functions being defined in a mutual recursive definition, then `f` should always occur fully applied within the mutual recursive definition.

A functional programmer could have the idea of replacing the occurrence of `f` with its η-expansion:

```
f (S n) = (iter (\x -> (f x)) n n) + 1
```

In this way the occurrence of `f` is applied to the variable `x`, thus satisfying the restriction. However, since the variable is bound inside the right-hand side of the equation, the constructor of fAcc would have to require that every possible value of the variable x satisfies fAcc:

$$\mathsf{f\_acc_s} \in (n \in \mathsf{N}; H \in (x \in \mathsf{N})\mathsf{fAcc}(x))\mathsf{fAcc}(\mathsf{s}(n))$$

This clearly makes it impossible to prove $\mathsf{fAcc}(\mathsf{s}(n))$, since we would first need to prove the totality of fAcc to deduce it. In section 7, we will further discuss the treatment of λ-abstractions in the right-hand side of recursive equations.

Another restriction is that each function definition should be self-standing, by which we mean that it should not call other previously defined functions unless they are structurally recursive functions that introduce no partiality. That is, the functions that can be freely used inside other definitions are such that the recursive calls on those functions are performed on structurally smaller arguments, they are defined for all inputs and they do not have internal calls to any general recursive function. In what follows, we shortly refer to these functions as structurally recursive functions, but the reader should keep in mind that structurally recursive functions should be such that they do not introduce partiality.

Then, if `f` is a general recursive function, it should be translated in type theory as a pair consisting of a predicate fAcc and a function f. Thus, we cannot call it inside the definition of another function `g`. This restriction is imposed by type-checking requirements and it will become clearer below. If, instead, `f` is structurally recursive, it can be directly translated in type theory as a structurally recursive function f, without the need of our auxiliary predicate fAcc. In this case, the use of `f` inside the definition of another function `g` is allowed, as it has been seen throughout this section.

We illustrate the reason for this restriction with the following example.

```
nub_map :: (N -> N) -> [N] -> [N]
nub_map f [] = []
nub_map f (x:xs) = f x : nub_map f (filter (/= x) xs)

f :: N -> N
f 0 = 0
f (S n) = f (S (S n))

g :: [N] -> [N]
g xs = nub_map f xs
```

where `/=` is the inequality operator in Haskell.

When we apply our method to each of the functions in this program, we first get the translation of `nub_map`:

$$\begin{array}{l}\mathsf{nub\_mapAcc} \in (f \in (N)N; l \in \mathsf{List}(\mathsf{N}))\mathsf{Set}\\ \mathsf{nub\_map} \in (f \in (\mathsf{N})\mathsf{N}; l \in \mathsf{List}(\mathsf{N}); \mathsf{nub\_mapAcc}(f, l))\mathsf{List}(\mathsf{N})\end{array}$$

Similarly, the partial function `f` is translated as:

$$\begin{array}{l}\mathsf{fAcc} \in (m \in \mathsf{N})\mathsf{Set} \\ \mathsf{f} \in (m \in \mathsf{N}; \mathsf{fAcc}(m))\mathsf{N}\end{array}$$

The problem arises when we try to translate `g`. The translation should be given by a predicate and a function with the following types:

$$\begin{array}{l}\mathsf{gAcc} \in (l \in \mathsf{List}(\mathsf{N}))\mathsf{Set} \\ \mathsf{g} \in (l \in \mathsf{List}(\mathsf{N}); \mathsf{gAcc}(l))\mathsf{List}(\mathsf{N})\end{array}$$

Even though `g` is not recursive (it does not call itself), it inherits a termination condition from `nub_map`. Thus, we have to translate `g` with a predicate gAcc and a function `g`. The difficulty now consists in how to formulate the constructors of gAcc and the equations that define `g`. The problem here is that the translation of the term `(nub_map f xs)` would not type-check because `f` does not have the type (N)N anymore.

For this reason, we require that the only previously defined functions allowed in a new function definition are the structurally recursive ones. In reality, this condition could be relaxed by allowing any function that can be proved total in type theory. As we have seen in the formalisation of the `quicksort` algorithm, we can sometimes define a total function that does not depend on the special accessibility predicate anymore, even when the algorithm is a general recursive one. The function QuickSort is an example of such a function. It would be safe to also allow these functions inside the definition of other functions. Then, the class of functions to which our method would apply would depend on what we can prove in type theory. To keep the definition of this class of functions simple, we choose not to follow this path. Although this is a severe restriction, we show in section 5 that the class of functions that we consider still allows us to define all recursive functions.

One might think that a possible way around this problem could be to define `nub_map`, `f`, and `g` as mutually dependent functions. However, this does not work for this particular example because we would fall into the first restriction. The function `f` is one of the functions being defined and the occurrence of `f` inside `g` is not fully applied and thus disallowed.

A solution to the problem stated above can be given if we adopt an impredicative type system. Using impredicativity, we can define the type of partial functions from α to β by making the domain predicate part of the object:

$$\alpha \rightharpoonup \beta \equiv \Sigma\, P \in (\alpha)\mathsf{Set}.(x \in \alpha; P(x))\beta$$

Thus, an object of type $\alpha \rightharpoonup \beta$ is a pair $\langle P, f\rangle$ consisting of a predicate P over α and a function f defined over the elements of α that satisfy the predicate. To be precise, we should also require, as a third component, a proof that f does not depend on its second argument, that is, a proof that $f(x, h_1) = f(x, h_2)$ for x in α and any two proofs h_1 and h_2 of $P(x)$. For the sake of simplicity, we leave this third component out since it is not necessary in the definition of

the translation, but just to guarantee that the function does not depend on the proof of the predicate.

Having defined the type of partial functions, we can translate functional programs into type theory by consistently interpreting any functional type `A -> B` as $\alpha \rightharpoonup \beta$, where α and β are the interpretation of `A` and `B`, respectively. Then, the function `nub_map` becomes:

$$\begin{array}{l}
\mathsf{nub_mapAcc} \in (p \in (\mathsf{N} \rightharpoonup \mathsf{N}) \times \mathsf{List(N)})\mathsf{Set} \\
\mathsf{nub_map} \in (p \in (\mathsf{N} \rightharpoonup \mathsf{N}) \times \mathsf{List(N)}; \mathsf{nub_mapAcc}(p))\mathsf{List(N)}
\end{array}$$

$$\begin{array}{l}
\mathsf{NubMap} \in (\mathsf{N} \rightharpoonup \mathsf{N}) \times \mathsf{List(N)} \rightharpoonup \mathsf{List(N)} \\
\mathsf{NubMap} = \langle \mathsf{nub_mapAcc}, \mathsf{nub_map} \rangle
\end{array}$$

The function `f` is translated as above, except that now we can eventually pack its special accessibility predicate and its structural function definition into one object of a partial function type:

$$\begin{array}{l}
\mathsf{fAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\mathsf{f} \in (m \in \mathsf{N}; \mathsf{fAcc}(m))\mathsf{N}
\end{array}$$

$$\begin{array}{l}
\mathsf{F} \in \mathsf{N} \rightharpoonup \mathsf{N} \\
\mathsf{F} = \langle \mathsf{fAcc}, \mathsf{f} \rangle
\end{array}$$

Finally, we are able to give a translation for `g`:

$$\begin{array}{l}
\mathsf{gAcc} \in (\mathsf{N})\mathsf{Set} \\
\mathsf{g_acc} \in (l \in \mathsf{List(N)}; \mathsf{nub_mapAcc}(\langle \mathsf{F}, l \rangle))\mathsf{gAcc}(l)
\end{array}$$

$$\begin{array}{l}
\mathsf{g} \in (l \in \mathsf{N}; \mathsf{gAcc}(l))\mathsf{List(N)} \\
\mathsf{g}(l, \mathsf{g_acc}(l, h)) = \mathsf{nub_map}(\langle \mathsf{F}, l \rangle, h)
\end{array}$$

$$\begin{array}{l}
\mathsf{G} \in \mathsf{List(N)} \rightharpoonup \mathsf{List(N)} \\
\mathsf{G} = \langle \mathsf{gAcc}, \mathsf{g} \rangle
\end{array}$$

Since in this work we use predicative type theory, we want to avoid such impredicative definitions. This is the reason why we decide to restrict the use of general recursive functions inside other function definitions.

4 General Recursive Definitions

In this section, we specify the class $\mathcal{FP}$ of functional programs that we consider. It is a subclass of the class of functions that can be defined in any functional programming language like Haskell, ML or Clean.

In the previous section we explained that we must impose some restrictions on this subclass. Here, we formalise these restrictions, namely, we require that all recursive calls in a recursive definition are fully applied and that only structurally recursive functions can be used inside the definition of a function.

4.1 The Class of Types

First of all, let us characterise the class of types that can appear in a program. These may be basic types, that are either variable types or inductive data types, or function types.

Let us assume that we have an infinite set of type variables $\mathcal{TV}$. The class of types that are allowed in our programs are inductively defined by:

- All elements of $\mathcal{TV}$ are types.
- Inductive data types are types. An inductive data type is introduced by a definition of the form

$$\begin{array}{rl} \texttt{Inductive T} ::= & \texttt{c}_1\ \tau_{11}\ \ldots\ \tau_{1k_1} \mid \\ & \vdots \\ & \texttt{c}_o\ \tau_{o1}\ \cdots\ \tau_{ok_o} \end{array}$$

  where $0 \leqslant o$ and $0 \leqslant k_i$ for $0 \leqslant i \leqslant o$. Here, if we consider T as a type variable, then every τ is a type with the extra condition that T occurs only strictly positively in it. This means that in every τ, the type T can only occur to the right of arrows.
- If σ and τ are types, then $\sigma \to \tau$ is a type.

In what follows, σ and τ denote types.

With each type σ, we associate an infinite set of variables. For simplicity, we assume that the sets of variables associated with two different types are disjoint.

Besides types, we also use *specifications* of the form

$$\sigma_1, \ldots, \sigma_m \Rightarrow \tau$$

If e has the above specification, it must be interpreted as follows: e is an expression that, when applied to arguments $a_1, \ldots, a_m$ of type $\sigma_1, \ldots, \sigma_m$, respectively, produces a term $e(a_1, \ldots, a_m)$ of type τ. The expression e itself is not a term of any type; in particular, it is not an element of the functional type $\sigma_1 \to \cdots \to \sigma_m \to \tau$. We introduce specifications to be able to formalise the requirement, explained in the previous section, that a function must be fully applied to be allowed to appear in the right-hand side of any of the equations within the block that defines the function. As we explain below, we also use specifications to force constructors to be fully applied.

In what follows, we write $a\colon A$ to denote that a is an expression of type A or that a has the specification A.

Given $a\colon \sigma$, the reader should keep in mind the difference between $f(a)$, that denotes the application of a function f with specification $\sigma \Rightarrow \tau$ to a, and $(f\ a)$, that denotes the application of a function f of type $\sigma \to \tau$ to a.

The definition of the inductive data type above introduces not only the new type T but also its constructors:

$$\texttt{c}_i\colon \tau_{i1}, \ldots, \tau_{ik_i} \Rightarrow \texttt{T}$$

Hence, we are not allowed to use $\mathtt{c}_i$ applied only to a few of its arguments. This is why we do not directly give to $\mathtt{c}_i$ the type $\tau_{i1} \to \cdots \to \tau_{ik_i} \to \mathtt{T}$.

As examples, we show how to define some common inductive data types. The types of boolean values and of natural numbers are defined as

$$\mathtt{Inductive\ Bool} ::= \mathtt{true} \mid \mathtt{false} \qquad \text{and} \qquad \mathtt{Inductive\ Nat} ::= \mathtt{0} \mid \mathtt{s\ Nat},$$

respectively.

Type variables can appear in an inductive data type definition. Then, we have a parametric data type. Usually, the type variables are explicitly written in the left-hand side of the definition:

$$\begin{array}{lll} \mathtt{Inductive\ T}\ \gamma_1\ \cdots\ \gamma_w ::= & \mathtt{c}_1\ \tau_{11}\ \cdots\ \tau_{1k_1} & \mid \\ & \vdots & \\ & \mathtt{c}_o\ \tau_{o1}\ \cdots\ \tau_{ok_o} & \end{array}$$

where $\gamma_1, \ldots, \gamma_w$ are type variables, for $1 \leqslant w$. Here, every occurrence of $\mathtt{T}$ in the τ's must be of the form $(\mathtt{T}\ \gamma_1 \cdots \gamma_w)$.

To instantiate the above definition, we simply write $(\mathtt{T}\ \sigma_1 \cdots \sigma_w)$ for specific types $\sigma_1, \ldots, \sigma_w$. This expression denotes the type obtained by substituting each γ_g by σ_g in the definition of $\mathtt{T}$, for $1 \leqslant g \leqslant w$.

Let γ and δ be type variables. A typical example of a parametric data type is the type of lists, defined as

$$\mathtt{Inductive\ List}\ \gamma ::= \mathtt{nil} \mid \mathtt{cons}\ \gamma\ (\mathtt{List}\ \gamma).$$

Product and sum types can also be seen as parametric data types. Here, we write the type constructors $\times$ and $+$ with infix notation:

$$\mathtt{Inductive}\ \gamma \times \delta ::= \mathtt{pair}\ \gamma\ \delta \qquad \text{and} \qquad \mathtt{Inductive}\ \gamma + \delta := \mathtt{inl}\ \gamma \mid \mathtt{inr}\ \delta.$$

We can directly translate any of the types that can occur in a functional program into type theory. The equivalent type-theoretic definitions are almost the same, except for a change in notation.

4.2 Terms and Patterns

Functional programs are defined by pattern matching. In addition, each function is defined by a sequence of recursive equations. Here, we formally define patterns and the terms that are allowed in the equations.

Definition 1. Let $\mathtt{T}$ be a type. A *pattern* of type $\mathtt{T}$ is an expression built up according to the following two rules.

- A variable of type $\mathtt{T}$ is a pattern of type $\mathtt{T}$.
- If $\mathtt{T}$ is an inductive type, $\mathtt{c}\colon \tau_1, \ldots, \tau_k \Rightarrow \mathtt{T}$ is one of its constructors and $p_1, \ldots, p_k$ are patterns of type $\tau_1, \ldots, \tau_k$, respectively, then $\mathtt{c}(p_1, \ldots, p_k)$ is a pattern of type $\mathtt{T}$.

Variables occurring in a pattern are called *pattern variables*. A pattern is *linear* if every pattern variable occurs only once in the pattern.

We consider here only linear patterns. Usually, we say just *pattern* when we refer to a *linear pattern*.

Definition 2. A sequence of patterns $p_1 \cdots p_m$ of type `T` is said to be *exclusive* if it is not possible to obtain the same term by instantiating two different patterns, that is, by substituting the pattern variables in those patterns by other terms.

The sequence is called *exhaustive* if every value of type `T` is an instance of at least one of the patterns. By *value* we mean a closed term that cannot be further reduced according to the reduction rules that we present later on.

The terms that are allowed in a recursive definition depend on three parameters: the variables that can occur free in the terms, the functions being defined, which can be used in the recursive calls, and the previously defined functions that we allow within a definition. Let us assume that we have a class $\mathcal{F}$ of functions together with their types such that every element $\mathtt{f}\colon \sigma_1 \to \cdots \to \sigma_m \to \tau$ in $\mathcal{F}$ can be translated into type theory with the same functional type. In what follows, we assume that $\mathcal{F}$ contains all structurally recursive functions that introduce no partiality, as described in section 3.1. As we have already mentioned, it is possible to extend $\mathcal{F}$ to a larger class of functions by adding all the functions that can be proved total in type theory. As we can consider the class $\mathcal{F}$ to be fixed, below we give the definition of the terms that we allow in a recursive definition considering only as parameters the variables that can occur free in the terms and the functions being defined. Let $\mathcal{X}$ be the set of variables that can occur free in a term together with their types. The idea is that, when defining an equation, this set only contains the pattern variables of the equation together with their corresponding types. Let $\mathcal{SF}$ be the set of the names of the functions being defined together with their specifications. When defining a single function, $\mathcal{SF}$ contains only one function name and its specification. When defining several functions within a mutually recursive block, $\mathcal{SF}$ contains a function name and its specification for each of the functions being mutually defined. Formally, we define terms as follows.

Definition 3. Let $\mathcal{X}$ be a set of variables together with their types. Let $\mathcal{SF}$ be a set of function names together with their specifications, that is, every element of $\mathcal{SF}$ is of the form $\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau$. Let the set of names of the variables in $\mathcal{X}$, the set of names of the functions in $\mathcal{SF}$ and the set of names of the functions in $\mathcal{F}$ be disjoint. The class of *valid terms with respect to* $\mathcal{X}$ *and* $\mathcal{SF}$ is built up according to the rules below. When the sets of variables and of functions name and their specifications remain the same, we simply refer to the terms in the class as *valid terms*. Below, together with the definition of the valid terms we give their corresponding types. If τ is the type of the valid term t, we say "t is a valid term of type τ" or "$t\colon \tau$ is a valid term", indistinctly.

- The variables in $\mathcal{X}$ are valid terms of the corresponding type.
- The functions in $\mathcal{F}$ are valid terms of the corresponding type.
- If $\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau$ is an element of $\mathcal{SF}$ and if $a_1, \ldots, a_m$ are valid terms of type $\sigma_1, \ldots, \sigma_m$, respectively, then $f(a_1, \ldots, a_m)$ is a valid term of type τ.
- If $\mathtt{c}\colon \tau_1, \ldots, \tau_k \Rightarrow \mathtt{T}$ is one of the constructors of the inductive data type $\mathtt{T}$ and if $a_1\colon \tau_1, \ldots, a_k\colon \tau_k$ are valid terms, then $\mathtt{c}(a_1, \ldots, a_k)\colon \mathtt{T}$ is a valid term.
- Let $x\colon \sigma$ be a variable and its type. Let $(\mathcal{X}/x)$ be the set $\mathcal{X}$ without any association for the variable x. If $b\colon \tau$ is a valid term with respect to $(\mathcal{X}/x)\bigcup\{x\colon \sigma\}$ and $\mathcal{SF}$, then $[x]b\colon \sigma \to \tau$ is a valid term.
- If $f\colon \sigma \to \tau$ and $a\colon \sigma$ are valid terms, then $(f\ a)\colon \tau$ is a valid term. If there is no place for confusion, we might just write $f\ a$ for the application of the function f to the argument a.
- Let t be a valid term of an inductive data type $\mathtt{T}$. Let $0 \leqslant v$ and $0 \leqslant s \leqslant v$. Let $p_1, \ldots, p_v$ be exclusive patterns of type $\mathtt{T}$ and let $\mathcal{Y}_s$ be the set of pattern variables in p_s together with their types. Finally, let $e_1, \ldots, e_v$ be valid terms of type τ with respect to $(\mathcal{X}, \mathcal{Y}_1), \ldots, (\mathcal{X}, \mathcal{Y}_v)$,respectively, and $\mathcal{SF}$. Then, the *case* expression

$$\mathtt{Cases}\ t\ \mathtt{of} \begin{cases} p_1 \mapsto e_1 \\ \quad\vdots \\ p_v \mapsto e_v \end{cases}$$

  is a valid term of type τ.

Notice that we do not require the patterns in a case expression to be exhaustive. This is to be consistent with the fact that we allow partiality in the definitions. We could also drop the requirement that the patterns should be exclusive and just say that, in a case expression, the first matching pattern is used, which is usually done in functional programming. However, this makes the semantics of case expressions depend on the order of the patterns and it complicates their interpretation in type theory. Requiring that the patterns are mutually exclusive does not seriously limit the expressiveness of the definitions.

We adopt a strict semantics to compute the programs in $\mathcal{FP}$ (see section 7 for further discussion on this matter). This said, the computation rules for terms are the usual ones. Below, we give a brief overview of them.

Let $\rightsquigarrow$ denote one step reduction over terms and $\rightsquigarrow^*$ its reflexive and transitive closure. Let $t\colon \tau$ be a valid term, p be a pattern for τ and $\overline{y}$ its pattern variables. We write $t \overset{\rightsquigarrow}{=} p[\overline{y := b}]$ whenever there exists a fully evaluated valid term $t^*\colon \tau$ and a sequence of terms $\overline{b}$ such that $t \rightsquigarrow^* t^*$, t^* matches p, and $[\overline{y := b}]$

is the substitution which results from this matching, that is, $t^* = p_s[\overline{y := b}]$ when we simultaneously substitute all pattern variables $\overline{y}$ by terms $\overline{b}$.

For a β-redex $([x]b\ a)$, we have the β-reduction rule $([x]b\ a) \rightsquigarrow b[x := a]$.

The computation behaviour of recursive calls is given in the next subsection.

Case expressions are computed by pattern matching. If $t\colon \mathtt{T}$ is the term on which we perform the case analysis, we first need to reduce it. If we have that $t \stackrel{\sim}{=} p_s[\overline{y := a}]$, for some s, then the computation rule for case expressions is

$$\mathtt{Cases}\ t\ \mathtt{of} \begin{cases} p_1 \mapsto e_1 & \\ \quad\vdots & \rightsquigarrow e_s[\overline{y := a}] \\ p_v \mapsto e_v & \end{cases}$$

If t does not reduce to an instance of any of the patterns, it is not possible to further reduce the case expression.

A special kind of case expression occurs when we consider cases over the type of booleans. Since this is a particularly common instance, some programming languages use the notation

$$\mathtt{if}\ b\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \qquad \text{for} \qquad \mathtt{Cases}\ b\ \mathtt{of} \begin{cases} \mathtt{true} & \mapsto e_1 \\ \mathtt{false} & \mapsto e_2. \end{cases}$$

4.3 Fixed Point Function Definition

We define the class $\mathcal{FP}$ of recursive functions that we want to translate into type theory. These functions are given by single or mutual fixed point equations satisfying some conditions. The general form of a single recursive definition is

$$\begin{aligned} \mathtt{fix}\ \ & \mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau \\ & \mathtt{f}\ p_{11}\ \cdots\ p_{1m} = e_1 \\ & \qquad\vdots \\ & \mathtt{f}\ p_{l1}\ \cdots\ p_{lm} = e_l \end{aligned}$$

where $p_{1u}, \ldots, p_{lu}$ are patterns of type σ_u, for $0 \leqslant u \leqslant m$. We call a sequence of patterns $p_1 \cdots p_m$ of types $\sigma_1, \ldots, \sigma_m$, respectively, a *multipattern* for $\mathtt{f}$. Usually, we just say *pattern* when referring to *multipattern*, if there is no confusion. We extend the notions of linearity, exclusivity, and exhaustivity from patterns to multipatterns in the straightforward way. The multipatterns that appear in the left-hand side of the definition of a function $\mathtt{f}$ must be linear and mutually exclusive. In this way, at most one equation in the definition of f can be applied to compute f on a given sequence of input arguments.

Let $1 \leqslant i \leqslant l$ and let $\mathcal{Y}_i$ be the set of pattern variables that occurs in the ith equation together with their types. The right-hand sides of the equations in the definition of $\mathtt{f}$, that is, $e_1, \ldots, e_l$, are valid terms of type τ with respect to $\mathcal{Y}_1, \ldots, \mathcal{Y}_l$, respectively, and $\{\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau\}$. Hence, each e_i can contain subterms of the form $\mathtt{f}(a_1, \ldots, a_m)$, where $a_1, \ldots, a_m$ are valid terms of type $\sigma_1, \ldots, \sigma_m$, respectively. Each a_u can, in turn, contain occurrences of $\mathtt{f}$, giving rise to nested recursive definitions, for $1 \leqslant u \leqslant m$.

Let us give some examples of recursive functions defined in this way.

The Fibonacci function is defined as

$$\begin{array}{ll} \texttt{fix} & \texttt{fib}\colon \texttt{Nat} \Rightarrow \texttt{Nat} \\ & \texttt{fib}\ \texttt{0} = \texttt{s(0)} \\ & \texttt{fib}\ \texttt{s(0)} = \texttt{s(0)} \\ & \texttt{fib}\ \texttt{s(s}(n)) = \texttt{fib}(n) + \texttt{fib(s}(n)) \end{array}$$

where $+\colon \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}$ is one of the functions defined in the class $\mathcal{F}$.

The concatenation of lists is defined as

$$\begin{array}{ll} \texttt{fix} & \texttt{concat}\colon \texttt{List}\ \gamma, \texttt{List}\ \gamma \Rightarrow \texttt{List}\ \gamma \\ & \texttt{concat}\ \texttt{nil}\ l_2 = l_2 \\ & \texttt{concat}\ \texttt{cons}(b, l_1)\ l_2 = \texttt{cons}(b, \texttt{concat}(l_1, l_2)) \end{array}$$

Since this function is actually defined by pattern matching only on the first argument, it can also be defined as

$$\begin{array}{ll} \texttt{fix} & \texttt{concat}\colon \texttt{List}\ \gamma \Rightarrow \texttt{List}\ \gamma \to \texttt{List}\ \gamma \\ & \texttt{concat}\ \texttt{nil} = [l_2]\ l_2 \\ & \texttt{concat}\ \texttt{cons}(b, l_1) = [l_2]\texttt{cons}(b, \texttt{concat}(l_1, l_2)) \end{array}$$

In general, it is better to put to the left of the symbol $\Rightarrow$ only the arguments that play an actual role in the recursion. This point will become clear in section 6 when we describe the method to translate the functions in $\mathcal{FP}$ into type theory.

The computation rules for $\texttt{f}$ are given by the different equations in its definition. If we want to compute the expression $\texttt{f}(a_1, \ldots, a_m)$, we first have to reduce the sequence $a_1 \cdots a_m$. Let us have $\overline{a} \overset{\rightsquigarrow}{=} \overline{p}[\overline{y := b}]$, that is, we have that $a_1 \overset{\rightsquigarrow}{=} p_1[\overline{y := b}]$, $\ldots$, $a_m \overset{\rightsquigarrow}{=} p_m[\overline{y := b}]$ for a multipattern $p_1 \cdots p_m$ in the left-hand side of one of the equations defining $\texttt{f}$. Let $\texttt{f}\ p_1 \cdots p_m = e$ be the corresponding equation. Then, we have the following computation rule

$$\texttt{f}(a_1, \ldots, a_m) \rightsquigarrow e[\overline{y := b}]$$

Otherwise, the function $\texttt{f}$ is undefined on the input $(a_1, \ldots, a_m)$.

The class of functions $\mathcal{FP}$ also contains mutually recursive definitions. The general form for defining n mutually recursive functions is as follows:

$$\begin{array}{ll} \texttt{mutual fix} & \texttt{f}_1\colon \sigma_{11}, \ldots, \sigma_{1m_1} \Rightarrow \tau_1 \\ & \texttt{f}_1\ p_{111}\ \cdots\ p_{11m_1} = e_{11} \\ & \qquad\vdots \\ & \texttt{f}_1\ p_{1l_11}\ \cdots\ p_{1l_1m_1} = e_{1l_1} \\ & \qquad\vdots \\ & \texttt{f}_\texttt{n}\colon \sigma_{n1}, \ldots, \sigma_{nm_n} \Rightarrow \tau_n \\ & \texttt{f}_\texttt{n}\ p_{n11}\ \cdots\ p_{n1m_n} = e_{n1} \\ & \qquad\vdots \\ & \texttt{f}_\texttt{n}\ p_{nl_n1}\ \cdots\ p_{nl_nm_n} = e_{nl_n} \end{array}$$

and defines n functions $\mathtt{f_1}, \ldots, \mathtt{f_n}$ at the same time. Let $\mathcal{Y}_{ji}$ be the set of pattern variables together with their types that occurs in the ith equation of the jth function, for $1 \leqslant j \leqslant n$ and $1 \leqslant i \leqslant l_j$, and let $\mathcal{SF}$ be the set that contains the specifications of the functions $\mathtt{f_1}, \ldots, \mathtt{f_n}$. Then, each right-hand side e_{ji} must be a valid term of type τ_j with respect to $\mathcal{Y}_{ji}$ and $\mathcal{SF}$. Thus, each function $\mathtt{f_j}$ may only occur fully applied on the right-hand side of any of the equations.

The computation rules for $\mathtt{f_1}, \ldots, \mathtt{f_n}$ are defined as before. The difference is that now, we have to find the matching pattern within the definition of the function that we want to compute.

4.4 Conditional equations

In functional programming, conditional equations are often allowed within fixed point equations. That is, equations of the following form are allowed:

$$\mathtt{f}\ p_1\ \cdots\ p_m = \begin{cases} e_1 \text{ if } c_1 \\ \vdots \\ e_r \text{ if } c_r \end{cases}$$

If $\mathcal{Y}$ is the set of pattern variables of the equation together with their types, then the conditions $c_1, \ldots, c_r$ are valid terms of boolean type with respect to $\mathcal{Y}$ and $\{\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau\}$. Hence, they can contain recursive calls to $\mathtt{f}$. Also in this case, we will require that the boolean expressions are exclusive. This is really not a strong restriction since we could define $c'_1 \equiv c_1$ and, for $2 \leqslant s \leqslant r$, $c'_s \equiv c_s \ \wedge\ \neg c_{s-1} \ \wedge \cdots \wedge\ \neg c_1$, and then replace the above equation with a similar one that uses the conditional expressions c' instead, where $\wedge$ and $\neg$ are the boolean operators for conjunction and negation, respectively.

The computation rule associated with a conditional equation consists simply in reducing the application of the function to the branch corresponding to the only condition that evaluates to `true`, if any. Let us have that $\overline{a} \cong \overline{p}[\overline{y := b}]$. If $c_s[\overline{y := b}]$ evaluates to `true`, for some s, then we have the following computation rule

$$\mathtt{f}(a_1, \ldots, a_m) \rightsquigarrow e_s[\overline{y := b}]$$

If none of the conditional expressions evaluates to `true`, then `f` is undefined on that sequence.

We end this section with the observation that a conditional equation can be seen as r equations of the form

$$\begin{aligned} \mathtt{f}\ p_1\ \cdots\ p_m &= e_1 \text{ if } c_1 \\ &\vdots \\ \mathtt{f}\ p_1\ \cdots\ p_m &= e_r \text{ if } c_r \end{aligned}$$

For a sequence of arguments $a_1 \cdots a_m$ matching the pattern $p_1 \cdots p_m$, at most one of the conditions $c_1, \ldots, c_r$ evaluates to `true` and hence, still only one

equation can be applied to compute $\mathtt{f}(a_1, \ldots, a_m)$. To simplify the presentation of our method, in section 6 we consider this latter kind of conditional equations. Although the patterns in the definition of a function might not be exclusive if one writes conditional equations in this way, the fact that still only one equation can be applied for a certain input makes both ways of defining functions with conditional equations semantically equivalent.

5 Turing Completeness

We prove that the class $\mathcal{FP}$ of functional programs allows the definition of all partial recursive functions. This is not immediately clear, because of the restrictions that we have imposed on the recursive definitions. In particular, we do not have a general fixed point operator, that is, given any functional $F\colon (\sigma \to \tau) \to \sigma \to \tau$, we cannot directly define a fixed point for it in our formalism. The tentative definition

$$\begin{aligned} &\mathtt{fix\ f}\colon \sigma \Rightarrow \tau \\ &\quad \mathtt{f}\ a = F\ \mathtt{f}\ a \end{aligned}$$

is not correct, since the function `f` appears in the right-hand side of the above equation without being applied to an argument.

On the other hand, the alternative definition

$$\begin{aligned} &\mathtt{fix\ f}\colon\ \Rightarrow \sigma \to \tau \\ &\quad \mathtt{f} = F\ \mathtt{f} \end{aligned}$$

is a valid recursive definition in our formalism, but it does not define the desired function since, according to the explanation that we give at the end of section 7, it does not actually define anything because the object `f`, which is being defined, has the specification $\mathtt{f}\colon\ \Rightarrow \sigma \to \tau$ and occurs in the right-hand side of its own definition.

Therefore, the fixed point can be defined only when F `f` can be unfolded into an expression where `f` occurs only fully applied.

To show that every recursive function can be defined, we exploit *Kleene normal form theorem* (see for example, Theorem 10.1 in [BM77] or Theorem 1.5.6 in [Phi92]). Below, let $\mathbb{N}$ be the mathematical set of natural numbers and $A \to_\perp B$ the set of partial functions from a set A to a set B. Predicates are total functions into natural numbers that assume only the values 0, meaning false, and 1, meaning true.

Theorem 1. [**Kleene normal form**] There exist primitive recursive predicates $T\colon \mathbb{N}^{n+2} \to \mathbb{N}$ for every natural number n, and a primitive recursive function $U\colon \mathbb{N} \to \mathbb{N}$ such that, for every partial recursive function $f\colon \mathbb{N}^n \to_\perp \mathbb{N}$ there exists a natural number e_f such that

$$f\ \overline{x} = U(\mu y.T(e_f, \overline{x}, y))$$

where μ is the minimisation operator.

Since both T and U are primitive recursive functions, they can be programmed in a functional programming language by structurally recursive algorithms. Therefore, there are programs $\mathtt{T}\colon \mathtt{Nat}^{n+2} \to \mathtt{Bool}$ and $\mathtt{U}\colon \mathtt{Nat} \to \mathtt{Nat}$ in the class $\mathcal{F}$, that we can use inside the recursive definition of f.

In [BC01], we have already used our method to translate the minimisation function in type theory. That formalisation contained a λ-abstraction in the right-hand side. As we show in section 7, the occurrence of λ-abstractions in the right-hand side of equations might cause problems in the translation, so here we give a slightly different formulation that avoids a λ-abstraction in the right-hand side of the definition. Since only elements of $\mathcal{F}$ and recursive calls can occur inside a recursive definition, we cannot directly use the minimisation function in the definition of `f`. Instead, we define a specific minimisation function and `f` within a mutual recursive definition. This minimisation function does not really depend on `f`, but the use of a mutual recursive definition is a trick to be able to use minimisation inside the definition of `f`.

$$\begin{array}{l}\mathtt{mutual\ fix}\ \mathtt{min_f}\colon \mathtt{Nat}^n, \mathtt{Nat} \Rightarrow \mathtt{Nat} \\ \qquad\qquad \mathtt{min_f}\ \overline{x}\ y = \begin{cases} y & \text{if } (\mathtt{T}\ e_f\ \overline{x}\ y) \\ \mathtt{min_f}(\overline{x}, \mathtt{s}(y)) & \text{if } \neg(\mathtt{T}\ e_f\ \overline{x}\ y) \end{cases} \\ \\ \qquad\qquad \mathtt{f}\colon \mathtt{Nat}^n \Rightarrow \mathtt{Nat} \\ \qquad\qquad \mathtt{f}\ \overline{x} = (\mathtt{U}\ \mathtt{min_f}(\overline{x}, 0))\end{array}$$

This definition shows that every function definable by a Kleene normal form can be implemented in our system.

Theorem 2. Every (partial) recursive function is definable in $\mathcal{FP}$.

6 Translation into Type Theory

We give a formal presentation of how to translate a general recursive definition in type theory. The translation applies to the class of functions $\mathcal{FP}$ defined in section 4.

We assume that the user is familiar with constructive type theory and knows how to translate types and expressions from functional programming into their type-theoretic equivalents. All types in functional programming have a corresponding type defined in type theory in the same way, except for the difference in notation. Hence, the values in those types have a type-theoretic equivalent. Structurally recursive functions, that is, the elements of the class $\mathcal{F}$, can also be directly translated in type theory with the corresponding types. If A is a type or an expression in functional programming, we denote its corresponding translation into type theory by $\widehat{A}$.

Let `f` be a general recursive function in $\mathcal{FP}$. Thus,

$$\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau$$

and `f` is defined by a sequence of recursive equations. There are two possible kind of equations, with or without conditionals. They have the following shapes, respectively:

$$\texttt{f}\ p_1\ \cdots\ p_m = e \qquad\qquad \texttt{f}\ p_1\ \cdots\ p_m = e \text{ if } c \qquad (*)$$

Notice that the equation on the left is a special case of the equation on the right, where the condition c is constantly true. For this reason, we only consider conditional equations in the rest of this section.

Let $\widehat{\sigma_1}, \ldots, \widehat{\sigma_m}$, and $\widehat{\tau}$ be the type-theoretic translation of $\sigma_1, \ldots, \sigma_m$, and τ, respectively. To translate `f` into type theory, we define a special-purpose accessibility predicate fAcc and the type-theoretic version of `f`, which we call f and that has the predicate fAcc as part of its domain. These two components have the following types:

$$\begin{array}{l} \mathsf{fAcc} \in (x_1 \in \widehat{\sigma_1}; \ldots; x_m \in \widehat{\sigma_m})\mathsf{Set} \\ \mathsf{f} \in (x_1 \in \widehat{\sigma_1}; \ldots; x_m \in \widehat{\sigma_m}; h \in \mathsf{fAcc}(x_1, \ldots, x_m))\widehat{\tau} \end{array} \qquad (*')$$

The function f is defined by structural recursion on the argument h. Hence, we have one equation in f for each constructor of fAcc.

If the function `f` is defined by nested recursion, we should define fAcc and f simultaneously using Dybjer's schema for simultaneous inductive-recursive definitions [Dyb00]. Otherwise, we first define fAcc and then use that predicate to define f.

Let us start by discussing how to define the predicate fAcc. This predicate has at least one constructor for each of the equations in the definition of the function `f`. The number of constructors associated with each equation depends on the structure of the expressions c and e in the equation. All constructors associated with the equation $(*)$ produce a proof of $\mathsf{fAcc}(\widehat{p_1}, \ldots, \widehat{p_m})$, where $\widehat{p_u}$ is the straightforward translation[3] of p_u, for $1 \leqslant u \leqslant m$. The type of each of the constructors associated with the equation $(*)$ depends on the structure and on the recursive calls that occur in c and in e. The fact that at most one equation can be used for the computation of $\texttt{f}(a_1, \ldots, a_m)$, with $a_u : \sigma_u$, and the way we break down the structure of c and e to establish the type of each constructor, guarantees that at most one constructor can be used to build a proof of $\mathsf{fAcc}(\widehat{a_1}, \ldots, \widehat{a_m})$, where each $\widehat{a_u}$ is the type-theoretic translation of a_u.

Case expressions are the only kind of expressions that might impose the need of several constructors associated with an equation. The reason is that each branch of a case expression needs to be treated separately since it contributes to the type of the corresponding constructor in a different way. Case expressions may occur anywhere within a term; there may be case expressions inside a conditional expression, a $\lambda-$abstraction, a function application, a constructor application, or even inside other case expressions, which implies that any expression might impose the need of several constructors associated with an equation.

[3] As we will see later when we formally define the translation of an expression, patterns are translated in a straightforward way.

However, not all case expressions introduce several constructors. Some case expressions, that we call *safe*, can be directly translated into type theory as case expressions. Safe case expressions are such that none of their branches introduce partiality, hence they can be straightforwardly translated into type theory without further analysis. The expression on which we perform case analysis might still introduce partiality. An example of a safe case expression is the following:

$$\texttt{Cases}\ \texttt{f}(n)\ \texttt{of}\ \begin{cases} \texttt{0} & \mapsto \texttt{0} \\ \texttt{s}(m) & \mapsto \texttt{Cases}\ xs\ \texttt{of}\ \begin{cases} \texttt{nil} & \mapsto \texttt{s(0)} \\ \texttt{cons}(y, ys) & \mapsto n + y \end{cases} \end{cases}$$

where $\texttt{f}\colon \texttt{Nat} \Rightarrow \texttt{Nat}$ is the function being defined, n is a natural number and xs is a list, and m, y and ys are fresh variables of the corresponding types.

First of all, let us explain the general idea of the translation. We associate a series of constructors for fAcc and a corresponding series of equations for f with each equation in the definition of $\texttt{f}$. The most important part of the translation is the definition of the types of the constructors of fAcc. For that purpose, we analyse the structure of the conditional expression c and of the right-hand side e of the equation and, from them, we construct a context of assumptions for each of the different constructors of the predicate associated with that equation.

We start with the context Γ comprising the variables introduced in the pattern $p_1 \cdots p_m$ of the equation. Each variable is assumed with type $\widehat{\sigma}$ if σ is its type in functional programming. Then, we associate context extensions Φ_c and Θ_e with the boolean expression c and with the defining term e, respectively. The extension Φ_c is such that the context $\Gamma; \Phi_c$ contains assumptions sufficient to make the condition c meaningful. Together with Φ_c, we get a translation $\widehat{c}$ of the condition.

This leads us to the final definition of the type for the corresponding constructor of the predicate fAcc:

$$\mathsf{f\_acc}_{c,e} \in (\Gamma; \Phi_c; q \in \widehat{c} = \mathsf{true}; \Theta_e)\mathsf{fAcc}(\widehat{p_1}, \ldots, \widehat{p_m}).$$

where $=$ is the propositional equality in type theory, true is the type-theoretic boolean value $\texttt{true}$ and Θ_e does not depend neither on Φ_c nor on q.

Together with the definition of the context extension Θ_e, we also get a translation $\widehat{e}$ of the term e itself. Then, the equation of f associated with the constructor $\mathsf{f\_acc}_{c,e}$ becomes

$$\mathsf{f}(\widehat{p_1}, \ldots, \widehat{p_m}, \mathsf{f\_acc}_{c,e}(\overline{y}, \overline{x}, q, \overline{z})) = \widehat{e}$$

where $\overline{y}$ is the sequence of variables assumed in Γ, $\overline{x}$ is the sequence of variables assumed in Φ_c, q is a variable of type $(\widehat{c} = \mathsf{true})$ and $\overline{z}$ is the sequence of variables assumed in Θ_e.

A single equation may be associated with several constructors of the special accessibility predicate and, consequently, with several equations of the translated function. So, we associate a sequence $\mathcal{P}_c(\Gamma)$ of pairs $\langle \Phi_c, \widehat{c} \rangle$ of context

extensions Φ_c and type-theoretic boolean terms $\widehat{c}$ with the boolean term c. Similarly, we associate a sequence $\mathcal{P}_e(\Gamma)$ of pairs $\langle \Theta_e, \widehat{e} \rangle$ with the term e. The definition of $\mathcal{P}_{\_}(\Gamma)$ is the same for the conditional expression c and for the term e, so we treat them together in the sequel.

In what follows, given an expression a, we define $\mathcal{P}_a(\Gamma)$ using list comprehension or list enumeration. In addition, $\langle \Phi_a, t_a \rangle$ denotes a generic element of $\mathcal{P}_a(\Gamma)$ and $\#\mathcal{P}_a$ denotes the number of elements in $\mathcal{P}_a(\Gamma)$.

Sometimes a subterm of a contains bound variables with the same name as the variables in Γ. The definition of $\mathcal{P}_a(\Gamma)$ may require that those variables are introduced in the context extension. To avoid naming conflicts we assume, without always explicitly saying it, that those variables are renamed with a fresh name before being introduced in the context. The need to rename the variables becomes clear if we consider the following equation:

$$\mathtt{f}\ x = \mathtt{Cases}\ x\ \mathtt{of} \begin{cases} \mathtt{0} & \mapsto e_{\mathtt{0}} \\ \mathtt{s}(x) & \mapsto e_{\mathtt{s}} \end{cases}$$

This equation presents no problem in functional programming. Any occurrence of the variable x in $e_{\mathtt{s}}$ is bound by the variable x in the pattern $\mathtt{s}(x)$, hence it does not refer to the variable x that occurs in the left-hand side of the equation. Thus, the binding $\mathtt{s}(x)$ shadows the variable x in the left-hand side of the equation inside the expression $e_{\mathtt{s}}$. However, we need to be able to refer to both variables x in type theory. As it will become clear below, we might need to add $(x = \mathtt{s}(x))$ to the assumptions of one of the pairs in the definition of $\mathcal{P}_a(\Gamma)$. In type theory, the two x's in that proposition would refer to the same object. Thus, we need to rename the second occurrence of x to a fresh variable name y of the corresponding type in order to have the proposition $(x = \mathtt{s}(y))$.

We define $\mathcal{P}_a(\Gamma)$ by recursion on the structure of the expression a. If $\langle \Phi_a, t_a \rangle \in \mathcal{P}_a(\Gamma)$, then the term t_a can be seen as the translation $\widehat{a}$ of a under the assumptions in $\Gamma; \Phi_a$. The reader can verify that if σ_a is the type of a in functional programming, then $\widehat{\sigma_a}$ is the type of t_a in the context $\Gamma; \Phi_a$.

$a \equiv z$**:** If the expression a is a variable, then $\mathcal{P}_a(\Gamma) \equiv \{\langle\ , z\rangle\}$.

$e \equiv \mathtt{c}(a_1, \ldots, a_o)$**:** Here, $0 \leqslant o$. First, we determine $\mathcal{P}_{a_1}(\Gamma), \ldots, \mathcal{P}_{a_o}(\Gamma)$ by structural recursion and then we combine these sequences into the definition of $\mathcal{P}_a(\Gamma)$. Formally, if $\mathtt{c} \equiv \widehat{\mathtt{c}}$, we have that

$$\begin{aligned} \mathcal{P}_a(\Gamma) \equiv\ & \{\langle \Phi_{a_1}; \ldots; \Phi_{a_o},\ \mathtt{c}(t_{a_1}, \ldots, t_{a_o}) \rangle \mid \\ & \langle \Phi_{a_1}, t_{a_1} \rangle \in \mathcal{P}_{a_1}(\Gamma), \ldots, \langle \Phi_{a_o}, t_{a_o} \rangle \in \mathcal{P}_{a_o}(\Gamma)\}. \end{aligned}$$

$a \equiv \mathtt{f}(a_1, \ldots, a_m)$**:** Again, we first determine $\mathcal{P}_{a_1}(\Gamma)$, ..., $\mathcal{P}_{a_m}(\Gamma)$ by structural recursion. As before, we combine these sequences into the definition of $\mathcal{P}_a(\Gamma)$. In addition, we have to add the assumption corresponding to the recursive call $\mathtt{f}(a_1, \ldots, a_m)$, stating that the tuple $(\widehat{a_1}, \ldots, \widehat{a_m})$ satisfies the predicate fAcc. Remember that $\mathtt{f} \equiv \widehat{\mathtt{f}}$ and that $\mathtt{f}$ takes an extra parameter,

which is a proof that the input values satisfy the predicate fAcc. Hence, we have that

$$\mathcal{P}_a(\Gamma) \equiv \{\langle \Phi_{a_1}; \ldots; \Phi_{a_m}; h \in \mathsf{fAcc}(t_{a_1}, \ldots, t_{a_m}),\ \mathsf{f}(t_{a_1}, \ldots, t_{a_m}, h)\rangle \mid \langle \Phi_{a_1}, t_{a_1}\rangle \in \mathcal{P}_{a_1}(\Gamma), \ldots, \langle \Phi_{a_m}, t_{a_m}\rangle \in \mathcal{P}_{a_m}(\Gamma)\}.$$

$a \equiv (a_1\ a_2)$**:** This case is treated similarly to the previous two cases.

$$\mathcal{P}_a(\Gamma) \equiv \{\langle \Phi_{a_1}; \Phi_{a_2},\ t_{a_1}(t_{a_2})\rangle \mid \langle \Phi_{a_1}, t_{a_1}\rangle \in \mathcal{P}_{a_1}(\Gamma), \langle \Phi_{a_2}, t_{a_2}\rangle \in \mathcal{P}_{a_2}(\Gamma)\}.$$

$a \equiv [z]b$**:** Let σ be the type of z. We first calculate $\mathcal{P}_b(\Gamma; z \in \widehat{\sigma})$ recursively.

If $\mathcal{P}_b(\Gamma; z \in \widehat{\sigma}) = \{\langle\ , t_b\rangle\}$, that is, if $\mathcal{P}_b(\Gamma; z \in \widehat{\sigma})$ contains only one pair and the context extension in that pair is empty, then

$$\mathcal{P}_b(\Gamma) \equiv \{\langle\ , [z]\, t_b\rangle\}.$$

In other words, in this case, the method does not produce any assumptions, and the λ-abstraction can be directly translated into type theory.

Otherwise, let $\mathcal{P}_b(\Gamma; z \in \widehat{\sigma}) = \{\langle \Phi_{b_1}, t_{b_1}\rangle, \ldots, \langle \Phi_{b_{\#\mathcal{P}_b}}, t_{b_{\#\mathcal{P}_b}}\rangle\}$. To translate this term as a λ-abstraction in type theory, we must impose that the translation $\widehat{b}$ of the abstracted term b is well-defined for every value of the variable z. Therefore, the assumption generated by a must be the universal quantification over z of the all the assumptions for $\widehat{b}$.

Let $\Sigma\Phi$ be the conjunction of all the assumptions in a non-empty context Φ and let $\overline{y_\Phi}$ be the variables in Φ. We define

$$\mathcal{P}_a(\Gamma) \equiv \{\langle H \in (z \in \widehat{\sigma})\ \Sigma\Phi_{b_1} + \ldots + \Sigma\Phi_{b_{\#\mathcal{P}_b}},\ t_a\rangle\}$$

with

$$t_a \equiv [z]\mathsf{Cases}\ H(z)\ \mathsf{of}\ \begin{cases} \mathsf{in}_1(\langle\overline{y_{\Phi_{b_1}}}\rangle) & \mapsto t_{b_1} \\ \vdots & \\ \mathsf{in}_{\#\mathcal{P}_b}(\langle\overline{y_{\Phi_{b_{\#\mathcal{P}_b}}}}\rangle) & \mapsto t_{b_{\#\mathcal{P}_b}} \end{cases}$$

If $\#\mathcal{P}_b = 1$, then we do not need to construct a disjoint union type

$$\mathcal{P}_a(\Gamma) \equiv \{\langle H \in (z \in \widehat{\sigma})\ \Sigma\Phi_b,\ t_a\rangle\}$$

with

$$t_a \equiv [z]\mathsf{Cases}\ H(z)\ \mathsf{of}\ \{\ \langle\overline{y_{\Phi_b}}\rangle \mapsto t_b$$

If, moreover, Φ_b contains only one assumption, then we do not need to construct a Σ-type. We can just call y_{Φ_b} the variable introduced in the assumption in Φ_b, and then

$$\mathcal{P}_b(\Gamma) \equiv \{\langle H \in (z \in \widehat{\sigma})\ \Phi_b,\ t_a\rangle\}$$

with

$$t_a \equiv [z]t_b[y_{\Phi_b} := H(z)]$$

which is actually equivalent to $t_a \equiv [z]\mathsf{Cases}\ H(z)\ \mathsf{of}\ \{\ y_{\Phi_b} \mapsto t_b$.

In the examples, we always use the simplest possible option.

$a \equiv \texttt{Cases}\ b\ \texttt{of}\ \left\{\begin{array}{l} p_1 \mapsto a_1 \\ \quad\vdots \\ p_v \mapsto a_v \end{array}\right.$: Here, $0 \leqslant v$. First, observe that variables, constructors and constructor applications are translated into their type-theoretic equivalents in a straightforward way. Hence, the translation $\widehat{p}$ of a pattern p is also straightforward. In addition, notice that if we compute $\mathcal{P}_p(\Gamma)$ we obtain $\{\langle\ , \widehat{p}\rangle\}$.

If the different branches of the case expression do not contain recursive calls or do not introduce partiality, we can give a straightforward translation. We call such case expressions *safe* and we translate them directly as case expressions in type theory. Formally, a *safe* case expression $\texttt{Cases}\ b\ \texttt{of}\ \left\{\begin{array}{l} p_1 \mapsto a_1 \\ \quad\vdots \\ p_v \mapsto a_v \end{array}\right.$ is such that the patterns $p_1, \ldots, p_v$ are exclusive and exhaustive and $\mathcal{P}_{a_s}(\Gamma) = \{\langle\ , t_{a_s}\rangle\}$, for $0 \leqslant s \leqslant v$. Notice that, for a case expression to be safe, there should be no recursive call in any of the expressions $a_1, \ldots, a_v$.

For a safe case expression we define

$$\mathcal{P}_a(\Gamma) = \{\langle \Phi, t_a\rangle \mid \langle \Phi, t_b\rangle \in \mathcal{P}_b(\Gamma)\}$$

where

$$t_a = \mathsf{Cases}\ t_b\ \mathsf{of}\ \left\{\begin{array}{l} \widehat{p_1} \mapsto t_{a_1} \\ \quad\vdots \\ \widehat{p_v} \mapsto t_{a_v} \end{array}\right.$$

If the case expression is not safe, each of the different branches imposes the need of a different constructor. Let $\overline{y_s}$ be the variables introduced by the pattern p_s and let $\overline{\sigma_s}$ be the types of those variables. Let $\overline{\widehat{y_s}}$ be a renaming of the variables in $\overline{y_s}$ by fresh variable names with respect to Γ. Observe that the renaming of the variables in $\overline{y_s}$ forces the corresponding renaming in $\overline{\sigma_s}$, which will be performed together with the translation of $\overline{\sigma_s}$ into its type-theoretic equivalent $\overline{\widehat{\sigma_s}}$.

Let us denote $\overline{(\widehat{y_s} \in \widehat{\sigma_s})}$ by Γ_s. As we have said before, each branch in the case expression imposes the need of at least one different constructor. Notice that each a_s may impose the need of several constructors, namely $\#\mathcal{P}_{a_s}$. Then, the number of constructors corresponding to the sth branch is also $\#\mathcal{P}_{a_s}$. The constructors associated with the sth branch should assume the variables introduced in the branch, that is, Γ_s. In addition, to ensure that these constructors are used only when we are inside the branch s, they should also assume $q_s \in \widehat{b} = \widehat{p_s}$, where q_s is a fresh variable name for each s. The expression b might also impose the need of several constructors and thus it might contribute to the type of the different constructors. Hence, as before, we should combine the elements

in $\mathcal{P}_b(\Gamma)$ and in $\mathcal{P}_{a_s}(\Gamma;\Gamma_s)$ in all possible ways. Formally, we determine $\mathcal{P}_b(\Gamma), \mathcal{P}_{a_1}(\Gamma;\Gamma_1), \ldots, \mathcal{P}_{a_v}(\Gamma;\Gamma_v)$ by structural recursion and then we define

$$\begin{aligned}\mathcal{P}_a(\Gamma) \equiv \; & \{\langle \Phi_b; \Gamma_s; q_s \in (t_b = \widehat{p_s}); \Phi_{a_s},\ t_{a_s}\rangle \mid \\ & \langle \Phi_b, t_b\rangle \in \mathcal{P}_b(\Gamma), 1 \leq s \leq v, \langle \Phi_{a_s}, t_{a_s}\rangle \in \mathcal{P}_{a_s}(\Gamma;\Gamma_s)\}\end{aligned}$$

This completes the definition of $\mathcal{P}_a(\Gamma)$.

Let us now return to the translation of the function `f` in type theory. To complete the definitions of fAcc and f, whose types where introduced in $(*')$, we need to give the type of the different constructors of the predicate fAcc, and the different equations that define the function f. We recall that the function `f` is defined by (conditional) equations. The shape of each equation that define `f` is given in $(*)$. Let us assume that $\overline{y}$ is the sequence of pattern variables in the equation with types $\overline{\sigma}$. Let Γ be $\overline{(y \in \widehat{\sigma})}$.

Using the definition we presented above, we determine $\mathcal{P}_c(\Gamma)$ and $\mathcal{P}_e(\Gamma)$. Observe that all the terms t_c in the sequence of pairs $\mathcal{P}_c(\Gamma)$ are boolean terms. To define the constructors of fAcc and the equations that define f, we should combine the elements in $\mathcal{P}_c(\Gamma)$ and $\mathcal{P}_e(\Gamma)$ in all possible ways. Let $\langle \Phi_{c_r}, t_{c_r}\rangle$ be the rth element of $\mathcal{P}_c(\Gamma)$ and $\langle \Phi_{e_l}, t_{e_l}\rangle$ the lth element of $\mathcal{P}_e(\Gamma)$. We recall that patterns are straightforwardly translated into type theory. Then, the corresponding constructor of fAcc is as follows:

$$\mathsf{facc}_{rl} \in (\Gamma; \Phi_{c_r}; q_r \in t_{c_r} = \mathsf{true}; \Phi_{e_l})\mathsf{fAcc}(\widehat{p_1}, \ldots, \widehat{p_m})$$

If the corresponding equation is a non-conditional equation, then $\mathcal{P}_c(\Gamma)$ is empty and the assumptions $\Phi_{c_r}; q_r \in t_{c_r} = \mathsf{true}$ are not present in the constructor. The presence or not of these premises is the only difference between the constructors associated with a conditional equation and the constructors associated with a non-conditional equation. Notice also that in the examples we gave in section 3, we converted c into a predicate rather than a boolean function. This is not a problem since we can always define $t'_{c_r} \equiv t_{c_r} = \mathsf{true}$.

The equation in the definition of f that corresponds to the above constructor is the following:

$$\mathsf{f}(\widehat{p_1}, \ldots, \widehat{p_m}, \mathsf{facc}_{rl}(\overline{y}, \overline{x_{\Phi_{c_r}}}, q_r, \overline{z_{\Phi_{e_l}}})) = t_{e_l}$$

where $\overline{x_{\Phi_{c_r}}}$ is the sequence of variables assumed in Φ_{c_r}, q is a variable of type $(\widehat{c} = \mathsf{true})$ and $\overline{z_{\Phi_{e_l}}}$ is the sequence of variables assumed in Φ_{e_l}.

This completes the definition of fAcc and f in type theory.

We have several observations at this point. First, notice that besides the introduction of the pattern variables of an equation, abstraction, recursive calls, non-safe case expressions and conditionals are the expressions that contribute to the type of a constructor. Observe also that if we have two or more syntactically equal recursive calls in an equation, our method will duplicate the assumptions corresponding to that call. This problem can be easily eliminated if we add an assumption corresponding to a recursive call into a sequence of assumptions

only when that assumption has not yet been added to the sequence. This is what we have done in the examples we presented in section 3. The number of constructors associated with an equation is strongly related to the structure of the expressions in the equation, in particular to the case expressions and the $\lambda-$abstractions that are present in the equation. The user should keep this in mind when choosing the definition of a function. Finally, observe that the choice of where to put the symbol $\Rightarrow$ within the specification of the type of a function `f` makes a difference in its translation, since it determines the type of the corresponding fAcc.

Let us now analyse what actually happens when we translate a general recursive algorithm. Hence, let us consider one of the equations that define a general recursive function `f`. For the sake of generality, let us assume that we have nested recursive calls in the equation. For the sake of simplicity, let us assume that the equation is a non-conditional equation with no case expressions in the right-hand side. Finally, let us assume here that there are no recursive calls to `f` inside a $\lambda-$abstraction in the right-hand side of the equation. We consider that case later. Hence, we have an equation of the following form

$$\mathtt{f}\ p_1 \cdots\ p_m = \cdots\ \mathtt{f}(a_1, \ldots, \mathtt{f}(a'_1, \ldots, a'_m), \ldots, a_m) \cdots$$

As usual, let Γ comprise the pattern variables together with their types. Let us call the right-hand side of the above equation $e_{\mathtt{f}}$. As there are no case expressions in the equation, for any subexpression a of $e_{\mathtt{f}}$, $\mathcal{P}_a$ has only one element. In order to calculate $\mathcal{P}_{e_{\mathtt{f}}}(\Gamma)$, we first have to calculate $\mathcal{P}_{\mathtt{f}(a'_1,\ldots,a'_m)}(\Gamma)$

$$\mathcal{P}_{\mathtt{f}(a'_1,\ldots,a'_m)}(\Gamma) \equiv \{\langle \Phi_{a'_1}; \ldots; \Phi_{a'_m}; h \in \mathsf{fAcc}(\widehat{a'_1}, \ldots, \widehat{a'_m}),\ \mathsf{f}(\widehat{a'_1}, \ldots, \widehat{a'_m}, h)\rangle\}$$

Hence, $\mathtt{f}(\widehat{a'_1, \ldots, a'_m})$ is defined as $\mathsf{f}(\widehat{a'_1}, \ldots, \widehat{a'_m}, h)$. Now, we calculate

$$\begin{aligned}\mathcal{P}_{\mathtt{f}(a_1,\ldots,\mathtt{f}(a'_1,\ldots,a'_m),\ldots,a_m)}(\Gamma) \equiv \\ \{\ \langle \Phi_{a_1}; \ldots; \Phi_{a'_1}; \ldots; \Phi_{a'_m}; h \in \mathsf{fAcc}(\widehat{a'_1}, \ldots, \widehat{a'_m}); \ldots; \Phi_{a_m};\\ h' \in \mathsf{fAcc}(\widehat{a_1}, \ldots, \mathsf{f}(\widehat{a'_1}, \ldots, \widehat{a'_m}, h), \ldots, \widehat{a_m}),\\ \mathsf{f}(\widehat{a_1}, \ldots, \mathsf{f}(\widehat{a'_1}, \ldots, \widehat{a'_m}, h), \ldots, \widehat{a_m}, h')\ \rangle\ \}\end{aligned}$$

Thus, the translation of $\mathtt{f}(a_1, \ldots, \mathtt{f}(a'_1, \ldots, a'_m), \ldots, a_m)$ is defined as the term $\mathsf{f}(\widehat{a_1}, \ldots, \mathsf{f}(\widehat{a'_1}, \ldots, \widehat{a'_m}, h), \ldots, \widehat{a_m}, h')$.

The constructor associated with the equation is

$$\begin{aligned}\mathsf{facc} \in\ & (\Gamma; \ldots; \Phi_{a_1}; \ldots; \Phi_{a'_1}; \ldots; \Phi_{a'_m}; h \in \mathsf{fAcc}(\widehat{a'_1}, \ldots, \widehat{a'_m}); \ldots; \Phi_{a_m};\\ & h' \in \mathsf{fAcc}(\widehat{a_1}, \ldots, \mathsf{f}(\widehat{a'_1}, \ldots, \widehat{a'_m}, h), \ldots, \widehat{a_m}); \ldots\\ &)\mathsf{fAcc}(\widehat{p_1}, \ldots, \widehat{p_m})\end{aligned}$$

and the corresponding equation in the definition of f would be

$$\begin{aligned}\mathsf{f}(\widehat{p_1}, \ldots, \widehat{p_m}, \mathsf{facc}(\ldots, h, \ldots, h', \ldots)) = \\ \cdots\ \mathsf{f}(\widehat{a_1}, \ldots, \mathsf{f}(\widehat{a'_1}, \ldots, \widehat{a'_m}, h), \ldots, \widehat{a_m}, h') \cdots\end{aligned}$$

Observe that the recursive calls to the function f are structurally smaller on the proof that the corresponding values satisfy the predicate fAcc. Notice that we have the same property even if there are no nested recursive calls.

Now, let us consider an equation with a $\lambda-$abstraction in the right-hand side and a recursive call to $\mathtt{f}$ inside the $\lambda-$abstraction. We have then an equation of the form

$$\mathtt{f}\ p_1 \cdots\ p_m = \cdots [z](\cdots \mathtt{f}(a_1, \ldots, a_m) \cdots) \cdots$$

Let Γ be as usual and let us call e_λ the expression $(\cdots \mathtt{f}(a_1, \ldots, a_m) \cdots)$. Here, we first need to compute $\mathcal{P}_{[z]e_\lambda}(\Gamma)$. If σ is the type of z, we have that

$$\mathcal{P}_{[z]e_\lambda}(\Gamma) \equiv \{\langle H \in (z \in \widehat{\sigma})\ \Sigma\Phi_{e_{\lambda 1}} + \ldots + \Sigma\Phi_{e_{\lambda \#\mathcal{P}_{e_\lambda}}},\ \widehat{[z]e_\lambda}\rangle\}$$

where

$$\widehat{[z]e_\lambda} \equiv [z]\ \mathsf{Cases}\ H(z)\ \mathsf{of} \begin{cases} \mathsf{in}_1(\langle \overline{y_{\Phi_{e_{\lambda 1}}}} \rangle) & \mapsto t_{e_{\lambda 1}} \\ \vdots & \\ \mathsf{in}_s(\langle \ldots, h, \ldots \rangle) & \mapsto \cdots \mathsf{f}(\widehat{a_1}, \ldots, \widehat{a_m}, h) \cdots \\ \vdots & \\ \mathsf{in}_{\#\mathcal{P}_{e_\lambda}}(\langle \overline{y_{\Phi_{e_{\lambda \#\mathcal{P}_{e_\lambda}}}}} \rangle) & \mapsto t_{e_{\lambda \#\mathcal{P}_{e_\lambda}}} \end{cases}$$

Here, $1 \leqslant s \leqslant \#\mathcal{P}_{e_\lambda}$, $h \in \mathsf{fAcc}(\widehat{a_1}, \ldots, \widehat{a_m})$ and $t_{e_{\lambda s}}$ is the translation of the part of e_λ where the recursive call actually occurs. Thus, the context extension $\Phi_{e_{\lambda s}}$ should contain the assumption $\mathsf{fAcc}(\widehat{a_1}, \ldots, \widehat{a_m})$. There can, of course, be recursive calls in any of the others $t_{e_{\lambda k}}$, with $1 \leqslant k \leqslant \#\mathcal{P}_{e_\lambda}$.

The constructor associated with the equation is

$$\mathsf{facc} \in (\Gamma; \ldots; H \in (z \in \widehat{\sigma})\ \Sigma\Phi_{e_{\lambda 1}} + \ldots + \Sigma\Phi_{e_{\lambda \#\mathcal{P}_{e_\lambda}}}; \ldots)\mathsf{fAcc}(\widehat{p_1}, \ldots, \widehat{p_m})$$

and the corresponding equation in the definition of f would be

$$\mathsf{f}(\widehat{p_1}, \ldots, \widehat{p_m}, \mathsf{facc}(\ldots, H, \ldots)) = \cdots\ \widehat{[z]e_\lambda} \cdots$$

Although it is less obvious here, the recursive calls to the function f are, also in this case, structurally smaller on the proof that the corresponding values satisfy the predicate fAcc. To convince ourselves of this, let us analyse the term $\widehat{[z]e_\lambda}$. Observe that all the pattern variables in $\overline{y_{\Phi_{e_{\lambda k}}}}$ are structurally smaller than $H(z)$. In addition, the term $H(z)$ is considered structurally smaller than the term H. Hence, the variable h in the sth branch of the case expression is a term structurally smaller than H.

If, instead of a single function, we face the mutual definition of n functions

$$\begin{array}{c} \mathtt{mutual\ fix}\ \mathsf{f_1}\colon \sigma_{11}, \ldots, \sigma_{1m_1} \Rightarrow \tau_1 \\ \vdots \\ \mathsf{f_n}\colon \sigma_{n1}, \ldots, \sigma_{nm_n} \Rightarrow \tau_n \\ \vdots \end{array}$$

then, we need to define n special-purpose accessibility predicates and n type-theoretic functions with the following types:

$$\begin{array}{l}
\mathsf{fAcc_1} \in (x_{11} \in \widehat{\sigma_{11}}; \ldots; x_{1m_1} \in \widehat{\sigma_{1m_1}})\mathsf{Set} \\
\qquad\qquad\qquad\qquad\vdots \\
\mathsf{fAcc_n} \in (x_{n1} \in \widehat{\sigma_{n1}}; \ldots; x_{nm_n} \in \widehat{\sigma_{nm_n}})\mathsf{Set} \\
\\
\mathsf{f_1} \in (x_{11} \in \widehat{\sigma_{11}}; \ldots; x_{1m_1} \in \widehat{\sigma_{1m_1}}; h_1 \in \mathsf{fAcc}(x_{11}, \ldots, x_{1m_1}))\widehat{\tau_1} \\
\qquad\qquad\qquad\qquad\vdots \\
\mathsf{f_n} \in (x_{n1} \in \widehat{\sigma_{n1}}; \ldots; x_{nm_n} \in \widehat{\sigma_{nm_n}}; h_n \in \mathsf{fAcc}(x_{n1}, \ldots, x_{nm_n}))\widehat{\tau_n}
\end{array}$$

Similarly to what happens in the translation of a single function definition, if a function $\mathtt{f_j}$ is defined by nested recursion, for $1 \leqslant j \leqslant n$, we should define the fAcc's and the f's simultaneously. In order to do so, we need the generalisation of Dybjer's schema presented in [Bov02]. Otherwise, we first define the fAcc's and we then use those predicates to define the f's.

Each special-purpose accessibility predicate $\mathsf{fAcc_j}$ and each function $\mathsf{f_j}$ is defined as for a single function. Observe that now, the case in the definition of $\mathcal{P}_a(\Gamma)$ that deals with recursive calls should consider the recursive calls to any of the n functions. Each recursive call is translated as in the definition of $\mathcal{P}_a(\Gamma)$.

7 Lazy, Strict and Totally Strict Semantics

We must be careful to state in which sense our type-theoretic translation of a functional program is equivalent to the original one. Given a program f in $\mathcal{FP}$, our general method produces a pair consisting of a predicate fAcc and a function f that takes a proof that the input values satisfy the predicate as extra argument. For example, if f has the specification $\sigma \Rightarrow \tau$, we obtain $\mathsf{fAcc} \in (\widehat{\sigma})\mathsf{Set}$ and $\mathsf{f} \in (x \in \widehat{\sigma}; h \in \mathsf{fAcc}(x))\widehat{\tau}$ in type theory, where $\widehat{\sigma}$ and $\widehat{\tau}$ are the type-theoretic translation of σ and τ, respectively. Then, we would like to state a soundness and completeness conjecture as follows:

> The program f terminates on the input t if and only if $\mathsf{fAcc}(\widehat{t})$ is provable. Moreover, if $h \in \mathsf{fAcc}(\widehat{t})$ then the output produced by the computation of $\mathtt{f}(\widehat{t}, h)$ is $\widehat{\mathtt{f}(t)}$.

Unfortunately, this conjecture is not always true neither for *lazy* nor for *strict* computational models. For this conjecture to be valid we need a *totally strict* semantics, by which we mean a strict semantics that requires the functions that occur in the right-hand side to be total. However, this is not a real computational model for functional programming and hence, we prefer to weaken the conjecture and instead state only the soundness of our method.

> Given an input t for f, if $h \in \mathsf{fAcc}(\widehat{t})$ then the output produced by the computation of $\mathtt{f}(\widehat{t}, h)$ is $\widehat{\mathtt{f}(t)}$.

Let us first explain why a soundness and completeness conjecture would only be valid in a totally strict semantics.

When evaluating an expression, a lazy computational model evaluates only the part of the expression that is necessary for the computation to continue. For example, in the expression $\mathtt{f}(n) \leqslant 0$ we might not need to fully evaluate $\mathtt{f}(n)$. If, at a certain stage, the computation produces the value $\mathtt{s}(e)$, where e is still an unevaluated expression, there is no need to further evaluate e to produce a result for $\mathtt{f}(n) \leqslant 0$, since we already know that the value of this expression must be `false`.

Similarly, in the definition of a recursive function, when we have a recursive equation of the form

$$\mathtt{f}\ \overline{p} = \cdots \mathtt{f}(\overline{a}) \cdots$$

the lazy evaluation strategy requires that $\mathtt{f}(\overline{a})$ is computed only if (and when) it is needed. Moreover, $\mathtt{f}(\overline{a})$ does not necessarily need to be fully evaluated since it might be enough to just partially evaluate it.

On the other hand, a *strict* evaluation strategy requires that the arguments of a function are always fully evaluated before the function is computed. Hence, in the example above, $\mathtt{f}(\overline{a})$ must be computed before the computation of $\mathtt{f}(\overline{p})$ begins, even if the value of $\mathtt{f}(\overline{a})$ may not actually be needed for the final result.

Our translation of a functional program in type theory corresponds to a strict evaluation strategy. In the definition of fAcc, to prove $\mathsf{fAcc}(\overline{p})$ we first need to prove $\mathsf{fAcc}(\overline{a})$.

The distinction between lazy and strict evaluation strategy is relevant not only to the efficiency of computation, but also to the question of termination since a lazy program may terminate while the corresponding strict program diverges. Suppose the computation of $\mathtt{f}(\overline{a})$ diverges and that its value is not actually needed for the computation of $\mathtt{f}(\overline{p})$. Then, a lazy evaluation strategy would just ignore the call $\mathtt{f}(\overline{a})$ and produce a result anyway, while a strict evaluation strategy would try to compute $\mathtt{f}(\overline{a})$ and therefore diverge.

A good illustration of the difference between lazy and strict evaluation is the following mutually recursive definition, which is a variant of the one given in section 3.

$$\begin{aligned}
&\mathtt{mutual\ fix\ f}\colon \mathtt{Nat} \Rightarrow \mathtt{Nat}\\
&\qquad\qquad \mathtt{f}\ 0 = 0\\
&\qquad\qquad \mathtt{f}\ \mathtt{s}(n) = \mathtt{f}(\mathtt{g}(n)) + \mathtt{g}(n)\\
&\\
&\qquad\qquad \mathtt{g}\colon \mathtt{Nat} \Rightarrow \mathtt{Nat}\\
&\qquad\qquad \mathtt{g}\ 0 = 0\\
&\qquad\qquad \mathtt{g}\ \mathtt{s}(n) = \begin{cases} \mathtt{g}(\mathtt{f}(n)) + n & \text{if } \mathtt{f}(n) \leqslant n\\ 0 & \text{if } \mathtt{f}(n) > n \end{cases}
\end{aligned}$$

Here is a table with the values of `f` and `g` for a few initial inputs in a lazy computational model with the right definitions of + and >:

input value x	$\mathtt{f}(x)$	$\mathtt{g}(x)$
0	0	0
1	0	0
2	0	1
3	1	2
4	2	3
5	4	5
6	9	8
7	8	0

input value x	$\mathtt{f}(x)$	$\mathtt{g}(x)$
8	0	0
9	0	8
10	8	9
11	9	10
12	18	19
13	`undefined`	0
14	0	0
15	0	14

First of all, let us consider the computation of $\mathtt{f}(7)$. From the definition of $\mathtt{f}$ and $\mathtt{g}$, we have that $\mathtt{f}(7) = \mathtt{f}(\mathtt{g}(6)) + \mathtt{g}(6) = \mathtt{f}(8) + 8$. This shows that $\mathtt{f}$ and $\mathtt{g}$ as above are not defined by primitive recursion since $\mathtt{f}(7)$ calls itself on a larger argument. Therefore, this definition is a good candidate to be translated with our method.

As it is shown in the table, the computation of $\mathtt{f}(13)$ diverges, independently of what evaluation strategy we adopt. However, if we partially evaluate it, we obtain

$$\mathtt{f}(13) = \mathtt{f}(\mathtt{g}(12)) + \mathtt{g}(12) = \mathtt{f}(19) + 19$$

Now, when computing $\mathtt{g}(14)$, we have to decide which of the two branches of the definition of $\mathtt{g}$ we should take depending on whether $\mathtt{f}(13) \leqslant 13$ or $\mathtt{f}(13) > 13$. A strict evaluation strategy would, at this point, try to fully evaluate $\mathtt{f}(13)$ and diverge. On the other hand, in a lazy evaluation strategy, with the right definitions of $+$ and $>$, one step of the computation of $\mathtt{f}(13)$ would be enough to determine that the value of

$$\mathtt{f}(13) = \mathtt{f}(19) + 19 > 13$$

is `true`. Therefore, the second branch in the definition of $\mathtt{g}$ would be taken and the desired result would be $\mathtt{g}(14) = 0$.

Using our method, we obtain the following two predicates and two functions.

$$\begin{array}{l}
\mathsf{fAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{f_acc}_0 \in \mathsf{fAcc}(0) \\
\quad \mathsf{f_acc}_\mathsf{s} \in (n \in \mathsf{N}; h_1 \in \mathsf{gAcc}(n); h_2 \in \mathsf{fAcc}(\mathsf{g}(n, h_1)))\mathsf{fAcc}(\mathsf{s}(n)) \\
\\
\mathsf{gAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{g_acc}_0 \in \mathsf{gAcc}(0) \\
\quad \mathsf{g_acc}_{\mathsf{s}_1} \in (n \in \mathsf{N}; h_1 \in \mathsf{fAcc}(n); q \in (\mathsf{f}(n, h_1) \leqslant n); h_2 \in \mathsf{gAcc}(\mathsf{f}(n, h_1)) \\
\qquad\qquad)\mathsf{gAcc}(\mathsf{s}(n)) \\
\quad \mathsf{g_acc}_{\mathsf{s}_2} \in (n \in \mathsf{N}; h_1 \in \mathsf{fAcc}(n); q \in (\mathsf{f}(n, h_1) > n))\mathsf{gAcc}(\mathsf{s}(n)) \\
\\
\mathsf{f} \in (m \in \mathsf{N}; \mathsf{fAcc}(m))\mathsf{N} \\
\quad \mathsf{f}(0, \mathsf{f_acc}_0) = 0 \\
\quad \mathsf{f}(\mathsf{s}(n), \mathsf{f_acc}_\mathsf{s}(n, h_1, h_2)) = \mathsf{f}(\mathsf{g}(n, h_1), h_2) + \mathsf{g}(n, h_1)
\end{array}$$

$$
\begin{array}{l}
\mathsf{g} \in (m \in \mathsf{N}; \mathsf{gAcc}(m))\mathsf{N} \\
\quad \mathsf{g}(0, \mathsf{g\_acc}_0) = 0 \\
\quad \mathsf{g}(\mathsf{s}(n), \mathsf{g\_acc}_{\mathsf{s}1}(n, h_1, q, h_2)) = \mathsf{g}(\mathsf{f}(n, h_1), h_2) + n \\
\quad \mathsf{g}(\mathsf{s}(n), \mathsf{g\_acc}_{\mathsf{s}2}(n, h_1, q)) = 0
\end{array}
$$

Then, to compute g on 14, we must first prove gAcc(14). If such a proof exists, it must be constructed using either $\mathsf{g\_acc}_{\mathsf{s}1}$ or $\mathsf{g\_acc}_{\mathsf{s}2}$. Both constructors require a proof $h_1 \in \mathsf{fAcc}(13)$. But fAcc(13) cannot be proved, since f(13) diverges. Thus, gAcc(14) cannot be proved either and g is not defined on 14.

Therefore, our method corresponds to a strict semantics for functional programs.

However, even with strict evaluation, an additional problem may arise if a λ-abstraction occurs in the right-hand side of one of the equations in the definition of a recursive program.

A strict semantics requires that, for an expression to be defined, all its subexpressions should be defined. This means that if the definition of a function f contains an equation of the form

$$\mathtt{f}\ p_1\ \cdots\ p_m = e$$

to evaluate f on arguments that instantiate the pattern $p_1 \cdots p_m$, we must strictly evaluate e. In other words, if the computation of any subexpression of e diverges, then the computation of e also diverges, independently of whether the value of that subexpression is needed for the computation of e or not. In terms of definedness, we require that all the subexpressions of e are defined for e to be defined.

Let us suppose that e contains a λ-abstraction, thus e is of the form

$$e = \cdots [x]e' \cdots$$

According to a strict interpretation, the subexpression $[x]e'$ must be defined for e to be defined. Notice however that $[x]e'$ denotes a function, and that functions are defined if their values are defined on every input. In other words, $[x]e'$ is defined if e' is defined for every value of x. This would amount to requiring that, whenever a λ-abstraction occurs in the right-hand side of an equation, the corresponding function must be total. Since the totality of recursive functions is in general undecidable, a semantics that requires the functions in the right-hand side to be total is not a real computational model for functional programming. This is why even in functional programming languages with strict semantics, a higher type term is considered computed whenever it has been reduced to a λ-abstraction form, and not when the corresponding function is total.

In our method, we need to translate e into type theory to be able to translate the equation above. Since there are no partially defined terms in type theory, all subexpressions of e must be translated into totally defined terms.

Let us consider the following example:

$$\texttt{Inductive Lift } a ::= \texttt{lift } a$$

$$\begin{array}{l}\texttt{f}: \texttt{Nat} \Rightarrow \texttt{Nat}\\ \texttt{f } n = \texttt{Cases lift } [x]\texttt{f}(x) \texttt{ of } \{ \texttt{ lift } y \mapsto 0\end{array}$$

With a strict semantics, the function `f` computes the value 0 for any input. However, things are very different in type theory. If we define the predicate fAcc using our method, its only constructor has the following type:

$$\begin{array}{l}\mathsf{fAcc} \in (\mathsf{N})\mathsf{Set}\\ \quad \mathsf{facc} \in (n \in \mathsf{N}; H \in (x \in \mathsf{N})\mathsf{fAcc}(x))\mathsf{fAcc}(n)\end{array}$$

That is, we need to prove $\mathsf{fAcc}(x)$ for all x's in order to construct a proof of $\mathsf{fAcc}(n)$. Obviously, this makes it impossible to prove fAcc for any input and thus, in type theory, we are not able to compute $\mathsf{f}(n)$ for any natural number n. Hence, the soundness and completeness conjecture also fails for strict semantics.

With a totally strict semantics, the function `f` defined above diverges for any input and hence, the soundness and completeness conjecture would be true. However, as we have already mentioned, totally strict semantics is not a real computational model of functional programming. Thus, we prefer to make our statement weaker and only require the soundness of our method.

Another example where a λ-abstraction in the right-hand side of the definition of a function causes problems is the following:

$$\begin{array}{l}\texttt{fix fdel}: \texttt{List Nat} \Rightarrow \texttt{Bool}\\ \quad \texttt{fdel nil} = \texttt{true}\\ \quad \texttt{fdel cons}(n,l) = \texttt{and}(\texttt{map } [x]\texttt{fdel}(\texttt{cons}(n, \texttt{delete } x\ l))\ l)\end{array}$$

Both the function `and` and the function `delete` are structurally recursive functions. The former returns the conjunction of the boolean elements in its argument list and the latter deletes the first occurrence of its first argument in its second argument, if there exists such an occurrence.

In the second equation, we have a λ-abstraction as the functional argument of the function `map`. A recursive call to the function `fdel` occurs in the scope of this abstraction. The lists on which we perform the recursive calls are of the form $\texttt{cons}(n, \texttt{delete } x\ l)$ and they are all smaller than the original list $\texttt{cons}(n,l)$ because the list $\texttt{delete } x\ l$ is smaller than l whenever x is an element in l. Hence, the function `fdel` always terminates. When we apply our method to this example, we obtain

$$\begin{array}{l}\mathsf{fdelAcc} \in (\mathsf{List}(\mathsf{N}))\mathsf{Set}\\ \quad \mathsf{fdel\_acc}_{\mathsf{nil}} \in \mathsf{fdelAcc}(\mathsf{nil})\\ \quad \mathsf{fdel\_acc}_{\mathsf{cons}} \in (n \in \mathsf{N}; l \in \mathsf{List}(\mathsf{N}); H \in (x \in \mathsf{N})\mathsf{fdelAcc}(\mathsf{cons}(n, \mathsf{delete}(x,l)))\\ \qquad\qquad)\mathsf{fdelAcc}(\mathsf{cons}(n,l))\end{array}$$

$$\begin{array}{l}\mathsf{fdel} \in (l' \in \mathsf{List(N)}; \mathsf{fdelAcc}(l'))\mathsf{Bool} \\ \quad \mathsf{fdel}(\mathsf{nil}, \mathsf{fdel\_acc}_{\mathsf{nil}}) = \mathsf{true} \\ \quad \mathsf{fdel}(\mathsf{cons}(n, l), \mathsf{fdel\_acc}_{\mathsf{cons}}(n, l, H)) = \\ \qquad\qquad \mathsf{and}(\mathsf{map}([x]\mathsf{fdel}(\mathsf{cons}(n, \mathsf{delete}(x, l)), H(x)), l))\end{array}$$

To compute fdel on the list $\mathsf{cons}(n, l)$ we have to prove $\mathsf{fdelAcc}(\mathsf{cons}(n, l))$. For this, we need to give a proof

$$H \in (x \in \mathsf{N})\mathsf{fdelAcc}(\mathsf{cons}(n, \mathsf{delete}(x, l)))$$

and therefore, we must prove $\mathsf{fdelAcc}(\mathsf{cons}(n, \mathsf{delete}(x, l)))$ for every x. However, for those x's that do not belong to the list l such a proof is not possible to construct.

An analysis of the algorithm shows that the assumption H is unnecessarily strong. It requires the function

$$[x]\mathtt{fdel}(\mathtt{cons}(n, \mathtt{delete}\ x\ l))$$

to be defined everywhere. However, in practice, since the function above is given as the functional argument of `map` and since the second argument of `map` is l, we just need the function to be defined on the elements of l.

Our translation does not analyse the behaviour of the occurrences of other functions in the definition. Thus, it does not try to determine what `map` does with its arguments. Instead, it considers the worst case scenario, that is, `map` could use its arguments in any possible way. Therefore, the translation requires that the function argument is defined for every value. It is possible to modify the definition of the function `fdel` to force the interpretation to look into the definition of `map` by defining `fdel` by mutual recursion with a specialised version of `map`.

$$\begin{array}{l}\mathtt{mutual\ fix\ fdel\_map}\colon \mathtt{Nat}, \mathtt{List\ Nat}, \mathtt{List\ Nat} \Rightarrow \mathtt{List\ Bool} \\ \qquad\qquad \mathtt{fdel\_map}\ m\ l_1\ \mathtt{nil} = \mathtt{nil} \\ \qquad\qquad \mathtt{fdel\_map}\ m\ l_1\ \mathtt{cons}(n, l_2) = \\ \qquad\qquad\quad \mathtt{cons}(\mathtt{fdel}(\mathtt{cons}(m, \mathtt{delete}\ n\ l_1)), \mathtt{fdel\_map}(m, l_1, l_2)) \\ \\ \qquad\qquad \mathtt{fdel}\colon \mathtt{List\ Nat} \Rightarrow \mathtt{Bool} \\ \qquad\qquad \mathtt{fdel\ nil} = \mathtt{true} \\ \qquad\qquad \mathtt{fdel\ cons}(n, l) = \mathtt{and}(\mathtt{fdel\_map}(n, l, l))\end{array}$$

The reader can verify that when we apply the translation for mutually recursive functions to this example, we get a much better condition for the termination of `fdel`. In the second equation of the function fdel, we require a proof of $\mathsf{fdel\_mapAcc}(n, l, l)$, which is equivalent to $\mathsf{fdelAcc}(\mathsf{cons}(n, \mathsf{delete}(x, l)))$ for all elements x in l, but not for every natural number x.

From these examples it becomes clear that it is a good programming style, in terms of our type-theoretic translation, to avoid λ-abstractions inside the definition of a recursive function in $\mathcal{FP}$. The user should instead try to replace

every λ-abstraction with a new function mutually defined with the original one. If the function we want to formalise has the specification

$$\mathtt{fix}\,\mathsf{f}\colon \sigma \Rightarrow \tau_1 \to \tau_2$$

an easier solution might be to define it as

$$\mathtt{fix}\,\mathsf{f}\colon \sigma, \tau_1 \Rightarrow \tau_2$$

In this way, the fAcc predicate contains an unnecessary argument but we avoid the need of a λ-abstraction in the right-hand side of the equations.

A limit case occurs when the sequence of types to the left of the symbol $\Rightarrow$ is empty, that is, when defining a function $\mathtt{f}\colon\ \Rightarrow \tau$. Notice that if $\mathtt{f}$ occurs in the right-hand side of the (necessarily unique) equation in its definition, the only constructor of the predicate fAcc is of the form:

$$\begin{array}{l}\mathsf{fAcc} \in \mathsf{Set} \\ \quad \mathsf{facc} \in (\cdots \mathsf{fAcc} \cdots)\mathsf{fAcc}\end{array}$$

Here, we need a proof of fAcc in order to prove fAcc. In conclusion, such a definition of $\mathtt{f}$ does not actually define anything.

The fact that lazy semantics is not an appropriate semantics for $\mathcal{FP}$ goes beyond the impossibility of proving a completeness theorem for $\mathcal{FP}$ with such semantics. In a lazy computational model we can define data types and functions whose type-theoretic translations behave in a completely different way.

Let us consider the following data type definition:

$$\mathtt{Inductive\ Stream\ ::=\ scons\ Nat\ Stream}$$

In a lazy computational model `Stream` defines the type of infinite lists of natural numbers. In general, lazy computational models allow the definition of data types with infinite objects and of recursive functions over those data types that might terminate for some inputs. An example of the latter is the function that returns the position of the first natural number smaller than 100, if any, in an infinite list.

$$\begin{array}{l}\mathtt{f}\colon \mathtt{Stream} \Rightarrow \mathtt{Nat} \\ \mathtt{f}\ \mathtt{scons}(n,l) = \begin{cases} 1 & \text{if } n < 100 \\ 1 + \mathtt{f}\ l & \text{if } n \geqslant 100 \end{cases}\end{array}$$

This function terminates if the input list has at least one element that is smaller than 100 and it diverges otherwise. Observe that, following the description we gave in section 3.1, the function $\mathtt{f}$ as above is structurally recursive.

However, in a strict semantics, the type `Stream` defines an empty data type and the function $\mathtt{f}$ defines a function that always terminates simply because there is no input to which we can apply the function.

In a similar way, the type-theoretic version of `Stream` also defines an empty set. In addition, $\mathtt{f}$ can be directly translated into type theory as a total function

f over an empty domain and hence, f will be a terminating function that can actually never be applied to any input.

We can now state that the translation of the functions in $\mathcal{FP}$ into type theory is sound with respect to a strict semantics.

Theorem 3. (*Soundness*) Let $\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau$ be a function in $\mathcal{FP}$. Let fAcc and f be the special accessibility predicate for $\mathtt{f}$ and the type-theoretic version of $\mathtt{f}$, respectively. For every sequence of arguments $t_1 \in \sigma_1, \ldots, t_m \in \sigma_m$ we have that if $h \in \mathsf{fAcc}(\widehat{t_1}, \ldots, \widehat{t_m})$, then $\mathtt{f}(t_1, \ldots, t_m)$ terminates and

$$\widehat{\mathtt{f}(t_1, \ldots, t_m)} = \mathsf{f}(\widehat{t_1}, \ldots, \widehat{t_m}, h)$$

We do not present a formal proof here but the ideas behind it.

First, we recall that the types in our functional language $\mathcal{FP}$ and the elements in those types have their equivalents in type theory, which we denote by using the symbol $\widehat{\ }$.

Then, notice that both $\mathcal{FP}$ and the subpart of type theory with no dependent types have the same computation rules.

Finally, given the input $(t_1, \ldots, t_m)$, the value of h, which is a canonical element in $\mathsf{fAcc}(\widehat{t_1}, \ldots, \widehat{t_m})$, is a trace of the computation of $\mathtt{f}(t_1, \ldots, t_m)$. Moreover, our method preserves the structure of the right-hand sides of the equations if there are no unsafe case expressions in them. If there are unsafe case expressions in a particular equation, our method produces several constructors associated (to the right-hand side of) that equation. However, the type of each constructor ensures that the constructor is only applicable when the corresponding branch in the case expression is used for the computation of $\mathtt{f}(t_1, \ldots, t_m)$.

Theorem 4. (*Completeness*) If there are no λ-abstractions in the right-hand side of the equations defining the function $\mathtt{f}$ then

$$\mathsf{fAcc}(\widehat{t_1}, \ldots, \widehat{t_m}) \text{ is provable} \iff \mathtt{f}(t_1, \ldots, t_m) \text{ terminates}$$

The implication from left to right is obvious since it can be deduced from the above theorem.

For the implication from right to left, recall from our previous discussion that λ-abstractions in the right-hand side of the equations defining a function were the source of problems. Examples like the function `fdel` show that the function itself can terminate in $\mathcal{FP}$ (with a non-totally semantics) but we are not able to prove its special accessibility predicate and hence, to compute it in type theory.

8 Conclusions

We described a method to translate a vast class of algorithms from functional programming into type theory. We defined the class $\mathcal{FP}$ of algorithms to which

the method applies. This class is large enough to allow the implementation of all partial recursive functions. We gave a formal definition of the translation of the elements of $\mathcal{FP}$ into type theory. In addition, we proved that the translation is sound with respect to a strict semantics.

As a final remark we would like to point out that given a function $\mathtt{f}$ in $\mathcal{FP}$ and a particular input $\overline{a}$ for the function, the canonical proof of the special predicate for that input, that is, the canonical proof of $\mathsf{fAcc}(\overline{\overline{a}})$, is a trace of the computation of $\mathtt{f}(\overline{a})$. Therefore, the structural complexity of the canonical proofs in fAcc are proportional to the number of steps in the algorithm.

Future work will try to improve the translation of lambda abstractions, that we find unsatisfactory at the moment. We hope that improving this part of the translation might allow us to also prove a completeness theorem. We also plan to develop the idea briefly explained at the end of section 3 and use impredicative type theory to formalise the method using an explicit type constructor for partial functions.

8.1 Related Work

There are few studies on formalising general recursion in type theory.

One can adopt a set-theoretic approach and see functions as relations. Specifically, the behaviour of a recursive function can be described by an inductive relation giving its operational semantics (see, for example, [Win93]). Operational semantics has been developed in type theory in [BCB02] and [ZMHH02]. However, relations do not have any computational content in type theory. The real challenge consists in representing general recursive programs as elements of some functional type.

In [Nor88], Nordström uses the predicate Acc for that purpose.

Using classical logic it is possible to extend every partial function to a total one. This fact is used by Finn, Fourman, and Longley [FFL97] to give a formalisation of partial recursive functions inside a classical axiomatic logic system. Their implementation associates a domain predicate to each function, in a similar way to our approach. However, the predicate is not used to define the function but just to restrict the scope of the recursive equations. Except for the translation of case expressions, for which they take the conjunction of all the assumptions that arise in the different branches of the case expression instead of considering each branch in a separate way as we do, an algorithm is analysed in [FFL97] as it is in our method. Moreover, Finn et al give a similar interpretation when bound variables are present in the right-hand side of the equations and they arrive to similar conclusions about the semantics associated to the formalisation of their programs. In addition, they have similar problems when using higher order functions in the definition of other functions. However, the different settings in which the two works are performed give rise to some differences. A function $\mathtt{f}$ is formalised in [FFL97] with the type that it has in a functional programming language, without the need of our extra parameter fAcc as part of the type of f. However, a function f obeys its definition in [FFL97] provided that its arguments can be proved to be in the domain of the function, which is

called $\mathsf{DOM}'\mathsf{f}$. Once (and if) the function has been proved total, one can forget about $\mathsf{DOM}'\mathsf{f}$, which is not possible in type theory. Another important difference is that in [FFL97], an application ($\mathsf{f}\ e$) is always considered defined since the cases in which ($\mathtt{f}\ e$) is not defined are considered as returning an unspecified value of the corresponding type. However, this causes some problems since the semantics they use is not capable of reflecting this distinction and sometimes one can prove things like $\mathsf{f}\ 0 = 0$ when $\mathtt{f}\ 0$ diverges.

Wiedijk and Zwanenburg [WZ02] show how standard classical first order logic can be used to reason about partial functions in type theory. They prove an equivalence between a system in which function application requires a proof term certifying that the argument is in the domain, like in our case, and a simple first order language without proof terms in which every function is total.

In [DDG98], Dubois and Viguié Donzeau-Gouge take also a similar approach to the problem. They also formalise an algorithm with a predicate that characterises the domain of the algorithm and the formalisation of the algorithm itself. However, they consider neither case expressions nor $\lambda-$abstractions as possible expressions, which simplifies the translation a lot. In addition, they only present the translation for expressions in canonical form which also helps in the simplification. The most important difference is their use of post-conditions. In order to be able to deal with nested recursion without the need of simultaneous inductive-recursive definitions, they require that, together with the algorithm, the user provides a post-condition that characterises the results of the algorithm.

Balaa and Bertot [BB00] use fix-point equations to obtain the desired equalities for the recursive definitions. The solution they present is rather complex and it does not really succeed in separating the actual algorithms and their termination proofs. In a later work [BB02], Balaa and Bertot use fix-points again to approach the problem. Their new solution produces nicer formalisations and although one has to provide proofs concerning the well-foundedness of the recursive calls when one defines the algorithms, there is a clear separation between the algorithms and these proofs. In any case, it is not very clear how their methods can be used to formalise partial or nested recursive algorithms.

In a recent work, Bertot et al [BCB02] present a technique to encode the method we describe in [BC01] for partial and nested algorithms in type theories that do not support Dybjer's schema for simultaneous inductive-recursive definitions. They do so by combining the way we define our special-accessibility predicate with the functionals in [BB02].

Some work has been done for simply typed λ-calculus with inductive types where the termination of recursive functions is ensured by types. Barthe et al [BFG$^+$00] present a type-based system $\lambda^\frown$ that ensures the termination of recursive functions through the notion of stage, which is used to restrict the arguments of recursive calls. In [BFG$^+$00], the system $\lambda^\frown$ is proved to be strongly normalising and to enjoy the property of subject reduction. Although this system seems a good candidate to be used in proof-assistants based on type theory, some work should still be done before this can be actually carried out in practice, namely, scale $\lambda^\frown$ up to dependent types and develop type checking and type inference algorithms for the system.

In the same line but with some differences in the type system, we can find the work by Abel [Abe02]. The core language in [Abe02] and the properties of the system presented there are basically the same as in [BFG+00]. In addition, in [Abe02], a type checking algorithms is given for the system. The key concept in this work is the use of decorated type variables, which play a similar role to stages in [BFG+00]. As well as with the work by Barthe et al, the system should be scaled up to dependent types before it can be used in proof-assistants for type theory.

The primary idea behind the work by McBride and McKinna [MM02] is similar to ours: to take the best of functional languages and of type theory in order to actually be able to nicely program in type theory. They do so by introducing a new notation for functional programming on top of the existing dependent type theory. Although all the examples they present in [MM02] are of structurally recursive functions, there is reason to believe that their work and ours can be put together. Our special accessibility predicate gives, in their sense, the *view* of a function's domain which allows the machine to check the recursive calls of the function without further notational overhead. Putting the two works together is actually a very interesting further work.

Acknowledgement. We want to thank Yves Bertot and Björn von Sydow for carefully reading and commenting on previous versions of this paper. We also want to thank Jörgen Gustavsson for fruitful discussions on the semantics of $\mathcal{FP}$.

References

[Abe02] A. Abel. Termination checking with types - Strong normalization for Mendler-style course-of-value recursion. Technical Report 0201, Institut für Informatik, Ludwig-Maximilians - Universität München, 2002.

[Acz77] P. Aczel. An Introduction to Inductive Definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland Publishing Company, 1977.

[Bar92] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 117–309. Oxford University Press, 1992.

[BB00] A. Balaa and Y. Bertot. Fix-point equations for well-founded recursion in type theory. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2000.

[BB02] A. Balaa and Y. Bertot. Fonctions récursives générales par itération en théorie des types. *Journées Francophones des Langages Applicatifs - JFLA02, INRIA*, January 2002.

[BC01] A. Bove and V. Capretta. Nested general recursion and partiality in type theory. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science, Springer-Verlag*, pages 121–135, September 2001.

[BCB02] Y. Bertot, V. Capretta, and K. Das Barman. Type-theoretic functional semantics. In *Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002*, 2002.

[BFG+00] G. Barthe, M.J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. Under consideration for publication in Math. Struct. in Comp. Science, December 2000.

[BG01] H. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 18, pages 1149–1238. Elsevier Science Publishers, 2001.

[BM77] J. L. Bell and M. Machover. *A course in mathematical logic*. North-Holland, 1977.

[Bov99] A. Bove. Programming in Martin-Löf type theory: Unification - A non-trivial example, November 1999. Licentiate Thesis of the Department of Computer Science, Chalmers University of Technology. Available on the WWW `http://cs.chalmers.se/~bove/Papers/lic_thesis.ps.gz`.

[Bov01] A. Bove. Simple general recursion in type theory. *Nordic Journal of Computing*, 8(1):22–42, Spring 2001.

[Bov02] A. Bove. Mutual general recursion in type theory, May 2002. Available on the WWW `http://cs.chalmers.se/~bove/Papers/mutual_rec.ps.gz`.

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

[CNSvS94] T. Coquand, B. Nordström, J. M. Smith, and B. von Sydow. Type theory and programming. *EATCS*, 52, February 1994.

[DDG98] C. Dubois and V. Viguié Donzeau-Gouge. A step towards the mechanization of partial functions: Domains as inductive predicates. In M. Kerber, editor, *CADE-15, The 15th International Conference on Automated Deduction*, pages 53–62, July 1998. WORKSHOP Mechanization of Partial Functions.

[dMJB+01] P. de Mast, J.-M. Jansen, D. Bruin, J. Fokker, P. Koopman, S. Smetsers, M. van Eekelen, and R. Plasmeijer. *Functional Programming in Clean.* Computing Science Institute, University of Nijmegen, 2001.

[Dyb00] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.

[FFL97] S. Finn, M.P. Fourman, and J. Longley. Partial functions in a total setting. *Journal of Automated Reasoning*, 18(1):85–104, 1997.

[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.

[JHe+99] S. Peyton Jones, J. Hughes, (editors), L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[MM70] Z. Manna and J. McCarthy. Properties of programs and partial function logic. *Machine Intelligence*, 5:27–37, 1970.

[MM02] C. McBride and J. McKinna. The view from the left, 2002. Under consideration for publication in Journal of Functional Programming.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML.* MIT Press, 1997.

[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.

[Phi92] J. C. C. Phillips. *Recursion Theory*, pages 79–187. Oxford University Press, 1992.

[SU98] M. H. B. Sørensen and P. Urzyczyn. Lectures on the curry-howard isomorphism. Available as DIKU Rapport 98/14, 1998.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages. An Introduction.* The MIT Press, 1993.

[WZ02] Freek Wiedijk and Jan Zwanenburg. First order logic with domain conditions. unpublised, available at `http://www.cs.kun.nl/~freek/notes/index.html`, 2002.

[ZMHH02] Xingyuan Zhang, Malcolm Munro, Mark Harman, and Lin Hu. Weakest precondition for general recursive programs formalized in Coq. In V. A Carreno, C. A. Munoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002*, volume 2410 of *Lecture Notes in Computer Science*, pages 332–347. Springer-Verlag, 2002.

Paper V

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

Programming in Martin-Löf Type Theory

Unification

A non-trivial Example

Ana Bove
e-mail: bove@cs.chalmers.se

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

Göteborg, 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.

Contents

1 Introduction **129**
1.1 Overview of this Work 134

2 Introduction to Martin-Löf's Type Theory and ALF **135**
2.1 Brief Introduction to Martin-Löf's Type Theory 135
2.2 Brief Introduction to ALF 136
2.3 Working with ALF 138
2.3.1 Logical Constants 139
2.3.2 Some Useful General Predicates 140
2.3.3 Some Useful General Data Types 142

3 A Small Example **143**
3.1 The Haskell Version of the Algorithm 143
3.2 Using the Standard Accessibility Predicate for the Formalisation 144
3.3 Using a Special-Purpose Accessibility Predicate for the Formalisation 146
3.4 Towards Program Extraction 148

4 The Unification Algorithm **149**
4.1 The Haskell Version of the Unification Algorithm 149
4.2 Termination of the Unification Algorithm 153
4.3 Terms, Lists of Pairs of Terms and Substitutions in ALF 155
4.4 The Unification Algorithm in Type Theory: First Attempt . . . 157
4.4.1 The Unification Algorithm using the Accessibility Predicate157
4.4.2 Problems of this Formalisation 158
4.5 The Unification Algorithm in Type Theory: Second Attempt . . 160
4.5.1 The UniAcc Predicate 160
4.5.2 The Unification Algorithm using the UniAcc Predicate . . 163
4.6 Towards Program Extraction 165

5 More about Substitutions **171**
5.1 Application of Substitutions 171
5.2 Idempotent Substitutions 172
5.3 Most General Unifier 173

5.4 Some Properties involving Substitutions 174

6 Partial Correctness of the Unification Algorithm 179
6.1 About the Result of the Unification Algorithm 179
6.2 Variables Property 181
6.3 Idempotence Property 183
6.4 Most General Unifier Property 183

7 The Integrated Approach 187

8 Conclusions 191
8.1 Related Work 193
8.2 Future Work 197

A ALF Formalisation of the Inequalities over Lists of Pairs of Terms 199

B All Lists of Pairs of Terms Satisfy UniAcc **203**

Bibliography 207

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".

This papers was originally printed as a thesis for the degree of Licentiate of Technology of the Department of Computing Science at Chalmers University of Technology. Here, we have omitted the appendix with the complete ALF code of the formalisation of the unification algorithm in Martin-Löf's type theory. The reader who is interested in the code of the formalisation should refer to appendix C of [Bov99].

1.1 Overview of this Work

This work 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.

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

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 of [Bov99] 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 of [Bov99] for the complete ALF code.

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

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

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

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 of [Bov99] for the complete ALF code.

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

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

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

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

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

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 of [Bov99] 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 of [Bov99] 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 of [Bov99] 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 of [Bov99] 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 of [Bov99] 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 of [Bov99] 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 of [Bov99] 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 of [Bov99] 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.

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

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.

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

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 of [Bov99] 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 of [Bov99]. 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 of [Bov99] 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 of [Bov99] 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 of [Bov99] 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

$\#\mathsf{eqs}_{\mathrm{LPT}}((x,\ f(lt)) : lp) < \#\mathsf{eqs}_{\mathrm{LPT}}((f(lt),\ x) : lp)$. 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 $\#\mathsf{vars}_{\mathrm{LPT}}((\mathsf{zip}\ \ lt_1\ lt_2) ++ lp) = \#\mathsf{vars}_{\mathrm{LPT}}((f(lt_1),\ f(lt_2)) : lp)$. We also know that $\#\mathsf{funs}_{\mathrm{LPT}}((\mathsf{zip}\ \ lt_1\ lt_2) ++ lp) < \#\mathsf{funs}_{\mathrm{LPT}}((f(lt_1),\ f(lt_2)) : lp)$ 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
    []v ∈ Vector(0, A)
    :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                          Fun ∈ Set
    Var ≡ N                            Fun ≡ N
Vardec ∈ (x, y ∈ Var) Dec(=(x, y))  Fundec ∈ (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.

```
Varto A ∈ (a, b ∈ A; Dec(=(x, y))) A
    Varto A(a, b, yes(h1)) ≡ a
    Varto A(a, b, no(h1)) ≡ 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 of [Bov99] 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 of [Bov99] 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:

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

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

$\mathrm{unify}_{\mathrm{acc}} \in (lp \in \mathrm{ListPT};\ sb \in \mathrm{Subst};\ \mathrm{Acc}(\mathrm{ListPT}, <_{\mathrm{LPT}}, lp)) \vee (\mathrm{Subst}, \mathrm{Error})$
$\mathrm{unify}_{\mathrm{acc}}([], sb, \mathrm{acc}(\_, h_1)) \equiv \vee_{\mathrm{L}}(sb)$
$\mathrm{unify}_{\mathrm{acc}}(:(lp_1, .(\mathrm{var}(x), \mathrm{var}(x_1))), sb, \mathrm{acc}(\_, h_1)) \equiv$
case $\mathrm{Var}_{\mathrm{dec}}(x, x_1) \in \mathrm{Dec}(=(x, x_1))$ **of**
$\mathrm{yes}(\mathrm{refl}(\_)) \Rightarrow ?_{\mathrm{unify\_acc.1.0.E}}$
$\mathrm{no}(h) \Rightarrow ?_{\mathrm{unify\_acc.1.1.E}}$
end
$\mathrm{unify}_{\mathrm{acc}}(:(lp_1, .(\mathrm{var}(x), \mathrm{fun}(f, lt))), sb, \mathrm{acc}(\_, h_1)) \equiv$
case $\in_{\mathrm{dec}}(x, \mathrm{vars}_{\mathrm{T}}(\mathrm{fun}(f, lt))) \in \mathrm{Dec}(\in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{T}}(\mathrm{fun}(f, lt))))$ **of**
$\mathrm{yes}(h) \Rightarrow \vee_{\mathrm{R}}(\mathrm{error})$
$\mathrm{no}(h) \Rightarrow ?_{\mathrm{unify\_acc.2.1.E}}$
end
$\mathrm{unify}_{\mathrm{acc}}(:(lp_1, .(\mathrm{fun}(f, lt), \mathrm{var}(x))), sb, \mathrm{acc}(\_, h_1)) \equiv ?_{\mathrm{unify\_acc.3.E}}$
$\mathrm{unify}_{\mathrm{acc}}(:(lp_1, .(\mathrm{fun}(f_1, lt_1), \mathrm{fun}(f_2, lt_2))), sb, \mathrm{acc}(\_, h_1)) \equiv$
case $\mathrm{Fun}_{\mathrm{dec}}(f_1, f_2) \in \mathrm{Dec}(=(f_1, f_2))$ **of**
$\mathrm{yes}(\mathrm{refl}(\_)) \Rightarrow$ **case** $\mathrm{N}_{\mathrm{dec}}(n_1, n_2) \in \mathrm{Dec}(=(n_1, n_2))$ **of**
$\mathrm{yes}(\mathrm{refl}(\_)) \Rightarrow ?_{\mathrm{unify\_acc.4.1.0.E}}$
$\mathrm{no}(h_2) \Rightarrow \vee_{\mathrm{R}}(\mathrm{error})$
end
$\mathrm{no}(h) \Rightarrow \vee_{\mathrm{R}}(\mathrm{error})$
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,

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

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

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

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' 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 $x \in \mathsf{vars}_{\mathrm{T}}(t)$ 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 $x \notin_{\mathrm{L}} \mathsf{vars}_{\mathrm{T}}(t)$, 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 $a \neq b$

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, varsT(t));
                       ¬(=(var(x), t))
                      ) UniAcc(:(lp, .(var(x), t)))
    uniacc:=_var_term ∈ (∉L(x, varsT(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

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 function as the ALF function zip (see section C.4.3 of [Bov99] 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.

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

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:

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

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

$$\text{unify} \in (lp \in \text{ListPT};\ sb \in \text{Subst};\ \text{UniAcc}(lp)) \vee (\text{Subst}, \text{Error})$$
$$\text{unify}(\_, sb, \text{uniacc}_{[]}) \equiv \vee_{\text{L}}(sb)$$
$$\text{unify}(\_, sb, \text{uniacc}_{\text{var\_var}}(x, h_1)) \equiv \text{unify}(lp_1, sb, h_1)$$
$$\text{unify}(\_, sb, \text{uniacc}_{\text{var\_term}}(lp_1, h_1, h_2)) \equiv \vee_{\text{R}}(\text{error})$$
$$\text{unify}(\_, sb, \text{uniacc}_{:=\text{var\_term}}(h_1, h_2)) \equiv \text{unify}(:=_{\text{LPT}}(x, t, lp_1), :(:=_{\text{S}}(x, t, sb), .(x, t)), h_2)$$
$$\text{unify}(\_, sb, \text{uniacc}_{\text{var\_fun}}(h_1)) \equiv \text{unify}(:(lp_1, .(\text{var}(x), \text{fun}(f, lt))), sb, h_1)$$
$$\text{unify}(\_, sb, \text{uniacc}_{\text{fun\_fun}}(lt_1, lt_2, lp_1, h_1)) \equiv \vee_{\text{R}}(\text{error})$$
$$\text{unify}(\_, sb, \text{uniacc}_{\text{zip\_fun\_fun}}(f, h_1)) \equiv \text{unify}(++(\text{zip}(lt_1, lt_2), lp_1), sb, h_1)$$

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

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 of [Bov99]). 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 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:

$$\text{unify} \in (lp \in \text{ListPT};\ sb \in \text{Subst};\ \text{UniAcc}(lp)) \vee (\text{Subst}, \text{Error})$$
$$\text{unify}(\_, sb, \text{uniacc}_{[]}) \equiv \vee_{\text{L}}(sb)$$
$$\text{unify}(\_, sb, \text{uniacc}_{\text{var\_var}}(x, h_1)) \equiv \text{unify}(lp_1, ?_{\text{sb1}}, h_1)$$
$$\text{unify}(\_, sb, \text{uniacc}_{\text{var\_term}}(lp_1, h_1, h_2)) \equiv \vee_{\text{R}}(\text{error})$$
$$\text{unify}(\_, sb, \text{uniacc}_{:=\text{var\_term}}(h_1, h_2)) \equiv \text{unify}(:=_{\text{LPT}}(x, t, lp_1), ?_{\text{sb2}}, h_2)$$
$$\text{unify}(\_, sb, \text{uniacc}_{\text{var\_fun}}(h_1)) \equiv \text{unify}(:(lp_1, .(\text{var}(x), \text{fun}(f, lt))), ?_{\text{sb3}}, h_1)$$
$$\text{unify}(\_, sb, \text{uniacc}_{\text{fun\_fun}}(lt_1, lt_2, lp_1, h_1)) \equiv \vee_{\text{R}}(\text{error})$$
$$\text{unify}(\_, sb, \text{uniacc}_{\text{zip\_fun\_fun}}(f, h_1)) \equiv \text{unify}(++(\text{zip}(lt_1, lt_2), lp_1), ?_{\text{sb4}}, h_1)$$

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 $:(lp, .(\mathsf{var}(x), t))$, $:=_{\mathrm{LPT}}(x, t, lp)$, $a = b$ or $\in_{\mathrm{L}}(x, \mathsf{vars}_{\mathrm{T}}(t))$ 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:

$\mathrm{zip2} \in (lt_1 \in \mathrm{VTerm}(n_1);\ lt_2 \in \mathrm{VTerm}(n_2))\ \mathrm{ListPT}$
$\quad \mathrm{zip2}([]_v,\ lt_2) \equiv []$
$\quad \mathrm{zip2}(:_v(lt'_1,\ t_1),\ []_v) \equiv []$
$\quad \mathrm{zip2}(:_v(lt'_1,\ t_1),\ :_v(lt'_2,\ t_2)) \equiv :(\mathrm{zip2}(lt'_1,\ lt'_2),\ .(t_1,\ t_2))$

$\mathrm{UniAcc2} \in (lp \in \mathrm{ListPT})\ \mathbf{Set}$
$\quad \mathrm{uniacc2}_{[]} \in \mathrm{UniAcc2}([])$
$\quad \mathrm{uniacc2}_{\mathrm{var\_var}} \in (=(x,\ y);$
$\qquad\qquad \mathrm{UniAcc2}(lp)$
$\qquad\qquad) \mathrm{UniAcc2}(:(lp,\ .(\mathrm{var}(x),\ \mathrm{var}(y))))$
$\quad \mathrm{uniacc2}_{\mathrm{var\_term}} \in (lp \in \mathrm{ListPT};$
$\qquad\qquad \in_{\mathrm{L}}(x,\ \mathrm{vars}_{\mathrm{T}}(t));$
$\qquad\qquad \neg(=(\mathrm{var}(x),\ t))$
$\qquad\qquad) \mathrm{UniAcc2}(:(lp,\ .(\mathrm{var}(x),\ t)))$
$\quad \mathrm{uniacc2}_{:=\mathrm{var\_term}} \in (\notin_{\mathrm{L}}(x,\ \mathrm{vars}_{\mathrm{T}}(t));$
$\qquad\qquad \mathrm{UniAcc2}(:=_{\mathrm{LPT}}(x,\ t,\ lp))$
$\qquad\qquad) \mathrm{UniAcc2}(:(lp,\ .(\mathrm{var}(x),\ t)))$
$\quad \mathrm{uniacc2}_{\mathrm{var\_fun}} \in (\mathrm{UniAcc2}(:(lp,\ .(\mathrm{var}(x),\ \mathrm{fun}(f,\ lt))))$
$\qquad\qquad) \mathrm{UniAcc2}(:(lp,\ .(\mathrm{fun}(f,\ lt),\ \mathrm{var}(x))))$
$\quad \mathrm{uniacc2}_{\mathrm{fun\_fun}} \in (lt_1 \in \mathrm{VTerm}(n_1);$
$\qquad\qquad lt_2 \in \mathrm{VTerm}(n_2);$
$\qquad\qquad lp \in \mathrm{ListPT};$
$\qquad\qquad \vee(\neg(=(f,\ g)),\ \neg(=(n_1,\ n_2)))$
$\qquad\qquad) \mathrm{UniAcc2}(:(lp,\ .(\mathrm{fun}(f,\ lt_1),\ \mathrm{fun}(g,\ lt_2))))$
$\quad \mathrm{uniacc2}_{\mathrm{zip\_fun\_fun}} \in (=(n_1,\ n_2);$
$\qquad\qquad =(f,\ g);$
$\qquad\qquad \mathrm{UniAcc2}(++(\mathrm{zip2}(lt_1,\ lt_2),\ lp))$
$\qquad\qquad) \mathrm{UniAcc2}(:(lp,\ .(\mathrm{fun}(f,\ lt_1),\ \mathrm{fun}(g,\ lt_2))))$

$\mathrm{unify2} \in (lp \in \mathrm{ListPT};\ sb \in \mathrm{Subst};\ \mathrm{UniAcc2}(lp)) \vee(\mathrm{Subst},\ \mathrm{Error})$
$\quad \mathrm{unify2}(\_,\ sb,\ \mathrm{uniacc2}_{[]}) \equiv \vee_{\mathrm{L}}(sb)$
$\quad \mathrm{unify2}(\_,\ sb,\ \mathrm{uniacc2}_{\mathrm{var\_var}}(c_1,\ h)) \equiv \mathrm{unify2}(lp_1,\ sb,\ h)$
$\quad \mathrm{unify2}(\_,\ sb,\ \mathrm{uniacc2}_{\mathrm{var\_term}}(lp_1,\ c_1,\ c_2)) \equiv \vee_{\mathrm{R}}(\mathrm{error})$
$\quad \mathrm{unify2}(\_,\ sb,\ \mathrm{uniacc2}_{:=\mathrm{var\_term}}(c_1,\ h)) \equiv \mathrm{unify2}(:=_{\mathrm{LPT}}(x,\ t,\ lp_1),\ :(:=_{\mathrm{S}}(x,\ t,\ sb),\ .(x,\ t)),\ h)$
$\quad \mathrm{unify2}(\_,\ sb,\ \mathrm{uniacc2}_{\mathrm{var\_fun}}(h)) \equiv \mathrm{unify2}(:(lp_1,\ .(\mathrm{var}(x),\ \mathrm{fun}(f,\ lt))),\ sb,\ h)$
$\quad \mathrm{unify2}(\_,\ sb,\ \mathrm{uniacc2}_{\mathrm{fun\_fun}}(lt_1,\ lt_2,\ lp_1,\ c_1)) \equiv \vee_{\mathrm{R}}(\mathrm{error})$
$\quad \mathrm{unify2}(\_,\ sb,\ \mathrm{uniacc2}_{\mathrm{zip\_fun\_fun}}(c_1,\ c_2,\ h)) \equiv \mathrm{unify2}(++(\mathrm{zip2}(lt_1,\ lt_2),\ lp_1),\ sb,\ h)$

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

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

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 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:

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

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

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

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:

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

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:

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

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:

$\mathrm{appP_T} \in (sb \in \mathrm{Subst};\ t \in \mathrm{Term})\ \mathrm{Term}$
$\mathrm{appP_T}([], \mathrm{var}(x)) \equiv \mathrm{var}(x)$
$\mathrm{appP_T}(:(sb_1, .(x_1, t)), \mathrm{var}(x)) \equiv \mathrm{Var_{to}A}(t, \mathrm{appP_T}(sb_1, \mathrm{var}(x)), \mathrm{Var_{dec}}(x_1, x))$
$\mathrm{appP_T}(sb, \mathrm{fun}(f, lt)) \equiv \mathrm{fun}(f, \mathrm{appP_{VT}}(sb, lt))$
$\mathrm{appP_{VT}} \in (sb \in \mathrm{Subst};\ lt \in \mathrm{VTerm}(n))\ \mathrm{VTerm}(n)$
$\mathrm{appP_{VT}}(sb, []_v) \equiv []_v$
$\mathrm{appP_{VT}}(sb, :_v(lt', t)) \equiv :_v(\mathrm{appP_{VT}}(sb, lt'), \mathrm{appP_T}(sb, t))$

where the function $\mathrm{Var_{to}A}$ 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:

appS$_T$ ∈ (sb ∈ Subst; t ∈ Term) Term
appS$_T$([], t) ≡ t
appS$_T$(:(sb_1, .(x, t_1)), t) ≡ 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:

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))

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:

Idempotent ∈ (sb ∈ Subst) **Set**
Idempotent ≡ [sb]∀(Term, [t]=(appP$_T$(sb, appP$_T$(sb, t)), 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(vars_T(t), dom(:(sb, .(x, t))));
             ∉_L(x, vars_S(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:

```
unifies_LPT ∈ (sb ∈ Subst; lp ∈ ListPT) Set
    unifies_LPT_[] ∈ (sb ∈ Subst) unifies_LPT(sb, [])
    unifies_LPT_: ∈ (unifies_LPT(sb, lp');
                     =(appP_T(sb, t1), appP_T(sb, t2))
                    ) unifies_LPT(sb, :(lp', .(t1, t2)))
```

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

```
unifies_S ∈ (sb1, sb2 ∈ Subst) Set
    unifies_S_[] ∈ (sb ∈ Subst) unifies_S(sb, [])
    unifies_S_: ∈ (unifies_S(sb, sb1);
                   =(appP_T(sb, var(x)), appP_T(sb, t))
                  ) unifies_S(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 of [Bov99] 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 of [Bov99] for the complete ALF proofs of these lemmas.

$$\in_\wedge\neq_\wedge\mathrm{unify}_{\mathrm{to}}\bot\ \in\ (t \in \mathrm{Term};\ \in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{T}}(t));\ \neg(=(\mathrm{var}(x), t));\ =(\mathrm{appP}_{\mathrm{T}}(sb, \mathrm{var}(x)), \mathrm{appP}_{\mathrm{T}}(sb, t)))\ \bot$$

$$\in_\wedge\neq_\wedge\mathrm{unify}_{\mathrm{to}}\bot(\mathrm{var}(x_1), \in_{\mathrm{hd}}(\_, \_), \Rightarrow_{\mathrm{I}}(f_1), h_2)\ \equiv\ f_1(\mathrm{refl}(\mathrm{var}(x_1)))$$

$$\in_\wedge\neq_\wedge\mathrm{unify}_{\mathrm{to}}\bot(\mathrm{var}(x_1), \in_{\mathrm{tl}}(\_, \_, h_3), h_1, h_2)\ \equiv\ \textbf{case}\ h_3 \in \in_{\mathrm{L}}(x, [])\ \textbf{of}\ \textbf{end}$$

$$\in_\wedge\neq_\wedge\mathrm{unify}_{\mathrm{to}}\bot(\mathrm{fun}(f, lt), h, h_1, h_2)\ \equiv\ <\wedge=_{\mathrm{to}}\bot(\leq_{\mathrm{to}}<_{\mathrm{sL}}(\in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}}(sb, lt, h)), =_{\mathrm{congl}}(\#\mathrm{funs}_{\mathrm{T}}, h_2))$$

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:

$$\in_{\mathrm{to}}\leq_{\#\mathrm{funsT}}\ \in\ (sb \in \mathrm{Subst};\ t \in \mathrm{Term};\ \in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{T}}(t))) \leq (\#\mathrm{funs}_{\mathrm{T}}(\mathrm{appP}_{\mathrm{T}}(sb, \mathrm{var}(x))), \#\mathrm{funs}_{\mathrm{T}}(\mathrm{appP}_{\mathrm{T}}(sb, t)))$$

$$\in_{\mathrm{to}}\leq_{\#\mathrm{funsT}}(sb, \mathrm{var}(x_1), \in_{\mathrm{hd}}(\_, \_))\ \equiv\ \vee_{\mathrm{R}}(\mathrm{refl}(\#\mathrm{funs}_{\mathrm{T}}(\mathrm{appP}_{\mathrm{T}}(sb, \mathrm{var}(x_1)))))$$

$$\in_{\mathrm{to}}\leq_{\#\mathrm{funsT}}(sb, \mathrm{var}(x_1), \in_{\mathrm{tl}}(\_, \_, h_1))\ \equiv\ \textbf{case}\ h_1 \in \in_{\mathrm{L}}(x, [])\ \textbf{of}\ \textbf{end}$$

$$\in_{\mathrm{to}}\leq_{\#\mathrm{funsT}}(sb, \mathrm{fun}(f, lt), h)\ \equiv\ \leq_{\mathrm{to}}\leq_{+\mathrm{R}}(\mathrm{s}(0), \in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}}(sb, lt, h))$$

$$\in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}}\ \in\ (sb \in \mathrm{Subst};\ lt \in \mathrm{VTerm}(n);\ \in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{VT}}(lt))) \leq (\#\mathrm{funs}_{\mathrm{T}}(\mathrm{appP}_{\mathrm{T}}(sb, \mathrm{var}(x))), \#\mathrm{funs}_{\mathrm{VT}}(\mathrm{appP}_{\mathrm{VT}}(sb, lt)))$$

$$\in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}}(sb, []_{\mathrm{v}}, h)\ \equiv\ \textbf{case}\ h \in \in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{VT}}([]_{\mathrm{v}}))\ \textbf{of}\ \textbf{end}$$

$$\in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}}(sb, :_{\mathrm{v}}(lt', t'), h)\ \equiv$$

$$\textbf{case}\ \in+\!\!+_{\mathrm{to}}\in\vee(\mathrm{vars}_{\mathrm{T}}(t'), h) \in \vee(\in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{VT}}(lt')), \in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{T}}(t')))\ \textbf{of}$$

$$\vee_{\mathrm{L}}(h_1)\ \Rightarrow\ \leq_{\mathrm{to}}\leq_{+\mathrm{R}}(\#\mathrm{funs}_{\mathrm{T}}(\mathrm{appP}_{\mathrm{T}}(sb, t')), \in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}}(sb, lt', h_1))$$

$$\vee_{\mathrm{R}}(h_2)\ \Rightarrow$$

$$=_{\mathrm{subst1}}(+_{\mathrm{comm}}(\#\mathrm{funs}_{\mathrm{T}}(\mathrm{appP}_{\mathrm{T}}(sb, t')), \#\mathrm{funs}_{\mathrm{VT}}(\mathrm{appP}_{\mathrm{VT}}(sb, lt'))),$$

$$\leq_{\mathrm{to}}\leq_{+\mathrm{R}}(\#\mathrm{funs}_{\mathrm{VT}}(\mathrm{appP}_{\mathrm{VT}}(sb, lt')), \in_{\mathrm{to}}\leq_{\#\mathrm{funsT}}(sb, t', h_2)))$$

$$\textbf{end}$$

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 of [Bov99] 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:

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

To prove these properties we use the following lemma:

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

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.

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

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 of [Bov99] 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 of [Bov99] 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 of [Bov99] 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 of [Bov99]. 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 of [Bov99].

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.

$$
\begin{aligned}
&\mathrm{vars}_{\mathrm{lemma}} \in (p \in \mathrm{UniAcc}(lp);\\
&\qquad =(\mathrm{unify}(sb, p), \vee_{\mathrm{L}}(sb'));\\
&\qquad \subseteq(\mathrm{vars}_{\mathrm{LPT}}(lp), l);\\
&\qquad \subseteq(\mathrm{vars}_{\mathrm{S}}(sb), l)\\
&\qquad) \subseteq(\mathrm{vars}_{\mathrm{S}}(sb'), l)\\
&\quad \mathrm{vars}_{\mathrm{lemma}}(\mathrm{uniacc}_{[]}, \mathrm{refl}(\_), h_1, h_2) \equiv h_2\\
&\quad \mathrm{vars}_{\mathrm{lemma}}(\mathrm{uniacc}_{\mathrm{var\_var}}(x, h_3), h, h_1, h_2) \equiv \mathrm{vars}_{\mathrm{lemma}}(h_3, h, \subseteq_{\mathrm{trans}}(\subseteq\mathrm{vars}_{\mathrm{var\_var}}(x, lp_1), h_1), h_2)\\
&\quad \mathrm{vars}_{\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{vars}_{\mathrm{lemma}}(\mathrm{uniacc}_{:=_{\mathrm{var\_term}}}(h_3, h_4), h, h_1, h_2) \equiv\\
&\qquad \mathrm{vars}_{\mathrm{lemma}}(h_4,\\
&\qquad\qquad h,\\
&\qquad\qquad \subseteq\mathrm{vars}_{:=_{\mathrm{LPT}}}(x, lp_1, \subseteq{+}{+}_{\mathrm{redL}}(\subseteq{+}{+}_{\mathrm{redL}}(h_1)), \subseteq{+}{+}_{\mathrm{redR}}(+{+}(\mathrm{vars}_{\mathrm{T}}(\mathrm{var}(x)), \mathrm{vars}_{\mathrm{T}}(t)), h_1)),\\
&\qquad\qquad \subseteq_{:}(\subseteq{+}{+}_{\mathrm{L}}(\subseteq\mathrm{vars}_{:=_{\mathrm{S}}}(x, sb, \subseteq{+}{+}_{\mathrm{redL}}(\subseteq{+}{+}_{\mathrm{redL}}(h_1)), h_2), \subseteq{+}{+}_{\mathrm{redL}}(\subseteq{+}{+}_{\mathrm{redL}}(h_1))),\\
&\qquad\qquad\quad \subseteq\in_{\mathrm{trans}}(\subseteq{+}{+}_{\mathrm{redR}}(\mathrm{vars}_{\mathrm{T}}(t), \subseteq{+}{+}_{\mathrm{redL}}(h_1)), \in_{\mathrm{hd}}(x, [])))\\
&\quad \mathrm{vars}_{\mathrm{lemma}}(\mathrm{uniacc}_{\mathrm{var\_fun}}(h_3), h, h_1, h_2) \equiv\\
&\qquad \mathrm{vars}_{\mathrm{lemma}}(h_3, h, \subseteq_{\mathrm{trans}}(\mathrm{fst}(\subseteq\mathrm{vars}_{\mathrm{var\_fun}}(f, x, lt, lp_1)), h_1), h_2)\\
&\quad \mathrm{vars}_{\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{vars}_{\mathrm{lemma}}(\mathrm{uniacc}_{\mathrm{zip\_fun\_fun}}(f, h_3), h, h_1, h_2) \equiv\\
&\qquad \mathrm{vars}_{\mathrm{lemma}}(h_3, h, \subseteq_{\mathrm{trans}}(\mathrm{fst}(\subseteq\mathrm{vars}_{\mathrm{fun\_fun}}(f, f, lt_1, lt_2, lp_1)), h_1), h_2)
\end{aligned}
$$

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 $x \in \mathrm{vars}_{\mathrm{T}}(t)$ 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 of [Bov99]. 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 of [Bov99] 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 of [Bov99], 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.

```
idem_lemma ∈ (p ∈ UniAcc(lp);
                =(unify(sb, p), ∨_L(sb'));
                Disjoint(vars_LPT(lp), dom(sb));
                Idem(sb)
              ) Idem(sb')
    idem_lemma(uniacc_[], refl(_), h1, h2) ≡ h2
    idem_lemma(uniacc_var_var(x, h3), h, h1, h2) ≡
        idem_lemma(h3, h, ⊆∧disj_todisj(⊆vars_var_var(x, lp1), h1), h2)
    idem_lemma(uniacc_var_term(lp1, h3, h4), h, h1, h2) ≡
        case h ∈ =(unify(sb, uniacc_var_term(lp1, h3, h4)), ∨_L(sb')) of
        end
    idem_lemma(uniacc:=var_term(h3, h4), h, h1, h2) ≡
        idem_lemma(
            h4,
            h,
            disj:R(=_subst1(=_dom(x, t, sb), ⊆∧disj_todisj(⊆vars_var_term(x, t, lp1), h1)), ∉:=_LPT=var(lp1, h3)),
            :_idem(idem:=(h2, h3, ⊆∧disj_todisj(⊆++_RR(vars_LPT(lp1), vars_T(var(x)), vars_T(t)), h1)),
                disj:R(
                    =_subst1(=_dom(x, t, sb), ⊆∧disj_todisj(⊆++_RR(vars_LPT(lp1), vars_T(var(x)), vars_T(t)), h1)),
                    h3),
                ∉:=_S=_var(
                    sb,
                    h3,
                    disj_to∉(disj_symm(⊆∧disj_todisj(⊆++_LR(vars_LPT(lp1), vars_T(var(x)), vars_T(t)), h1))))))
    idem_lemma(uniacc_var_fun(h3), h, h1, h2) ≡
        idem_lemma(h3, h, ⊆∧disj_todisj(fst(⊆vars_var_fun(f, x, lt, lp1)), h1), h2)
    idem_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
    idem_lemma(uniacc_zip_fun_fun(f, h3), h, h1, h2) ≡
        idem_lemma(h3, h, ⊆∧disj_todisj(fst(⊆vars_fun_fun(f, f, lt1, lt2, lp1)), h1), h2)
```

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

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

```
≡_Lp_Unify ∈ (=(Unify(lp), ∨_L(sb))) ≡_LpSb(lp, sb)
    ≡_Lp_Unify(h) ≡
        ∀_I([sb1]
                ∧_I(⇒_I([h1]unifies_LpSbtounifies_Sb(allUniAcc_LPT(lp), fst(h1), snd(h1), h)),
                    ⇒_I([h1]unifies_Sbtounifies_LpSb(allUniAcc_LPT(lp), h1, h))))
```

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

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

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

$\text{unifies}_{\text{LpSbto}}\text{unifies}_{\text{Sb}}(\text{uniacc}_{\text{var\_term}}(lp_1, h_3, h_4), h, h_1, h_2) \equiv$
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**
$\wedge_{\text{I}}(h_3, h_4) \Rightarrow \wedge_{\text{I}}(\text{unifies}_{\text{LPT\_:}}(h_3, \text{refl}(\text{app}_{\text{T}}(sb', \text{var}(x)))), h_4)$
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

The parameter h_1 in the first argument of the conjunction is a proof that the 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 of [Bov99] 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:

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

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:

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

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.

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

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 of [Bov99] 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 of [Bov99]. 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_{\mathrm{I}}(\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=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=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=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=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 of [Bov99].

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 can be found in appendix C of [Bov99], 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 of [Bov99] 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].

In [MW81], the deductive synthesis of the unification algorithm by Manna and Waldinger 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 $t_1 \rightarrow t_2$. 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 of [Bov99].

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

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

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

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

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

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 $\#\text{eqs}_{\text{LPT}}$. 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 of [Bov99] 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:

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

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

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

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

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

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

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

Given the lists of variables l_1 and l_2, to prove that $\text{len}(l_1) < \text{len}(l_2)$ 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 $\text{len}(l_1) \leq \text{len}(l_2)$ it is sufficient to prove that l_1 is included in l_2. Finally, to prove that $\text{len}(l_1) = \text{len}(l_2)$ 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 $\text{len}(l_1) \leq \text{len}(l_2)$ and that $\text{len}(l_2) \leq \text{len}(l_1)$, which clearly gives us a proof that $\text{len}(l_1) = \text{len}(l_2)$.

[1] See section C.3.3 of [Bov99] 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:

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

where the lemma $\mathsf{uniacc}_{\mathsf{aux2}}$ has the following type:

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

To prove this last lemma, we proceed in a similar way as the one presented in section 4.4.1 to define the algorithm $\mathsf{Unify}_{\mathsf{acc}}$: 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 $\mathsf{Unify}_{\mathsf{acc}}$. 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 $\mathsf{uniacc}_{\mathsf{aux2}}$ 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 $\mathsf{Unify}_{\mathsf{acc}}$ 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 $\text{uniacc}_{\text{aux2}}$

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.

[Bov99] A. Bove. Programming in Martin-Löf type theory: Unification - A non-trivial example, November 1999. Licentiate Thesis of the Department of Computer Science, Chalmers University of Technology. Available on the WWW `http://cs.chalmers.se/~bove/Papers/lic_thesis.ps.gz`.

[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] S. Peyton Jones, J. Hughes, (editors), L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. 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.