

Nested General Recursion and Partiality in Type Theory

Ana Bove¹ and Venanzio Capretta²

¹ 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

² 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

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 $(x_1 \in \alpha_1; \dots; x_n \in \alpha_n)\mathbf{Set}$, for $\alpha_1, \dots, \alpha_n$ types. We must specify the constructors that generate the elements of $A(a_1, \dots, a_n)$ by giving their types, for $a_1 \in A_1, \dots, a_n \in A_n$.

Abstractions are written as $[x_1, \dots, x_n]e$ and theorems are introduced as dependent types of the form $(x_1 \in \alpha_1; \dots; x_n \in \alpha_n)\beta(x_1, \dots, x_n)$. 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 $(x_1, x_2, \dots, x_n \in \alpha)$ instead of $(x_1 \in \alpha; x_2 \in \alpha; \dots; x_n \in \alpha)$.

General recursive algorithms are defined by cases where the recursive calls are performed on objects that satisfy no 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 \mathbf{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 $\mathbf{Acc}(A, \prec, a)$. The only introduction rule for the accessibility predicate is

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

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

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

and its computation rule is

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

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.

From [Bov99] we can extract 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.

Given the Haskell version of an algorithm f_alg , the method in [Bov99] uses an inductive special-purpose accessibility predicate called $fAcc$. We construct this predicate directly from f_alg , and we regard it as a characterization 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 [Bov99] 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 [Bov99]. 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 [Bov99] 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 [Bov99] 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 algorithm¹:

```
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 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 lemmas²:

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

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 $s(m)$, then the result of subtracting $s(m)$ from n is less than n .

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

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

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

accessibility predicate Acc to handle the recursive call is³

```

modacc ∈ (n, m ∈ ℕ; Acc(ℕ, <, n))ℕ ∨ Error
  modacc(n, 0, acc(n, p)) = inr(error)
  modacc(n, s(m1), acc(n, p)) =
    case less-dec(n, s(m1)) ∈ Dec(n < s(m1)) of
      inl(q1) ⇒ inl(n)
      inr(q2) ⇒ modacc(n - s(m1), s(m1), p(n - s(m1), min-less(n, m1, q2)))
    end.

```

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 $s(m_1)$ for some natural number m_1 , we distinguish cases on whether n is smaller 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 on the values $n - s(m_1)$ and $s(m_1)$. The recursive call needs a proof that the value $n - s(m_1)$ is accessible. This proof is given by the expression $p(n - s(m_1), \text{min-less}(n, m_1, q_2))$, which is structurally smaller than $\text{acc}(n, p)$.

We can easily define a function $\text{allacc}_\mathbb{N}$ 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:

$$\begin{aligned} \text{Mod}_{\text{acc}} &\in (n, m \in \mathbb{N})\mathbb{N} \vee \text{Error} \\ \text{Mod}_{\text{acc}}(n, m) &= \text{mod}_{\text{acc}}(n, m, \text{allacc}_\mathbb{N}(n)). \end{aligned}$$

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 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 for the modulo algorithm, ModAcc , containing useful information that can help 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:

³ The set former \vee represents the disjunction of two sets, and inl and inr the two constructors of the set.

- 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 zero⁴, 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)

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

This predicate can easily be formalised in type theory:

$$\begin{aligned} \text{ModAcc} &\in (n, m \in \mathbb{N})\text{Set} \\ \text{modacc}_0 &\in (n \in \mathbb{N})\text{ModAcc}(n, 0) \\ \text{modacc}_< &\in (n, m \in \mathbb{N}; n < m)\text{ModAcc}(n, m) \\ \text{modacc}_\geq &\in (n, m \in \mathbb{N}; \neg(m = 0); \neg(n < m); \text{ModAcc}(n - m, m)) \\ &\quad \text{ModAcc}(n, m). \end{aligned}$$

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

$$\begin{aligned} \text{mod} &\in (n, m \in \mathbb{N}; \text{ModAcc}(n, m))\mathbb{N} \vee \text{Error} \\ \text{mod}(n, 0, \text{modacc}_0(n)) &= \text{inr}(\text{error}) \\ \text{mod}(n, m, \text{modacc}_<(n, m, q)) &= \text{inl}(n) \\ \text{mod}(n, m, \text{modacc}_\geq(n, m, q_1, q_2, h)) &= \text{mod}(n - m, m, h). \end{aligned}$$

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 $\text{modacc}_{\text{aux}}$. Given a natural number m , this lemma proves $\text{ModAcc}(i, m)$, for i an accessible natural number, from the assumption that $\text{ModAcc}(j, m)$ holds for every natural number j smaller than i . The proof proceeds by case analysis on m and, when m is equal to $s(m_1)$ for some natural number m_1 , by cases on whether i is smaller than $s(m_1)$. The term $\text{nots0}(m_1)$ is

⁴ 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.

a proof that $s(m_1)$ is different from 0.

```

modaccaux ∈ (m, i ∈ N; Acc(N, <, i); f ∈ (j ∈ N; j < i)ModAcc(j, m))
  ModAcc(i, m)
modaccaux(0, i, h, f) = modacc0(i)
modaccaux(s(m1), i, h, f) =
  case less-dec(i, s(m1)) ∈ Dec(i < s(m1)) of
    inl(q1) ⇒ modacc<(i, s(m1), q1)
    inr(q2) ⇒ modacc≥(i, s(m1), nots0(m1), q2,
      f(i - s(m1), min-less(i, m1, q2)))
  end

```

Now, we prove that the modulo algorithm terminates on all inputs, that is, we prove that all pairs of natural numbers satisfy ModAcc ⁵:

```

allModAcc ∈ (n, m ∈ N)ModAcc(n, m)
allModAcc(n, m) = wfrec(n, allaccN(n), modaccaux(m)).

```

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

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

```

Mod ∈ (n, m ∈ N)N ∨ Error
Mod(n, m) = mod(n, m, allModAcc(n, m)).

```

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 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.

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).

```

⁵ Here, we use the general recursor wfrec with the elimination predicate $P(n) \equiv \text{ModAcc}(n, m)$.

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 $s(n)$ for some natural number n , then the algorithm terminates if it terminates on the inputs n and $\text{nest}(n)$.

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

$$\frac{}{\text{NestAcc}(0)}, \quad \frac{\text{NestAcc}(n) \quad \text{NestAcc}(\text{nest}(n))}{\text{NestAcc}(s(n))}.$$

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 $\text{NestAcc}(n)$. Using Dybjer's schema, we can simultaneously define NestAcc and nest :

$$\begin{aligned} \text{NestAcc} &\in (n \in \mathbb{N})\text{Set} \\ \text{nest} &\in (n \in \mathbb{N}; \text{NestAcc}(n))\mathbb{N} \\ \\ \text{nestacc0} &\in \text{NestAcc}(0) \\ \text{nestaccs} &\in (n \in \mathbb{N}; h_1 \in \text{NestAcc}(n); h_2 \in \text{NestAcc}(\text{nest}(n, h_1))) \\ &\quad \text{NestAcc}(s(n)) \\ \\ \text{nest}(0, \text{nestacc0}) &= 0 \\ \text{nest}(s(n), \text{nestaccs}(n, h_1, h_2)) &= \text{nest}(\text{nest}(n, h_1), h_2). \end{aligned}$$

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, $\text{NestAcc}(0)$ is well defined because it does not depend on any assumption and its only element is nestacc0 . Once $\text{NestAcc}(0)$ 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 $n = 0$, $h_1 =$

`nestacc0` and $h_2 = \text{nestacc0}$. This application is well typed since h_2 must be an element in $\text{NestAcc}(\text{nest}(0, \text{nestacc0}))$, that is, $\text{NestAcc}(0)$. At this point, we can compute the value of $\text{nest}(s(0), \text{nestaccs}(0, \text{nestacc0}, \text{nestacc0}))$ and obtain the value `zero`⁶, 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:

$$\text{allNestAcc} \in (n \in \mathbb{N})\text{NestAcc}(n).$$

This can be done by first proving that, given a natural number n and a proof h of $\text{NestAcc}(n)$, $\text{nest}(n, h) \leq n$ (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:

$$\begin{aligned} \text{Nest} &\in (n \in \mathbb{N})\mathbb{N} \\ \text{Nest}(n) &= \text{nest}(n, \text{allNestAcc}(n)). \end{aligned}$$

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))).
```

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):

$$\frac{\overline{\text{nmAcc}(\text{At})}, \quad \frac{\text{nmAcc}(y) \quad \text{nmAcc}(z)}{\text{nmAcc}(\text{If}(\text{At}, y, z))}, \quad \frac{\text{nmAcc}(\text{If}(v, y, z)) \quad \text{nmAcc}(\text{If}(w, y, z))}{\text{nmAcc}(\text{If}(u, \text{nm}(\text{If}(v, y, z)), \text{nm}(\text{If}(w, y, z))))}}{\text{nmAcc}(\text{If}(\text{If}(u, v, w), y, z))}.$$

⁶ Since $\text{nest}(s(0), \text{nestaccs}(0, \text{nestacc0}, \text{nestacc0})) = \text{nest}(\text{nest}(0, \text{nestacc0}), \text{nestacc0}) = \text{nest}(0, \text{nestacc0}) = 0$

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

$$\begin{aligned}
\text{nmAcc} &\in (e \in \text{CExp})\text{Set} \\
\text{nm} &\in (e \in \text{CExp}; \text{nmAcc}(e))\text{CExp} \\
\\
\text{nmacc}_1 &\in \text{nmAcc}(\text{At}) \\
\text{nmacc}_2 &\in (y, z \in \text{CExp}; \text{nmAcc}(y); \text{nmAcc}(z))\text{nmAcc}(\text{If}(\text{At}, y, z)) \\
\text{nmacc}_3 &\in (u, v, w, y, z \in \text{CExp}; \\
&\quad h_1 \in \text{nmAcc}(\text{If}(v, y, z)); h_2 \in \text{nmAcc}(\text{If}(w, y, z)); \\
&\quad h_3 \in \text{nmAcc}(\text{If}(u, \text{nm}(\text{If}(v, y, z), h_1), \text{nm}(\text{If}(w, y, z), h_2)))) \\
&\quad \text{nmAcc}(\text{If}(\text{If}(u, v, w), y, z))) \\
\\
\text{nm}(\text{At}, \text{nmacc}_1) &= \text{At} \\
\text{nm}(\text{If}(\text{At}, y, z), \text{nmacc}_2(y, z, h_1, h_2)) &= \text{If}(\text{At}, \text{nm}(y, h_1), \text{nm}(z, h_2)) \\
\text{nm}(\text{If}(\text{If}(u, v, w), y, z), \text{nmacc}_3(u, v, w, y, z, h_1, h_2, h_3)) &= \\
&\quad \text{nm}(\text{If}(u, \text{nm}(\text{If}(v, y, z), h_1), \text{nm}(\text{If}(w, y, z), h_2)), h_3).
\end{aligned}$$

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:

$$\text{allnmAcc} \in (e \in \text{CExp})\text{nmAcc}(e).$$

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:

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

With this measure, it is easy to prove that

$$\begin{aligned}
|\text{If}(v, y, z)| &< |\text{If}(\text{If}(u, v, w), y, z)|, \quad |\text{If}(w, y, z)| < |\text{If}(\text{If}(u, v, w), y, z)| \\
\text{and} \quad |\text{If}(u, v', w')| &< |\text{If}(\text{If}(u, v, w), y, z)|
\end{aligned}$$

for every v', w' such that $|v'| \leq |\text{If}(v, y, z)|$ and $|w'| \leq |\text{If}(w, y, z)|$. Therefore, to prove that the predicate nmAcc holds for a certain $e \in \text{CExp}$, we need to call nm only on those arguments that have smaller measure than e^7 .

Now, we can prove that every conditional expression satisfies nmAcc by first proving that, given a conditional expression e and a proof h of $\text{nmAcc}(e)$,

⁷ We could have done something similar in the case of the algorithm `nest` by defining the measure $|x| = x$ and proving the inequality $y < s(x)$ for every $y \leq x$

$|nm(e, h)| \leq |e|$ (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:

$$\begin{aligned} NM &\in (e \in \text{CExp})\text{CExp} \\ NM(e) &= nm(e, \text{allNmAcc}(e)). \end{aligned}$$

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 minimization operator for natural numbers, which takes a function $f \in (\mathbb{N})\mathbb{N}$ as input and gives the least $n \in \mathbb{N}$ such that $f(n) = 0$ as output, cannot be represented as an object of type $((\mathbb{N})\mathbb{N})\mathbb{N}$ because it does not terminate on all inputs. A standard representation of this operator in type theory is the inductive relation

$$\begin{aligned} \mu &\in (f \in (\mathbb{N})\mathbb{N}; n \in \mathbb{N})\text{Set} \\ \mu_0 &\in (f \in (\mathbb{N})\mathbb{N}; f(0) = 0)\mu(f, 0) \\ \mu_1 &\in (f \in (\mathbb{N})\mathbb{N}; f(0) \neq 0; n \in \mathbb{N}; \mu([m]f(s(m)), n))\mu(f, s(n)). \end{aligned}$$

The relation μ represents the graph of the minimization operator. If we indicate the minimization function by min , then $\mu(f, n)$ is inhabited if and only if $\text{min}(f) = n$. 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 $n, m \in \mathbb{N}$, 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 $\text{min}(f)$ for any f .

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

```
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 $[m]f(s(m))$. 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)

$$\frac{f(0) = 0}{\text{minAcc}(f)}, \quad \frac{f(0) \neq 0 \quad \text{minAcc}([m]f(s(m)))}{\text{minAcc}(f)}.$$

We can directly translate these rules into type theory:

$$\begin{aligned} \text{minAcc} &\in (f \in (\mathbb{N})\mathbb{N})\text{Set} \\ \text{minacc0} &\in (f \in (\mathbb{N})\mathbb{N}; f(0) = 0)\text{minAcc}(f) \\ \text{minacc1} &\in (f \in (\mathbb{N})\mathbb{N}; f(0) \neq 0; \text{minAcc}([m]f(s(m))))\text{minAcc}(f). \end{aligned}$$

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

$$\begin{aligned} \text{min} &\in (f \in (\mathbb{N})\mathbb{N}; \text{minAcc}(f))\mathbb{N} \\ \text{min}(f, \text{minacc0}(f, q)) &= 0 \\ \text{min}(f, \text{minacc1}(f, q, h)) &= s(\text{min}([m]f(s(m)), h)). \end{aligned}$$

In this case, it is not possible to prove that all elements in $(\mathbb{N})\mathbb{N}$ satisfy the special predicate, simply because it is not true. However, given a function f , we may first prove $\text{minAcc}(f)$ (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 minimization 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 A 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)

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

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

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

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

$$\begin{aligned} \text{nfAcc} &\in (x \in \Lambda)\text{Set} \\ \text{nf} &\in (x \in \Lambda; \text{nfAcc}(x))\Lambda \\ \\ \text{nfacc}_1 &\in (i \in \mathbb{N})\text{nfAcc}(\text{Var}(i)) \\ \text{nfacc}_2 &\in (i \in \mathbb{N}; a \in \Lambda; h_a \in \text{nfAcc}(a))\text{nfAcc}(\text{Abst}(i, a)) \\ \text{nfacc}_3 &\in (a, b \in \Lambda; h_a \in \text{nfAcc}(a); h_b \in \text{nfAcc}(b); i \in \mathbb{N}; \text{nf}(a, h_a) = \text{Var}(i)) \\ &\quad \text{nfAcc}(\text{App}(a, b)) \\ \text{nfacc}_4 &\in (a, b \in \Lambda; h_a \in \text{nfAcc}(a); i \in \mathbb{N}; a' \in \Lambda; \\ &\quad \text{nf}(a, h_a) = \text{Abst}(i, a'); \text{nfAcc}(\text{sub}(a', i, b))) \\ &\quad \text{nfAcc}(\text{App}(a, b)) \\ \text{nfacc}_5 &\in (a, b \in \Lambda; h_a \in \text{nfAcc}(a); h_b \in \text{nfAcc}(b); \\ &\quad a', a'' \in \Lambda; \text{nf}(a, h_a) = \text{App}(a', a'')) \\ &\quad \text{nfAcc}(\text{App}(a, b)) \\ \\ \text{nf}(\text{Var}(i), \text{nfacc}_1(i)) &= \text{Var}(i) \\ \text{nf}(\text{Abst}(i, a), \text{nfacc}_2(i, a, h_a)) &= \text{Abst}(i, \text{nf}(a, h_a)) \\ \text{nf}(\text{App}(a, b), \text{nfacc}_3(a, b, h_a, h_b, i, q)) &= \text{App}(\text{Var}(i), \text{nf}(b, h_b)) \\ \text{nf}(\text{App}(a, b), \text{nfacc}_4(a, b, h_a, i, a', q, h)) &= \text{nf}(\text{sub}(a', i, b), h) \\ \text{nf}(\text{App}(a, b), \text{nfacc}_5(a, b, h_a, h_b, a', a'', q)) &= \text{App}(\text{App}(a', a''), \text{nf}(b, h_b)). \end{aligned}$$

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.

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.

Most of 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 and uses strong inductive principles not available in type theory.

Some work has been done in the area of formalising partial functions. 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. A function f is a partial function from A to B , denoted $A \rightsquigarrow B$, if its domain can be computed as a predicate over A . In order to apply f to an element a in A , we need a proof that a is in the domain of f . In [CS87], Constable and Smith develop a partial type theory. Corresponding to every type T of the underlying total theory there is a type \overline{T} , which might contain diverging terms. They define a termination predicate $t \text{ in! } T$ which asserts that t is a term that terminates and they also present some induction principles applicable to partial types. 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. Balaá and Y. Bertot. Fix-point equations for well-founded recursion in type theory. In Harrison and Aagaard [HA00], pages 1–16.
- [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.
- [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] 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.
- [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.