Math. Struct. in Comp. Science (2005), *vol.* 15, *pp.* 671–708.
doi:10.1017/S0960129505004822 Printed in the United Kingdom

Modelling general recursion in type theory

ANA BOVE[†] and VENANZIO CAPRETTA[‡]

[†]*Department of Computing Science, Chalmers University of Technology, 412 96 Göteborg, Sweden*
Email: bove@cs.chalmers.se
[‡]*Department of Mathematics and Statistics, University of Ottawa, 585 King Edward Ave., Ottawa, ON, K1N 6N5, Canada*
Email: venanzio.capretta@mathstat.uottawa.ca

Received 13 February 2003; revised 8 January 2005

Constructive type theory is an expressive programming language in which both algorithms and proofs can be represented. A limitation of constructive type theory as a programming language is that only terminating programs can be defined in it. Hence, general recursive algorithms have no direct formalisation in type theory since they contain recursive calls that satisfy no syntactic condition guaranteeing termination. In this work, we present a method to formalise general recursive algorithms in type theory. Given a general recursive algorithm, our method is to define 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. The method separates the computational and logical parts of the definitions and thus the resulting type-theoretic algorithms are clear, compact and easy to understand. They are as simple as their equivalents in a functional programming language, where there is no restriction on recursive calls. Here, we give a formal definition of the method and discuss its power and its limitations.

1. Introduction

Constructive type theory is a very expressive programming language with dependent types (see, for example, Martin-Löf (1984) and Coquand and Huet (1988)). According to the Curry–Howard isomorphism (Howard 1980; Sørensen and Urzyczyn 1998), logic can also be represented in it by identifying propositions with types and proofs with terms of the corresponding types. Therefore, we can encode in a type a complete specification, requiring 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.

On the other hand, functional programming languages like Standard ML (Milner *et al.* 1997), Haskell (Jones 2003) and Clean (de Mast *et al.* 2001) are less expressive in the sense that they do not have dependent types and they cannot represent logic. Moreover, the existing frameworks for reasoning about the correctness of Haskell-like programs

are weaker than the framework provided by type theory, and it is the responsibility of the programmer to write correct programs. However, functional programming languages are computationally stronger because they impose no restriction on recursive programs, allowing the definition of general recursive algorithms. In addition, functional programs are usually short and self-explanatory.

General recursive algorithms are defined by equations in which the recursive calls are not required to have structurally smaller arguments. In other words, the recursive calls are performed on objects that satisfy no syntactic condition guaranteeing termination. Hence, there is no direct way of formalising algorithms of this kind in type theory.

The standard way to handle (terminating) general recursion in constructive type theory is to use a well-founded recursion principle derived from the accessibility predicate Acc (Aczel 1977; Nordström 1988; Balaa and Bertot 2000). 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, distracting our attention from the computational part of the algorithm (see, for example, Bove (1999), where we present the formalisation of a unification algorithm over lists of term pairs using the standard accessibility predicate Acc). In addition, partial algorithms cannot be represented in this way.

To bridge the gap between programming in type theory and programming in a functional language, we have developed a method for formalising 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. Given a general recursive algorithm, our method is to define an inductive special-purpose accessibility predicate that characterises the inputs on which the algorithm terminates. We can think of this predicate as the domain of the algorithm. The type-theoretic version of the algorithm is 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. Definitions of this kind are not allowed in ordinary type theory, but they are provided in type theories extended with Dybjer's schema for simultaneous induction-recursion (Dybjer 2000).

The method was introduced in Bove (2001) to formalise simple general recursive algorithms in constructive type theory (by *simple* we mean non-nested and non-mutually recursive). It was extended in Bove and Capretta (2001) to treat nested recursion, and in Bove (2002b) to treat mutually recursive algorithms, nested or not. All these papers have been collected in the first author's Ph.D. thesis (Bove 2002a), which also includes an earlier version of this paper. A tutorial on the method can also be found in Bove (2003). Since our method separates the computational part from the logical part of a definition, formalising partial functions becomes possible (Bove and Capretta 2001; Bove 2003). Proving that a certain function is total amounts to proving that the corresponding special-purpose accessibility predicate (or domain predicate) is satisfied by every input.

In previous publications (Bove 2001; Bove and Capretta 2001; Bove 2002b; Bove 2003), we have presented our method purely by means of examples. The purpose of the current paper 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 equations where the recursive calls 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 synonyms.

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 general recursive algorithms in type theory. In Section 4, we define the class $\mathscr{FP}$ of recursive definitions that can be translated into type theory by applying our method. In Section 5, we formally describe our method for translating general recursive functions into type theory. In Section 6, we discuss the operational semantics of functional programs and prove that our translation is sound with respect to strict semantics. In addition, we discuss limitations of our method that make a completeness result more difficult. In Section 7, we extend the class $\mathscr{FP}$ to allow guarded equations and case analysis, and we show how to modify the translation presented in Section 5 to account for these new features. Finally, in Section 8, we present some conclusions and related work.

2. Constructive type theory

This paper is mainly intended for readers who already have some knowledge of type theory. Here we recall the main concepts to fix notation and to highlight some extensions that we use. Readers familiar with type theory may skip this section.

The components of type theory are terms and types: terms are denotations of mathematical or computational objects and types are collections of terms. We write $\alpha\ \mathsf{Type}$ to indicate that α is a type, and $t \in \alpha$ to indicate that t is an element of α.

A *context* Γ is a sequence of assumptions

$$\Gamma \equiv x_1 \in \alpha_1; \ldots; x_n \in \alpha_n$$

where $x_1, \ldots, x_n$ are distinct variables and each α_i is a type that can contain occurrences of the variables that precede it, that is, the variables $x_1, \ldots, x_{i-1}$. We use capital Greek letters to denote contexts.

Type theory also contains rules for making *typing judgements* of the form

$$\Gamma \vdash t \in \alpha.$$

In Table 1 we present a sketch of the basic type formers in constructive type theory. For a complete description of the formal rules, see Martin-Löf (1984), Nordström *et al.* (1990), and Coquand *et al.* (1994). A general formulation of type systems and their use in formal verification can be found in Barendregt (1992) and Barendregt and Geuvers (2001).

We will now introduce some additional notation and terminology that we use in the rest of the paper.

A sequence of variable assumptions Δ is called a *context extension* of a context Γ if $\Gamma; \Delta$ is a context. If there is no danger of confusion, we may refer to context extensions simply

Table 1. *Basic type formers*

Small types			
Formation	$\mathsf{Set\ Type}$	Canonical elements	$(x \in \alpha)\beta, \Sigma x \in \alpha.\beta,$ $\mathsf{N}, \alpha \times \beta, \alpha + \beta, \ldots$
Equality			
Formation	$\dfrac{\alpha\ \mathsf{Type} \quad a, b \in \alpha}{\mathsf{Id}(\alpha, a, b)\ \mathsf{Type}}$	Notation	$a = b$
		Canonical elements	$\mathsf{refl}(a) \in a = a$
Dependent Products			
Formation	$\dfrac{\alpha\ \mathsf{Type} \quad x \in \alpha \vdash \beta\ \mathsf{Type}}{(x \in \alpha)\beta\ \mathsf{Type}}$	Canonical elements	$[x \in \alpha]b, [x]b$
		Application	$f(a)$
Function types			
Formation	$\dfrac{\alpha\ \mathsf{Type} \quad \beta\ \mathsf{Type}}{(\alpha)\beta\ \mathsf{Type}}$	Definition	$(\alpha)\beta \equiv (x \in \alpha)\beta$
Dependent Sums			
Formation	$\dfrac{\alpha\ \mathsf{Type} \quad x \in \alpha \vdash \beta\ \mathsf{Type}}{\Sigma x \in \alpha.\beta\ \mathsf{Type}}$	Canonical elements	$\langle a, b \rangle$
		Projections	π_1, π_2
Cartesian products			
Formation	$\dfrac{\alpha\ \mathsf{Type} \quad \beta\ \mathsf{Type}}{\alpha \times \beta\ \mathsf{Type}}$	Definition	$\alpha \times \beta \equiv \Sigma x \in \alpha.\beta$
Binary Sums			
Formation	$\dfrac{\alpha\ \mathsf{Type} \quad \beta\ \mathsf{Type}}{\alpha + \beta\ \mathsf{Type}}$	Canonical elements	$\mathsf{in_l}(a), \mathsf{in_r}(b)$
		Elimination	Cases
Finite Sums			
Formation	$\dfrac{\alpha_1\ \mathsf{Type} \quad \ldots \quad \alpha_n\ \mathsf{Type}}{\alpha_1 + \cdots + \alpha_n\ \mathsf{Type}}$	Canonical elements	$\mathsf{in_i}(a)$
		Elimination	Cases

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.

Given a context $\Gamma \equiv x_1 \in \alpha_1, \ldots, x_n \in \alpha_n$, an *instantiation* of Γ is a sequence of terms $a_1, \ldots, a_n$ such that $a_1 \in \alpha_1, \ldots, a_n \in \alpha_n[x_1 := a_1, \ldots, x_{n-1} := a_{n-1}]$. We write $\overline{a} \in \Gamma$ for such an instantiation.

If Γ is a context and β a type whose free variables are among those 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:

$$()\beta \equiv \beta \quad \text{and} \quad (x \in \alpha; \Gamma')\beta \equiv (x \in \alpha)((\Gamma')\beta).$$

We usually write sequential dependent products as $(x_1 \in \alpha_1; \ldots; x_n \in \alpha_n)\beta$ and sequential λ-abstractions as $[x_1, \ldots, x_n]b$. In addition, we write $(x_1, x_2, \ldots, x_n \in \alpha)$ instead of $(x_1 \in \alpha; x_2 \in \alpha; \ldots; x_n \in \alpha)$.

Similarly, $\Sigma(\Gamma)$ is the sum of all the types of the assumptions in Γ, for a non-empty context Γ. Again, it is formally defined by recursion on the length of Γ:

$$\Sigma(x \in \alpha) \equiv \alpha \quad \text{and} \quad \Sigma(x \in \alpha; \Gamma') \equiv \Sigma x \in \alpha.\Sigma(\Gamma').$$

If $\Gamma \equiv x_1 \in \alpha_1; \ldots; x_n \in \alpha_n$, we use n-tuple notation for the canonical elements of $\Sigma(\Gamma)$; concretely, we write $\langle a_1, a_2, \ldots, a_{n-1}, a_n \rangle$ for $\langle a_1, \langle a_2, \ldots, \langle a_{n-1}, a_n \rangle \cdots \rangle \rangle$ and the symbols $\pi_1, \ldots, \pi_n$ for the projections.

We allow inductive definitions in type theory. They are type formers defined by a sequence of constructors:

$$\begin{array}{l} \alpha \in (\Gamma)\mathsf{Set} \\ \quad \mathsf{c}_1 \in (\Theta_1)\alpha(\overline{t_1}) \\ \quad\quad \vdots \\ \quad \mathsf{c}_n \in (\Theta_n)\alpha(\overline{t_n}) \end{array}$$

with a positivity condition on the types of the constructors to guarantee that terms are well-founded. For a complete formal description of inductive definitions, see, for example, Hagino (1987), Coquand and Paulin (1990), Pfenning and Paulin-Mohring (1990), Werner (1994), or Chapter 2 of Capretta (2002).

We need to extend inductive definitions to allow simultaneous induction-recursion as introduced in Dybjer (2000), that is, to allow the simultaneous definition of an inductive type and a recursive function that depends on the type. Their general forms are

$$\begin{array}{ll} \alpha \in (\Gamma)\mathsf{Set} & f \in (\Delta; \alpha(d))\beta \\ \quad \mathsf{c}_1 \in (\Theta_1)\alpha(\overline{t_1}) & \quad f(\overline{x_1}, \mathsf{c}_1(\overline{z_1})) = e_1 \\ \quad\quad \vdots & \quad\quad \vdots \\ \quad \mathsf{c}_n \in (\Theta_n)\alpha(\overline{t_n}) & \quad f(\overline{x_n}, \mathsf{c}_n(\overline{z_n})) = e_n \end{array}$$

where the function f may occur in the type of some constructors c_i's, with certain restrictions guaranteeing that terms of type α are well-founded and that f is total.

Finally, theorems are considered as dependent types, thus they have the general form $(x_1 \in \alpha_1; \ldots; x_n \in \alpha_n)\beta$.

3. Some examples

We illustrate our method for formalising general recursive algorithms in type theory with a few easy examples. Detailed descriptions and more examples can be found in our previous work (see: Bove (2001), for simple recursive algorithms; Bove and Capretta (2001), for nested algorithms and partial functions; Bove (2002b), for mutually recursive algorithms; and Bove (2003)).

All the auxiliary functions that we use in the examples below are well-known structurally recursive functions. We call a function structurally recursive if the arguments of the recursive calls in its definition are structurally smaller than the input and the function contains no calls to general recursive functions. Therefore, these functions can be translated straightforwardly into type theory, and we can use their translation in the formalisation of the corresponding example. Unless we state otherwise, we assume that the type-theoretic translation of an auxiliary function has the same name as in the functional program.

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

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

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

— The algorithm `quicksort` terminates on the input `[]`.

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

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

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

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

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

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

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

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

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

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

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

Note that the proof of termination is given after the definition of quicksort. Hence, the formalisation of the function does not depend on its totality, as is the case when we use the general accessibility method, where the proof of termination of an algorithm is mixed with the definition of the algorithm. Therefore, it is possible to define, with our method, non-total functions. The only difference is that when formalising non-total functions, we cannot construct a proof that the accessibility predicate is always satisfied. See Bove and Capretta (2001) and Bove (2003) for examples showing how to use our method to define partial recursive functions in type theory.

Our method applies also to the formalisation of nested recursive algorithms. Here is the Haskell code of Paulson's normalisation function for conditional expressions (Paulson 1986). Its Haskell definition is

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

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

Using our method, we would obtain the following introduction rules for the inductive predicate nmAcc (for y, z, u, v and w conditional expressions):

$$\frac{}{\mathsf{nmAcc(at)}} \qquad \frac{\mathsf{nmAcc}(y) \quad \mathsf{nmAcc}(z)}{\mathsf{nmAcc}(\mathsf{if}(\mathsf{at}, y, z))} \qquad \frac{\begin{array}{c}\mathsf{nmAcc}(\mathsf{if}(v, y, z))\\ \mathsf{nmAcc}(\mathsf{if}(w, y, z))\\ \mathsf{nmAcc}(\mathsf{if}(u, \mathsf{nm}(\mathsf{if}(v, y, z)), \mathsf{nm}(\mathsf{if}(w, y, z))))\end{array}}{\mathsf{nmAcc}(\mathsf{if}(\mathsf{if}(u, v, w), y, z))}.$$

Unfortunately, this definition is not correct in ordinary type theory since the algorithm nm is not defined yet and, therefore, cannot be used in the definition of the predicate. Moreover, the purpose of defining the predicate nmAcc is to be able to define the algorithm nm by structural recursion on the proof that its input value satisfies nmAcc, so we need nmAcc to define nm. However, there is an extension of type theory that gives us the means to define the predicate nmAcc and the function nm at the same time. This

extension was introduced in Dybjer (2000) and 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 nmAcc and nm simultaneously as follows:

$$
\begin{array}{l}
\mathsf{nmAcc} \in (e \in \mathsf{CExp})\mathsf{Set} \\
\quad \mathsf{nmacc}_{at} \in \mathsf{nmAcc}(\mathsf{at}) \\
\quad \mathsf{nmacc}_{if/at} \in (y, z \in \mathsf{CExp}; \mathsf{nmAcc}(y); \mathsf{nmAcc}(z))\mathsf{nmAcc}(\mathsf{if}(\mathsf{at}, y, z)) \\
\quad \mathsf{nmacc}_{if/if} \in (u, v, w, y, z \in \mathsf{CExp}; \\
\qquad\qquad h_1 \in \mathsf{nmAcc}(\mathsf{if}(v, y, z)); h_2 \in \mathsf{nmAcc}(\mathsf{if}(w, y, z)); \\
\qquad\qquad h_3 \in \mathsf{nmAcc}(\mathsf{if}(u, \mathsf{nm}(\mathsf{if}(v, y, z), h_1), \mathsf{nm}(\mathsf{if}(w, y, z), h_2))) \\
\qquad\qquad)\mathsf{nmAcc}(\mathsf{if}(\mathsf{if}(u, v, w), y, z))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{nm} \in (e \in \mathsf{CExp}; \mathsf{nmAcc}(e))\mathsf{CExp} \\
\quad \mathsf{nm}(\mathsf{at}, \mathsf{nmacc}_{at}) = \mathsf{at} \\
\quad \mathsf{nm}(\mathsf{if}(\mathsf{at}, y, z), \mathsf{nmacc}_{if/at}(y, z, h_1, h_2)) = \mathsf{if}(\mathsf{at}, \mathsf{nm}(y, h_1), \mathsf{nm}(z, h_2)) \\
\quad \mathsf{nm}(\mathsf{if}(\mathsf{if}(u, v, w), y, z), \mathsf{nmacc}_{if/if}(u, v, w, y, z, h_1, h_2, h_3)) = \\
\qquad\qquad \mathsf{nm}(\mathsf{if}(u, \mathsf{nm}(\mathsf{if}(v, y, z), h_1),\ \mathsf{nm}(\mathsf{if}(w, y, z), h_2)), h_3).
\end{array}
$$

Mutually recursive algorithms, with or without nested recursive calls, can also be formalised with our method. If the mutually recursive algorithms are not nested, their formalisation is similar to the formalisation of the quicksort algorithm in the sense that we first define the accessibility predicate for each function, and then we formalise the algorithms by structural recursion on the proofs that the input values satisfy the corresponding predicate. When we have mutually recursive algorithms, the termination of one function depends on the termination of the others, so the domain 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. The soundness of the extension follows from the known fact that mutual recursion can be implemented by simple recursion using some indexing. See Bove (2002b; 2003) for a description and examples of our method in the formalisation of mutual recursive functions.

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

As a final remark, we should draw 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.

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

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

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

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

A functional programmer might consider 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 satisfy 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 6, we will say more about 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. Recall that structurally recursive functions are guaranteed to be total and can be defined directly in type theory.

If `f` is a general recursive function, it should be translated in type theory as a pair consisting of a predicate fAcc and a function f. Thus, we cannot call it inside the definition of another function g. This restriction is imposed by type-checking requirements and will become clearer below. If, instead, `f` is structurally recursive, it can be directly translated into type theory as a structurally recursive function f, without the need for 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`:

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

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

$$\begin{aligned}&\mathsf{fAcc} \in (m \in \mathsf{N})\mathsf{Set}\\&\mathsf{f} \in (m \in \mathsf{N}; \mathsf{fAcc}(m))\mathsf{N}.\end{aligned}$$

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

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

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 using a predicate gAcc and a function g. The difficulty now is 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 no longer has the type (N)N.

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 no longer depends on the special accessibility predicate, even when the algorithm is a general recursive one. QuickSort is an example of such a function. It would also be safe to allow these functions inside the definition of other functions. Then, the class of functions to which our method applies 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, it is an easy exercise to show that the class of functions that we consider still allows us to define all recursive functions. See Section 4.4 for a more detailed discussion on this matter.

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 foul of the first restriction: the function `f` is one of the functions being defined and the occurrence of `f` inside `g` is not fully applied, thus disallowed.

In a forthcoming article (Bove and Capretta 2004), we show that all these restrictions can be lifted if we use an impredicative type theory.

4. General recursive definitions

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

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

4.1. *The class of types*

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

Definition 1. Let $\mathscr{TV}$ be an infinite set of type variables. The class of types that are allowed in our programs are inductively defined by:

— All elements of $\mathscr{TV}$ are types.

— Inductive data types are types. An inductive data type is introduced by a definition of the form

$$\begin{aligned} \texttt{Inductive T}\ \gamma_1\ \cdots\ \gamma_w ::= \ & \mathsf{c}_1\ \tau_{11}\ \cdots\ \tau_{1k_1} \mid \\ & \vdots \qquad\quad \mid \\ & \mathsf{c}_o\ \tau_{o1}\ \cdots\ \tau_{ok_o} \end{aligned}$$

where $o \geqslant 0$, $k_i \geqslant 0$ for $0 \leqslant i \leqslant o$, and $\gamma_1, \ldots, \gamma_w$ are type variables, for $w \geqslant 0$. Every τ_{ij} is a type where `T` may occur only strictly positive and fully applied to its arguments, that is, only in the form $(\texttt{T}\ \gamma_1 \cdots \gamma_w)$ and on the right of the arrows.

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

As examples of inductive data types, we show how to define the types of boolean values, the type of natural numbers and the parametric type of lists:

$$\texttt{Inductive Bool} ::= \texttt{true} \mid \texttt{false}$$
$$\texttt{Inductive Nat} ::= \texttt{0} \mid \texttt{s Nat}$$
$$\texttt{Inductive List}\ \gamma ::= \texttt{nil} \mid \texttt{cons}\ \gamma\ (\texttt{List}\ \gamma).$$

To instantiate a parametric data type, we simply write ($\texttt{T}\ \sigma_1 \cdots \sigma_w$) for specific types $\sigma_1, \ldots, \sigma_w$. This expression denotes the type obtained by substituting each γ_h by σ_h in the definition of T, for $1 \leqslant h \leqslant w$.

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

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

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

If e has the above specification, it must be interpreted as follows: e is an expression that, when applied to arguments $a_1, \ldots, a_m$ of type $\sigma_1, \ldots, \sigma_m$, respectively, produces a term $e(a_1, \ldots, a_m)$ of type τ. The expression e itself is not a term of any type; in particular, it is not an element of the functional type $\sigma_1 \to \cdots \to \sigma_m \to \tau$. We introduce specifications so that we can 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.

In what follows, we write $a : A$ to express the fact that a is an expression of type A or that a has the specification A.

We also use specifications to force constructors to be fully applied. The definition of an inductive data type introduces not only a new type but also its constructors:

$$c_i : \tau_{i1}, \ldots, \tau_{ik_i} \Rightarrow \tau.$$

When instantiating a parametric data type, we should, of course, also instantiate its constructors. For example, the constructors of the instantiated data type (List Nat) are $\texttt{nil} : \Rightarrow \texttt{List Nat}$ and $\texttt{cons} : \texttt{Nat}, (\texttt{List Nat}) \Rightarrow \texttt{List Nat}$.

Given $a : \sigma$, the reader should bear in mind the difference between $f(a)$, which denotes the application of a function f with specification $\sigma \Rightarrow \tau$ to a, and $(f\ a)$, which denotes the application of a function f of type $\sigma \to \tau$ to a. If there is no risk of confusion, we may simply write $f\ a$ for the latter.

Note that we can directly translate every type occurring 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. Each function is defined by a sequence of recursive equations. We now formally define patterns and the terms that are allowed in the equations.

Definition 2. Let τ be a type. A *pattern* of type τ is an expression built up according to the following two rules:

— A variable of type τ is a pattern of type τ.
— If τ is an inductive type, $\mathsf{c} : \tau_1, \ldots, \tau_k \Rightarrow \tau$ is one of its constructors and $p_1, \ldots, p_k$ are patterns of type $\tau_1, \ldots, \tau_k$, respectively, then $\mathsf{c}(p_1, \ldots, p_k)$ is a pattern of type τ.

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

We only allow linear patterns. Usually, when we want to refer to a *linear pattern*, we will just say *pattern*. .

Definition 3. A sequence of patterns $p_1, \ldots, p_m$ of type τ is said to be *exclusive* if it is impossible to obtain the same term by instantiating two different patterns, that is, by substituting the pattern variables in those patterns by other terms.

The sequence is called *exhaustive* if every value of type τ is an instance of at least one of the patterns. By *value*, we mean a closed normal form.

The terms that we allow in the definition of recursive functions depend on an environment comprising three components:

(a) the set $\mathscr{X}$ of variables that can occur free in the terms;
(b) the set $\mathscr{SF}$ of functions that are being defined, which can be used in the recursive calls; and
(c) a set $\mathscr{F}$ of total functions that we already know how to translate in type theory with the same functional type.

In what follows, we assume that $\mathscr{F}$ consists of all structurally recursive functions together with their types (the notion of structurally recursive function was described at the beginning of Section 3). As we have already explained, we could extend $\mathscr{F}$ to a larger class of functions by adding all the functions that can be proved total in type theory. The class $\mathscr{F}$ being fixed, the allowed terms depend only on the set $\mathscr{X}$ and on the set $\mathscr{SF}$. Formally, we define terms as follows.

Definition 4. Let $\mathscr{X}$ be a set of variables together with their types. Let $\mathscr{SF}$ be a set of function names together with their specifications. Let the set of names of the variables in $\mathscr{X}$, the set of names of the functions in $\mathscr{SF}$, and the set of names of the functions in $\mathscr{F}$ be disjoint. We say that *t is a valid term of type τ* or that *$t : \tau$ is a valid term* with respect to $\mathscr{X}$ and $\mathscr{SF}$, if the judgement $\mathscr{X}; \mathscr{SF} \vdash t : \tau$ can be derived from the rules in Table 2, where $(\mathscr{X} \backslash x)$ stands for the set $\mathscr{X}$ without any association for the variable x.

Normally, a functional programming language also allows terms obtained by case analysis on arguments of inductive types. Case analysis does not add any conceptual difficulty to our method but some notational overhead. Therefore, for the moment we will not allow it. We will explain how to translate programs containing case analysis in Section 7.

As we explain in Section 6, we need a strict semantics to compute the programs in $\mathscr{FP}$. This said, the computation (reduction) rules for terms are the usual ones. We use

Table 2. *Rules for deriving valid term judgements*

$$\frac{x : \sigma \in \mathcal{X}}{\mathcal{X}; \mathcal{SF} \vdash x : \sigma} \qquad \frac{f : \sigma \to \tau \in \mathcal{F}}{\mathcal{X}; \mathcal{SF} \vdash f : \sigma \to \tau}$$

$$\frac{\mathtt{f} : \sigma_1, \ldots, \sigma_m \Rightarrow \tau \in \mathcal{SF} \quad \mathcal{X}; \mathcal{SF} \vdash a_i : \sigma_i \text{ for } 1 \leqslant i \leqslant m}{\mathcal{X}; \mathcal{SF} \vdash f(a_1, \ldots, a_m) : \tau}$$

$$\frac{\mathtt{c} : \tau_1, \ldots, \tau_k \Rightarrow \tau \qquad \mathtt{c} \text{ constructor of } \tau \qquad \mathcal{X}; \mathcal{SF} \vdash a_i : \tau_i \text{ for } 1 \leqslant i \leqslant k}{\mathcal{X}; \mathcal{SF} \vdash \mathtt{c}(a_1, \ldots, a_k) : \tau}$$

$$\frac{(\mathcal{X} \backslash x) \cup \{x : \sigma\}; \mathcal{SF} \vdash b : \tau}{\mathcal{X}; \mathcal{SF} \vdash [x]b : \sigma \to \tau} \qquad \frac{\mathcal{X}; \mathcal{SF} \vdash f : \sigma \to \tau \quad \mathcal{X}; \mathcal{SF} \vdash a : \sigma}{\mathcal{X}; \mathcal{SF} \vdash (f\ a) : \tau}$$

the symbol $\rightsquigarrow$ to denote one step reduction over terms and the symbol $\rightsquigarrow^*$ to denote the reflexive and transitive closure of $\rightsquigarrow$.

4.3. *Fixed-point function definition*

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

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

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

Let $1 \leqslant i \leqslant l$, and let $\mathcal{Y}_i$ be the set of pattern variables that occur in the ith equation together with their types. The right-hand side of the ith equation in the definition of $\mathtt{f}$, that is e_i, is a valid term of type τ with respect to $\mathcal{Y}i$ and $\{\mathtt{f} : \sigma_1, \ldots, \sigma_m \Rightarrow \tau\}$. Hence, e_i can contain subterms of the form $\mathtt{f}(a_1, \ldots, a_m)$, where $a_1, \ldots, a_m$ are valid terms of type $\sigma_1, \ldots, \sigma_m$, respectively. Each a_u can, in turn, contain calls to $\mathtt{f}$, giving rise to nested recursive definitions.

The computation rules for $\mathtt{f}$ are given by the different equations in its definition. Recall that we need to compute our programs in a strict semantics. Hence, if we want to compute the expression $\mathtt{f}(a_1, \ldots, a_m)$, we first have to reduce the terms $a_1, \ldots, a_m$.

Assume that $\overline{a} \rightsquigarrow^* \overline{p}[\overline{y := b}]$, that is, $a_1 \rightsquigarrow^* p_1[\overline{y := b}]$, ..., $a_m \rightsquigarrow^* p_m[\overline{y := b}]$ where $\overline{y}$ are the pattern variables of a multipattern $(p_1, \ldots, p_m)$ in the left-hand side of one of the equations defining $\mathtt{f}$. Let $\mathtt{f}(p_1, \ldots, p_m) = e$ be the corresponding equation. Then, we have the following computation rule:

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

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

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

$$\begin{aligned}
\mathtt{mutual\ fix}\ \mathtt{f_1} : \sigma_{11}, \ldots, \sigma_{1m_1} &\Rightarrow \tau_1 \\
\mathtt{f_1}(p_{111}, \ldots, p_{11m_1}) &= e_{11} \\
&\vdots \\
\mathtt{f_1}(p_{1l_11}, \ldots, p_{1l_1m_1}) &= e_{1l_1} \\
&\vdots \\
\mathtt{f_n} : \sigma_{n1}, \ldots, \sigma_{nm_n} &\Rightarrow \tau_n \\
\mathtt{f_n}(p_{n11}, \ldots, p_{n1m_n}) &= e_{n1} \\
&\vdots \\
\mathtt{f_n}(p_{nl_n1}, \ldots, p_{nl_nm_n}) &= e_{nl_n},
\end{aligned}$$

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

4.4. *Turing completeness*

Because of the restrictions we have imposed on the recursive definitions, it may not be obvious that the class $\mathscr{FP}$ of functional programs allows the definition of all partial recursive functions. To see that this is actually the case, we can use the *Kleene normal form theorem* (see, for example, Bell and Machover (1977, Theorem 10.1) or Phillips (1992, Theorem 1.5.6)). It states that each general recursive function can be defined from primitive recursive operations by using minimisation once only.

The only problematic part when applying Kleene's theorem to define a concrete function f is the use of the minimisation function inside the definition of f. Recall that we do not allow previously defined functions inside the definition of a new function unless they are structurally recursive. Thus, although it is easy to translate the minimisation function with our method, we cannot use the general form of the minimisation operator inside the definition of f. Instead, we can define a specific minimisation function for f by mutual recursion with f. This minimisation function does not really depend on f, but the use

of a mutual recursive definition is a trick that allows us to use minimisation inside the definition of f.

5. Translation into type theory

In this section we give a formal presentation of the translation of the programs in $\mathscr{FP}$ into type theory.

We assume that the reader is familiar with constructive type theory and knows how to translate types and expressions from functional programming into their type-theoretic equivalents. All types in functional programming have a corresponding type defined in type theory in the same way, except for the difference in notation. Hence, the elements in those types have a type-theoretic equivalent. Structurally recursive functions, that is, the elements of the class $\mathscr{F}$, can also be translated directly into 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 $\mathtt{f}$ be a general recursive function in $\mathscr{FP}$. Thus,

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

where $\mathtt{f}$ is defined by a sequence of recursive equations of the form

$$\mathtt{f}(p_1, \ldots, p_m) = e. \tag{1}$$

To translate $\mathtt{f}$ into type theory, we define a special-purpose accessibility predicate fAcc and a type-theoretic version of $\mathtt{f}$, called $\mathtt{f}$, which has the predicate fAcc as part of its domain. These two components have the following types:

$$\begin{array}{l} \mathsf{fAcc} \in (x_1 \in \widehat{\sigma_1}; \ldots; x_m \in \widehat{\sigma_m})\mathsf{Set} \\ \mathtt{f} \in (x_1 \in \widehat{\sigma_1}; \ldots; x_m \in \widehat{\sigma_m}; h \in \mathsf{fAcc}(x_1, \ldots, x_m))\widehat{\tau}. \end{array} \tag{2}$$

The function $\mathtt{f}$ is defined by structural recursion on the argument h. Hence, we have one equation in $\mathtt{f}$ for each constructor of fAcc. If the function $\mathtt{f}$ is defined by nested recursion, we should define fAcc and $\mathtt{f}$ simultaneously using Dybjer's schema for simultaneous inductive-recursive definitions (Dybjer 2000). Otherwise, we first define fAcc and then use that predicate to define $\mathtt{f}$.

We begin by discussing how to define the predicate fAcc. This predicate has one constructor for each equation in the definition of $\mathtt{f}$. The constructor associated with Equation (1) produces a proof of $\mathsf{fAcc}(\widehat{p_1}, \ldots, \widehat{p_m})$, where $\widehat{p_u}$ is the straightforward translation of p_u, for $1 \leqslant u \leqslant m$. 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 e to establish the type of each constructor, guarantees that at most one constructor can be used to build a proof of $\mathsf{fAcc}(\widehat{a_1}, \ldots, \widehat{a_m})$.

Let us explain the general idea of the translation. We associate with each equation of form (1) in the definition of $\mathtt{f}$, a constructor for fAcc and an equation for $\mathtt{f}$. To this end, we analyse the structure of the right-hand side e of Equation (1).

Let Γ be the type-theoretic context containing type assumptions for the variables introduced in the pattern $(p_1, \ldots, p_m)$ of the equation. Each variable is assumed with type

$\widehat{\sigma}$ if σ is its type in functional programming. From the analysis of e, we construct a context extension Φ_e containing assumptions that provide for the type-theoretic translation of e.

Let Equation (1) be the ith equation in the definition of `f`. The type for the constructor of fAcc associated with this equation can now be defined as

$$\mathsf{f\_acc}_i \in (\Gamma; \Phi_e)\mathsf{fAcc}(\widehat{p_1}, \ldots, \widehat{p_m}).$$

Concurrently with the definition of the context extension Φ_e, we also get a translation $\widehat{e}$ of the term e itself. Then, the equation of f associated with the constructor $\mathsf{f\_acc}_i$ is

$$\mathsf{f}(\widehat{p_1}, \ldots, \widehat{p_m}, \mathsf{f\_acc}_i(\overline{y}, \overline{z})) = \widehat{e}$$

where $\overline{y}$ and $\overline{z}$ are the sequences of variables assumed in Γ and in Φ_e, respectively.

To complete the formal translation, we need to define precisely the context extension Φ_e and the translation $\widehat{e}$ associated with Equation (1). The reader can verify that $\widehat{e}$ has type $\widehat{\tau}$ in context $\Gamma; \Phi_e$.

Definition 5. Given an equation of `f` in the form (1) and a type-theoretic context Γ containing type assumptions for the variables introduced in the pattern of the equation, we define the context extension Φ_e and the type-theoretic term $\widehat{e}$ by recursion on the structure of the expression e. Note that, since Φ_e extends Γ, we should only introduce fresh variables in Φ_e.

$e \equiv z$**:** If the expression e is a variable z, then $\Phi_e \equiv ()$ and $\widehat{e} \equiv z$.

$e \equiv \mathsf{g}$**:** If e is one of the functions g in $\mathscr{F}$, then $\Phi_e \equiv ()$ and $\widehat{e} \equiv \widehat{\mathsf{g}}$.

$e \equiv \mathsf{c}(a_1, \ldots, a_o)$**:** First we determine $\Phi_{a_1}, \ldots, \Phi_{a_o}$ and $\widehat{a_1}, \ldots, \widehat{a_o}$ by structural recursion, and then we combine these translations into the definition of Φ_e and $\widehat{e}$. Formally, we define

$$\Phi_e \equiv \Phi_{a_1}; \ldots; \Phi_{a_o} \qquad \text{and} \qquad \widehat{e} \equiv \widehat{\mathsf{c}}(\widehat{a_1}, \ldots, \widehat{a_o}).$$

$e \equiv \mathsf{f}(a_1, \ldots, a_m)$**:** Again we first determine $\Phi_{a_1}, \ldots, \Phi_{a_m}$ and $\widehat{a_1}, \ldots, \widehat{a_m}$ by structural recursion. As before, we combine these translations into the definition of Φ_e and $\widehat{e}$. Recall that we have to add the assumption corresponding to the recursive call $\mathsf{f}(a_1, \ldots, a_m)$ stating that the tuple $(\widehat{a_1}, \ldots, \widehat{a_m})$ satisfies the predicate fAcc. Remember that $\mathsf{f} \equiv \widehat{\mathsf{f}}$ and that f takes an extra parameter, which is a proof that the input values satisfy the predicate fAcc. Hence, we have

$$\Phi_e \equiv \Phi_{a_1}; \ldots; \Phi_{a_m}; h \in \mathsf{fAcc}(\widehat{a_1}, \ldots, \widehat{a_m}) \qquad \text{and} \qquad \widehat{e} \equiv \mathsf{f}(\widehat{a_1}, \ldots, \widehat{a_m}, h).$$

$e \equiv (a_1\ a_2)$**:** This case is treated similarly to the previous two cases, giving

$$\Phi_e \equiv \Phi_{a_1}; \Phi_{a_2} \qquad \text{and} \qquad \widehat{e} \equiv \widehat{a_1}(\widehat{a_2}).$$

$e \equiv [z]b$**:** Let σ be the type of z. We start by calculating Φ_b and $\widehat{b}$ recursively. Now, the context with the assumptions for the free variables in b is $(\Gamma; z \in \widehat{\sigma})$. Recall that variables are directly translated as variables in type theory.

If $\Phi_b = ()$, that is, if the context of translation for b is empty, then

$$\Phi_e \equiv () \qquad \text{and} \qquad \widehat{e} \equiv [z]\widehat{b}\,.$$

In other words, in this case the method does not produce any assumption and the λ-abstraction can be translated straightforwardly into type theory.
Otherwise, to translate this term as a λ-abstraction in type theory, we must impose the condition that the term $\widehat{b}$ is well-defined for every value of the variable z. Therefore, the assumption generated by e must be the universal quantification over z of all the assumptions for $\widehat{b}$. Let $\Sigma\Phi_b$ be the conjunction of all the assumptions in the non-empty context Φ_b and let $y_1 \cdots y_o$ be the sequence of variables assumed in Φ_b. We define

$$\Phi_e \equiv H \in (z \in \widehat{\sigma})\Sigma\Phi_b \qquad \text{and} \qquad \widehat{e} \equiv [z]\widehat{b}[y_1 := (\pi_1\ H(z)); \ldots ; y_o := (\pi_o\ H(z))]\,.$$

To simplify the notation, in the following we write $[\overline{y := (\pi\ H(z))}]$ to denote the above substitution.
If Φ_b contains only one assumption, we do not need to construct a Σ-type. If we let $\Phi_b \equiv y \in \alpha$, then we simply have

$$\Phi_e \equiv H \in (z \in \widehat{\sigma})\alpha \qquad \text{and} \qquad \widehat{e} \equiv [z]\widehat{b}[y := H(z)]\,.$$

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

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

$$\begin{array}{l}\texttt{mutual fix}\ \mathsf{f}_1 : \sigma_{11}, \ldots, \sigma_{1m_1} \Rightarrow \tau_1 \\ \qquad\qquad \vdots \\ \qquad \mathsf{f}_n : \sigma_{n1}, \ldots, \sigma_{nm_n} \Rightarrow \tau_n \\ \qquad\qquad \vdots\end{array}$$

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

$$\begin{array}{l}\mathsf{fAcc}_1 \in (x_{11} \in \widehat{\sigma_{11}}; \ldots ; x_{1m_1} \in \widehat{\sigma_{1m_1}})\mathsf{Set} \\ \qquad\qquad \vdots \\ \mathsf{fAcc}_n \in (x_{n1} \in \widehat{\sigma_{n1}}; \ldots ; x_{nm_n} \in \widehat{\sigma_{nm_n}})\mathsf{Set}\end{array}$$

$$\begin{array}{l}\mathsf{f}_1 \in (x_{11} \in \widehat{\sigma_{11}}; \ldots ; x_{1m_1} \in \widehat{\sigma_{1m_1}}; h_1 \in \mathsf{fAcc}(x_{11}, \ldots, x_{1m_1}))\widehat{\tau_1} \\ \qquad\qquad \vdots \\ \mathsf{f}_n \in (x_{n1} \in \widehat{\sigma_{n1}}; \ldots ; x_{nm_n} \in \widehat{\sigma_{nm_n}}; h_n \in \mathsf{fAcc}(x_{n1}, \ldots, x_{nm_n}))\widehat{\tau_n}.\end{array}$$

In a similar way to what happens in the translation of a single function definition, if a function $\mathtt{f_j}$ is defined by nested recursion, for $1 \leqslant j \leqslant n$, we should define the fAcc's and the f's simultaneously. In order to do so, we need the generalisation of Dybjer's schema presented in Bove (2002b). Otherwise, we first define the fAcc's and then use those predicates to define the f's.

Each predicate $\mathsf{fAcc_j}$ and each function $\mathsf{f_j}$ is defined as in the case of a single function, with the only difference being that now the case in the definition of Φ_e and $\widehat{e}$ 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 for the non-mutually recursive case.

We conclude this section with a couple of observations. First, notice 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 easily be 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 in Section 3.

Second, 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.

6. Lazy and 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 `f` in $\mathscr{FP}$, our general method produces a pair consisting of a predicate fAcc and a function `f` that depends on the predicate. For example, if `f` has the specification $\sigma \Rightarrow \tau$, we obtain $\mathsf{fAcc} \in (\widehat{\sigma})\mathsf{Set}$ and $\mathtt{f} \in (x \in \widehat{\sigma}; h \in \mathsf{fAcc}(x))\widehat{\tau}$ in the translation into 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 `f` *terminates on the input* a *if and only if* $\mathsf{fAcc}(a)$ *is provable. Moreover, if* $h \in \mathsf{fAcc}(a)$*, the type-theoretic equivalent to the output produced by the computation of* $\mathtt{f}(a)$ *is* $\mathtt{f}(a, h)$.

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

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

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

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

On the other hand, a *strict* evaluation strategy requires that the arguments of a function are reduced to a value before the function is computed. In the example above, this implies that $\mathtt{f}(\overline{a})$ 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 method corresponds to a strict semantics for functional programs: the accessibility predicate fAcc characterises the arguments on which the function `f` is defined, and its constructors are such that, to prove $\mathsf{fAcc}(\overline{p})$ in the example above, we first need to prove $\mathsf{fAcc}(\overline{a})$, that is, `f` must be defined on $\overline{a}$ in order to be defined on $\overline{p}$.

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 function. Let us consider an equation of the form

$$\mathtt{f}(p_1, \ldots, p_m) = \cdots [x]e' \cdots .$$

According to our interpretation, the subexpression $[x]e'$ must be defined for the right-hand side to be defined. Notice, however, that $[x]e'$ denotes a function, and that functions are defined if their values are defined on every input, that is, if they are total. Since the totality of recursive functions it is, in general, undecidable, this interpretation is not a computational model of functional programming. Even in functional programming languages with strict semantics, a term of higher type 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 above equation. Since there are no partially defined terms in type theory, all subexpressions of e must be translated into totally defined terms.

We can illustrate this point with an example. Let us assume that we have a function

$$\mathtt{fix\ factors} : \mathtt{Nat} \Rightarrow \mathtt{List\ Nat}$$

that gives the list of prime factors of a number, repeated as many times as their multiplicity. For example, $\mathtt{factors}(6) = [2, 3]$ and $\mathtt{factors}(300) = [2, 2, 3, 5, 5]$. Let us assume that $\mathtt{factors}(0) = [\,]$. Obviously, this is a total function, although not a structurally recursive one.

We now define the function

$$\begin{aligned}
&\mathtt{fix\ facrec} : \mathtt{Nat} \Rightarrow \mathtt{Nat}\\
&\quad \mathtt{facrec}(0) = 0\\
&\quad \mathtt{facrec}(\mathtt{s}(n)) = \mathtt{sum}\ (\mathtt{map}\ ([x]x * \mathtt{facrec}(x))\ \mathtt{factors}(n)) + n.
\end{aligned}$$

Both the function `sum` and the function `map` are structurally recursive functions. The former returns the addition of all the numbers in a list and the latter maps a function over all the elements in a list and returns a list with the results of the applications. Observe that, formally, both `factors` and `facrec` should be defined in a mutually recursive way in $\mathscr{FP}$.

In the second equation of `facrec`, we have a λ-abstraction as the functional argument of the function `map`. A recursive call to the function `facrec` occurs in the scope of this abstraction. Given the argument $\mathtt{s}(n)$, the recursive calls are performed on the elements of the list that results from $\mathtt{factors}(n)$. Hence, the recursive calls are performed on elements that are smaller than $\mathtt{s}(n)$, so `facrec` is a terminating function.

When we apply our method to this example, the part of the translation corresponding to `facrec` is

$$\begin{array}{l}\mathsf{facrecAcc} \in (\mathsf{N})\mathsf{Set}\\ \quad \mathsf{facrec\_acc}_0 \in \mathsf{facrecAcc}(0)\\ \quad \mathsf{facrec\_acc}_\mathsf{s} \in (n \in \mathsf{N}; H \in (x \in \mathsf{N})\mathsf{facrecAcc}(x); h \in \mathsf{factorsAcc}(n))\mathsf{facrecAcc}(\mathsf{s}(n))\\ \\ \mathsf{facrec} \in (m \in \mathsf{N}; \mathsf{facrecAcc}(m))\mathsf{N}\\ \quad \mathsf{facrec}(0, \mathsf{facrec\_acc}_0) = 0\\ \quad \mathsf{facrec}(\mathsf{s}(n), \mathsf{facrec\_acc}_\mathsf{s}(n, H, h)) = \mathsf{sum}(\mathsf{map}([x]x * \mathsf{facrec}(x, H(x)), \mathsf{factors}(n, h))) + n.\end{array}$$

To compute facrec on the number $\mathsf{s}(n)$, we have to prove $\mathsf{facrecAcc}(\mathsf{s}(n))$. For this, we need to give a proof $H \in (x \in \mathsf{N})\mathsf{facrecAcc}(x)$, that is, we must prove $\mathsf{facrecAcc}(x)$ for every x.

An analysis of the algorithm shows that the assumption H is unnecessarily strong. It requires the function $[x]x * \mathtt{facrec}(x)$ to be defined everywhere. However, in practise, since the function above is given as the functional argument of `map`, we just need the function to be defined on the elements of the list $\mathtt{factors}(n)$.

Our method does not analyse the behaviour of the occurrences of other functions in the functional program we are translating. 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 facrec to force the interpretation to look into the definition of `map` by defining facrec (and factors) by mutual recursion with a specialised version of `map`.

$$\begin{array}{l}\mathtt{mutual\ fix\ factors} : \mathtt{Nat} \Rightarrow \mathtt{List\ Nat}\\ \qquad\qquad\qquad\qquad\quad \vdots\\ \qquad\quad \mathtt{map\_facrec} : \mathtt{List\ Nat} \Rightarrow \mathtt{List\ Nat}\\ \qquad\quad \mathtt{map\_facrec}(\mathtt{nil}) = \mathtt{nil}\\ \qquad\quad \mathtt{map\_facrec}(\mathtt{cons}(m, l)) = \mathtt{cons}(m * \mathtt{facrec}(m), \mathtt{map\_facrec}(l))\\ \\ \qquad\quad \mathtt{facrec} : \mathtt{Nat} \Rightarrow \mathtt{Nat}\\ \qquad\quad \mathtt{facrec}(0) = 0\\ \qquad\quad \mathtt{facrec}(\mathtt{s}(n)) = \mathtt{sum}(\mathtt{map\_facrec}(\mathtt{factors}(n))) + n\,.\end{array}$$

The reader can verify that when we apply the translation for mutually recursive functions to this example, we get a much better condition for the termination of `facrec`. In the second equation of the function facrec, we require a proof of $\mathsf{map\_facrecAcc}(\mathsf{factors}(n, h))$, which is equivalent to $\mathsf{facrecAcc}(x)$ for all elements x in $\mathsf{factors}(n, h)$, but not for every natural number x.

From this example, it is clear that our type-theoretic translation does not give the expected result when applied to recursive functions containing λ-abstractions in the right-hand side of their equations. 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

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

an alternative, and possibly easier solution, might be to define it as

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

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

This said, we now investigate the relation between computations in functional programming and reductions in type theory. Let $e : \tau$ be a valid term in $\mathscr{FP}$ with respect to Γ and $\mathscr{SF}$. In general, there is no correspondence between steps of computation of e in $\mathscr{FP}$ and steps of reduction in the type-theoretic translation $\widehat{e}$ obtained by our method. The reason is that $\widehat{e}$ depends on extra parameters Φ_e (the context of termination conditions constructed by the translation algorithm), and it can only be reduced if those extra parameters are instantiated with actual terms. However, we have a weaker result: if all the extra parameters can be instantiated with terms whose variables are among those in $\widehat{\Gamma}$, then the *result* of the computation of e in $\mathscr{FP}$ corresponds to the normal form of $\widehat{e}$ in type theory. Moreover, the existence of such instantiations for the extra parameters guarantees the termination of e.

Below, when referring to instantiations of Φ_e, we will always mean instantiations whose variables are among those in $\widehat{\Gamma}$. We write $\overline{d_e} \in \Phi_e$ for such an instantiation. Observe that $\overline{d_e}$ cannot contain variables that stand for proofs of domain predicates, since Γ is a context in $\mathscr{FP}$ and there are no such objects in functional programming. That is, $\overline{d_e}$ cannot depend on a variable of type $\mathsf{gAcc}(\overline{\widehat{b}})$ for any function g and arguments $\overline{b}$. This means that given a constraint $h \in \mathsf{fAcc}(\overline{\widehat{a}})$ in Φ_e, it can only be instantiated with a term that reduces (in zero or more steps) to a canonical proof of fAcc.

This, in turn, implies that a result cannot contain a subterm of the form $\texttt{f}(\overline{a})$ where f is a general recursive function. When translating this subterm, our translation method adds a constraint $h \in \mathsf{fAcc}(\overline{\widehat{a}})$ to Φ_e. According to the above observation, any instantiation of h should reduce to a term the form $\mathsf{facc_j}(\ldots)$, where $\mathsf{facc_j}$ is the jth constructor of fAcc. This means that we are able to further reduce the application $\texttt{f}(\overline{\widehat{a}}, \mathsf{facc_j}(\ldots))$ to $\widehat{e_j}[\ldots]$ in type theory, where $\widehat{e_j}$ is the right-hand side of the equation in f that corresponds to the jth constructor. Then, a result cannot contain applications of general recursive functions and thus, it is either a variable, or a constructor or a structurally recursive function (partially) applied to results.

In what follows, the equality symbol $=$ denotes convertibility in type theory and the symbol $\equiv$ denotes syntactic identity.

Lemma 1. Let $e : \tau$ be a valid term in $\mathscr{FP}$ with respect to Γ and $\mathscr{SF}$. Let Φ_e be the context of premises generated by the translation method, $\overline{z}$ the sequence of variables in Φ_e and $\overline{d_e} \in \Phi_e$. Then, the evaluation of e in $\mathscr{FP}$ terminates and gives a result r_e such that

$$\widehat{r_e} = \widehat{e}[\overline{z := d_e}]\,.$$

Proof. We use induction on the pair (l, e) where l is the maximum length of a normalisation path of $\widehat{e}[\overline{z := d_e}]$. Hence, our inductive hypothesis states that the lemma is true for all pairs $(l', e') < (l, e)$ where $<$ is the lexicographic order on pairs, that is

$$(l', e') < (l, e) \text{ iff either } l' < l \text{ or } l' \leqslant l \text{ and } e' \text{ is structurally smaller than } e.$$

We proceed by cases on the structure of e:

— If $e \equiv x$ is a variable, then $\Phi_e \equiv ()$ and the statement is trivially true:

$$r_e \equiv x \qquad \widehat{r_e} \equiv x \equiv \widehat{e}.$$

— If $e \equiv \mathsf{g} \in \mathscr{F}$ is a structurally recursive function, then again $\Phi_e \equiv ()$ and the statement is trivially true:

$$r_e \equiv \mathsf{g} \qquad \widehat{r_e} \equiv \widehat{\mathsf{g}} \equiv \widehat{e}.$$

— If $e \equiv \mathsf{c}(a_1, \ldots, a_o)$ where c is a constructor, then

$$\Phi_e \equiv \Phi_{a_1}; \ldots; \Phi_{a_o} \qquad \overline{d_e} \equiv \overline{d_{a_1}}, \ldots, \overline{d_{a_o}}$$

where $\overline{d_{a_i}} \in \Phi_{a_i}$ for $1 \leqslant i \leqslant o$. The computation of e is just the computation of the arguments $a_1, \ldots, a_o$ of the constructor. The inductive hypothesis can be applied to each of the a_i's, since a reduction path for $\widehat{a_i}[\overline{z_i := d_{a_i}}]$ extends to a reduction path for $\widehat{e}[\overline{z := d_e}]$ (first element of the inductive argument) and a_i is a proper subterm of e (second element of the inductive argument). Thus, each a_i terminates with the result r_{a_i} such that

$$\widehat{r_{a_i}} = \widehat{a_i}[\overline{z_i := d_{a_i}}] \equiv \widehat{a_i}[\overline{z := d_e}]$$

where the syntactic identity is justified by the fact that the variables z_j do not occur free in a_i if $j \neq i$. Therefore the computation of e terminates with the result

$$\widehat{r_e} \equiv \mathsf{c}(\widehat{r_{a_1}, \ldots, r_{a_o}}) = \mathsf{c}(\widehat{a_1}[\overline{z := d_e}], \ldots, \widehat{a_o}[\overline{z := d_e}]) \equiv \widehat{e}[\overline{z := d_e}]\,.$$

— If $e \equiv \mathtt{f}(a_1, \ldots, a_m)$ is an application of a function in $\mathscr{SF}$, then

$$\Phi_e \equiv \Phi_{a_1}; \ldots; \Phi_{a_m}; h \in \mathsf{fAcc}(\widehat{a_1}, \ldots, \widehat{a_m}) \qquad \overline{d_e} \equiv \overline{d_{a_1}}, \ldots, \overline{d_{a_m}}, d_h\,.$$

As in the previous case, our induction hypothesis states that the computation of each a_i terminates with result r_{a_i} such that

$$\widehat{r_{a_i}} = \widehat{a_i}[\overline{z_i := d_{a_i}}]\,.$$

Notice that we need to use these hypotheses to prove that the computation of e terminates since we are assuming a strict evaluation strategy for $\mathscr{FP}$, hence e normalises only if every a_i normalises.

As we pointed out earlier, the proof d_h reduces to a canonical proof of fAcc, let us say $\mathsf{f\_acc}_j$ applied to some arguments. Because of the way we defined the constructor of our domain predicate, this means that the sequence $(r_{a_1}, \ldots, r_{a_m})$ matches the multipattern $(p_{j1}, \ldots, p_{jm})$ in the jth equation in the definition of $\mathtt{f}$. Let Δ be the context of variables occurring free in $\overline{p_j}$ and $\overline{y}$ be the sequence of variables in Δ. Since $\overline{r_a}$ matches $\overline{p_j}$, we know that there is an instantiation $\overline{c} \in \Delta$ such that

$$r_{a_i} \equiv p_{ji}[\overline{y := c}] \qquad \text{for } 1 \leqslant i \leqslant m\,,$$

and then d_h reduces to $\mathtt{f\_acc}_j(\overline{\widehat{c}}, \overline{u})$ for some instantiation $\overline{u} \in \Phi_{e_j}[\overline{y := \widehat{c}}]$, where e_j is the right-hand side of the jth equation in the definition of $\mathtt{f}$.
Observe that, since r_{a_i} does not contain calls to the functions in $\mathscr{SF}$, its translation is straightforward, so we have

$$\widehat{r_{a_i}} \equiv \widehat{p_{ji}}[\overline{y := \widehat{c}}] \qquad \text{for } 1 \leqslant i \leqslant m\,.$$

If we use $\overline{w}$ to denote the variables in Φ_{e_j}, in type theory we have

$$\begin{aligned}
\widehat{e}[\overline{z := d_e}] &\equiv \mathtt{f}(\widehat{a_1}[\overline{z_1 := d_{a_1}}], \ldots, \widehat{a_m}[\overline{z_m := d_{a_m}}], d_h) \\
&\rightsquigarrow^* \mathtt{f}(\widehat{r_{a_1}}, \ldots, \widehat{r_{a_m}}, d_h) \\
&\rightsquigarrow^* \mathtt{f}(\widehat{r_{a_1}}, \ldots, \widehat{r_{a_m}}, \mathtt{f_acc}_j(\overline{\widehat{c}}, \overline{u})) \\
&\equiv \mathtt{f}(\widehat{p_{j1}}[\overline{y := \widehat{c}}], \ldots, \widehat{p_{jm}}[\overline{y := \widehat{c}}], \mathtt{f_acc}_j(\overline{y}, \overline{w})[\overline{y := \widehat{c}, w := u}]) \\
&\equiv \mathtt{f}(\widehat{p_{j1}}, \ldots, \widehat{p_{jm}}, \mathtt{f_acc}_j(\overline{y}, \overline{w}))[\overline{y := \widehat{c}, w := u}] \\
&\rightsquigarrow \widehat{e_j}[\overline{y := \widehat{c}, w := u}] \\
&\equiv e_j[\widehat{\overline{y := c}}][\overline{w := u}]\,,
\end{aligned}$$

where the last reduction step is obtained by the definition of $\mathtt{f}$ in type theory.
The last term is one step closer to its normal form than the original one. So, by the induction hypothesis, we have that the computation of $e_j[\overline{y := c}]$ terminates and gives, as a result, the term $r_{e_j[\overline{y:=c}]}$ such that

$$\widehat{r_{e_j[\overline{y:=c}]}} = e_j[\widehat{\overline{y := c}}][\overline{w := u}]\,.$$

Due to the exclusivity of the patterns defining $\mathtt{f}$, we know that e reduces to $e_j[\overline{y := c}]$ in $\mathscr{FP}$. Concretely, we have

$$e \equiv \mathtt{f}(a_1, \ldots, a_m) \rightsquigarrow^* \mathtt{f}(p_{j1}, \ldots, p_{jm})[\overline{y := c}] \rightsquigarrow e_j[\overline{y := c}]\,.$$

If we put both results together, we know that the computation of e terminates with result $r_e \equiv r_{e_j[\overline{y:=c}]}$ such that

$$\widehat{r_e} \equiv \widehat{r_{e_j[\overline{y:=c}]}} = e_j[\widehat{\overline{y := c}}][\overline{w := u}] = \widehat{e}[\overline{z := d_e}]\,,$$

as required by the statement.

— If $e \equiv (a_1\ a_2)$, we proceed as in the third case. Hence, we have

$$\Phi_e \equiv \Phi_{a_1}; \Phi_{a_2} \qquad \overline{d} \equiv \overline{d_{a_1}}, \overline{d_{a_2}}$$

with $\overline{d_{a_1}} \in \Phi_{a_1}$ and $\overline{d_{a_2}} \in \Phi_{a_2}$. By the induction hypothesis (second condition in the lexicographic order), the computation of a_1 and a_2 terminates with results r_{a_1} and r_{a_2} such that

$$\widehat{r_{a_1}} = \widehat{a_1}[\overline{z_1 := d_{a_1}}] \qquad \widehat{r_{a_2}} = \widehat{a_2}[\overline{z_2 := d_{a_2}}]\,.$$

Therefore, the evaluation of $(a_1\ a_2)$ also terminates (remember that the notation $(f\ a)$ is used only if f is a total function, hence it is guaranteed to terminate for every input and can be translated in a straightforward way into type theory) with result $r_e \equiv (r_{a_1}\ r_{a_2})$, so

$$\widehat{r_e} \equiv \widehat{(r_{a_1}\ r_{a_2})} = (\widehat{a_1}[\overline{z_1 := d_{a_1}}]\ \widehat{a_2}[\overline{z_2 := d_{a_2}}]) \equiv \widehat{e}[\overline{z := d}]\,.$$

— If $e \equiv [x]b$, we have two cases, according to whether Φ_b is empty or not.
If $\Phi_b \equiv ()$, we also have $\Phi_e \equiv ()$, and the statement is trivially true (the λ-abstraction is directly translated into type theory):

$$\widehat{e} \equiv [x]\widehat{b} \equiv \widehat{[x]b}\,.$$

If Φ_b is non-empty, the sequence of variables $\overline{z}$ in Φ_e contains only one element. Concretely,

$$\Phi_e \equiv H \in (x \in \widehat{\sigma})\Sigma\Phi_b \qquad \overline{z} \equiv H \qquad \overline{d_e} \equiv D \in (x \in \widehat{\sigma})\Sigma\Phi_b\,.$$

Recall that here $\widehat{e} \equiv [x]\widehat{b}[\overline{y := (\pi\ H(x))}]$, where $\overline{y}$ are the variables in Φ_b.
The computation of e terminates if and only if the computation of b terminates. By the induction hypothesis (second condition in the lexicographic order), we have that, if there is an instantiation $\overline{d_b} \in \Phi_b$, the computation of b terminates with a result r_b such that

$$\widehat{r_b} = \widehat{b}[\overline{y := d_b}]\,.$$

Observe that $\overline{(\pi\ D(x))}$, that is, the tuple of projections of $D(x)$, is an instantiation of Φ_b, so we can use $\overline{(\pi\ D(x))}$ as our $\overline{d_b}$. Then, the computation of b terminates and

$$\widehat{r_b} = \widehat{b}[\overline{y := (\pi\ D(x))}].$$

Therefore, the computation of e terminates with result $r_e \equiv [x]r_b$, so

$$\begin{aligned}
\widehat{r_e} &\equiv \widehat{[x]r_b} \\
&\equiv [x]\widehat{r_b} \\
&= [x]\widehat{b}[\overline{y := (\pi\ D(x))}] \\
&\equiv ([x]\widehat{b}[\overline{y := (\pi\ H(x))}])[H := D] \\
&\equiv \widehat{e}[H := D]\,.
\end{aligned}$$

□

In particular, if we apply the lemma to a term obtained by the application of a general recursive function to some values, we obtain a result stating that our translation of a general recursive function computes the same function as the original one.

In the rest of this paper, when we say that an expression is *defined*, we will mean that its computation terminates with a result (as described above).

Theorem 1. Let $\mathtt{f} : \sigma_1, \ldots, \sigma_m \Rightarrow \tau$ be a general recursive function in $\mathscr{FP}$. Let $\langle \mathsf{fAcc}, \mathsf{f} \rangle$ be its translation into type theory. Then, for every sequence of values $v_1 : \sigma_1, \ldots, v_m : \sigma_m$ in $\mathscr{FP}$, we have

$$\text{if } \mathsf{fAcc}(\widehat{v_1}, \ldots, \widehat{v_m}) \text{ is provable then } \mathtt{f} \text{ is defined on the inputs } v_1, \ldots, v_m$$

and, if $d_h \in \mathsf{fAcc}(\widehat{v_1}, \ldots, \widehat{v_m})$ is a closed proof, then

$$\mathsf{f}(\widehat{v_1}, \ldots, \widehat{v_m}, d_h) = \widehat{\mathtt{f}(v_1, \ldots, v_m)}\,.$$

Proof. The proof is immediate by applying the previous lemma to $e \equiv \mathtt{f}(v_1, \ldots, v_m)$. Observe that since $v_1, \ldots, v_m$ are values, that is, closed normal forms, they do not generate

any termination condition in the translation, and hence the instantiation $\overline{d} \in \Phi_e$ of the lemma is simply the proof d_h. □

The only unsatisfactory restriction to the power of Theorem 1 is that the inverse of the first part is not always true. It may happen that the computation of $\mathtt{f}(v_1, \ldots, v_m)$ terminates without $\mathsf{fAcc}(\widehat{v_1}, \ldots, \widehat{v_m})$ being provable. This bars us from stating that the translation of a functional program defines exactly the same function as the original one. We know that this unfortunate state of affairs is caused by the overly restrictive translation of λ-abstractions, thus we can obtain the full result if we restrict recursive definitions so that λ-abstractions are not used.

Lemma 2. Let $e : \tau$ be a valid term in $\mathscr{FP}$ with respect to Γ and $\mathscr{SF}$. If e does not contain any λ-abstraction and e is defined (that is, its computation terminates with a result), then there is a sequence of instantiations $\overline{d_e} \in \Phi_e$.

Proof. We use induction on the pair (l, e) where l is the length of the trace of the computation of e. Again, the inductive hypothesis is that the lemma is true for all pairs $(l', e') < (l, e)$, where $<$ is the lexicographic order on pairs.

We proceed by cases on the structure of e:

— If $e = x$ is a variable or $e \equiv \mathtt{g} \in \mathscr{F}$ is a structurally recursive function, then $\Phi_e \equiv ()$ and there is nothing to prove.

— If $e \equiv \mathtt{c}(a_1, \ldots, a_o)$, then $\Phi_e \equiv \Phi_{a_1}; \ldots; \Phi_{a_o}$. The induction hypothesis (second condition in the lexicographic order) states that there is an instantiation $\overline{d_{a_i}}$ of Φ_{a_i}, for $1 \leqslant i \leqslant o$. By putting these instantiations together, we obtain an instantiation $\overline{d_e} \equiv \overline{d_{a_1}}, \ldots, \overline{d_{a_o}}$ of Φ_e.

— If $e \equiv \mathtt{f}(a_1, \ldots, a_m)$, then $\Phi_e \equiv \Phi_{a_1}; \ldots; \Phi_{a_m}; h \in \mathsf{fAcc}(\widehat{a_1}, \ldots, \widehat{a_m})$. By the induction hypothesis (second condition in the lexicographic order), there are instantiations of all the translation contexts $\Phi_{a_1}, \ldots, \Phi_{a_m}$. Thus, we just need to find a proof of $\mathsf{fAcc}(\widehat{a_1}, \ldots, \widehat{a_m})$. By hypothesis, e is defined, so the sequence $(a_1, \ldots, a_m)$ must match one (and only one, by the exclusivity of the patterns) of the multipatterns in the recursive equations that define $\mathtt{f}$. Let us say that the input matches the multipattern $(p_{j1}, \ldots, p_{jm})$ in the jth equation of $\mathtt{f}$. This means that if $\overline{y}$ are the free variables occurring in the multipattern $\overline{p_j}$, there exists an instantiation $[\overline{y := c}]$ of those variables such that $a_i \rightsquigarrow^* p_{ji}[\overline{y := c}]$, for $1 \leqslant i \leqslant m$. So the first few steps in the computation of e are

$$e \equiv \mathtt{f}(a_1, \ldots, a_m) \rightsquigarrow^* \mathtt{f}(p_{j1}, \ldots, p_{jm})[\overline{y := c}] \rightsquigarrow e_j[\overline{y := c}]$$

where e_j is the right-hand side of the jth equation in the definition of $\mathtt{f}$.

By the induction hypothesis (first condition in the lexicographic order), there are instantiations $\overline{u} \in \Phi_{e_j[\overline{y:=c}]}$, and, therefore, we obtain the desired proof by

$$d_h = \mathtt{f\_acc}_j(\overline{\widehat{c}}, \overline{u}).$$

— If $e \equiv (a_1\ a_2)$, then $\Phi_e \equiv \Phi_{a_1}; \Phi_{a_2}$. This case is similar to the second case.

Since there are no λ-abstractions in e, we have completed all the possible cases and the statement is proved. □

Theorem 2. Let $\mathtt{f} : \sigma_1, \ldots, \sigma_m \Rightarrow \tau$ be a general recursive function in $\mathscr{FP}$ such that no λ-abstraction occurs in the right-hand sides of the equations defining $\mathtt{f}$. Let $\langle \mathsf{fAcc}, \mathsf{f} \rangle$ be its translation into type theory. Then, for any values $v_1 : \sigma_1, \ldots, v_m : \sigma_m$ in $\mathscr{FP}$, we have

$$\mathsf{fAcc}(\widehat{v_1}, \ldots, \widehat{v_m}) \text{ is provable} \iff \mathtt{f} \text{ is defined on the inputs } v_1, \ldots, v_m .$$

Proof. One direction has already been proved in Theorem 1. The other direction follows from Lemma 2 by noticing that no λ-abstraction can be introduced by the computation rules of $\mathscr{FP}$. □

Theorems 1 and 2 extend in the natural way to mutually recursive functions.

7. Guarded equations and case expressions

We have so far examined a very simple functional language $\mathscr{FP}$. In this section, we consider two possible extensions that make the language more expressive, but for which our translation method still works, albeit with some modifications. First, we consider guarded expressions in the equations that define a recursive function. Second, we extend the definition of valid terms to consider also case analysis on a term of an inductive data type.

The most dramatic change in the translation into type theory brought about by these extensions is that a single recursive equation in the definition of a function $\mathtt{f}$ may no longer translate to a single constructor for fAcc. A guarded equation generates one constructor for each guard. A case expression may also generate several constructors, one for each of its cases. Therefore, when we translate a term e in the right-hand side of an equation in the definition of $\mathtt{f}$, we no longer obtain a single context extension Φ_e and a single translated expression $\widehat{e}$, but a finite set of pairs $\langle \Phi_e, \widehat{e} \rangle$.

The proofs of Theorems 1 and 2 are still valid with slight changes to account for the translation of guarded equations and multiple translations of equations, and with extra cases in the secondary inductions because of the addition of case expressions as valid terms.

7.1. *Guarded equations*

In functional programming, guarded equations are often allowed in recursive definitions. That is, equations of the following form are quite common when defining a function:

$$\mathtt{f}(p_1, \ldots, p_m) = \begin{cases} e_1 \text{ if } c_1 \\ \quad \vdots \\ e_r \text{ if } c_r . \end{cases}$$

To support guarded equations, we should extend the set of types of $\mathscr{FP}$ with a built-in boolean type. Then, if $\mathscr{Y}$ is the set of pattern variables of the equation together with their types, the conditions $c_1, \ldots, c_r$ should be valid terms of boolean type with respect to $\mathscr{Y}$ and $\{\mathtt{f} : \sigma_1, \ldots, \sigma_m \Rightarrow \tau\}$. Hence, they can contain recursive calls to $\mathtt{f}$. To simplify the translation, we require that the boolean expressions are exclusive. This is

not really a very 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.

Observe that a guarded equation can be seen as r equations of the form

$$\begin{aligned} \mathtt{f}(p_1, \ldots, p_m) &= e_1 \text{ if } c_1 \\ &\vdots \\ \mathtt{f}(p_1, \ldots, p_m) &= e_r \text{ if } c_r \,. \end{aligned}$$

Given a tuple of arguments $(a_1, \ldots, a_m)$ matching the pattern $(p_1, \ldots, p_m)$, only the equation (if any) whose guard is satisfied can be used to compute $\mathtt{f}(a_1, \ldots, a_m)$. Remember that because of the exclusivity of the guards, at most one of the conditions $c_1, \ldots, c_r$ evaluates to `true` and hence, it is still the case that at most one equation can be applied to compute $\mathtt{f}(a_1, \ldots, a_m)$.

In the following we assume that all guarded equations have only one case determined by one condition. Then the general form of a guarded equation is

$$\mathtt{f}(p_1, \ldots, p_m) = e \text{ if } c \,.$$

Let this equation be the ith equation in the definition of `f` and let Γ be the type-theoretic context of the pattern variables in the equation.

Since c is a valid term of boolean type, we use Definition 5 for the translation of c into type theory. Notice that c might contain calls to the function `f`, and hence we also need to associate a context of assumptions to the translation of c. By means of Definition 5, we associate a context extension Φ_c and a type-theoretic term $\widehat{c}$ with the boolean expression c. Recall that the extension Φ_c is such that the context $\Gamma; \Phi_c$ contains the necessary assumptions to make $\widehat{c}$ a valid type-theoretic expression.

Similarly, we obtain the context extension Φ_e and the translation $\widehat{e}$ of e.

We change the definition of the constructor of fAcc associated with this equation by including the translation of c:

$$\mathsf{f\_acc}_i \in (\Gamma; \Phi_c; q \in \widehat{c} = \mathsf{true}; \Phi_e)\mathsf{fAcc}(\widehat{p_1}, \ldots, \widehat{p_m})$$

where $=$ is the propositional identity in type theory and true is the type-theoretic version of the boolean value `true`. Observe that this constructor can only be used to construct a proof of $\mathsf{fAcc}(\overline{a})$, for an input $\overline{a}$, when the term $\widehat{c}$ is true for $\overline{a}$.

The equation of f associated with the constructor $\mathsf{f\_acc}_i$ is

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

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

We conclude this subsection with an example that uses guarded equations: McCarthy's f_{91} function (Manna and McCarthy 1970). Here is its definition:

$$\begin{array}{l}\texttt{fix f\_91} : \texttt{Nat} \Rightarrow \texttt{Nat} \\ \quad \texttt{f\_91}(n) = n - 10 \qquad\qquad \text{if } n > 100 \\ \quad \texttt{f\_91}(n) = \texttt{f\_91}(\texttt{f\_91}(n + 11)) \quad \text{if } n \leqslant 100\end{array}$$

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

Following our method for guarded equations, we define $\mathsf{f}_{91}\mathsf{Acc}$ and f_{91} simultaneously as follows:

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

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

7.2. *Case expressions*

To allow case analysis on a term of an inductive data type, we extend the definition of valid terms with respect to the set of free variables $\mathscr{X}$ and the set of functions $\mathscr{SF}$ (Definition 4) and add the following item:

— Let t be a valid term of an inductive data type τ'. Let $p_1, \ldots, p_v$ be exclusive patterns of type τ' and let $\mathscr{Y}_s$ be the set of pattern variables in p_s together with their types, for $0 \leqslant v$ and $0 \leqslant s \leqslant v$. Finally, let $e_1, \ldots, e_v$ be valid terms of type τ with respect to $(\mathscr{X} \cup \mathscr{Y}_1), \ldots, (\mathscr{X} \cup \mathscr{Y}_v)$, respectively, and $\mathscr{SF}$. Then, the *case* expression

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

is a valid term of type τ.

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

Case expressions are computed by pattern matching. If $t : \tau'$ matches the case pattern p_s, that is, $t \rightsquigarrow^* p_s[\overline{y := a}]$ where $\overline{y}$ are the pattern variables in p_s, for some s, then the

computation rule is

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

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

Case expressions might impose a need for several constructors associated with an equation. The reason is that each branch of a case expression must be treated separately since it contributes to the type of the corresponding constructor in a different way.

We illustrate this point with the `quicksort` algorithm, which we will now present in a slightly different way:

$$\begin{array}{l} \texttt{fix quicksort} : \texttt{List Nat} \Rightarrow \texttt{List Nat} \\ \quad \texttt{quicksort}(xs) = \\ \qquad \texttt{Cases}\ xs\ \texttt{of} \begin{cases} \texttt{nil} & \mapsto \texttt{nil} \\ \texttt{cons}(x, xs') & \mapsto \texttt{quicksort}(\texttt{filter}\ (<x)\ xs') +\!+ \\ & \qquad \texttt{cons}(x, \texttt{quicksort}(\texttt{filter}\ (>=x)\ xs')). \end{cases} \end{array}$$

It is easy to see that, from the computational point of view, this version of `quicksort` is equivalent to the Haskell version that we presented in Section 3. It is logical to expect a similar domain predicate for both versions of the algorithm. We hope that by now it is clear that the two branches in the case analysis should impose different constraints to the domain predicate, despite both branches belonging to the same equation in the definition of `quicksort`. Concretely, the first branch indicates that `quicksort` terminates on the empty list, while the second branch tells us that we can only establish the termination of `quicksort` on a non-empty list if we have proofs that `quicksort` terminates on the lists on which we perform the recursive calls.

However, not all case expressions introduce multiple constructors. Some case expressions, which 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 translated straightforwardly into type theory without further analysis. Note that the expression on which we perform case analysis might still introduce partiality.

Since a single equation in `f` may now be associated with several constructors of fAcc and, consequently, with several equations of f, we need to associate a sequence $\mathscr{P}_e(\Gamma)$ of pairs $\langle \Phi_e, \widehat{e} \rangle$ of context extensions Φ_e and type-theoretic terms $\widehat{e}$ with the right-hand side e of the equation, where Γ is the usual type-theoretic context for the pattern variables of the equation.

The translation strictly follows the one given in Section 5 except that it has an extra case for case expressions, and it deals with a list of pairs ⟨context extension, term⟩ instead of a single one. Hence, when using the recursive translations of the subterms, we need to combine every element of every recursive call in all possible ways. Recall that since each context extension extends Γ, we should only introduce fresh variables in them.

Definition 6. Given a valid term e, we define $\mathscr{P}_e(\Gamma)$ by recursion on the structure of e. We use $\langle \Phi_e, \widehat{e} \rangle$ to denote a generic element of $\mathscr{P}_e(\Gamma)$, and $\#\mathscr{P}_e$ to denote the number of elements in $\mathscr{P}_e(\Gamma)$.

$e \equiv z$**:** If the expression e is a variable, then $\mathscr{P}_e(\Gamma) \equiv \{\langle (), z \rangle\}$.

$e \equiv \mathtt{g}$**:** If e is one of the functions $\mathtt{g}$ in $\mathscr{F}$, then $\mathscr{P}_e(\Gamma) \equiv \{\langle (), \widehat{\mathtt{g}} \rangle\}$.

$e \equiv \mathtt{c}(a_1, \ldots, a_o)$**:** First we determine $\mathscr{P}_{a_1}(\Gamma), \ldots, \mathscr{P}_{a_o}(\Gamma)$ by structural recursion, and then combine these sequences into the definition of $\mathscr{P}_e(\Gamma)$. Formally, we have

$$\mathscr{P}_e(\Gamma) \equiv \{\langle \Phi_{a_1}; \ldots; \Phi_{a_o},\ \widehat{\mathtt{c}}(\widehat{a_1}, \ldots, \widehat{a_o}) \rangle \mid \langle \Phi_{a_i}, \widehat{a_i} \rangle \in \mathscr{P}_{a_i}(\Gamma), 1 \leqslant i \leqslant o\}.$$

$e \equiv \mathtt{f}(a_1, \ldots, a_m)$**:** Again we first determine $\mathscr{P}_{a_1}(\Gamma)$, …, $\mathscr{P}_{a_m}(\Gamma)$ by structural recursion, and then combine these sequences into the definition of $\mathscr{P}_e(\Gamma)$. Recall that we have to add the assumption corresponding to the recursive call $\mathtt{f}(a_1, \ldots, a_m)$. Hence, we have

$$\begin{aligned}\mathscr{P}_e(\Gamma) \equiv\ & \{\langle \Phi_{a_1}; \ldots; \Phi_{a_m}; h \in \mathsf{fAcc}(\widehat{a_1}, \ldots, \widehat{a_m}),\ \mathtt{f}(\widehat{a_1}, \ldots, \widehat{a_m}, h) \rangle \mid \\ & \langle \Phi_{a_i}, \widehat{a_i} \rangle \in \mathscr{P}_{a_i}(\Gamma), 1 \leqslant i \leqslant m\}.\end{aligned}$$

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

$$\mathscr{P}_e(\Gamma) \equiv \{\langle \Phi_{a_1}; \Phi_{a_2},\ \widehat{a_1}(\widehat{a_2}) \rangle \mid \langle \Phi_{a_1}, \widehat{a_1} \rangle \in \mathscr{P}_{a_1}(\Gamma), \langle \Phi_{a_2}, \widehat{a_2} \rangle \in \mathscr{P}_{a_2}(\Gamma)\}.$$

$e \equiv [z]b$**:** Let σ be the type of z. We first calculate $\mathscr{P}_b(\Gamma; z \in \widehat{\sigma})$ recursively.
If $\mathscr{P}_b(\Gamma; z \in \widehat{\sigma}) = \{\langle (), \widehat{b} \rangle\}$, that is, if $\mathscr{P}_b(\Gamma; z \in \widehat{\sigma})$ contains only one pair and the context extension in that pair is empty, then

$$\mathscr{P}_b(\Gamma) \equiv \{\langle (), [z]\,\widehat{b} \rangle\}.$$

Otherwise, let $\mathscr{P}_b(\Gamma; z \in \widehat{\sigma}) = \{\langle \Phi_{b1}, \widehat{b}_1 \rangle, \ldots, \langle \Phi_{b\#\mathscr{P}_b}, \widehat{b}_{\#\mathscr{P}_b} \rangle\}$. The assumption generated by e must be the universal quantification over z of all the assumptions for $\widehat{b}$. Let $\overline{y_{\Phi_{bi}}}$ be the variables in Φ_{bi}, for $1 \leqslant i \leqslant \#\mathscr{P}_b$. We define

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

with

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

If $\#\mathscr{P}_b = 1$, we do not need to construct a disjoint union type. Hence,

$$\mathscr{P}_e(\Gamma) \equiv \{\langle H \in (z \in \widehat{\sigma})\, \Sigma\Phi_b,\ \widehat{e} \rangle\} \quad \text{with} \quad \widehat{e} \equiv [z]\widehat{b}[\overline{y_{\Phi_b} := (\pi\ H(z))}].$$

If Φ_b contains only one assumption, we do not need to construct a Σ-type. Let $\Phi_b \equiv y_{\Phi_b} \in \alpha$, and we simply have

$$\mathscr{P}_e(\Gamma) \equiv \{\langle H \in (z \in \widehat{\sigma})\alpha,\ [z]\widehat{b}[y_{\Phi_b} := H(z)] \rangle\}.$$

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

$a \equiv \texttt{Cases}\ b\ \texttt{of}\ \begin{cases} p_1 \mapsto a_1 \\ \quad\vdots \quad : \\ p_v \mapsto a_v \end{cases}$ First let us consider the translation of patterns. Variables, constructors and constructor applications are translated into type theory in a straightforward way, without generating new termination conditions. Hence, the translation of a pattern p is direct and $\Phi_p(\Gamma)$ is just $\{\langle (), \widehat{p}\rangle\}$.

If none of the different branches of the case expression contain recursive calls or introduce partiality, we can give a straightforward translation. We call such case expressions *safe*, and translate them directly as case expressions in type theory. Formally, a case expression

$$\texttt{Cases}\ b\ \texttt{of}\ \begin{cases} p_1 \mapsto a_1 \\ \quad\vdots \\ p_v \mapsto a_v \end{cases}$$

is *safe* if the patterns $p_1, \ldots, p_v$ are exclusive and exhaustive and $\mathscr{P}_{a_s}(\Gamma) = \{\langle (), \widehat{a_s}\rangle\}$, for $0 \leqslant s \leqslant v$.

For a safe case expression we define

$$\mathscr{P}_e(\Gamma) = \{\langle \Phi, \widehat{e}\rangle \mid \langle \Phi, \widehat{b}\rangle \in \mathscr{P}_b(\Gamma)\} \quad \text{with} \quad \widehat{e} = \mathsf{Cases}\ \widehat{b}\ \mathsf{of}\ \begin{cases} \widehat{p_1} \mapsto \widehat{a_1} \\ \quad\vdots \\ \widehat{p_v} \mapsto \widehat{a_v}\,. \end{cases}$$

If the case expression is not safe, each of the different branches imposes the need for 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. We assume that the variables $\overline{y_s}$ are fresh with respect to Γ.

Let us use Γ_s to denote $\overline{(y_s \in \widehat{\sigma_s})}$. As we said earlier, each branch in the case expression imposes the need for at least one different constructor. Notice that each a_s may impose a need for several constructors, namely $\#\mathscr{P}_{a_s}$. Then, the number of constructors corresponding to the sth branch is also $\#\mathscr{P}_{a_s}$. 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 a need for several constructors, and thus it might contribute to the type of the different constructors. Hence, we should combine the elements in $\mathscr{P}_b(\Gamma)$ and in $\mathscr{P}_{a_s}(\Gamma; \Gamma_s)$ in all possible ways. Formally, we determine $\mathscr{P}_b(\Gamma), \mathscr{P}_{a_1}(\Gamma; \Gamma_1), \ldots, \mathscr{P}_{a_v}(\Gamma; \Gamma_v)$ by structural recursion, and then we define

$$\begin{aligned} \mathscr{P}_e(\Gamma) \equiv\ & \{\langle \Phi_b; \Gamma_s; q_s \in (\widehat{b} = \widehat{p_s}); \Phi_{a_s},\ \widehat{a_s}\rangle \mid \\ & \langle \Phi_b, \widehat{b}\rangle \in \mathscr{P}_b(\Gamma), \langle \Phi_{a_s}, \widehat{a_s}\rangle \in \mathscr{P}_{a_s}(\Gamma; \Gamma_s), 1 \leqslant s \leqslant v\}\,. \end{aligned}$$

This completes the definition of $\mathscr{P}_e(\Gamma)$.

It remains to explain how these changes modify the definition of the constructors of fAcc and of the equations of f. For the sake of generality, we consider a guarded equation

here. So, let us assume that the ith equation in the definition of f is

$$\mathtt{f}(p_1, \ldots, p_m) = e \text{ if } c\,.$$

Using the definition presented above, we determine $\mathscr{P}_c(\Gamma)$ and $\mathscr{P}_e(\Gamma)$ for Γ the usual type-theoretic context of pattern variables. To define the constructors of fAcc and the equations of f, we should combine the elements in $\mathscr{P}_c(\Gamma)$ and $\mathscr{P}_e(\Gamma)$ in all possible ways. Let $\langle \Phi_{cr}, \widehat{c}_r \rangle$ be the rth element of $\mathscr{P}_c(\Gamma)$ and $\langle \Phi_{el}, \widehat{e}_l \rangle$ be the lth element of $\mathscr{P}_e(\Gamma)$. Then, the rlth constructor of fAcc corresponding to the ith equation is

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

If the corresponding equation is a non-conditional equation, then $\mathscr{P}_c(\Gamma)$ is empty and the assumptions $\Phi_{c_r}; q_r \in (t_{c_r} = \mathsf{true})$ are absent from 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.

The equation of f corresponding to the above constructor is

$$\mathsf{f}(\widehat{p_1}, \ldots, \widehat{p_m}, \mathsf{facc}_{irl}(\overline{y}, \overline{x_{\Phi_{cr}}}, q_r, \overline{z_{\Phi_{el}}})) = \widehat{e}_l$$

where $\overline{y}, \overline{x_{\Phi_{cr}}}$ and $\overline{z_{\Phi_{el}}}$ are the sequences of variables assumed in Γ, Φ_{cr} and Φ_{el}, respectively, and q_r is a variable of type $(\widehat{c}_r = \mathsf{true})$.

This completes the type-theoretic definition of fAcc and f for the extension of our method with guarded equations and case expressions.

Following this description, the translation of the quicksort algorithm presented with a case expression is as follows:

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

8. Conclusions

We have described a method for translating a vast class of algorithms from functional programming into type theory. We have defined the class $\mathscr{FP}$ of algorithms to which the method applies. This class is large enough to allow the implementation of all partial recursive functions. We have given a formal definition of the translation of the programs in $\mathscr{FP}$ into type theory. In addition, we have proved that the translation is sound with respect to strict semantics.

Although we do not focus on formal verification in this paper, we should mention that our method also simplifies the task of proving properties about the defined algorithms.

Since our method succeeds in separating the logical part from the computational part, the resulting definitions are simple and the formal verification of their properties becomes dramatically easier. Readers interested in this aspect should refer to Bove (1999; 2003).

As a final remark, we would like to point out that given a function f in $\mathscr{FP}$ and a particular input $\overline{a}$ for the function, the canonical form of the proof of $\mathsf{fAcc}(\overline{a})$ is a trace of the computation of $\mathtt{f}(\overline{a})$. Therefore, the structural complexity of the proofs in fAcc are proportional to the number of steps in the algorithm.

In a recent work (Bove and Capretta 2004), we explored the formalisation of our method in an impredicative type theory. To do this, we define a new type of partial function containing the domain predicate as part of the information in the type. Then, we translate a function in the functional program as a member of the type of partial functions, rather than as an instance of the type of total functions as we do in the current paper. This allows us to lift all the restrictions we imposed on our programs in this paper.

Related work

In recent years, there has been increasing interest in the formalisation of general recursion. There are a few different approaches to the matter.

Nordström (1988) uses the predicate Acc for that purpose.

One can adopt a set-theoretic approach and view functions as relations. Specifically, the behaviour of a recursive function can be described by an inductive relation giving its operational semantics (see, for example, Winskel (1993)). Operational semantics has been developed in type theory in Bertot *et al.* (2002) and Zhang *et al.* (2002). However, relations do not have any computational content in type theory. The real challenge consists in representing general recursive programs as elements of some functional type.

Using classical logic, it is possible to extend every partial function to a total one. This fact is used in Finn *et al.* (1997) to give a formalisation of partial recursive functions inside a classical axiomatic logic system. Their implementation associates a domain predicate to each function, in a way similar to our approach. However, the predicate is not used to define the function, but just to restrict the scope of the recursive equations. Except for the translation of case expressions (for which they take the conjunction of all the assumptions that arise in the different branches of the case expression, instead of considering each branch in a separate way, as we do), an algorithm is analysed as in our method. Moreover, they give a similar interpretation when bound variables are present in the right-hand side of the equations, and they arrive at similar conclusions about the semantics associated to the formalisation of their programs. In addition, they encounter analogous problems when using higher order functions in recursive definitions. However, the different settings in which the two works are formulated give rise to some differences. A function f is formalised in Finn *et al.* (1997) with the type that it has in a functional programming language, without the need for our extra parameter fAcc as part of the type of f. However, a function f obeys its definition in Finn *et al.* (1997) provided its arguments can be proved to be in the domain of the function, which is called DOM′f. Once (and if) the function has been proved total, one can forget about DOM′f, which is not possible in type theory. Another important difference is that in Finn *et al.* (1997), an application (f *e*) is always

defined since the cases in which (`f` *e*) is not defined are considered to return an unspecified value of the corresponding type. However, this causes some problems since the semantics they use is not capable of reflecting this distinction, and sometimes one can prove things like `f` $0 = 0$ when `f` 0 diverges.

Another similar technique is the extraction of special induction schemes from recursive function definitions (see Walther (1992) and the *Recursion analysis* section of Bundy (2001)), which is specified in Hutter (1992) and fully automated in the INKA theorem prover.

Wiedijk and Zwanenburg (2003) shows how standard classical first-order logic can be used to reason about partial functions in type theory. The authors prove an equivalence between a system in which function application requires a proof term certifying that the argument is in the domain, as in our case, and a simple first-order language without proof terms in which every function is total.

Dubois and Donzeau-Gouge (1998) takes an approach to the problem similar to ours. 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 λ-abstractions as possible expressions, which greatly simplifies the translation. 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 for 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 (2000) uses fix-point equations to obtain the desired equalities for the recursive definitions. However, the solution presented is rather complex and does not really succeed in separating the actual algorithms and their termination proofs. Balaa and Bertot (2002) again uses fix-points to approach the problem, but this 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. On the other hand, it is not very clear how their methods can be used to formalise partial or nested recursive algorithms.

Bertot *et al.* (2002) presents a technique to encode the method we describe in Bove and Capretta (2001) 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 Balaa and Bertot (2002).

Some work has been done for simply typed λ-calculus with inductive types where the termination of recursive functions is ensured by types. Barthe *et al.* (2004) presents a type-based system $\lambda\hat{}$ that ensures the termination of recursive functions through the notion of stage, which is used to restrict the arguments of recursive calls. In Barthe *et al.* (2004), the system $\lambda\hat{}$ is proved to be strongly normalising and to satisfy the property of subject reduction. Although this system seems a good candidate for use in proof-assistants based on type theory, some work still needs to be done before this can be actually carried out in practise, namely, the scaling of $\lambda\hat{}$ up to dependent types and the development of type checking and type inference algorithms for the system.

Abel (2004) proceeds along the same lines, but with some differences in the type system. The core language in Abel (2004), and the properties of the system presented there, are basically the same as in Barthe *et al.* (2004). In addition, in Abel (2004), a type checking algorithm is given for the system. The key concept in this work is the use of decorated type variables, which play a similar role to stages in Barthe *et al.* (2004). In this case also, the system needs to be scaled up to dependent types before it can be used in proof-assistants for type theory.

Another relevant publication that treats the problem of formalising partial functions and proving termination is McBride and McKinna (2004).

Acknowledgements

We want to thank Yves Bertot and Björn von Sydow for carefully reading and commenting on previous versions of this paper. We are grateful to Jörgen Gustavsson for fruitful discussions on the semantics of $\mathscr{FP}$. Finally, we thank an anonymous referee for his/her valuable comments.

References

Abel, A. (2004) Termination checking with types. In: Fix Points in Computer Science 2003 (FICS03). Special issue of *Theoretical Informatics and Applications* (RAIRO).

Aczel, P. (1977) An Introduction to Inductive Definitions. In: Barwise, J. (ed.) *Handbook of Mathematical Logic*, North-Holland Publishing Company 739–782.

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

Balaa, A. and Bertot, Y. (2002) Fonctions récursives générales par itération en théorie des types. *Journées Francophones des Langages Applicatifs* JFLA02, INRIA.

Barendregt, H. and Geuvers, H. (2001) Proof-assistants using dependent type systems. In: Robinson, A. and Voronkov, A. (eds.) *Handbook of Automated Reasoning*, Chapter 18, Elsevier 1149–1238.

Barendregt, H. P. (1992) Lambda calculi with types. In: Abramsky, S., Gabbay, D. M. and Maibaum, T. S. E. (eds.) *Handbook of Logic in Computer Science* **2**, Oxford University Press, 117–309.

Barthe, G., Frade, M., Giménez, E., Pinto, L. and Uustalu, T. (2004) Type-based termination of recursive definitions. *Mathematical Structures in Computer Science* **14** (1) 97–141.

Bell, J. L. and Machover, M. (1977) *A course in mathematical logic*, North-Holland.

Bertot, Y., Capretta, V. and Barman, K. D. (2002) Type-theoretic functional semantics. In: Carreno, V. A., Munoz, C. A. and Tahar, S. (eds.) Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002. *Springer-Verlag Lecture Notes in Computer Science* **2410** 83–97.

Bove, A. (1999) *Programming in Martin-Löf type theory: Unification – A non-trivial example*, Licentiate Thesis of the Department of Computer Science, Chalmers University of Technology. (Available at `http://www.cs.chalmers.se/~bove/Papers/lic_thesis.ps.gz`.)

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

Bove, A. (2002a) *General Recursion in Type Theory*, Ph.D. thesis, Chalmers University of Technology, Department of Computing Science. (Available at `http://www.cs.chalmers.se/~bove/Papers/phd_thesis.ps.gz`.)

Bove, A. (2002b) Mutual general recursion in type theory. Technical Report, Chalmers University of Technology. (Available at `http://www.cs.chalmers.se/~bove/Papers/mutual_rec.ps.gz`.)

Bove, A. (2003) General recursion in type theory. In: Geuvers, H. and Wiedijk, F. (eds.) Types for Proofs and Programs, International Workshop TYPES 2002, The Netherlands. *Springer-Verlag Lecture Notes in Computer Science* **2646** 39–58.

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

Bove, A. and Capretta, V. (2005) Recursive functions with higher order domains. In: Urzyczyn, P. (ed.) Typed Lambda Calculi and Applications. 7th International Conference, TLCA 2005, Nara, Japan. *Springer-Verlag Lecture Notes in Computer Science* **3461** 116–130.

Bundy, A. (2001) The automation of proof by mathematical induction. In: Robinson, A. and Voronkov, A. (eds.) *Handbook of Automated Reasoning* **1**, Elsevier Science and MIT Press 845–912.

Capretta, V. (2002) *Abstraction and Computation*, Ph.D thesis, Computing Science Institute, University of Nijmegen.

Carreno, V. A., Munoz, C. A. and Tahar, S. (eds.) (2002) Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002. *Springer-Verlag Lecture Notes in Computer Science* **2410**.

Coquand, T. and Huet, G. (1988) The Calculus of Constructions. *Information and Computation* **76** 95–120.

Coquand, T., Nordström, B., Smith, J. M. and von Sydow, B. (1994) Type theory and programming. *EATCS* **52**.

Coquand, T. and Paulin, C. (1990) Inductively defined types. In: Martin-Löf, P. (ed.) Proceedings of Colog '88. *Springer-Verlag Lecture Notes in Computer Science* **417**.

de Mast, P., Jansen, J.-M., Bruin, D., Fokker, J., Koopman, P., Smetsers, S., van Eekelen, M. and Plasmeijer, R. (2001) Functional Programming in Clean, Computing Science Institute, University of Nijmegen.

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

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

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

Hagino, T. (1987) *A Categorical Programming Language*, Ph.D. thesis, University of Edinburgh.

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

Hutter, D. (1992) *Automatisierung der vollständigen Induktion*, Oldenbourg Verlag.

Jones, S. P., (ed.) (2003) *Haskell 98 Language and Libraries. The Revised Report*, Cambridge University Press.

Manna, Z. and McCarthy, J. (1970) Properties of programs and partial function logic. *Machine Intelligence* **5** 27–37.

Martin-Löf, P. (1984) *Intuitionistic Type Theory*, Bibliopolis, Napoli.

McBride, C. and McKinna, J. (2004) The view from the left. *Journal of Functional Programming* **14** (1) 69–111.

Milner, R., Tofte, M., Harper, R. and MacQueen, D. (1997) *The Definition of Standard ML*, MIT Press.

Nordström, B. (1988) Terminating General Recursion. *BIT* **28** (3) 605–619.

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

Paulson, L. C. (1986) Proving Termination of Normalization Functions for Conditional Expressions. *Journal of Automated Reasoning* **2** 63–74.

Pfenning, F. and Paulin-Mohring, C. (1990) Inductively defined types in the Calculus of Constructions. In: Proceedings of Mathematical Foundations of Programming Semantics. *Springer-Verlag Lecture Notes in Computer Science* **442**. (Also, Technical report CMU-CS-89-209.)

Phillips, J. C. C. (1992) Recursion theory. *Handbook of Logic in Computer Science* **1** (Background: Mathematical Structures), Oxford University Press 79–187.

Sørensen, M. H. B. and Urzyczyn, P. (1998) Lectures on the Curry–Howard isomorphism. Available as DIKU Rapport 98/14.

Walther, C. (1992) Computing induction axioms. In: International Conference on Logic Programming and Automated Reasoning, LPAR 92. *Springer-Verlag Lecture Notes in Artificial Intelligence* **624**.

Werner, B. (1994) *Méta-théorie du Calcul des Constructions Inductives*, Ph.D. thesis, Université Paris 7.

Wiedijk, F. and Zwanenburg, J. (2003) First order logic with domain conditions. In: Theorem Proving in Higher Order Logics, TPHOLs 2003. *Springer-Verlag Lecture Notes in Computer Science* **2758** 221–237.

Winskel, G. (1993) *The formal semantics of programming languages: An introduction*, The MIT Press.

Zhang, X., Munro, M., Harman, M. and Hu, L. (2002) Weakest precondition for general recursive programs formalized in Coq. In: Carreno, V. A., Munoz, C. A. and Tahar, S. (eds.) Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002. *Springer-Verlag Lecture Notes in Computer Science* **2410** 332–347.