Modelling General Recursion in Type Theory

Ana Bove[*] Venanzio Capretta[†]

July 8, 2002

Abstract

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

1 Introduction

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

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

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

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

General recursive algorithms are defined by cases where the recursive calls are performed on non-structurally smaller arguments. In other words, the recursive calls are performed on objects that satisfy 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, BB00]). However, the use of this predicate in the type-theoretic formalisation of general recursive algorithms often results in unnecessarily long and complicated code. Moreover, its use adds a considerable amount of code with no computational content, that distracts our attention from the computational part of the algorithm (see for example [Bov99], where we present the formalisation of a unification algorithm over lists of pairs of terms using the standard accessibility predicate Acc).

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

This method was introduced by Bove [Bov01] to formalise simple general recursive algorithms in type theory (by simple we mean non-nested and non-mutually recursive). It was extended by Bove and Capretta [BC01] to treat nested recursion and by Bove [Bov02] to treat mutually recursive algorithms, nested or not. Since our method separates the computational part from the logical part of a definition, formalising partial functions becomes possible [BC01]. Proving that a certain function is total amounts to proving that the corresponding accessibility predicate is satisfied by every input.

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

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

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

2 Constructive Type Theory

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

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

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

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

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

Theorems have the general form of dependent types and thus, they have the form $(x_1 \in \alpha_1; \ldots ; x_n \in \alpha_n)\beta$.

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

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

Beside product types, we also use dependent sum types and disjoint unions.

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

If α and β are types, the disjoint union of α and β is denoted by $\alpha + \beta$. If $a \in \alpha$ and $b \in \beta$, then $\mathsf{in_l}(a)$ and $\mathsf{in_r}(b)$ are elements in $\alpha + \beta$.

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

If Γ is a context and β a type whose free variables are among the ones assumed in Γ, we write $(\Gamma)\beta$ for the sequential product of all the assumptions in Γ over β. Formally it is defined by recursion on the length of Γ. If Γ is empty, then $()\beta \equiv \beta$. If $\Gamma \equiv x \in \alpha; \Gamma'$, then $(x \in \alpha; \Gamma')\beta \equiv (x \in \alpha)((\Gamma')\beta)$. Our notation for several consecutive dependent products already makes this definition clear.

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

Finally, the disjoint union of n types $\alpha_1, \ldots, \alpha_n$ is denoted by $\alpha_1 + \ldots + \alpha_n$ and defined as $(\cdots(\alpha_1 + \alpha_2) + \ldots + \alpha_n)$. For the sake of simplicity, we call $\mathsf{in_1}, \mathsf{in_2}, \ldots, \mathsf{in_n}$ to the corresponding constructors.

3 Some Examples

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

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

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

Below, the type-theoretic translation of the boolean functions `<` and `>=` are called $\prec$ and $\succcurlyeq$, respectively. We do not use the symbols $<$ and $\geqslant$ for the formalisation of those functions because, later on, we use them to denote relations in type theory, that is, terms of type $(\mathsf{N}, \mathsf{N})\mathsf{Set}$, while in this example we need terms of type $(\mathsf{N}, \mathsf{N})\mathsf{Bool}$.

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

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

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

$$\frac{}{\mathsf{qsAcc}(\mathsf{nil})} \qquad \frac{\mathsf{qsAcc}(\mathsf{filter}((\prec x), xs)) \qquad \mathsf{qsAcc}(\mathsf{filter}((\not\prec x), xs))}{\mathsf{qsAcc}(\mathsf{cons}(x, xs))}$$

where $(\prec x)$ denotes the function $[y](y \prec x)$ as in functional programming, similarly for $\not\prec$. We formalise this predicate in type theory as follows:

$$\begin{array}{l}
\mathsf{qsAcc} \in (zs \in \mathsf{List}(\mathsf{N}))\mathsf{Set} \\
\quad \mathsf{qs\_acc}_{\mathsf{nil}} \in \mathsf{qsAcc}(\mathsf{nil}) \\
\quad \mathsf{qs\_acc}_{\mathsf{cons}} \in (x \in \mathsf{N}; xs \in \mathsf{List}(\mathsf{N}); h_1 \in \mathsf{qsAcc}(\mathsf{filter}((\prec x), xs)); \\
\qquad\qquad\qquad h_2 \in \mathsf{qsAcc}(\mathsf{filter}((\not\prec x), xs)) \\
\qquad\qquad\quad)\mathsf{qsAcc}(\mathsf{cons}(x, xs))
\end{array}$$

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

$$\begin{array}{l}
\mathsf{quicksort} \in (zs \in \mathsf{List}(\mathsf{N}); \mathsf{qsAcc}(zs))\mathsf{List}(\mathsf{N}) \\
\quad \mathsf{quicksort}(\mathsf{nil}, \mathsf{qs\_acc}_{\mathsf{nil}}) = \mathsf{nil} \\
\quad \mathsf{quicksort}(\mathsf{cons}(x, xs), \mathsf{qs\_acc}_{\mathsf{cons}}(x, xs, h_1, h_2)) = \\
\qquad \mathsf{quicksort}(\mathsf{filter}((\prec x), xs), h_1) +\!\!+\ \mathsf{cons}(x, \mathsf{quicksort}(\mathsf{filter}((\not\prec x), xs), h_2))
\end{array}$$

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 $\mathsf{QuickSort}$.

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

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

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

where `+` is the addition operation and `div2` is the division by two over natural numbers defined in a structurally recursive way, respectively.

Following the description given above for the algorithm $\mathsf{quicksort}$, we define the special-purpose accessibility predicate fAcc that characterises the inputs on which the algorithm f terminates. In this example, we have conditional recursive equations depending on the boolean conditions (usually called *guards* in Haskell literature) `(even n)` and `(odd n)`. These conditions are translated using the predicates Even and Odd in type theory and they are added as arguments of the corresponding constructor of the accessibility predicate. Here is its type-theoretical definition:

$$\begin{array}{l}
\mathsf{fAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{f\_acc_0} \in \mathsf{fAcc}(0) \\
\quad \mathsf{f\_acc_{s1}} \in (n \in \mathsf{N}; q \in \mathsf{Even}(n); h \in \mathsf{fAcc}(\mathsf{div}(n, 2)))\mathsf{fAcc}(\mathsf{s}(n)) \\
\quad \mathsf{f\_acc_{s2}} \in (n \in \mathsf{N}; q \in \mathsf{Odd}(n); h \in \mathsf{fAcc}(n + 4))\mathsf{fAcc}(\mathsf{s}(n))
\end{array}$$

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

$$\begin{array}{l}
\mathsf{f} \in (m \in \mathsf{N}; \mathsf{fAcc}(m))\mathsf{N} \\
\quad \mathsf{f}(0, \mathsf{f\_acc_0}) = 0 \\
\quad \mathsf{f}(\mathsf{s}(n), \mathsf{f\_acc_{s1}}(n, q, h)) = \mathsf{f}(\mathsf{div}(n, 2), h) + 1 \\
\quad \mathsf{f}(\mathsf{s}(n), \mathsf{f\_acc_{s2}}(n, q, h)) = \mathsf{f}(n + 4, h)
\end{array}$$

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

Our method applies also in the formalisation of nested recursive algorithms. Here is the Haskell code of McCarthy's $\mathsf{f_{91}}$ 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. 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 our method, we would construct the predicate $\mathsf{f_{91}Acc}$ defined by the following introduction rules (for n a natural number):

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

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

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

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

Mutually recursive algorithms, with or without nested recursive calls, can also be formalised with our method. If the mutually recursive algorithms are not nested, their formalisation is similar to the formalisation of the quicksort algorithm in the sense that we first define the accessibility predicates for each function and then, we formalise the algorithms by structural recursion on the proof that the input values satisfy the corresponding predicate. With mutually recursive algorithms, the termination of one function depends on the termination of the others. Hence, the accessibility predicates are also mutually recursive. If, in addition to mutual recursion, we have nested calls, we again need to define the predicates simultaneously with the algorithms. In order to do so, we need to extend Dybjer's schema for cases where we have several mutually recursive predicates defined simultaneously with several functions (originally, Dybjer's schema considers one predicate and one function). This extension and its application to the formalisation of mutually recursive functions in type theory was given in [Bov02]. Let us consider a simple example where we define two mutually recursive algorithms. In Haskell we write them as follows[1]:

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

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

where `<=` and `>` are inequalities over `N` defined in the usual way.

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

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

$$
\begin{array}{l}
\mathsf{fAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{f\_acc_0} \in \mathsf{fAcc}(0) \\
\quad \mathsf{f\_acc_{s1}} \in (n \in \mathsf{N}; h_1 \in \mathsf{gAcc}(n); q \in (\mathsf{g}(n, h_1) \leqslant n); h_2 \in \mathsf{fAcc}(\mathsf{g}(n, h_1)) \\
\qquad\qquad)\mathsf{fAcc}(\mathsf{s}(n)) \\
\quad \mathsf{f\_acc_{s2}} \in (n \in \mathsf{N}; h_1 \in \mathsf{gAcc}(n); q \in (\mathsf{g}(n, h_1) > n))\mathsf{fAcc}(\mathsf{s}(n))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{gAcc} \in (m \in \mathsf{N})\mathsf{Set} \\
\quad \mathsf{g\_acc_0} \in \mathsf{gAcc}(0) \\
\quad \mathsf{g\_acc_{s1}} \in (n \in \mathsf{N}; h_1 \in \mathsf{fAcc}(n); q \in (\mathsf{f}(n, h_1) \leqslant n); h_2 \in \mathsf{gAcc}(\mathsf{f}(n, h_1)) \\
\qquad\qquad)\mathsf{gAcc}(\mathsf{s}(n)) \\
\quad \mathsf{g\_acc_{s2}} \in (n \in \mathsf{N}; h_1 \in \mathsf{fAcc}(n); q \in (\mathsf{f}(n, h_1) > n))\mathsf{gAcc}(\mathsf{s}(n))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{f} \in (m \in \mathsf{N}; \mathsf{fAcc}(m))\mathsf{N} \\
\quad \mathsf{f}(0, \mathsf{f\_acc_0}) = 0 \\
\quad \mathsf{f}(\mathsf{s}(n), \mathsf{f\_acc_{s1}}(n, h_1, q, h_2)) = \mathsf{f}(\mathsf{g}(n, h_1), h_2) + n \\
\quad \mathsf{f}(\mathsf{s}(n), \mathsf{f\_acc_{s2}}(n, h_1, q)) = 0
\end{array}
$$

$$
\begin{array}{l}
\mathsf{g} \in (m \in \mathsf{N}; \mathsf{gAcc}(m))\mathsf{N} \\
\quad \mathsf{g}(0, \mathsf{g\_acc_0}) = 1 \\
\quad \mathsf{g}(\mathsf{s}(n), \mathsf{g\_acc_{s1}}(n, h_1, q, h_2)) = \mathsf{g}(\mathsf{f}(n, h_1), h_2) + \mathsf{f}(n, h_1) \\
\quad \mathsf{g}(\mathsf{s}(n), \mathsf{g\_acc_{s2}}(n, h_1, q)) = \mathsf{f}(n, h_1) + n
\end{array}
$$

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

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

3.1 Necessary Restrictions

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

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

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

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

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

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

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

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

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

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

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

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

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

Another restriction is that each function definition should be self-standing, by which we mean that it should not call other previously defined functions unless they are structurally recursive. If `f` is a general recursive function, it should be translated in type theory as a pair consisting of a predicate fAcc and a function f. Then, we cannot call it inside the definition of another function g. This restriction is imposed by type-checking requirements and it will become clearer below. If, instead, `f` is structurally recursive, it can be directly translated in type theory as a structurally recursive function f, without the need of our auxiliary predicate fAcc. In this case, the use of `f` inside the definition of another function g is allowed, as it has been seen throughout this section.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\begin{aligned}&\mathsf{nub\_mapAcc} \in (p \in (\mathsf{N} \rightharpoonup \mathsf{N}) \times \mathsf{List}(\mathsf{N}))\mathsf{Set}\\ &\mathsf{nub\_map} \in (p \in (\mathsf{N} \rightharpoonup \mathsf{N}) \times \mathsf{List}(\mathsf{N}); \mathsf{nub\_mapAcc}(p))\mathsf{List}(\mathsf{N})\\ \\ &\mathsf{NubMap} \in (\mathsf{N} \rightharpoonup \mathsf{N}) \times \mathsf{List}(\mathsf{N}) \rightharpoonup \mathsf{List}(\mathsf{N})\\ &\mathsf{NubMap} = \langle \mathsf{nub\_mapAcc}, \mathsf{nub\_map}\rangle\end{aligned}$$

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

$$\begin{aligned}&\mathsf{fAcc} \in (m \in \mathsf{N})\mathsf{Set}\\ &\mathsf{f} \in (m \in \mathsf{N}; \mathsf{fAcc}(m))\mathsf{N}\\ \\ &\mathsf{F} \in \mathsf{N} \rightharpoonup \mathsf{N}\\ &\mathsf{F} = \langle \mathsf{fAcc}, \mathsf{f}\rangle\end{aligned}$$

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

$$\begin{array}{l}
\mathsf{gAcc} \in (\mathsf{N})\mathsf{Set} \\
\mathsf{g\_acc} \in (l \in \mathsf{List}(\mathsf{N}); \mathsf{nub\_mapAcc}(\langle \mathsf{F}, l\rangle))\mathsf{gAcc}(l) \\
\\
\mathsf{g} \in (l \in \mathsf{N}; \mathsf{gAcc}(l))\mathsf{List}(\mathsf{N}) \\
\mathsf{g}(l, \mathsf{g\_acc}(l, h)) = \mathsf{nub\_map}(\langle \mathsf{F}, l\rangle, h) \\
\\
\mathsf{G} \in \mathsf{List}(\mathsf{N}) \rightharpoonup \mathsf{List}(\mathsf{N}) \\
\mathsf{G} = \langle \mathsf{gAcc}, \mathsf{g}\rangle
\end{array}$$

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

4 General Recursive Definitions

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

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

4.1 The Class of Types

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

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

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

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

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

- If σ and τ are types, then $\sigma \to \tau$ is a type.

In what follows, σ and τ denote types.

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

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

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

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

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

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

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

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

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

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

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

respectively.

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

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

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

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

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

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

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

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

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

4.2 Terms and Patterns

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

Definition 1. Let `T` be a type. A *pattern* of type `T` is an expression build up according to the following two rules.

- A variable of type `T` is a pattern of type `T`.
- If `T` is an inductive type, $\texttt{c}\colon \tau_1, \ldots, \tau_k \Rightarrow \texttt{T}$ is one of its constructors and $p'_1, \ldots, p'_k$ are patterns of type $\tau_1, \ldots, \tau_k$, respectively, then $\texttt{c}(p'_1, \ldots, p'_k)$ is a pattern of type `T`.

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

We consider here only linear patterns, since this restriction is present in some functional programming languages. Usually, we say just *pattern* when we refer to *linear pattern*.

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

The sequence is called *exhaustive* if every term in normal form of type `T` is an instance of at least one of the patterns.

Now, we define the terms that are allowed in a recursive definition. These terms depend on three parameters: the variables that can occur free in the term, the functions being defined, which can be used in the recursive calls, and the previously defined functions that we allow within a definition. Let us assume that we have a class $\mathcal{F}$ of functions together with their type such that every element $\mathtt{f}\colon \sigma_1 \to \cdots \to \sigma_m \to \tau$ in $\mathcal{F}$ can be translated into type theory with the same functional type. In what follows, we assume that $\mathcal{F}$ contains all structurally recursive functions. As we have already mention, it is possible to extend $\mathcal{F}$ to a larger class of functions by adding all the functions that can be proved total in type theory. As we can consider the class $\mathcal{F}$ to be fix, below we give the definition of the terms that we allow in a recursive definition considering only as parameters the variables that can occur free in the term and the functions being defined. Let $\mathcal{X}$ be the set of variables together with their type that can occur free in a term. The idea is that, when defining an equation, this set only contains the pattern variables of the equation together with their corresponding type. Let $\mathcal{SF}$ be the set of specifications of the functions being defined. When defining a single function, $\mathcal{SF}$ contains only one specification. When defining several functions within a mutual recursive block, $\mathcal{SF}$ contains one specification for each of the functions being mutually defined. Formally, we define terms as follows.

Definition 3. Let $\mathcal{X}$ be a set of variables together with their type. Let $\mathcal{SF}$ be a set of functions specifications, that is, every element of $\mathcal{SF}$ is of the form $\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau$. Let the set of names of the variables in $\mathcal{X}$, the set of names of the functions in $\mathcal{SF}$ and the set of names of the functions in $\mathcal{F}$ be disjoint. The class of *valid terms with respect to* $\mathcal{X}$ *and* $\mathcal{SF}$ is build up according to the rules below. When the sets of variables and of function specifications remain the same, we simply refer to the terms in the class as *valid terms.*

- The variables in $\mathcal{X}$ are valid terms of the corresponding type.
- The functions in $\mathcal{F}$ are valid terms of the corresponding type.
- If $\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau$ is an element of $\mathcal{SF}$ and if $a_1, \ldots, a_m$ are valid terms of type $\sigma_1, \ldots, \sigma_m$, respectively, then $f(a_1, \ldots, a_m)$ is a valid term of type τ.
- If $\mathtt{c}\colon \tau_1, \ldots, \tau_k \Rightarrow \mathtt{T}$ is one of the constructors of the inductive data type $\mathtt{T}$ and if $a_1\colon \tau_1, \ldots, a_k\colon \tau_k$ are valid terms, then $\mathtt{c}(a_1, \ldots, a_k)\colon \mathtt{T}$ is a valid term.
- If $x\colon \sigma$ is a variable and $b\colon \tau$ is a valid term with respect to $(\mathcal{X}, x\colon \sigma)$ and $\mathcal{SF}$, that is, x may occur free in b, then $[x]b\colon \sigma \to \tau$ is a valid term.
- If $f\colon \sigma \to \tau$ and $a\colon \sigma$ are valid terms, then $(f\ a)\colon \tau$ is a valid term.
- Let t be a valid term of an inductive data type $\mathtt{T}$. Let $0 \leqslant v$ and $0 \leqslant s \leqslant v$. Let $p_1, \ldots, p_v$ be exclusive patterns of type $\mathtt{T}$ and let $\mathcal{Y}_s$ be the set of pattern variables in p_s together with their type. Finally, let $e_1, \ldots, e_v$ be valid terms of type τ with respect to $(\mathcal{X}, \mathcal{Y}_1), \ldots, (\mathcal{X}, \mathcal{Y}_v)$,respectively,

and $\mathcal{SF}$. Then, a *case* expressions on t is a valid term of the form

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

and it has type τ.

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

The computation rules for terms are the usual ones. For a β-redex $([x]b\ a)$ we have the β-reduction rule $([x]b\ a) \rightsquigarrow b[x := a]$.

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

Cases expressions are computed by pattern-matching. Let us assume that the expression $t\colon \mathtt{T}$ is an instance of the pattern p_s. In other words, $t = p_s[\overline{y := a}]$, by which we mean the simultaneous substitution of all pattern variables $\overline{y}$ by terms $\overline{a}$, that is, $[y_1 := a_1, y_2 := a_2, \ldots]$. Then, we have the following computation rule for case expressions:

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

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

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

4.3 Fixed Point Function Definition

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

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

where $p_{1u}, \ldots, p_{lu}$ are patterns of type σ_u, for $0 \leqslant u \leqslant m$. We call a sequence of patterns $p_1 \cdots p_m$ of type $\sigma_1, \ldots, \sigma_m$, respectively, a *multipattern* for $\mathtt{f}$. Usually, we just say *pattern* when referring to *multipattern*, if there is no confusion. We extend the notions of linearness, exclusiveness and exhaustiveness from patterns to multipatterns in the normal way. The multipatterns that appear in the definition of a function $\mathtt{f}$ must be linear and mutually exclusive.

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

The computation rules for $\mathtt{f}$ are given by the different equations in its definition. If we want to compute the expression $\mathtt{f}(a_1, \ldots, a_m)$, we first have to find a pattern $p_1 \cdots p_m$ in the left-hand side of one of the equations defining $\mathtt{f}$ that matches the sequence $a_1 \cdots a_m$. Let $\mathtt{f}\ p_1 \cdots p_m = e$ be the corresponding equation. As $p_1 \cdots p_m$ matches $a_1 \cdots a_m$, let $\overline{b}$ be the sequence of terms that instantiate the pattern variables $\overline{y}$. Then, we have the following computation rule

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

If, on the contrary, there is no pattern that matches the sequence $a_1 \cdots a_m$, then the function $\mathtt{f}$ is undefined on that sequence.

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

The Fibonacci function is defined as

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

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

The concatenation of lists is defined as

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

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

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

We will see in section 6 that this definition is preferable when we translate this particular function into type theory. In general, it is better to put to the left of the symbol $\Rightarrow$ only the arguments that play an actual role in the recursion.

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

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

and defines n functions $\texttt{f}_1, \ldots, \texttt{f}_n$ at the same time. Let $\mathcal{Y}_{ji}$ be the set of pattern variables together with their type that occurs in the *ith* equation of the jth function, for $1 \leqslant j \leqslant n$ and $1 \leqslant i \leqslant l_j$, and let $\mathcal{SF}$ be the set that contains the specification of the functions $\texttt{f}_1, \ldots, \texttt{f}_n$. Then, each right-hand side e_{ji} must be a valid term of type τ_j with respect to $\mathcal{Y}_{ji}$ and $\mathcal{SF}$. That is, each function $\texttt{f}_\texttt{j}$ may only occur fully applied on the right-hand side of any of the equations.

The computation rules for $\texttt{f}_1, \ldots, \texttt{f}_n$ are defined as before. The difference is that now, we have to find the pattern within the definition of the function that we want to compute.

4.4 Conditional equations

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

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

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

The computation rule associated with a conditional expression consists simply in reducing the application of the function to the branch corresponding to the only true condition, if any. If, for a sequence of arguments instantiating the patterns, none of the conditional expressions evaluates to `true`, then `f` is undefined on that sequence. If $a_1 \cdots a_m$ is a sequence of arguments for `f` that matches the pattern $p_1 \cdots p_m$, that is, $\overline{a} = \overline{p}[\overline{y := b}]$, and if the condition $c_s[\overline{y := b}]$ evaluates to `true`, then we have the following computation rule

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

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

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

since for a sequence of arguments $a_1 \cdots a_m$ matching the pattern $p_1 \cdots p_m$, at most one of the conditions $c_1, \ldots, c_r$ evaluates to `true`. Hence, only one equation can be used to compute $\mathtt{f}(a_1, \ldots, a_m)$.

5 Turing Completeness

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

$$\begin{array}{l} \mathtt{fix}\ \mathtt{f}\colon \sigma \Rightarrow \tau \\ \qquad \mathtt{f}\ a = F\ \mathtt{f}\ a \end{array}$$

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

On the other hand, the alternative definition

$$\begin{array}{l} \mathtt{fix}\ \mathtt{f}\colon \Rightarrow \sigma \to \tau \\ \qquad \mathtt{f} = F\ \mathtt{f} \end{array}$$

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

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

To show that every recursive function can be defined, we exploit *Kleene normal form theorem* (see for example, Theorem 10.1 in [BM77] or Theorem 1.5.6 in [Phi92]):

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

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

where μ is the minimisation operator.

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

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

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

where $\neg$ is the boolean negation.

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

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

6 Translation into Type Theory

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

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

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

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

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

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

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

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

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

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

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

Let us start by discussing how to define the predicate fAcc. This predicate has at least one constructor for each of the equations in the definition of the function `f`. The number of constructors associated with each equation depends on the structure of the expressions c and e in the equation. All constructors associated with the equation $(*)$ produce a proof of $\mathsf{fAcc}(\widehat{p_1}, \ldots, \widehat{p_m})$, where $\widehat{p_u}$ is the straightforward translation[2] of p_u, for $1 \leqslant u \leqslant m$. The type of each of the constructors associated with the equation $(*)$ depends on the structure and on the recursive calls that occur in c and in e. The fact that at most one equation can be used for the computation of $\mathtt{f}(a_1, \ldots, a_m)$, with $a_u : \sigma_u$, and the way we break down the structure of c and e to establish the type of each

[2]As we will see later when we formally define the translation of an expression, patterns are translated in a straightforward way.

constructor, guarantees that at most one constructor can be used to build a proof of $\mathsf{fAcc}(\widehat{a_1}, \ldots, \widehat{a_m})$, where each $\widehat{a_u}$ is the type-theoretic translation of a_u.

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

However, not all case expressions introduce several constructors. Some case expressions, that we call *safe*, can be directly translated into type theory as case expressions. Safe case expressions are such that none of their branches introduce partiality, hence they can be straightforwardly translated into type theory without further analysis. The expression on which we perform case analysis might still introduce partiality. An example of a safe case expression is the following:

$$\mathtt{Cases}\ \mathtt{f}(n)\ \mathtt{of} \begin{cases} \mathtt{0} & \mapsto \mathtt{0} \\ \mathtt{s}(m) & \mapsto \mathtt{Cases}\ xs\ \mathtt{of} \begin{cases} \mathtt{nil} & \mapsto \mathtt{s(0)} \\ \mathtt{cons}(y, ys) & \mapsto \mathtt{n} + \mathtt{y} \end{cases} \end{cases}$$

where $\mathtt{f}\colon \mathtt{Nat} \Rightarrow \mathtt{Nat}$ is the function being defined, n is a natural number and xs is a list, and m, y and ys are fresh variables of the corresponding types.

First of all, let us explain the general idea of the translation. We associate a series of constructors for fAcc and a corresponding series of equations for f with each equation in the definition of $\mathtt{f}$. The most important part of the translation is the definition of the types of the constructors fAcc. For that purpose, we analyse the structure of the conditional expression c and of the right-hand side e of the equation and, from them, we construct a context of assumptions for the different constructors of the predicate.

We start with the context Γ comprising the variables introduced in the patters $p_1, \ldots, p_m$ of the equation; each variable is assumed with type $\widehat{\sigma}$ if σ is its type in functional programming. Then, we associate classes of context extensions Φ_c and Θ_e with the boolean expression c and with the defining term e, respectively. Putting these contexts together we get a context $\Gamma; \Phi_c; \Theta_e$ comprising all the assumptions needed for the corresponding constructor of fAcc. The extension Φ_c is such that the context $\Gamma; \Phi_c$ contains assumptions sufficient to make the condition c meaningful. Together with Φ_c, we get a translation $\widehat{c}$ of the condition.

This leads us to the final definition of the type for the corresponding constructor of the predicate fAcc:

$$\mathsf{f\_acc}_{c,e} \in (\Gamma; \Phi_c; q \in \widehat{c} = \mathsf{true}; \Theta_e)\mathsf{fAcc}(\widehat{p_1}, \ldots, \widehat{p_m}).$$

where $=$ is the propositional identity in type theory and true is the type-theoretic boolean value $\mathtt{true}$.

Simultaneously with the definition of the context extension Θ_e, we also get a translation $\widehat{e}$ of the term e itself. Then, the equation of f associated with the constructor $\mathsf{f\_acc}_{c,e}$ becomes

$$\mathsf{f}(\widehat{p_1}, \ldots, \widehat{p_m}, \mathsf{f\_acc}_{c,e}(\overline{x}, \overline{y}, q, \overline{z})) = \widehat{e}$$

where $\overline{x}$ is the sequence of variables assumed in Γ, $\overline{y}$ is the sequence of variables assumed in Φ_c, q is a variable of type $(\widehat{c} = \mathsf{true})$, and $\overline{z}$ is the sequence of variables assumed in Θ_e.

A single equation may be associated with several constructors of the accessibility predicate and, consequently, with several equations of the translated function. So, we associate a sequence $\mathcal{P}_c(\Gamma)$ of pairs $\langle \Phi_c, \widehat{c}\rangle$ of context extensions Φ_c and boolean terms $\widehat{c}$ in type theory with the boolean term c. Similarly, we associate a sequence $\mathcal{P}_e(\Gamma)$ of pairs $\langle \Theta_e, \widehat{e}\rangle$ with the term e. The definition of $\mathcal{P}_{\_}(\Gamma)$ is the same for the conditional expression c and for the term e, so we treat them together in the sequel.

In what follows, given an expression a, we write the sequence of pairs in $\mathcal{P}_a(\Gamma)$ between $\{,\}$, that is, we write $\mathcal{P}_a(\Gamma) \equiv \{\cdots\}$ when defining $\mathcal{P}_a(\Gamma)$. In addition, $\langle \Phi_a, t_a\rangle$ denotes a generic element of $\mathcal{P}_a(\Gamma)$ and $\#\mathcal{P}_a$ denotes the number of elements in $\mathcal{P}_a(\Gamma)$.

Sometimes a subterm of a contains bound variables with the same name as the variables in Γ. The definition of $\mathcal{P}_a(\Gamma)$ may require that those variables are introduced in the context extension. To avoid naming conflicts we assume, without always explicitly saying it, that those variables are renamed with a fresh name before being introduced in the context. The need of the renaming of the variables becomes clear if we consider the following equation:

$$\mathtt{f}\ x = \mathtt{Cases}\ x\ \mathtt{of}\ \begin{cases} \mathsf{0} & \mapsto e_{\mathsf{0}} \\ \mathsf{s}(x) & \mapsto e_{\mathsf{s}} \end{cases}$$

This equation presents no problem in functional programming. Any occurrence of the variable x in e_{s} is bound by the variable x in the pattern $\mathsf{s}(x)$, hence it does not refer to the variable x that occurs in the left-hand side of the equation. Thus, the binding $\mathsf{s}(x)$ shadows the variable x in the left-hand side of the equation inside the expression e_{s}. However, we need to be able to refer to both variables x in type theory. As it will become clear below, we might need to add $(x = \mathsf{s}(x))$ to the assumptions of one of the pairs in the definition of $\mathcal{P}_a(\Gamma)$. While the type $(x = \mathsf{s}(x))$ is empty, the type $(x = \mathsf{s}(y))$ might not be, for y a fresh variable name of the corresponding type.

We define $\mathcal{P}_a(\Gamma)$ by recursion on the structure of the term a. If $\langle \Phi, t\rangle \in \mathcal{P}_a(\Gamma)$, then the term t can be seen as the translation $\widehat{a}$ of a under the assumptions in $\Gamma; \Phi$. The reader can verify that if σ_a is the type of a in functional programming, then $\widehat{\sigma_a}$ is the type of t in the context $\Gamma; \Phi$.

$a \equiv z$: If the expression a is a variable, then $\mathcal{P}_a(\Gamma) \equiv \{\langle\ , z\rangle\}$.

$e \equiv \mathsf{c}(a_1, \ldots, a_o)$: Here, $0 \leqslant o$. First, we determine $\mathcal{P}_{a_1}(\Gamma), \ldots, \mathcal{P}_{a_o}(\Gamma)$ by structural recursion and then, we combine these sequences into the definition of $\mathcal{P}_a(\Gamma)$. For each $\langle\Phi_{a_1}, t_{a_1}\rangle$ in $\mathcal{P}_{a_1}(\Gamma)$, ..., $\langle\Phi_{a_o}, t_{a_o}\rangle$ in $\mathcal{P}_{a_o}(\Gamma)$, we add the pair $\langle\Phi_{a_1};\ldots;\Phi_{a_o},\ \mathsf{c}(t_{a_1},\ldots,t_{a_o})\rangle$ to $\mathcal{P}_a(\Gamma)$, where $\mathsf{c} \equiv \widehat{\mathtt{c}}$. That is, we add a pair to $\mathcal{P}_a(\Gamma)$ for each of the possible combinations of the elements in $\mathcal{P}_{a_1}(\Gamma),\ldots,\mathcal{P}_{a_o}(\Gamma)$. Formally, we have

$$\begin{aligned}\mathcal{P}_a(\Gamma) \equiv\ & \{\langle\Phi_{a_1};\ldots;\Phi_{a_o},\ \mathsf{c}(t_{a_1},\ldots,t_{a_o})\rangle \mid \\ & \quad \langle\Phi_{a_1}, t_{a_1}\rangle \in \mathcal{P}_{a_1}(\Gamma),\ldots,\langle\Phi_{a_o}, t_{a_o}\rangle \in \mathcal{P}_{a_o}(\Gamma)\}.\end{aligned}$$

$a \equiv \mathtt{f}(a_1,\ldots,a_m)$: Again, we first determine $\mathcal{P}_{a_1}(\Gamma)$, ..., $\mathcal{P}_{a_m}(\Gamma)$ by structural recursion. As before, we combine these sequences into the definition of $\mathcal{P}_a(\Gamma)$. In addition, we have to add the assumption corresponding to the recursive call $\mathtt{f}(a_1,\ldots,a_m)$, stating that the tuple $(\widehat{a_1},\ldots,\widehat{a_m})$ satisfies the predicate fAcc. Remember that $\mathsf{f} \equiv \widehat{\mathtt{f}}$ and that f takes an extra parameter, which is a proof that the input values satisfy the predicate fAcc. Hence, we have that

$$\begin{aligned}\mathcal{P}_a(\Gamma) \equiv\ & \{\langle\Phi_{a_1};\ldots;\Phi_{a_m}; h \in \mathsf{fAcc}(t_{a_1},\ldots,t_{a_m}),\ \mathsf{f}(t_{a_1},\ldots,t_{a_m},h)\rangle \mid \\ & \quad \langle\Phi_{a_1}, t_{a_1}\rangle \in \mathcal{P}_{a_1}(\Gamma),\ldots,\langle\Phi_{a_o}, t_{a_o}\rangle \in \mathcal{P}_{a_o}(\Gamma)\}.\end{aligned}$$

$a \equiv (a_1\ a_2)$: This case is treated similarly to the previous two cases.

$$\mathcal{P}_a(\Gamma) \equiv \{\langle\Phi_{a_1};\Phi_{a_2},\ t_{a_1}(t_{a_2})\rangle \mid \langle\Phi_{a_1}, t_{a_1}\rangle \in \mathcal{P}_{a_1}(\Gamma), \langle\Phi_{a_2}, t_{a_2}\rangle \in \mathcal{P}_{a_2}(\Gamma)\}.$$

$a \equiv [z]b$: Let σ be the type of z. We first calculate $\mathcal{P}_b(\Gamma; z \in \widehat{\sigma})$ recursively.

If $\mathcal{P}_b(\Gamma; z \in \widehat{\sigma}) = \{\langle\ , t_b\rangle\}$, that is, if $\mathcal{P}_b(\Gamma; z \in \widehat{\sigma})$ contains only one pair and the context extension in that pair is empty, then

$$\mathcal{P}_b(\Gamma) \equiv \{\langle\ , [z]\, t_b\rangle\}.$$

In other words, in this case, the method does not produce any assumptions, and the λ-abstraction can be directly translated into type theory.

Otherwise, let $\mathcal{P}_b(\Gamma; z \in \widehat{\sigma}) = \{\langle\Phi_{b1}, t_{b1}\rangle,\ldots,\langle\Phi_{b\#\mathcal{P}_b}, t_{b\#\mathcal{P}_b}\rangle\}$. To translate this term as a λ-abstraction in type theory, we must impose that the translation $\widehat{b}$ of the abstracted term b is well-defined for every value of the variable z. Therefore, the assumption generated by a must be the universal quantification over z of the all the assumptions for $\widehat{b}$.

Let $\Sigma\Phi$ be the conjunction of all the assumptions in a non-empty context Φ and let $\overline{y_\Phi}$ be the variables in Φ. We define

$$\mathcal{P}_a(\Gamma) \equiv \{\langle H \in (z \in \widehat{\sigma})\ \Sigma\Phi_{b1} + \ldots + \Sigma\Phi_{b\#\mathcal{P}_b},\ t_a\rangle\}$$

with

$$t_a \equiv [z]\ \mathtt{Cases}\ H(z)\ \mathtt{of}\ \begin{cases} \mathsf{in}_1(\langle\overline{y_{\Phi_{b1}}}\rangle) & \mapsto t_{b1} \\ \vdots & \\ \mathsf{in}_{\#\mathcal{P}_b}(\langle\overline{y_{\Phi_{b\#\mathcal{P}_b}}}\rangle) & \mapsto t_{b\#\mathcal{P}_b}\end{cases}$$

If $\#\mathcal{P}_b = 1$, then we do not need to construct a disjoint union type and we just put

$$\mathcal{P}_a(\Gamma) \equiv \{\langle H \in (z \in \widehat{\sigma})\, \Sigma\Phi_b,\ t_a\rangle\}$$

with

$$t_a \equiv \texttt{Cases}\ H(z)\ \texttt{of}\ \left\{\ \langle\overline{y_{\Phi_b}}\rangle \mapsto t_b.\right.$$

If, moreover, Φ_b contains only one assumption, then we do not need to construct a Σ-type. We can just call y_{Φ_b} the variable introduced in the assumption in Φ_b, and put

$$\mathcal{P}_b(\Gamma) \equiv \{\langle H \in (z \in \widehat{\sigma})\, \Phi_b,\ t_a\rangle\}$$

with

$$t_a \equiv \texttt{Cases}\ H(z)\ \texttt{of}\ \left\{\ y_{\Phi_b} \mapsto t_b.\right.$$

In the examples, we always use the simplest possible version.

$a \equiv \texttt{Cases}\ b\ \texttt{of} \begin{cases} p_1 \mapsto a_1 \\ \vdots \\ p_v \mapsto a_v \end{cases}$ **:** Here, $0 \leqslant v$. First, observe that variables, constructors and constructor applications are translated into their type-theoretic equivalents in a straightforward way. Hence, the translation $\widehat{p}$ of a pattern p is also straightforward. In addition, notice that if we compute $\mathcal{P}_p(\Gamma)$ we obtain $\{\langle\ ,\widehat{p}\rangle\}$.

If the different branches of the case expression do not contain recursive calls or do not introduce partiality, we can give a straightforward translation. We call such case expressions *safe* and we translate them directly as case expressions in type theory. Formally, a *safe* case expression $\texttt{Cases}\ b\ \texttt{of} \begin{cases} p_1 \mapsto a_1 \\ \vdots \\ p_v \mapsto a_v \end{cases}$ is such that the patterns $p_1, \ldots, p_v$ are exclusive and exhaustive and, for each branch in the case expression, $\mathcal{P}_{a_s}(\Gamma) = \{\langle\ , t_{a_s}\rangle\}$, for $1 \leqslant s \leqslant v$. Notice that, for a case expression to be safe, there should be no recursive call in any of the expressions $a_1, \ldots, a_v$.

For a safe case expression we define

$$\mathcal{P}_a(\Gamma) = \{\langle\Phi, t_a\rangle \mid \langle\Phi, t_b\rangle \in \mathcal{P}_b(\Gamma)\}$$

where

$$t_a = \texttt{Cases}\ t_b\ \texttt{of} \begin{cases} \widehat{p_1} \mapsto t_{a_1} \\ \vdots \\ \widehat{p_v} \mapsto t_{a_v} \end{cases}$$

If the case expression is not safe, each of the different branches imposes the need of a different constructor. Let $\overline{y_s}$ be the variables introduced by the pattern p_s and let $\overline{\sigma_s}$ be the types of those variables. Let $\overline{\widehat{y_s}}$ be a renaming of the variables in $\overline{y_s}$ by fresh variable names with respect to Γ. Observe that the renaming of the variables in $\overline{y_s}$ forces the corresponding renaming in $\overline{\sigma_s}$, which will be performed together with the translation of $\overline{\sigma_s}$ into its type-theoretic equivalent $\overline{\widehat{\sigma_s}}$.

Let us denote $\overline{(\widehat{y_s} \in \widehat{\sigma_s})}$ by Γ_s. As we have said before, each branch in the case expression imposes the need of at least one different constructor. Notice that each a_s may impose the need of several constructors, namely $\#\mathcal{P}_{a_s}$. Then, the number of constructors corresponding to the sth branch is also $\#\mathcal{P}_{a_s}$. The constructors associated with the sth branch should assume the variables introduced in the branch, that is, Γ_s. In addition, to ensure that these constructors are used only when we are inside the branch s, they should also assume $q_s \in \widehat{b} = \widehat{p_s}$, where q_s is a fresh variable name for each s. The expression b might also impose the need of several constructors and that it might contribute to the type of the constructors. Hence, as before, we should combine the elements in $\mathcal{P}_b(\Gamma)$ and in $\mathcal{P}_{a_s}(\Gamma;\Gamma_s)$ in all possible ways. Now, we determine $\mathcal{P}_b(\Gamma), \mathcal{P}_{a_1}(\Gamma;\Gamma_1), \ldots, \mathcal{P}_{a_v}(\Gamma;\Gamma_v)$ by structural recursion and then we define

$$\begin{aligned}\mathcal{P}_a(\Gamma) \equiv \ & \{\langle \Phi_b; \Gamma_s; q_s \in (t_b = \widehat{p_s}); \Phi_{a_s},\ t_{a_s}\rangle \mid \\ & \quad \langle \Phi_b, t_b\rangle \in \mathcal{P}_b(\Gamma), 1 \leq s \leq v, \langle \Phi_{a_s}, t_{a_s}\rangle \in \mathcal{P}_{a_s}(\Gamma;\Gamma_s)\}\end{aligned}$$

This completes the definition of $\mathcal{P}_a(\Gamma)$.

Let us now return to the translation of the function `f` in type theory. To complete the definitions of fAcc and f, whose types where introduced in $(*')$, we need to give the type of the different constructors of the predicate fAcc and the different equations that define the function f. We recall that the function `f` is defined by (conditional) equations. The shape of each equation that define `f` is given in $(*)$. Let us assume that $\overline{y}$ is the sequence of pattern variables in the equation with types $\overline{\tau}$. Let Γ be $\overline{(y \in \widehat{\tau})}$.

Using the definition we presented above, we determine $\mathcal{P}_c(\Gamma)$ and $\mathcal{P}_e(\Gamma)$. Observe that all the terms t_c in the sequence of pairs $\mathcal{P}_c(\Gamma)$ are boolean terms. To define the constructors of fAcc and the equations that define f, we should combine in all possible ways the elements in $\mathcal{P}_c(\Gamma)$ and $\mathcal{P}_e(\Gamma)$. Let $\langle \Phi_{cr}, t_{cr}\rangle$ be the rth element of $\mathcal{P}_c(\Gamma)$ and $\langle \Phi_{el}, t_{el}\rangle$ the lth element of $\mathcal{P}_e(\Gamma)$. We recall that patterns are straightforwardly translated into type theory. Then, the corresponding constructor of fAcc is as follows:

$$\mathsf{facc}_{rl} \in (\Gamma; \Phi_{cr}; q_r \in t_{cr} = \mathsf{true}; \Phi_{el})\mathsf{fAcc}(\widehat{p_1}, \ldots, \widehat{p_m}).$$

If the corresponding equation is a non-conditional equation, then $\mathcal{P}_c(\Gamma)$ is empty and the assumptions $\Phi_{cr}; q_r \in t_{cr} = \mathsf{true}$ are not present in the constructor. The presence or not of these premises is the only difference between the constructors associated with a conditional equation and the constructors associated with a non-conditional equation. Notice also that in the examples we gave in section 3, we converted c into a predicate rather than a boolean function. This is not a problem since we can always define $t'_{cr} \equiv t_{cr} = \mathsf{true}$.

The equation in the definition of f that corresponds to the above constructor is the following:

$$\mathsf{f}(\widehat{p_1}, \ldots, \widehat{p_m}, \mathsf{facc}_{rl}(\overline{y}, \overline{x_{\Phi_{cr}}}, q_r, \overline{x_{\Phi_{el}}}) = t_{el}.$$

This completes the definition of fAcc and f in type theory.

We have several observations at this point. First, notice that besides the introduction of the pattern variables of an equation, abstraction, recursive calls, non-safe case expressions and conditionals are the expressions that contribute to the type of a constructor. Observe also that if we have two or more syntactically equal recursive calls in an equation, our method will duplicate the assumptions corresponding to that call. This problem can be easily eliminated if we add an assumption corresponding to a recursive call into a sequence of assumptions only when that assumption has not yet been added to the sequence. This is what we have done in the examples we presented in section 3. The number of constructors associated with an equation is strongly related to the structure of the expressions in the equation, in particular to the case expressions and the $\lambda-$abstractions that are present in the equation. The user should keep this in mind when choosing the definition of a function. Finally, observe that the choice of where to put the symbol $\Rightarrow$ within the specification of the type of a function `f` makes a difference in its translation, since it determines the type of the corresponding fAcc.

Let us now analyse what actually happens when we translate a general recursive algorithm. Hence, let us consider one of the equations that define a general recursive function `f`. For the sake of generality, let us assume that we have nested recursive calls in the equation. For the sake of simplicity, let us assume that the equation is a non-conditional equation with no case expressions in the right-hand side. Finally, let us assume here that there are no recursive calls to `f` inside a $\lambda-$abstraction in the right-hand side of the equation. We consider that case later. Hence, we have an equation of the following form

$$\mathtt{f}\ p_1 \cdots\ p_m = \cdots\ \mathtt{f}(a_1, \ldots, \mathtt{f}(a'_1, \ldots, a'_m), \ldots, a_m) \cdots$$

Let us call $e_{\mathtt{f}}$ its right-hand side. As there are no case expressions in the equation, for any subexpression a of $e_{\mathtt{f}}$, $\mathcal{P}_a$ has only one element. In order to calculate $\mathcal{P}_{e_{\mathtt{f}}}(\Gamma)$, we first have to calculate $\mathcal{P}_{\mathtt{f}(a'_1,\ldots,a'_m)}(\Gamma)$

$$\mathcal{P}_{\mathtt{f}(a'_1,\ldots,a'_m)}(\Gamma) \equiv \{\langle \Phi_{a'_1}; \ldots; \Phi_{a'_m}; h \in \mathsf{fAcc}(\widehat{a'_1}, \ldots, \widehat{a'_m}),\ \mathsf{f}(\widehat{a'_1}, \ldots, \widehat{a'_m}, h)\rangle\}$$

Hence, $\mathtt{f}(\widehat{a'_1, \ldots, a'_m})$ is defined as $\mathsf{f}(\widehat{a'_1}, \ldots, \widehat{a'_m}, h)$. Now, we calculate

$$\begin{aligned}
&\mathcal{P}_{\mathtt{f}(a_1,\ldots,\mathtt{f}(a'_1,\ldots,a'_m),\ldots,a_m)}(\Gamma) \equiv \\
&\qquad\qquad \{\ \langle \Phi_{a_1}; \ldots; \Phi_{a'_1}; \ldots; \Phi_{a'_m}; h \in \mathsf{fAcc}(\widehat{a'_1}, \ldots, \widehat{a'_m}); \ldots; \Phi_{a_m}; \\
&\qquad\qquad\quad h' \in \mathsf{fAcc}(\widehat{a_1}, \ldots, \mathsf{f}(\widehat{a'_1}, \ldots, \widehat{a'_m}, h), \ldots, \widehat{a_m}), \\
&\qquad\qquad\quad \mathsf{f}(\widehat{a_1}, \ldots, \mathsf{f}(\widehat{a'_1}, \ldots, \widehat{a'_m}, h), \ldots, \widehat{a_m}, h')\ \rangle\ \}
\end{aligned}$$

Thus, the translation of $\mathtt{f}(a_1, \ldots, \mathtt{f}(a'_1, \ldots, a'_m), \ldots, a_m)$ is defined as the term $\mathsf{f}(\widehat{a_1}, \ldots, \mathsf{f}(\widehat{a'_1}, \ldots, \widehat{a'_m}, h), \ldots, \widehat{a_m}, h')$.

The constructor associated with the equation is

$$\begin{aligned}\mathsf{facc} \in\ &(\Gamma; \ldots; \Phi_{a_1}; \ldots; \Phi_{a'_1}; \ldots; \Phi_{a'_m}; h \in \mathsf{fAcc}(\widehat{a'_1}, \ldots, \widehat{a'_m}); \ldots; \Phi_{a_m};\\ &\quad h' \in \mathsf{fAcc}(\widehat{a_1}, \ldots, \mathsf{f}(\widehat{a'_1}, \ldots, \widehat{a'_m}, h), \ldots, \widehat{a_m}); \ldots\\ &\quad)\mathsf{fAcc}(\widehat{p_1}, \ldots, \widehat{p_m})\end{aligned}$$

and the corresponding equation in the definition of f would be

$$\begin{aligned}&\mathsf{f}(\widehat{p_1}, \ldots, \widehat{p_m}, \mathsf{facc}(\ldots, h, \ldots, h', \ldots)) =\\ &\qquad\qquad \cdots\ \mathsf{f}(\widehat{a_1}, \ldots, \mathsf{f}(\widehat{a'_1}, \ldots, \widehat{a'_m}, h), \ldots, \widehat{a_m}, h') \cdots\end{aligned}$$

Observe that the recursive calls to the function f are structurally smaller on the proof that the corresponding values satisfy the predicate fAcc. Notice that we have the same property even if there are no nested recursive calls. Hence, a general recursive definition is translated into type theory as a structurally smaller recursive definition.

Now, let us consider an equation with a $\lambda-$abstraction in the right-hand side and a recursive call to $\mathtt{f}$ inside the $\lambda-$abstraction. We have then an equation of the form

$$\mathtt{f}\ p_1 \cdots\ p_m = \cdots [z](\cdots \mathtt{f}(a_1, \ldots, a_m) \cdots) \cdots$$

Let us call e_λ the expression $(\cdots \mathtt{f}(a_1, \ldots, a_m) \cdots)$. Here, we first need to compute $\mathcal{P}_{[z]e_\lambda}(\Gamma)$. If σ is the type of z, we have that

$$\mathcal{P}_{[z]e\lambda}(\Gamma) \equiv \{\langle H \in (z \in \widehat{\sigma})\ \Sigma\Phi_{e_{\lambda 1}} + \ldots + \Sigma\Phi_{e_{\lambda \#\mathcal{P}_{e\lambda}}},\ \widehat{[z]e_\lambda}\rangle\}$$

where

$$\widehat{[z]e_\lambda} \equiv [z]\ \mathsf{Cases}\ H(z)\ \mathsf{of} \begin{cases} \mathsf{in}_1(\langle\overline{y_{\Phi_{e_{\lambda 1}}}}\rangle) & \mapsto t_{e_{\lambda 1}}\\ & \quad\vdots\\ \mathsf{in}_s(\ldots, h, \ldots) & \mapsto \cdots \mathsf{f}(\widehat{a_1}, \ldots, \widehat{a_m}, h) \cdots\\ & \quad\vdots\\ \mathsf{in}_{\#\mathcal{P}_{e\lambda}}(\langle\overline{y_{\Phi_{e_\lambda \#\mathcal{P}_{e\lambda}}}}\rangle) & \mapsto t_{e_{\lambda \#\mathcal{P}_{e\lambda}}}\end{cases}$$

Here, $1 \leqslant s \leqslant \#\mathcal{P}_{e_\lambda}$, $h \in \mathsf{fAcc}(\widehat{a_1}, \ldots, \widehat{a_m})$ and $t_{e_{\lambda s}}$ is the translation of the part of e_λ where the recursive call actually occurs. The context extension $\Phi_{e_\lambda s}$ contains the assumption $\mathsf{fAcc}(\widehat{a_1}, \ldots, \widehat{a_m})$. There can, of course, be recursive calls in any of the others $t_{e_{\lambda k}}$, with $1 \leqslant k \leqslant \#\mathcal{P}_{e_\lambda}$.

The constructor associated with the equation is

$$\mathsf{facc} \in (\Gamma; \ldots; H \in (z \in \widehat{\sigma})\ \Sigma\Phi_{e_{\lambda 1}} + \ldots + \Sigma\Phi_{e_\lambda \#\mathcal{P}_{e\lambda}}; \ldots)\mathsf{fAcc}(\widehat{p_1}, \ldots, \widehat{p_m})$$

and the corresponding equation in the definition of f would be

$$\mathsf{f}(\widehat{p_1}, \ldots, \widehat{p_m}, \mathsf{facc}(\ldots, H, \ldots)) = \cdots\ \widehat{[z]e_\lambda} \cdots$$

Although it is less obvious here, the recursive calls to the function f are, also in this case, structurally smaller on the proof that the corresponding values satisfy the predicate fAcc. To convince ourselves of this, let us analyse the term $\widehat{[z]e_\lambda}$. Observe that all the pattern variables in $\overline{y_{\Phi_{e_\lambda}k}}$ are structurally smaller than $H(z)$. In addition, the term $H(z)$ is considered structurally smaller than the term H. Hence, the variable h in the sth branch of the case expression is a term structurally smaller than H.

If, instead of a single function, we face the mutual definition of n functions

$$\begin{aligned}\texttt{mutual fix } \mathsf{f_1}:\, & \sigma_{11},\ldots,\sigma_{1m_1} \Rightarrow \tau_1 \\ & \vdots \\ \mathsf{f_n}:\, & \sigma_{n1},\ldots,\sigma_{nm_n} \Rightarrow \tau_n \\ & \vdots\end{aligned}$$

then, we need to define n special-purpose accessibility predicates and n type-theoretic functions with the following types:

$$\begin{aligned}&\mathsf{fAcc_1} \in (x_{11} \in \widehat{\sigma_{11}};\ldots;x_{1m_1} \in \widehat{\sigma_{1m_1}})\mathsf{Set} \\ &\qquad\vdots \\ &\mathsf{fAcc_n} \in (x_{n1} \in \widehat{\sigma_{n1}};\ldots;x_{nm_n} \in \widehat{\sigma_{nm_n}})\mathsf{Set} \\[1ex] &\mathsf{f_1} \in (x_{11} \in \widehat{\sigma_{11}};\ldots;x_{1m_1} \in \widehat{\sigma_{1m_1}}; h_1 \in \mathsf{fAcc}(x_{11},\ldots,x_{1m_1}))\widehat{\tau_1} \\ &\qquad\qquad\qquad\qquad\vdots \\ &\mathsf{f_n} \in (x_{n1} \in \widehat{\sigma_{n1}};\ldots;x_{nm_n} \in \widehat{\sigma_{nm_n}}; h_n \in \mathsf{fAcc}(x_{n1},\ldots,x_{nm_n}))\widehat{\tau_n}\end{aligned}$$

Similarly to what happens in the translation of a single function definition, if a function $\texttt{f}_\texttt{j}$ is defined by nested recursion, for $1 \leqslant j \leqslant n$, we should define the fAcc's and the f's simultaneously. Otherwise, we first define the fAcc's and we then use those predicates to define the f's.

Each special-purpose accessibility predicate $\mathsf{fAcc_j}$ and each function $\mathsf{f_j}$ is defined as for a single function. Observe that now, the case in the definition of $\mathcal{P}_a(\Gamma)$ that deals with recursive calls should consider the recursive calls to any of the n functions. Each recursive call is translated as in the definition of $\mathcal{P}_a(\Gamma)$.

7 Lazy, Strict and Totally Strict Semantics

We must be careful to state in what sense our type-theoretic translation of a functional program is equivalent to the original one. Given a program $\texttt{f}$ in $\mathcal{FP}$, our general method produces a pair consisting of a predicate fAcc and a function f that takes a proof that the input values satisfy the predicate as extra argument. For a program with type, for example, $\texttt{f}: \sigma \Rightarrow \tau$, we obtain $\mathsf{fAcc} \in (\widehat{\sigma})\mathsf{Set}$ and $\mathsf{f} \in (x \in \widehat{\sigma}; h \in \mathsf{fAcc}(x))\widehat{\tau}$ in type theory, where $\widehat{\sigma}$ and $\widehat{\tau}$ are the type-theoretic translation of σ and τ, respectively. Then, we would like to state:

> The program $\texttt{f}$ terminates on the input x if and only if $\mathsf{fAcc}(x)$ is provable. Moreover, if $h \in \mathsf{fAcc}(x)$ the output produced by the computation of $\texttt{f}(x)$ is $\mathsf{f}(x,h)$.

Unfortunately, this conjecture is not always true for *lazy* computational models. When evaluating an expression, a lazy computational model evaluates only the part of the expression that is necessary for the computation to continue. For example, in the expression $\mathtt{f}(n) \leqslant 0$ we do not need to fully evaluate $\mathtt{f}(n)$. If, at a certain stage, the computation produces the value $\mathtt{s}(e)$, where e is still an unevaluated expression, there is no need to further evaluate e to produce a result for $\mathtt{f}(n) \leqslant 0$, since we already know that the value of this expression must be `false`.

Similarly, in the definition of a recursive function, when we have a recursive equation of the form

$$\mathtt{f}\ \overline{p} = \cdots \mathtt{f}(\overline{a}) \cdots$$

the lazy evaluation strategy requires that $\mathtt{f}(\overline{a})$ is computed only if it is needed. Moreover, the computation is performed when it is needed.

On the other hand, a *strict* evaluation strategy requires that the arguments of a function are always fully evaluated before the function is computed. Hence, in the example above, $\mathtt{f}(\overline{a})$ must be computed before the computation of $\mathtt{f}(\overline{p})$ begins, even if the value of $\mathtt{f}(\overline{a})$ may not actually be needed for the final result.

Our translation of a functional program in type theory corresponds to a strict evaluation strategy. In the definition of fAcc, to prove $\mathsf{fAcc}(\overline{p})$ we first need to prove $\mathsf{fAcc}(\overline{a})$.

The distinction between lazy and strict evaluation strategy is relevant not only to the efficiency of computation, but also to the question of termination since a lazy program may terminate while the corresponding strict program diverges. Suppose the computation of $\mathtt{f}(\overline{a})$ diverges and that its value is not actually needed for the computation of $\mathtt{f}(\overline{p})$. Then, a lazy evaluation strategy would just ignore the call $\mathtt{f}(\overline{a})$ and produce a result anyway, while a strict evaluation strategy would try to compute $\mathtt{f}(\overline{a})$ and therefore diverge.

A good illustration of the difference between lazy and strict evaluation is the following mutual recursive definition, which is a variant of the one given in section 3.

$$
\begin{array}{ll}
\mathtt{mutual\ fix} & \mathtt{f}\colon \mathtt{Nat} \Rightarrow \mathtt{Nat} \\
& \mathtt{f}\ 0 = 0 \\
& \mathtt{f}\ \mathtt{s}(n) = \mathtt{f}(\mathtt{g}(n)) + \mathtt{g}(n) \\
& \\
& \mathtt{g}\colon \mathtt{Nat} \Rightarrow \mathtt{Nat} \\
& \mathtt{g}\ 0 = 0 \\
& \mathtt{g}\ \mathtt{s}(n) = \begin{cases} \mathtt{g}(\mathtt{f}(n)) + n & \text{if } \mathtt{f}(n) \leqslant n \\ 0 & \text{if } \mathtt{f}(n) > n \end{cases}
\end{array}
$$

Here is a table with the values of `f` and `g` for a few initial inputs:

input value x	$\mathtt{f}(x)$	$\mathtt{g}(x)$
0	0	0
1	0	0
2	0	1
3	1	2
4	2	3
5	4	5
6	9	8
7	8	0

input value x	$\mathtt{f}(x)$	$\mathtt{g}(x)$
8	0	0
9	0	8
10	8	9
11	9	10
12	18	19
13	`undefined`	0
14	0	0
15	0	14

First of all, let us consider the computation of $\mathtt{f}(7)$. From the definition of $\mathtt{f}$ and $\mathtt{g}$, we have that $\mathtt{f}(7) = \mathtt{f}(\mathtt{g}(6)) + \mathtt{g}(6) = \mathtt{f}(8) + 8$. This shows that $\mathtt{f}$ and $\mathtt{g}$ cannot be defined by primitive recursion since $\mathtt{f}(7)$ calls itself on a larger argument. Therefore, this definition is a good candidate to be translated with our method.

As it is shown in the table, the computation of $\mathtt{f}(13)$ diverges, independently of what evaluation strategy we adopt. However, if we partially evaluate it, we obtain

$$\mathtt{f}(13) = \mathtt{f}(\mathtt{g}(12)) + \mathtt{g}(12) = \mathtt{f}(19) + 19.$$

Now, when computing $\mathtt{g}(14)$, we have to decide which of the two branches of the definition of $\mathtt{g}$ to take depending on whether $\mathtt{f}(13) \leqslant 13$ or $\mathtt{f}(13) > 13$. A strict evaluation strategy would, at this point, try to fully evaluate $\mathtt{f}(13)$ and diverge. On the other hand, in a lazy evaluation strategy, with the right definition of $+$ and $>$, one step of the computation of $\mathtt{f}(13)$ would be enough to determine that the value of

$$\mathtt{f}(13) = \mathtt{f}(19) + 19 > 13$$

is `true`. Therefore, the second branch in the definition of $\mathtt{g}$ would be taken and the desired result would be $\mathtt{g}(14) = 0$.

Using our method we obtain the following two predicates and two functions.

$\mathsf{fAcc} \in (m \in \mathsf{N})\mathsf{Set}$
$\quad \mathsf{f\_acc_0} \in \mathsf{fAcc}(0)$
$\quad \mathsf{f\_acc_s} \in (n \in \mathsf{N}; h_1 \in \mathsf{gAcc}(n); h_2 \in \mathsf{fAcc}(\mathsf{g}(n, h_1)))\mathsf{fAcc}(\mathsf{s}(n))$

$\mathsf{gAcc} \in (m \in \mathsf{N})\mathsf{Set}$
$\quad \mathsf{g\_acc_0} \in \mathsf{gAcc}(0)$
$\quad \mathsf{g\_acc_{s1}} \in (n \in \mathsf{N}; h_1 \in \mathsf{fAcc}(n); q \in (\mathsf{f}(n, h_1) \leqslant n); h_2 \in \mathsf{gAcc}(\mathsf{f}(n, h_1))$
$\qquad\qquad)\mathsf{gAcc}(\mathsf{s}(n))$
$\quad \mathsf{g\_acc_{s2}} \in (n \in \mathsf{N}; h_1 \in \mathsf{fAcc}(n); q \in (\mathsf{f}(n, h_1) > n))\mathsf{gAcc}(\mathsf{s}(n))$

$$
\begin{aligned}
&\mathsf{f} \in (m \in \mathsf{N}; \mathsf{fAcc}(m))\mathsf{N}\\
&\quad \mathsf{f}(0, \mathsf{f\_acc_0}) = 0\\
&\quad \mathsf{f}(\mathsf{s}(n), \mathsf{f\_acc_s}(n, h_1, h_2)) = \mathsf{f}(\mathsf{g}(n, h_1), h_2) + \mathsf{g}(n, h_1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{g} \in (m \in \mathsf{N}; \mathsf{gAcc}(m))\mathsf{N}\\
&\quad \mathsf{g}(0, \mathsf{g\_acc_0}) = 0\\
&\quad \mathsf{g}(\mathsf{s}(n), \mathsf{g\_acc_{s1}}(n, h_1, q, h_2)) = \mathsf{g}(\mathsf{f}(n, h_1), h_2) + n\\
&\quad \mathsf{g}(\mathsf{s}(n), \mathsf{g\_acc_{s2}}(n, h_1, q)) = 0
\end{aligned}
$$

Then, to compute g on 14, we must first prove $\mathsf{gAcc}(14)$. If such a proof exists, it must be constructed using either $\mathsf{g\_acc_{s1}}$ or $\mathsf{g\_acc_{s2}}$. Both constructors require a proof $h_1 \in \mathsf{fAcc}(13)$. But $\mathsf{fAcc}(13)$ cannot be proved, since `f`(13) diverges. Thus, $\mathsf{gAcc}(14)$ cannot be proved either and g is not defined on 14.

Therefore, our method corresponds to a strict semantics for functional programs.

An additional problem arises if a λ-abstraction occurs in the right-hand side of one of the equations in the definition of a recursive program. A strict semantics requires that, for an expression to be defined, all its subexpressions should be defined. This means that if the definition of a function `f` contains an equation of the form

$$\texttt{f}\ p_1\ \cdots\ p_m = e,$$

to evaluate `f` on arguments that instantiate the patterns $p_1, \ldots, p_m$, we must strictly evaluate e. In other words, if the computation of any subexpression of e diverges, then the computation of e also diverges, independently of whether the value of that subexpression is needed for the computation of e or not. In terms of definedness, we require that all the subexpressions of e are defined for e to be defined.

Then, a problem arises if e contains some λ-abstraction. Suppose that e is of the form

$$e = \cdots [x]e' \cdots .$$

According to our interpretation, the subexpression $[x]e'$ must be defined for e to be defined. Notice however that $[x]e'$ denotes a function, and that functions are defined if their values are defined on every input. In other words, $[x]e'$ is defined if e' is defined for every value of x. This amounts to requiring that, whenever a λ-abstraction occurs in the right-hand side of an equation, the corresponding function must be total. That is why we call this kind of semantics *totally strict*. Since the totality of recursive functions is in general undecidable, the totally strict semantics is not a computational model of functional programming. This is why even in functional programming languages with strict semantics, a higher type term is considered computed whenever it has been reduced to a λ-abstraction form, and not when the corresponding function is total.

In our method, we need to translate e into type theory to be able to translate the equation above. Since there are no partially defined terms in type theory, all subexpressions of e must be translated into totally defined terms. Therefore, total strictness is the appropriate semantics for $\mathcal{FP}$.

This point is illustrated by the following recursive function.

$$
\begin{array}{l}
\texttt{fix fln}\colon \texttt{List Nat} \Rightarrow \texttt{Nat} \\
\quad \texttt{fln nil} = 0 \\
\quad \texttt{fln cons}(n,l) = \;\; \texttt{Cases (div2 (fln}(l)\texttt{)) of} \\
\qquad\qquad\qquad\qquad\quad \begin{cases} 0 & \mapsto \texttt{fln}(l)+1 \\ \texttt{s}(m) & \mapsto \texttt{fln(map }[x]\texttt{fln(cons}(x+\texttt{s}(m),l))\ l) \end{cases}
\end{array}
$$

In the second case of the second equation, we have a λ-abstraction as the functional argument of the function `map`. A recursive call to the function `fln` occurs in the scope of this abstraction. This causes the requirement of an infinite number of termination assumptions for the function. In fact, when we apply our method to this example, we obtain

$$
\begin{array}{l}
\mathsf{flnAcc} \in (\mathsf{List}(\mathsf{N}))\mathsf{Set} \\
\quad \mathsf{fln\_acc}_0 \in \mathsf{flnAcc}(\mathsf{nil}) \\
\quad \mathsf{fln\_acc}_{\mathsf{s}1} \in (n \in \mathsf{N}; l \in \mathsf{List}(\mathsf{N}); h_1 \in \mathsf{flnAcc}(l); q \in (\mathsf{div2}(\mathsf{fln}(l,h_1)) = 0) \\
\qquad\qquad\quad)\mathsf{flnAcc}(\mathsf{cons}(n,l)) \\
\quad \mathsf{fln\_acc}_{\mathsf{s}2} \in (n \in \mathsf{N}; l \in \mathsf{List}(\mathsf{N}); h_1 \in \mathsf{flnAcc}(l); \\
\qquad\qquad\quad m \in \mathsf{N}; q \in (\mathsf{div2}(\mathsf{fln}(l,h_1)) = \mathsf{s}(m)); \\
\qquad\qquad\quad H \in (x \in \mathsf{N})\mathsf{flnAcc}(\mathsf{cons}(x+\mathsf{s}(m),l)); \\
\qquad\qquad\quad h_2 \in \mathsf{flnAcc}(\mathsf{map}([x]\mathsf{fln}(\mathsf{cons}(x+\mathsf{s}(m),l),H(x)),l)) \\
\qquad\qquad\quad)\mathsf{flnAcc}(\mathsf{cons}(n,l))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fln} \in (l \in \mathsf{List}(\mathsf{N}); \mathsf{flnAcc}(l))\mathsf{N} \\
\quad \mathsf{fln}(\mathsf{nil},\mathsf{fln\_acc}_0) = 0 \\
\quad \mathsf{fln}(\mathsf{cons}(n,l),\mathsf{fln\_acc}_{\mathsf{s}1}(n,l,h_1,q)) = \mathsf{fln}(l,h)+1 \\
\quad \mathsf{fln}(\mathsf{cons}(n,l),\mathsf{fln\_acc}_{\mathsf{s}2}(n,l,h_1,m,q,H,h_2)) = \\
\qquad \mathsf{fln}(\mathsf{map}([x]\mathsf{fln}(\mathsf{cons}(x+\mathsf{s}(m),l),H(x)),l),h_2)
\end{array}
$$

To compute fln on a non-empty list $\mathsf{cons}(n,l)$ we have to prove $\mathsf{flnAcc}(\mathsf{cons}(n,l))$. For this, we must use either the constructor $\mathsf{fln\_acc}_{\mathsf{s}1}$ or the constructor $\mathsf{fln\_acc}_{\mathsf{s}2}$. In both cases, we first need to give a proof $h_1 \in \mathsf{flnAcc}(l)$. Then, whenever $\mathsf{div2}(\mathsf{fln}(l,h_1))$ is not zero, that is, it is the successor of m, we must also give a proof

$$H \in (x \in \mathsf{N})\mathsf{flnAcc}(\mathsf{cons}(x+\mathsf{s}(m),l))$$

and therefore, we must prove $\mathsf{flnAcc}(\mathsf{cons}(x+\mathsf{s}(m),l))$ for every x. Since we have an infinite number of x's, this interpretation does not correspond to a viable operational semantics, because we cannot compute all these values in practice.

An analysis of the algorithm shows that the assumption H is unnecessarily strong. It requires the function

$$[x]\texttt{fln}(\texttt{cons}(x+\texttt{s}(m),l))$$

to be defined everywhere. However, in practice, since the function above is given as the functional argument of `map` and since the second argument of `map` is l, we just need the function to be defined on the elements of l.

Our translation does not analyse the behaviour of the occurrences of other functions in the definition. Thus, it does not try to determine what `map` does with its arguments. Instead, it considers the worst case scenario, that is, `map` could use its arguments in any possible way. Therefore, the translation requires that the function argument is defined for every value. It is possible to modify the definition of the function `fln` to force the interpretation to look into the definition of `map` by defining `fln` by mutual recursion with a specialised version of `map`.

$$
\begin{array}{ll}
\texttt{mutual fix} & \texttt{map\_fln}\colon \texttt{Nat}, \texttt{List Nat}, \texttt{List Nat} \Rightarrow \texttt{List Nat} \\
& \texttt{map\_fln}\ m\ l_1\ \texttt{nil} = \texttt{nil} \\
& \texttt{map\_fln}\ m\ l_1\ \texttt{cons}(n, l_2) = \\
& \quad\quad \texttt{cons}(\texttt{fln}(\texttt{cons}(n + \texttt{s}(m), l_1)), \texttt{map\_fln}(m, l_1, l_2)) \\
& \\
& \texttt{fln}\colon \texttt{List Nat} \Rightarrow \texttt{Nat} \\
& \texttt{fln nil} = 0 \\
& \texttt{fln cons}(n, l) = \ \texttt{Cases (div2 (fln}(l)\texttt{)) of} \\
& \quad\quad\quad\quad\quad\quad\quad \begin{cases} 0 & \mapsto \texttt{fln}(l) + 1 \\ \texttt{s}(m) & \mapsto \texttt{fln}(\texttt{map\_fln}(m, l, l)) \end{cases}
\end{array}
$$

The reader can verify that when we apply the translation for mutual recursive functions to this example, we get a much better condition for the termination of `fln`. In the case where $(\texttt{div2 fln}(l)) = \texttt{s}(m)$, we require a proof of $\mathsf{map\_flnAcc}(m, l, l)$, which is equivalent to $\mathsf{flnAcc}(\mathsf{cons}(x + \mathsf{s}(m), l))$ for all elements x in l, but not for every natural number x.

From this example, it is clear that it is a good programming style, in terms of our type-theoretic translation, to avoid λ-abstractions inside the definition of a recursive function in $\mathcal{FP}$. The user should instead try to replace every λ-abstraction with a new function mutually defined with the original one. If the function we want to formalise has the specification

$$\texttt{fix f}\colon \sigma \Rightarrow \tau_1 \rightarrow \tau_2$$

an easier solution might be to define it as

$$\texttt{fix f}\colon \sigma, \tau_1 \Rightarrow \tau_2$$

In this way, the fAcc predicate contains an unnecessary argument but we avoid the need of a λ-abstraction in the right-hand side of the equations.

This said, the formal definition of the semantics of $\mathcal{FP}$ is straightforward. This semantics consists in a mathematical interpretation of every function definable in $\mathcal{FP}$. We assume that each data type σ is interpreted as a set $\ulcorner\sigma\urcorner$. A program $\texttt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau$ is interpreted as a partial function

$$[\![\texttt{f}]\!] \in \ulcorner\sigma_1\urcorner \otimes \cdots \otimes \ulcorner\sigma_m\urcorner \rightarrow_\perp \ulcorner\tau\urcorner$$

where $\otimes$ denotes the cartesian product of sets and $A \rightarrow_\perp B$ denotes the set of partial functions from a set A to a set B.

Definition 4. Given the definition of a functional program $\texttt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau$ in $\mathcal{FP}$, we define $[\![\texttt{f}]\!]$ as the least fixed point of an operator mapping partial functions to partial functions.

$$F\colon (\ulcorner\sigma_1\urcorner \otimes \cdots \otimes \ulcorner\sigma_m\urcorner \to_\perp \ulcorner\tau\urcorner) \to (\ulcorner\sigma_1\urcorner \otimes \cdots \otimes \ulcorner\sigma_m\urcorner \to_\perp \ulcorner\tau\urcorner)$$

The definition of F is standard, except for the fact that the value of $F(\texttt{f}, \overline{a})$ should be undefined whenever an occurrence of a λ-abstraction within the definition of $\texttt{f}$ is instantiated with a partial function.

For the mutual recursive definition of the functions $\texttt{f}_\texttt{1}, \ldots, \texttt{f}_\texttt{n}$, the associated operator maps n-tuples of partial functions to n-tuples of partial functions,

$$\begin{aligned} F\colon\ & (\ulcorner\sigma_{11}\urcorner \otimes \cdots \otimes \ulcorner\sigma_{1m_1}\urcorner \to_\perp \ulcorner\tau_1\urcorner) \otimes \cdots \otimes (\ulcorner\sigma_{n1}\urcorner \otimes \cdots \otimes \ulcorner\sigma_{nm_n}\urcorner \to_\perp \ulcorner\tau_n\urcorner) \to \\ & (\ulcorner\sigma_{11}\urcorner \otimes \cdots \otimes \ulcorner\sigma_{1m_1}\urcorner \to_\perp \ulcorner\tau_1\urcorner) \otimes \cdots \otimes (\ulcorner\sigma_{n1}\urcorner \otimes \cdots \otimes \ulcorner\sigma_{nm_n}\urcorner \to_\perp \ulcorner\tau_n\urcorner). \end{aligned}$$

and the partial function denoted by $[\![\texttt{f}_\texttt{i}]\!]$ is the i-th component of the least fixed point of F.

In the definition of $[\![\texttt{f}]\!]$, we did not precisely specify how to construct the operator F. However, this construction is the natural one, once we have pointed out that λ-abstractions must be interpreted totally strictly.

A limit case occurs when the sequence of types to the left of the symbol $\Rightarrow$ is empty, that is, for function definitions with the specification $\texttt{f}\colon \Rightarrow \tau$. In this case, the domain of the operator F is $\mathbf{1} \to_\perp \ulcorner\tau\urcorner$, where $\mathbf{1}$ is a singleton set. An element of $\mathbf{1} \to_\perp \ulcorner\tau\urcorner$ can be seen as a partial element of $\ulcorner\tau\urcorner$ in the sense that it is either undefined or it is defined and denotes an element of $\ulcorner\tau\urcorner$. So, here, the operator F maps partial elements of $\ulcorner\tau\urcorner$ to partial elements of $\ulcorner\tau\urcorner$. Notice that if $\texttt{f}$ occurs in the right-hand side of the (necessarily unique) equation in its definition, then, according to our strict interpretation, F would produce an undefined result when applied to an undefined argument. It follows that the least fixed point of F is an undefined element of $\ulcorner\tau\urcorner$. In conclusion, such a definition of $\texttt{f}$ does not actually define anything.

We can now state that the translation of the functions in $\mathcal{FP}$ into type theory is complete with respect to the totally strict semantics.

Theorem 3. Let $\texttt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau$ be a function in $\mathcal{FP}$. Let fAcc and f be the special accessibility predicate for $\texttt{f}$ and the type-theoretic version of $\texttt{f}$, respectively. Then, f defines the same function as $[\![\texttt{f}]\!] \in \ulcorner\sigma_1\urcorner \otimes \cdots \otimes \ulcorner\sigma_m\urcorner \to_\perp \ulcorner\tau\urcorner$, that is, for every sequence of arguments $t_1 \in \widehat{\sigma_1}, \ldots, t_m \in \widehat{\sigma_m}$ we have that

$$\mathsf{fAcc}(t_1, \ldots, t_m) \text{ is provable} \quad \Longleftrightarrow \quad [\![\texttt{f}]\!] \text{ is defined on } (t_1, \ldots, t_m)$$

and if $h \in \mathsf{fAcc}(t_1, \ldots, t_m)$, then

$$\mathsf{f}(t_1, \ldots, t_m, h) = [\![\texttt{f}]\!](t_1, \ldots, t_m).$$

8 Conclusions

We described a method to translate a vast class of algorithms from functional programming into type theory. We defined the class $\mathcal{FP}$ of algorithms to which the method applies. This class is large enough to allow the implementation of all partial recursive functions. We gave a formal definition of the translation of the elements of $\mathcal{FP}$ in type theory. Finally, we proved that the translation is sound and complete with respect to a certain mathematical model, that we call *totally strict*.

Future work will try to improve the translation of lambda abstractions, that is for the moment unsatisfactory. We also plan to develop the idea briefly explained at the end of section 3 and use impredicative type theory to formalise the method using an explicit type constructor for partial functions.

8.1 Related Work

There are few studies on formalising general recursion in type theory.

In [Nor88], Nordström uses the predicate Acc for that purpose.

In a different setting, Finn et al [FFL97] treat general recursive functions in a very similar way to our approach. Except for case expressions, for which they consider the conjunction of all the assumptions that arise in the different branches of the case expression instead of considering each branch in a separate way as we do, an algorithm is analysed in [FFL97] as it is in our method. Moreover, Finn et al give a similar interpretation when bound variables are present in the right-hand side of the equations and they arrive to similar conclusions about the semantics associated to the formalisation of their programs. In addition, they have similar problems when using higher order functions in the definition of other functions. However, the different settings in which both works are performed give rise to some differences. A function `f` is formalised in [FFL97] with the type that it has in a functional programming language, without the need of our extra parameter fAcc as part of the type of f. However, f obeys its definition in [FFL97] provided that its arguments can be proven to be in the domain of the function, which is called DOM′f. Once (and if) the function has been proven total, one can forget about DOM′f, which is not possible in type theory. Another important difference is that in [FFL97], an application (f e) is always considered defined since the cases in which (`f` e) is not defined are considered as returning an unknown value of the corresponding type. However, this causes some problems since the semantics they use is not capable of reflecting this distinction and some times one can prove things like f $0 = 0$ when `f` 0 diverges.

In [DDG98], Dubois and Viguié Donzeau-Gouge take also a similar approach to the problem. They also formalise an algorithm with a predicate that characterises the domain of the algorithm and the formalisation of the algorithm itself. However, they consider neither case expressions nor $\lambda-$abstractions as possible expressions, which simplifies the translation a lot. In addition, they only present the translation for expressions in canonical form which also helps in the simplification. The most important difference is their use of post-conditions. In order to be able to deal with nested recursion without the need of simultaneous inductive-recursive definitions, they require that, together with the algorithm, the user provides a post-condition that characterises the results of the algorithm.

Balaa and Bertot [BB00] use fix-point equations to obtain the desired equalities for the recursive definitions. The solution they present is rather complex and it does not really succeed in separating the actual algorithms and their termination proofs. In a later work [BB02], Balaa and Bertot use fix-points again to approach the problem. Their new solution produces nicer formalisations and although one has to provide proofs concerning the well-foundedness of the recursive calls when one defines the algorithms, there is a clear separation between the algorithms and these proofs. In any case, it is not very clear how their methods can be used to formalise partial or nested recursive algorithms.

In a recent work, Bertot et al [BCB02] present a technique to encode the method we described in [BC01] for partial and nested algorithms in type theories that do not support Dybjer's schema for simultaneous inductive-recursive definitions. They do so by combining the way we define our special-accessibility predicate with the functionals in [BB00].

Other relevant publications that treat the problem of formalising partial functions and proving termination are [Abe02], [MM02], [dB94], and [BFG$^+$00].

References

[Abe02] A. Abel. Termination checking with types - Strong normalization for Mendler-style course-of-value recursion. Technical Report 0201, Institut für Informatik, Ludwig-Maximilians - Universität München, 2002.

[Acz77] P. Aczel. An Introduction to Inductive Definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland Publishing Company, 1977.

[Bar92] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2*, pages 117–309. Oxford University Press, 1992.

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

[BB02] A. Balaa and Y. Bertot. Fonctions récursives générales par itération en théorie des types. *Journées Francophones des Langages Applicatifs - JFLA02, INRIA*, January 2002.

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

[BCB02] Y. Bertot, V. Capretta, and K. Das Barman. Type-theoretic functional semantics. In *Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002*, 2002.

[BFG$^+$00] G. Barthe, M.J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. Under consideration for publication in Math. Struct. in Comp. Science, December 2000.

[BG01] H. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 18, pages 1149–1238. Elsevier Science Publishers, 2001.

[BM77] J. L. Bell and M. Machover. *A course in mathematical logic.* North-Holland, 1977.

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

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

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

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

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

[dB94] N.G. de Bruijn. Computer program semantics in space and time. In J.H. Geuvers, R.P. Nederpelt, and R.C. de Vrijer, editors, *Selected Papers on Automath*, number 133 in Studies in Logic and the Foundations of Mathematics, pages 947–972. North-Holland, Amsterdam, 1994.

[DDG98] C. Dubois and V. Viguié Donzeau-Gouge. A step towards the mechanization of partial functions: Domains as inductive predicates. In M. Kerber, editor, *CADE-15, The 15th International Conference on Automated Deduction*, pages 53–62, July 1998. WORKSHOP Mechanization of Partial Functions.

[dMJB$^+$01] P. de Mast, J.-M. Jansen, D. Bruin, J. Fokker, P. Koopman, S. Smetsers, M. van Eekelen, and R. Plasmeijer. *Functional Programming in Clean.* Computing Science Institute, University of Nijmegen, 2001.

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

[FFL97] S. Finn, M.P. Fourman, and J. Longley. Partial functions in a total setting. *Journal of Automated Reasoning*, 18(1):85–104, 1997.

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

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

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

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

[MM02] C. McBride and J. McKinna. The view from the left, 2002. Under consideration for publication in Journal of Functional Programming.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML.* MIT Press, 1997.

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

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

[Phi92] J. C. C. Phillips. *Recursion Theory*, pages 79–187. Oxford University Press, 1992.

[SU98] M. H. B. Sørensen and P. Urzyczyn. Lectures on the curry-howard isomorphism. Available as DIKU Rapport 98/14, 1998.