Generalised Simultaneous Inductive-Recursive Definitions and their Application to Programming 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

Abstract. In this work we present a generalisation of Dybjer's schema for simultaneous inductive-recursive definitions 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. As an application, we can use the methodology developed by Bove and Capretta for formalising general recursive algorithms to define nested and mutually recursive algorithms in type theories extended with the generalised schema we present in this work. Hence, the resulting definitions are such that the computational and logical parts are clearly separated. Moreover, 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

Constructive type theory (see for example [ML84,CH88]) is a formal language in which to carry out constructive mathematics and where induction is one of the main notions. Following the Curry-Howard isomorphism [How80], it can also be seen as a very expressive programming language with dependent types where specifications are represented as types and programs as objects of those types. Therefore, we can encode in a type a complete specification, requiring also logical properties from an algorithm. As a consequence, algorithms are correct by construction or can be proved correct by using the expressive power of constructive type theory. This is clearly an advantage of constructive type theory over standard programming languages. A computational limitation of type theory is that, to keep the logic consistent and type-checking decidable, only structural recursive definitions are allowed, that is, definitions in which the recursive calls must have structurally smaller arguments.

General recursive algorithms are defined by cases where the recursive calls are not required to be on 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. 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.

In [BC02], 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. Since the method separates the computational part from the logical part of a definition, formalising partial functions becomes possible. Proving that a certain function is total amounts to proving that the corresponding accessibility predicate is satisfied by every input.

If 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]). When the algorithms are nested and mutually recursive, we have to define several predicates simultaneously with several functions. As Dybjer only considers the case of one predicate and one function in [Dyb00], 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 this paper.

As an application, we show how we can use the methodology developed in [BC02] to formalise nested and mutually recursive algorithms in type theories extended with the generalised schema we introduce here.

The rest of the paper is organised as follows. In section 2, we briefly recall the main concepts of constructive type theory. In section 3, we illustrate the methodology presented in [BC02] by showing the formalisation of some simple examples. In section 4, 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. Finally, in section 5, we present some conclusions.

2 Constructive Type Theory

Although this paper is intended mainly for those who already have some knowledge of type theory, we recall the basic ideas and notions that we use. For a complete presentation of constructive type theory, see [ML84,NPS90,CNSvS94].

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".

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

The basic type is the type of 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. The canonical elements of function types are λ-abstractions. 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 $\beta(a)$. Non-dependent functions are special cases of dependent functions where the type β does not depend on the type α.

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

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

3 Formalising General Recursion in Type Theory

In [BC02], we present a method to formalise general recursive algorithms in type theory in an easy way. Here, we illustrate that method by showing the formalisation of some simple examples.

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

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

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

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

From this description, we define the inductive predicate qsAcc over lists of natural numbers by the following introduction rules:

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

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

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

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

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

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

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

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

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

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

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

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

Following the same methodology, we would construct the predicate $\mathsf{f}_{91}\mathsf{Acc}$ defined by the following introduction rules (for n a natural number)[1]:

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

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

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

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

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. If the mutually recursive algorithms are not nested, their formalisation is similar to the formalisation of the quicksort algorithm in the sense that we first define the accessibility predicate for each function and then, we formalise the algorithms by structural recursion on the proof that the input values satisfy the corresponding predicate. If, in addition to mutual recursion, we have nested calls, we again need to define the predicates simultaneously with the algorithms. In order to do so, we need to extend Dybjer's schema for the cases where we have several mutually recursive predicates defined simultaneously with several functions (Dybjer's schema considers only one predicate and one function). This

[1] Observe that here, $>$ and $\leqslant$ denote relations over the natural numbers while $\geqslant$ and $<$ in the above example denoted boolean functions over N.

generalisation is given in section 4. In [Bov02], the reader can find more applications of the generalised schema to programming nested and mutual recursive algorithms in type theory.

As an example of a nested and mutual recursive algorithm, we present an algorithm to reverse the order of the elements in a list of natural numbers which 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 :: [N] -> [N]
rev [] = []
rev (x:xs) = last x xs : rev2 x xs

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

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

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

As usual, we first present the introduction rules for the special-purpose inductive predicates revAcc and rev2Acc. Notice that, since the algorithms rev and rev2 are nested, the two predicates need to know about the two algorithms. In the following rules, x and y are natural numbers and xs is a list of natural numbers.

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

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

Finally, in type theory we formalise the inductive predicates revAcc and rev2Acc simultaneously with the algorithms rev and rev2, recursively defined on the predicates.

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

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

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

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

Partial functions may also be defined in type theory using the method we illustrated above. See [BC01,BC02] for some examples.

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

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 presented in [BC02] and illustrated in the previous section.

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 closely follow through this section the formalisation of the nested and mutually recursive algorithms f and g that we introduce below. They are 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)
```

To define the special-purpose accessibility predicates we study the equations in the Haskell version of the algorithms. 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. Their type-theoretic definitions are given in the following subsections.

4.1 Formation Rules

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

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

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

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

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

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

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

If we carefully analyse the assumptions stated above, we see that none of our inductive predicates or recursive functions is known when we construct the

sequences of small types α's and the types ψ's. Hence, no one of our predicates or functions can be mentioned in those sequences or types, since they are not yet defined. As a consequence, no one of our predicates can have any of the other predicates or functions as part of its formation rule. On the other hand, each function is defined by recursion on one of our inductive predicates and thus, this predicate must be part of the domain of the function. However, no other of our predicates or functions can be part of the formation rule of the function.

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

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

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

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

4.2 Introduction Rules

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

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

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

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

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

where

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

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

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

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

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

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

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

Here, facc_0 is an introduction rule with no premises. The premises of the 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.

4.3 Possible Dependencies

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

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

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

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

$$(A :: \sigma; \ldots; b' : \beta'[A]; \ldots; v' : (x :: \xi'[A])\psi_t[A, p'[A, x]]; \ldots)^2.$$

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

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

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

4.4 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

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

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

$$\begin{aligned} &\mathsf{f}(0, \mathsf{facc}_0) = 0 \\ &\mathsf{f}(\mathsf{s}(n), \mathsf{facc}_\mathsf{s}(n, h_1, h_2)) = \mathsf{f}(\mathsf{g}(n, h_1), h_2) \\ \\ &\mathsf{g}(0, \mathsf{gacc}_0) = 0 \\ &\mathsf{g}(\mathsf{s}(n), \mathsf{gacc}_\mathsf{s}(n, h_1, h_2)) = \mathsf{g}(\mathsf{f}(n, h_1), h_2) \end{aligned}$$

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

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

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

4.5 Recursive Definitions

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

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

by recursion on P_k, where

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

Observe that f'_y might have a different set of parameters than those needed for the definitions of the m inductive predicates and the n recursive functions[3]. Note also that both the inductive predicates and the recursive functions are known when we define the function f'_y and hence, they can be mentioned as part of the type ψ'_y (compare ψ' here with the type ψ introduced in section 4.1; 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

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

[3] 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 4.1 for a similar and more detail explanation of how to construct σ as the union of the different sets of parameters).

where

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

in the context

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

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

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

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

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.

5 Conclusions

In this work, we generalise Dybjer's schema for simultaneous inductive-recursive definitions originally 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.

A direct application of the general schema for simultaneous inductive-recursive definitions is the type-theoretic formalisation of nested recursive functions (mutually recursive or not) following the methodology developed in [BC02]. As we have already mentioned, the formalisation of nested functions using the methodology in [BC02] 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 introduced in section 4. Thanks to this schema, we are able to formalise in type theory mainly any general recursive algorithm following the same methodology, as we describe in [BC02] (where we also mention some restrictions on the algorithms that can be formalised using the methodology). Thus, in type theories extended with our general schema, we can formalise nested (and possibly mutually) recursive algorithms such that the computational and logical parts of the definitions are clearly separated. 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. In addition, the simplicity of the definitions of the type-theoretic algorithms usually simplifies the task of their formal verification.

References

[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.

[BC02] A. Bove and V. Capretta. Modelling general recursion in type theory, 2002. Available on the WWW `http://cs.chalmers.se/~bove/Papers/general_rec.ps.gz`.

[Bov02] A. Bove. Mutual general recursion in type theory, May 2002. Available on the WWW `http://cs.chalmers.se/~bove/Papers/mutual_rec.ps.gz`.

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

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

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

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

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

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

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

[MM70] Z. Manna and J. McCarthy. Properties of programs and partial function logic. *Machine Intelligence*, 5:27–37, 1970.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction*. Oxford University Press, 1990.