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;
- 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.

[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.

$$
\begin{array}{l}
\mathsf{evenAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{evenacc0} \in \mathsf{evenAcc}(0) \\
\quad \mathsf{evenaccs} \in (n \in \mathsf{N}; h \in \mathsf{oddAcc}(n))\mathsf{evenAcc}(\mathsf{s}(n)) \\
\\
\mathsf{oddAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{oddacc0} \in \mathsf{oddAcc}(0) \\
\quad \mathsf{oddaccs} \in (n \in \mathsf{N}; h \in \mathsf{evenAcc}(\mathsf{s}(n)))\mathsf{oddAcc}(\mathsf{s}(n))
\end{array}
$$

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

$$
\begin{array}{l}
\mathsf{even} \in (m \in \mathsf{N}; \mathsf{evenAcc}(m))\mathsf{Bool} \\
\quad \mathsf{even}(0, \mathsf{evenacc0}) = \mathsf{True} \\
\quad \mathsf{even}(\mathsf{s}(n), \mathsf{evenaccs}(n, h)) = \mathsf{odd}(n, h) \\
\\
\mathsf{odd} \in (m \in \mathsf{N}; \mathsf{oddAcc}(m))\mathsf{Bool} \\
\quad \mathsf{odd}(0, \mathsf{oddacc0}) = \mathsf{False} \\
\quad \mathsf{odd}(\mathsf{s}(n), \mathsf{oddaccs}(n, h)) = \mathsf{not}(\mathsf{even}(\mathsf{s}(n), h))
\end{array}
$$

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.

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

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].

```
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.

[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.

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. Note the indirect nested recursion 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:

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

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

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

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.

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

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

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

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

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

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

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 pay attention just 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.

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

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.

$\mathsf{revAcc} \in (zs \in \mathsf{List(A)})\mathsf{Set}$
  $\mathsf{revacc1} \in \mathsf{revAcc}(\mathsf{nil})$
  $\mathsf{revacc2} \in (x \in A; xs \in \mathsf{List}(A); h \in \mathsf{rev2Acc}(x, xs))\mathsf{revAcc}(\mathsf{cons}(x, xs))$

$\mathsf{rev2Acc} \in (y \in A; zs \in \mathsf{List(A)})\mathsf{Set}$
  $\mathsf{rev2acc1} \in (y \in A)\mathsf{rev2Acc}(y, \mathsf{nil})$
  $\mathsf{rev2acc2} \in (y, x \in A; xs \in \mathsf{List}(A); h_1 \in \mathsf{rev2Acc}(x, xs);$
    $h_2 \in \mathsf{revAcc}(\mathsf{rev2}(x, xs, h_1)); h_3 \in \mathsf{revAcc}(\mathsf{cons}(y, \mathsf{rev}(\mathsf{rev2}(x, xs, h_1), h_2)))$
    $)\mathsf{rev2Acc}(y, \mathsf{cons}(x, xs))$

$\mathsf{rev} \in (zs \in \mathsf{List}(A); \mathsf{revAcc}(zs))\mathsf{List}(A)$
  $\mathsf{rev}(\mathsf{nil}, \mathsf{revacc1}) \equiv \mathsf{nil}$
  $\mathsf{rev}(\mathsf{cons}(x, xs), \mathsf{revacc2}(x, xs, h)) \equiv \mathsf{cons}(\mathsf{last}(x, xs), \mathsf{rev2}(x, xs, h))$

$\mathsf{rev2} \in (y \in A; zs \in \mathsf{List}(A); \mathsf{rev2Acc}(y, zs))\mathsf{List}(A)$
  $\mathsf{rev2}(y, \mathsf{nil}, \mathsf{rev2acc1}(y)) \equiv \mathsf{nil}$
  $\mathsf{rev2}(y, \mathsf{cons}(x, xs), \mathsf{rev2acc2}(y, x, xs, h_1, h_2, h_3)) \equiv \mathsf{rev}(\mathsf{cons}(y, \mathsf{rev}(\mathsf{rev2}(x, xs, h_1), h_2)), h_3)$

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 simultaneously 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 $(A :: \sigma)$;
- $\psi_l[A, a]$ 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 analyse carefully 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:

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

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:

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

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:

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

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 ones $u' : (x :: \xi'[A])P_g(A, p'[A,x])$, 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 that:

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

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

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

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 will 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 $m+1 \leqslant z \leqslant m+n$:

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

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

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

[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 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 proof 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 proofs 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 proof 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 the assumptions $(A :: \sigma; A' :: \sigma'; a :: \alpha_k[A]; c : P_k(A, a))$.

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

[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).

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

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

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 *Springer-Verlag, LNCS*, 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] Simon Peyton Jones, John Hughes, (editors), Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.

[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.