Recursive Functions with Higher Order Domains

Ana Bove[1] and Venanzio Capretta[2]

[1] Department of Computing Science, Chalmers University of Technology,
412 96 Göteborg, Sweden
telephone: +46-31-7721020, fax: +46-31-165655
bove@cs.chalmers.se
[2] Department of Mathematics and Statistics, University of Ottawa,
585 King Edward, Ottawa, Canada
telephone: +1-613-562-5800 extension 2103, fax: +1-613-562-5776
venanzio.capretta@mathstat.uottawa.ca

Abstract. In a series of articles, we developed a method to translate general recursive functions written in a functional programming style into constructive type theory. Three problems remained: the method could not properly deal with functions taking functional arguments, the translation of terms containing λ-abstractions was too strict, and partial application of general recursive functions was not allowed. Here, we show how the three problems can be solved by defining a type of partial functions between given types. Every function, including arguments to higher order functions, λ-abstractions and partially applied functions, is then translated as a pair consisting of a domain predicate and a function dependent on the predicate. Higher order functions are assigned domain predicates that inherit termination conditions from their functional arguments. The translation of a λ-abstraction does not need to be total anymore, but generates a local termination condition. The domain predicate of a partially applied function is defined by fixing the given arguments in the domain of the original function. As in our previous articles, simultaneous induction-recursion is required to deal with nested recursive functions. Since by using our method the inductive definition of the domain predicate can refer globally to the domain predicate itself, here we need to work on an impredicative type theory for the method to apply to all functions. However, in most practical cases the method can be adapted to work on a predicative type theory with type universes.

1 Introduction

In functional programming, functions can be defined by recursive equations where the arguments of the recursive calls are not required to be smaller than the input, hence allowing the definition of general recursive functions. Thus, the termination of a program is not guaranteed by its structure. On the other hand, in type theory, only structurally recursive functions are allowed, that is, functions where the recursive calls are performed only on arguments structurally smaller than the input. Thus, some functional programs have no direct translation into type theory.

P. Urzyczyn (Ed.): TLCA 2005, LNCS 3461, pp. 116–130, 2005.

In a series of articles, we have developed a method to translate functional programs into constructive type theory. Given a general recursive function, the method consists in defining an inductive predicate that characterises the inputs on which the function terminates. We can think of this predicate as the domain of the function. The type-theoretic version of the function can then be defined by structural recursion on the proof that the input values satisfy this predicate. A similar method was independently developed by Dubois and Viguié Donzeau-Gouge [DDG98], however they treat nested recursion in a different way and they do not consider the issues we tackle in the present article.

Given a general recursive function $\mathtt{f}$ from σ to τ, its formalisation in type theory following our method consists of an inductive predicate fAcc and a function f with types

$$\begin{aligned}&\mathsf{fAcc}\colon \widehat{\sigma} \to \mathsf{Prop}\\ &\mathsf{f}\colon (x\colon \widehat{\sigma}; \mathsf{fAcc}\ x) \to \widehat{\tau}\end{aligned}$$

where $\widehat{\sigma}$ and $\widehat{\tau}$ are the type-theoretic translations of σ and τ, respectively, and f is defined by structural recursion on its second argument.

Intuitively, if the ith recursive equation of the original program is

$$\mathtt{f}(p) = \cdots \mathtt{f}(a_1) \cdots \mathtt{f}(a_n) \cdots$$

calling itself recursively on the arguments $a_1, \ldots, a_n$, then the ith constructor of fAcc has type

$$\mathsf{facc}_i\colon (\ldots; \mathsf{fAcc}\ a_1; \ldots; \mathsf{fAcc}\ a_n)(\mathsf{fAcc}\ p)$$

and the ith structural recursive equation of f is

$$\mathsf{f}\ p\ (\mathsf{facc}_i\ \cdots\ h_1\ \cdots\ h_n) = \cdots (\mathsf{f}\ a_1\ h_1) \cdots (\mathsf{f}\ a_n\ h_n) \cdots .$$

In practise, the types of the constructors of fAcc and the structure of the equations of f may be more complex, but the idea remains the same: fAcc is inductively defined so that proving fAcc on an input p requires proofs of fAcc for the arguments of all the recursive calls that $\mathtt{f}$ performs when applied to p.

The method was introduced by Bove [Bov01] to formalise simple general recursive algorithms in constructive type theory (by simple we mean non-nested and non-mutually recursive). It was extended by Bove and Capretta [BC01] to treat nested recursion by using Dybjer's simultaneous induction-recursion, and by Bove [Bov02a] to treat mutually recursive algorithms, nested or not. A formal description of the method is given in [BC04] where we also prove a soundness and a weak completeness theorem. The first three papers mentioned above and a previous version of [BC04] have been put together into the first author's Ph.D. thesis [Bov02b]. A tutorial on the method can be found in [Bov03].

The method of [BC04] separates the computational and logical parts of the type-theoretic versions of the functional programs. An immediate consequence is that it allows the formalisation of partial functions: proving that a certain function is total amounts to proving that the corresponding domain predicate is satisfied by every input. Another consequence is that the resulting type-theoretic

algorithms are clear, compact and easy to understand. They are as simple as their counterparts in a functional programming language. However, our method has some problems and limitations which we have already mentioned in [BC04].

The first problem concerns higher order functions, that is, functions that take other functions as arguments. For example, let us consider the function `map`: $(\sigma \to \tau) \to [\sigma] \to [\tau]$, as defined in the Haskell [Jon03] prelude, which has a first argument in a functional type. Since `map` is structurally recursive on its list argument, in [BC04] we translate it into a structurally recursive function map in type theory with type $(\widehat{\sigma} \to \widehat{\tau}) \to [\widehat{\sigma}] \to [\widehat{\tau}]$. This means that the first argument of map can only be instantiated to a total function of type $\widehat{\sigma} \to \widehat{\tau}$. But in functional programming `map` could be applied to potentially non-terminating functions. Therefore, even if `map` is structurally recursive, it is still liable to non-termination since its functional argument f might be undefined on some (or all) of the elements to which it will be applied. In other words, `map` inherits the termination conditions from its functional argument. In [BC04] we had no way to express this fact. Therefore, our translation was too restrictive with respect to the original functional program.

Another problem is that whenever there is a λ-abstraction in the right-hand side of an equation, the method of [BC04] translates it as a total function in type theory. This interpretation is too strict, since the corresponding function could diverge on arguments to which it is not actually applied during execution, without jeopardising the termination behaviour of the program. More specifically, if under the scope of the λ-abstraction there is a call to one of the general recursive functions we are defining in our program, let us say `f`, the method inductively requires every instantiation of the argument of the λ-abstraction to be in the domain of `f`. However, this constraint might be stronger than actually needed since the λ-abstraction may just be applied to a proper subset of all the instantiations. This problem is already present in a classical setting in the work of Finn, Fourman, and Longley [FFL97].

The third problem is that partial applications of general recursive functions are not allowed in [BC04]. When applying a recursive function `f` taking m arguments $a_1, \ldots, a_m$, we must also provide a proof h of the accessibility of $a_1, \ldots, a_m$. If `f` is applied to an insufficient number of arguments $a_1, \ldots, a_k$ with $k < m$, then the accessibility condition cannot even be formulated and it is not possible to prove that the result of the application converges. Therefore, we barred partial applications of general recursive functions.

Here, we introduce a type of partial functions in type theory and we present a new method to translate functional programs into their type-theoretic equivalents. The method is based on the one presented in [BC04] but, now, it translates every function in the functional side into an element of the new type of partial functions. With this new approach, the problems we mention above disappear.

For our new method to be applicable to any function, we need to work in a type theory with an impredicative universe Set with inductive-recursive definitions à la Dybjer [Dyb00]. Both datatypes and propositions are represented as elements of Set. Therefore, we use the Calculus of Construction [CH88] extended

with a schema for simultaneous inductive-recursive definitions. A justification of the soundness of inductive-recursive definitions in the Calculus of Constructions is given by [Bla03] and [Cap04]. The method works also in the slightly different type architecture used in the proof assistant Coq [Coq02]. There are in Coq two impredicative universes, Set and Prop, both being elements of the predicative universe Type. For the purpose of this work, it is possible to use Prop for the domain predicate and Set for the functions. Usually, elimination of a proposition over a set (essential in our method) is not allowed in Coq, a feature intended to prevent mixing logical information with pure computational content. However, Christine Paulin [Pau] devised a way around this difficulty, consisting in defining structural deconstruction functions on propositions that have the property of uniqueness of proofs, as is the case of our domain predicates. Therefore, we use the notation Set for datatypes and Prop for propositions, to be interpreted as either the same impredicative universe or two distinct impredicative universes, in which case Paulin's method is to be used in place of structural recursion on the proofs of accessibility. As we point out later, it would be possible to adapt the method to work on a predicative type theory. However, we would loose generality since then the method could not be used to translate all functions (see function `itz` in section 4).

An extension of type theory with a constructor $A \rightsquigarrow B$ for partial functions from A to B was already proposed by Constable and Mendler in [CM85]. Together with the type $A \rightsquigarrow B$ they introduce a new form of canonical elements, which is not the case in our partial functions type. They are of the form $fix(f, x.F)$, to be intended as the functions with definition $f(x) = F$. From the definition of f one can construct its domain $dom(f)(x)$, essentially as in our method, as a recursive predicate generated structurally by the recursive calls to f in F. The roles of both domain predicates are however quite different. While our functions are defined by structural recursion on their domain predicates, the predicates in [CM85] serve only as a way to characterise the valid inputs and the functions in [CM85] are defined independently of their domain predicates.

This paper is mainly intended for readers with some knowledge in type theory. This said, what follows is just intended to fix the notation.

A *context* Γ is a sequence of assumptions $\Gamma \equiv x_1 \colon \alpha_1; \ldots ; x_n \colon \alpha_n$ 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. We call a sequence of variable assumptions Δ a *context extension* of the context Γ if $\Gamma; \Delta$ is a context.

If α is a type and β is a family of types over α, we write $(x \colon \alpha)\beta(x)$ for the type of dependent functions from α to β. If β does not depend on values of type α we might simply write $\alpha \to \beta$ for the type of functions from α to β. Functions have abstractions as canonical values, which we write $[x \colon \alpha]e$. Consecutive dependent function types and abstractions are written $(x_1 \colon \alpha_1; \ldots ; x_n \colon \alpha_n)\beta(x_1, \ldots , x_n)$ and $[x_1 \colon \alpha_1, \ldots , x_n \colon \alpha_n]e$, respectively. In either case, each α_i can contain occurrences of the variables that precede it. If β doest not depend on the last assumption x_n, then we can write $(x_1 \colon \alpha_1; \ldots ; x_{n-1} \colon \alpha_{n-1}; \alpha_n) \to \beta(x_1, \ldots , x_{n-1})$.

For the sake of simplicity, we will use the same notation as much as possible both in the functional programming side and in the type-theoretic side. In addition, names in the functional programming side will be written with `typewriter` font while names in the type-theoretic side will be written with Sans Serif font.

The article has the following organisation. Section 2 briefly presents the method of [BC04]. Section 3 defines the type of partial functions and gives a formal definition of the new translation. Section 4 illustrates the translation on some examples. Finally, Section 5 discusses advantages and disadvantages of the new method.

2 Brief Summary of the Original Translation

We start from a Haskell-like functional programming language $\mathcal{FP}$. The types allowed in $\mathcal{FP}$ are: variable types, inductive data types, and function types.

Elements of inductive types are generated by constructors which must always be used fully applied.

There are two kinds of functions, that is, of elements in the functional type: those defined by structural recursion and those defined by general recursion. Since the two kinds need to be translated differently, we distinguish them in the functional programming notation. Structurally recursive functions acquire the usual Haskell-like functional types, $\sigma \to \tau$. On the other hand, general recursive functions must always be used fully applied. We reflect this requirement in the syntax by assigning them a *specification* $\sigma_1, \ldots, \sigma_m \Rightarrow \tau$ rather than a proper functional type.

The two kinds of functions give rise to two kinds of applications: those dealing with proper functional types and those dealing with specifications.

The form of the definition of a general recursive function is:

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

where the p_{ij}'s are exclusive linear patterns of the corresponding types and the e_i's are valid terms of type τ (see Definition 1 below). We also allow guarded equations in the definition of a function, where the condition in the equation must be a valid term of type `Bool`. In any case, the equations must satisfy the exclusivity condition: for every particular argument, at most one equation can apply. It is possible that, for some argument, no equation applies, in which case the function is undefined on the given input.

Since the definition of the set of valid terms of a certain type (definition 4 in [BC04]) is important for the understanding of this work, we transcribe it below.

Let us call $\mathcal{F}$ the set of all structurally recursive functions together with their types. Then, the valid terms that we allow in the definition of a recursive function depend on two components: (a) the set $\mathcal{X}$ of variables that can occur

free in the terms, and (b) the set $\mathcal{SF}$ of functions that are being defined (we might define several mutually recursive functions simultaneously), which can be used in the recursive calls.

Definition 1. *Let $\mathcal{X}$ be a set of variables together with their types. Let $\mathcal{SF}$ be a set of function names together with their specifications. Let the set of names of the variables in $\mathcal{X}$, the set of names of the functions in $\mathcal{SF}$, and the set of names of the functions in $\mathcal{F}$ be disjoint. We say that t* is a valid term of type τ *with respect to $\mathcal{X}$ and $\mathcal{SF}$, if the judgement $\mathcal{X}; \mathcal{SF} \vdash t\colon \tau$ can be derived from the rules in Figure 1.* □

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

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

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

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

Fig. 1. Rules for deriving valid terms judgements

Next, we briefly explain the translation of programs into type theory presented in [BC04]. Below, we call $\widehat{\sigma}$ the translation of the type σ.

Variable types and variables, and inductive data types and their constructors are translated straightforwardly. A function type $\sigma \to \tau$ is translated as the total function type $\widehat{\sigma} \to \widehat{\tau}$ in type theory. Structurally recursive functions are directly translated as structurally recursive functions in type theory with the same functional type (except for some possible changes in the notation).

As we have already mentioned in Section 1, a function $\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau$ is translated as a pair

$$\begin{array}{l} \mathsf{fAcc}\colon \widehat{\sigma_1} \to \ldots \to \widehat{\sigma_m} \to \mathsf{Prop} \\ \mathtt{f}\colon (x_1\colon \widehat{\sigma_1}; \ldots; x_m\colon \widehat{\sigma_m}; \mathsf{fAcc}\ x_1 \cdots x_m) \to \widehat{\tau} \end{array}$$

In order to complete the translation of `f` we need to give the types of the constructors of fAcc and the equations defining `f`.

For each equation in `f` (i.e., in the functional side) we define a constructor for fAcc and an equation for `f` (i.e., in the type-theoretic side) as follows. Let

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

be a guarded equation of `f`, let Γ be the context of variables occurring in the patterns $p_1, \ldots, p_m$ and let $\widehat{\Gamma}$ be its type-theoretic translation. Given the term

e, we define a type-theoretic context Φ_e which extends $\widehat{\Gamma}$, and a type-theoretic translation $\widehat{e}$ of e whose free variables are included in $\widehat{\Gamma};\Phi_e$. Similarly, we define Φ_c and $\widehat{c}$.

The type of the constructor of fAcc corresponding to the above equation is

$$\mathsf{facc}\colon (\widehat{\Gamma};\Phi_c;q\colon \widehat{c} = \mathsf{true};\Phi_e)(\mathsf{fAcc}\ \widehat{p_1}\cdots\widehat{p_m})$$

and the corresponding equation of f is

$$\mathsf{f}\ \widehat{p_1}\cdots\widehat{p_m}\ (\mathsf{facc}\ \overline{x}\ \overline{y}\ q\ \overline{z}) = \widehat{e}$$

where $\overline{x}$, $\overline{y}$ and $\overline{z}$ are the variables defined in $\widehat{\Gamma}$, Φ_c and Φ_e, respectively, and $\widehat{p_i}$ is the type-theoretic version of p_i. For non-conditional equations, we simply omit all the parts related with the condition c.

The translation of the application of a function to an argument must consider two cases. If the function has a regular function type $\sigma \to \tau$, that is, it is defined without using general recursion, then the application is translated straightforwardly. When we translate the application of a general recursive function to *all* its arguments, we have to make sure that the arguments are in the domain of the function. Hence, we must add a constraint expressing this fact in the translation context of the application. Concretely, when translating a term of the form $\mathtt{f}(a_1,\ldots,a_m)$, we must add an assumption $(h\colon \mathsf{fAcc}\ \widehat{a_1}\cdots\widehat{a_m})$ to the translation context of the application, where $\widehat{a_i}$ is the type-theoretic translation of a_i. The application itself is translated as $(\mathsf{f}\ \widehat{a_1}\cdots\widehat{a_m}\ h)$. The argument h is needed to make sure that f is only applied to arguments in its domain.

The crucial part of the method in [BC04] is the definition of the context Φ_a and the type-theoretic term $\widehat{a}$ associated with a term a. Both Φ_a and $\widehat{a}$ are defined simultaneously by recursion over the structure of a.

For a formal description of the language $\mathcal{FP}$ and the translation of its programs into type theory, the reader can refer to [BC04].

We finish this section with the definition of the partial function `my_mod` and its translation into type theory following the method we have just described. This function is such that $\mathtt{my\_mod}(m,n) = n\ \mathtt{mod}\ m$ whenever $m \neq 0$, where `mod` is the standard modulo function defined as in the Haskell prelude.

We define the functional version of `my_mod` as:

$$\begin{array}{ll}\mathtt{fix\ my\_mod}\colon \mathtt{N},\mathtt{N} \Rightarrow \mathtt{N} & \\ \quad \mathtt{my\_mod}(m,n) = n & \mathtt{if}\ m \neq 0 \wedge n < m \\ \quad \mathtt{my\_mod}(m,n) = \mathtt{my\_mod}(m,n-m) & \mathtt{if}\ m \neq 0 \wedge n \geqslant m\end{array}$$

where $-$, $<$, $\geqslant$, $\neq$ and $\wedge$ are defined as expected.

This function is translated into type theory as follows:

$$\begin{array}{l}\mathsf{my\_modAcc}\colon \mathsf{N} \to \mathsf{N} \to \mathsf{Prop} \\ \quad \mathsf{my\_mod\_acc}_<\colon (m\colon \mathsf{N};n\colon \mathsf{N};q\colon (m \neq 0 \wedge n < m) = \mathsf{true})(\mathsf{my\_modAcc}\ m\ n) \\ \quad \mathsf{my\_mod\_acc}_\geqslant\colon (m\colon \mathsf{N};n\colon \mathsf{N};q\colon (m \neq 0 \wedge n \geqslant m) = \mathsf{true}; \\ \qquad\qquad h\colon \mathsf{my\_modAcc}\ m\ (n-m))(\mathsf{my\_modAcc}\ m\ n) \\ \\ \mathsf{my\_mod}\colon (m\colon \mathsf{N};n\colon \mathsf{N};\mathsf{my\_modAcc}\ m\ n) \to \mathsf{N} \\ \quad \mathsf{my\_mod}\ m\ n\ (\mathsf{my\_mod\_acc}_<\ m\ n\ q) = n \\ \quad \mathsf{my\_mod}\ m\ n\ (\mathsf{my\_mod\_acc}_\geqslant\ m\ n\ q\ h) = \mathsf{my\_mod}\ m\ (n-m)\ h\end{array}$$

Observe that we will never be able to apply the function my_mod to the arguments 0 and $i\colon \mathsf{N}$ since we cannot construct a proof of (my_modAcc 0 i).

3 New Translation of Functional Programs

We now introduce the type of *partial functions* in our impredicative type theory. If $\alpha\colon \mathsf{Set}$ and $\beta\colon \mathsf{Set}$, then we define the type of partial functions as

$$\alpha \rightharpoonup \beta \equiv \Sigma D\colon \alpha \to \mathsf{Prop}.(x\colon \alpha; D\ x) \to \beta\colon \mathsf{Set}.$$

Thus, a partial function $f\colon \alpha \rightharpoonup \beta$ is actually a pair consisting of the domain of the function and the function itself, which depends on a proof that the input value is in the domain of the function. The definition can be extended to consider partial functions of several arguments. If $\alpha_1, \ldots, \alpha_m\colon \mathsf{Set}$ and $\beta\colon \mathsf{Set}$, we define

$$\begin{aligned}\alpha_1, \ldots, \alpha_m \rightharpoonup \beta \equiv\ & \Sigma D\colon \alpha_1 \to \cdots \to \alpha_m \to \mathsf{Prop}.\\ & (x_1\colon \alpha_1; \ldots; x_m\colon \alpha_m; D\ x_1\ \cdots\ x_m) \to \beta.\end{aligned}$$

If $f\colon \alpha_1, \ldots, \alpha_m \rightharpoonup \beta$, we write dom_f for $(\pi_1\ f)$ and, if $a_i\colon \alpha_i$ for $1 \leqslant i \leqslant m$ and $(h\colon \mathsf{dom}_f\ a_1\ \cdots\ a_m)$, we use the notation $f_{[h]}(a_1, \ldots, a_m)$ for $(\pi_2\ f\ a_1\ \cdots\ a_m\ h)$.

In the case of functions of many arguments, we may partially apply the function to only $k < m$ arguments. Then we write:

$$\begin{aligned}& f(a_1, \ldots, a_k) = \langle D', f'\rangle\colon \alpha_{k+1}, \ldots, \alpha_m \rightharpoonup \beta\\ & \text{where } D'\ x_{k+1}\ \cdots\ x_m = \mathsf{dom}_f\ a_1\ \cdots\ a_k\ x_{k+1}\ \cdots\ x_m\\ & \qquad f'\ x_{k+1}\ \cdots\ x_m\ h = f_{[h]}(a_1, \ldots, a_k, x_{k+1}, \ldots, x_m)\end{aligned} \qquad (*)$$

This definition amounts to an outline of a proof of the left-to-right direction of the following equivalence:

$$\alpha_1, \ldots, \alpha_m \rightharpoonup \beta \cong \alpha_1 \to \cdots \to \alpha_k \to (\alpha_{k+1}, \ldots, \alpha_m \rightharpoonup \beta).$$

The right-to-left direction is straightforward.

In what follows, we give a modification of the method of [BC04] that uses the type of partial functions in place of the standard type of total functions.

First, we apply the translation method also to structurally recursive functions since we want all the functions to have a partial function type. Hence, structurally and general recursive functions are now all assigned specifications, so now the set $\mathcal{SF}$ contains also the functions previously in $\mathcal{F}$. As a consequence, the second rule in Definition 1 of valid terms simply disappears.

In addition, the definition of valid terms required every occurrence of a general recursive function to be fully applied. Now we lift this restriction and we replace the third rule in Definition 1 by the following rule:

$$\frac{\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau \in \mathcal{SF} \quad \mathcal{X}; \mathcal{SF} \vdash a_i\colon \sigma_i \text{ for } 1 \leqslant i \leqslant k \text{ and } k \leqslant m}{\mathcal{X}; \mathcal{SF} \vdash f(a_1, \ldots, a_k)\colon \sigma_{k+1}, \ldots, \sigma_m \Rightarrow \tau}$$

where if $k = m$, $\sigma_{k+1}, \ldots, \sigma_m \Rightarrow \tau$ is understood simply as τ.

A similar modification has to be made to the fourth rule of the same definition, which deals with constructors, since they must also be used fully applied in [BC04].

Next, we modify the definition of the translation into type theory.

Variable types and inductive datatypes are translated as before. The function type $\sigma \to \tau$ is now translated into the partial function type $\widehat{\sigma} \rightharpoonup \widehat{\tau}$. In our previous work, specifications were not translated into a type. Now, given the specification $\sigma_1, \ldots, \sigma_m \Rightarrow \tau$ we define its translation as the type of partial functions $\widehat{\sigma_1}, \ldots, \widehat{\sigma_m} \rightharpoonup \widehat{\tau}$.

Constructors are assigned specifications in [BC04] rather than types. With the translation we give of specifications, a constructor is now expected to have a domain predicate. However, the application of a constructor to arguments of the corresponding types should always be defined. Let $\mathsf{c}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \mathtt{T}$ be one of the constructors of the inductive type $\mathtt{T}$. In type theory, the corresponding constructor would have type $\mathsf{c}\colon \widehat{\sigma_1} \to \cdots \to \widehat{\sigma_m} \to \mathsf{T}$. The translation of c is then defined as $\mathsf{C} = \langle \mathsf{cAcc}, \mathsf{c}' \rangle \colon \widehat{\sigma_1}, \ldots, \widehat{\sigma_m} \rightharpoonup \mathsf{T}$ with

$$\mathsf{cAcc} = [x_1 \colon \widehat{\sigma_1}; \ldots; x_m \colon \widehat{\sigma_m}]\mathbb{T} \colon \widehat{\sigma_1} \to \cdots \to \widehat{\sigma_m} \to \mathsf{Prop}$$
$$\mathsf{c}'_{[h]}(a_1, \ldots, a_m) = \mathsf{c}(a_1, \ldots, a_m)$$

where $\mathbb{T}$ is a set containing only the element tt.

Below we present the definition of the context Φ_a and the type-theoretic expression $\widehat{a}$ associated with a term a. The cases of function calls and constructors are now split into two cases, according to whether the function or the constructor is fully or partially applied; this distinction was not relevant in [BC04].

Definition 2. *Given a term a and a context Γ containing type assumptions for the free variables in a, we define the context extension Φ_a and the type-theoretic term $\widehat{a}$ by recursion on the structure of a. Note that, since Φ_a extends Γ, we should only introduce fresh variables in Φ_a.*

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

$a \equiv \mathtt{f}(a_1, \ldots, a_m)$**:** *Here, $\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau$, and $a_1, \ldots, a_m$ are arguments of the appropriate types. Hence, the function is fully applied. First, we determine $\Phi_{a_1}, \ldots, \Phi_{a_m}$ and $\widehat{a_1} \ldots, \widehat{a_m}$ by structural recursion. We then define*

$$\Phi_a \equiv \Phi_{a_1}; \ldots; \Phi_{a_m}; (h \colon \mathsf{fAcc}\ \widehat{a_1}\ \cdots\ \widehat{a_m}) \quad \text{and} \quad \widehat{a} \equiv \mathsf{f}_{[h]}(\widehat{a_1}, \ldots, \widehat{a_m})$$

$a \equiv \mathtt{f}(a_1, \ldots, a_k)$**:** *Let $\mathtt{f}$ be as above and $k < m$. In this case the function is not fully applied. Under similar assumptions as in the previous case, we define $\Phi_a \equiv \Phi_{a_1}; \ldots; \Phi_{a_k}$ and $\widehat{a} \equiv \mathsf{f}(\widehat{a_1}, \ldots, \widehat{a_k})$. See (*) for the meaning of the partial application of f to only k of its m arguments.*

$a \equiv \mathsf{c}(a_1, \ldots, a_m)$**:** *Let c be a constructor fully applied to arguments of the appropriate types. Under similar assumptions as in the case of fully applied functions, we define $\Phi_a \equiv \Phi_{a_1}; \ldots; \Phi_{a_m}$ and $\widehat{a} \equiv \mathsf{c}(\widehat{a_1}, \ldots, \widehat{a_m})$. Notice that this is equal to $\mathsf{c}'_{[\mathsf{tt}]}(\widehat{a_1}, \ldots, \widehat{a_m})$, so the translation is consistent with that of recursive functions.*

$a \equiv \mathsf{c}(a_1, \ldots, a_k)$**:** *Let* c *be a constructor applied to only* k *of its* m *arguments. This case is similar to the case of partially applied functions.*

$a \equiv [z]b$**:** *Let* σ *be the type of* z*. We start by calculating* Φ_b *and* $\widehat{b}$ *recursively. Notice that now, the context with the assumptions for the free variables in* b *is* $(\Gamma; z\colon \widehat{\sigma})$*. We define* $\Phi_{[z]b} \equiv ()$ *and* $\widehat{[z]b} \equiv \langle \mathsf{gAcc}, \mathsf{g} \rangle$ *with*

$$\mathsf{gAcc} = [z\colon \widehat{\sigma}]\Sigma\Phi_b$$
$$\mathsf{g} = [z\colon \widehat{\sigma}; h\colon \mathsf{gAcc}\ z]\mathsf{Cases}\ h\ \mathsf{of}\ \left\{ \langle \overline{y_{\Phi_b}} \rangle \mapsto \widehat{b} \right.$$

where $\Sigma\Phi_b$ *is a big* Σ*-type defining the conjunction of all the preconditions contained in* Φ_b *and* $\overline{y_{\Phi_b}}$ *is the sequence of variables in* Φ_b*. If* Φ_b *is empty then* $\Sigma\Phi_b$ *is simply understood as* $\mathbb{T}$ *and the whole case-expression can just be replaced by* $\widehat{b}$*. On the other hand, if* Φ_b *contains only one assumption then* $\Sigma\Phi_b$ *is simply* Φ_b*. Moreover, the case-expression can just be replaced by* $\widehat{b}[y_b := h]$*, where* y_b *is the variable assumed in* Φ_b*.*

$a \equiv (g\ b)$**:** *Here* g *stands for any function with a functional type (not a specification). As usual, we define* Φ_a *and* $\widehat{a}$ *in terms of* $\Phi_g, \widehat{g}, \Phi_b$ *and* $\widehat{b}$*. Since* g *is (potentially) a partial function, it has a partial function type in type theory, so we have to make sure that it is only applied to elements in its domain. Hence, we have* $\Phi_a = \Phi_g; \Phi_b; (h\colon \mathsf{dom}_{\widehat{g}}\ \widehat{b})$ *and* $\widehat{a} = \widehat{g}_{[h]}(\widehat{b})$*.* □

Theorem 1 of [BC04] can be strengthened: now, it states that the functional program `f` and its type-theoretic translation `f` denote the same partial recursive function. The proof is similar to those of Theorems 1 and 2 in [BC04] and it relies on a correspondence between computation of terms in functional programming and reduction of their translations in type theory. Given the application of a general recursive function, this correspondence depends, in turn, on the correspondence between the trace of the computation of the application in functional programming and a normal form of the proof that the type-theoretic versions of the arguments satisfy the corresponding domain predicate.

Lemma 1. *Let* $e\colon \tau$ *be a valid term in* $\mathcal{FP}$ *with respect to* Γ *and* $\mathcal{SF}$*, and let* $\widehat{\Gamma}$ *be the type-theoretic translation of* Γ*. Let* Φ_e *be the translation context generated by the method,* $\overline{z}$ *the sequence of variables in* Φ_e*, and* $\overline{d}$ *an instantiation of* Φ_e *depending only on variables in* $\widehat{\Gamma}$*. Then the computation of* e *in* $\mathcal{FP}$ *terminates with a term* r *whose type-theoretic translation* $\widehat{r}$ *is convertible with* $\widehat{e}[\overline{z := d}]$*.*

Proof. We do induction on the pair (l, e) where l is the maximum length of a normalisation path of $\widehat{e}[\overline{z := d}]$ (notice that the path is finite because type theory enjoys strong normalisation) and the order $<$ is the lexicographic order on pairs, that is: $(l', e') < (l, e)$ iff either $l' < l$ or $l' \leqslant l$ and e' is structurally smaller than e. This part of the proof does not present any substantial change with respect to our previous work. □

Lemma 2. *Let* e *and* Φ_e *be as in the previous lemma. If the computation of* e *terminates in* $\mathcal{FP}$ *then there is an instantiation* $\overline{d}$ *of* Φ_e*.*

Proof. We do induction on the pair (l, e) where l is the length of the trace of the computation of e and the order is the lexicographic order on pairs as in lemma 1.

The fact that we assign domain predicates also to local functions generated by λ-abstraction entails that, when computing a local function on a specific argument, we can generate a proof of the local termination predicate whenever the application terminates in the functional side. This is an improvement on [BC04], where we needed to generate a proof of totality for the local function. □

Theorem 1. *Let* $\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau$ *be a function in* $\mathcal{FP}$. *Let* fAcc *and* f *be the domain predicate for* $\mathtt{f}$ *and the type-theoretic version of* $\mathtt{f}$, *respectively. Then, for every sequence of values* $v_1\colon \sigma_1, \ldots, v_m\colon \sigma_m$ *we have that*

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

and if $(h\colon \mathsf{fAcc}\ \widehat{v_1}\ \cdots\ \widehat{v_m})$ *is a closed proof, then*

$$\mathsf{f}_{[h]}(\widehat{v_1}, \ldots, \widehat{v_m}) = \widehat{\mathtt{f}(v_1, \ldots, v_m)}.$$

Proof. Immediate by applying the previous lemmas to $e \equiv \mathtt{f}(v_1, \ldots, v_m)$. □

4 Illustration of the Method

We illustrate the advantages of the new method on some examples containing the features that were problematic in our previous work. The function `map` shows how to deal with functional arguments, the function `sumdel` illustrates how λ-abstractions are treated, and the function `is_div2` shows how to deal with partial applications.

In addition, we demonstrate that the impredicativity of the sort Prop is necessary for the generality of the method, by giving an example `itz` that gives rise to a polymorphic domain predicate. The example `itz` is also the only one with nested recursion, so it is the only one that needs induction-recursion.

Functional Arguments: `map`. Functional version:

$$\begin{aligned}&\mathtt{fix\ map}\colon \gamma \to \delta, \mathtt{List}\ \gamma \Rightarrow \mathtt{List}\ \delta\\ &\qquad \mathtt{map}(f, \mathtt{nil}) = \mathtt{nil}\\ &\qquad \mathtt{map}(f, \mathtt{cons}(x, xs)) = \mathtt{cons}(f\ x, \mathtt{map}(f, xs))\end{aligned}$$

Type-theoretic version: $\mathsf{Map} = \langle \mathsf{mapAcc}, \mathsf{map}\rangle\colon (\gamma \rightharpoonup \delta), \mathsf{List}\ \gamma \rightharpoonup \mathsf{List}\ \delta$ with:

$$\begin{aligned}&\mathsf{mapAcc}\colon (\gamma \rightharpoonup \delta) \to \mathsf{List}\ \gamma \to \mathsf{Prop}\\ &\quad \mathsf{mapacc}_{\mathsf{nil}}\colon (f\colon \gamma \rightharpoonup \delta)(\mathsf{mapAcc}\ f\ \mathsf{nil})\\ &\quad \mathsf{mapacc}_{\mathsf{cons}}\colon (f\colon \gamma \rightharpoonup \delta; x\colon \gamma; xs\colon \mathsf{List}\ \gamma; h\colon \mathsf{dom}_f\ x; h_1\colon \mathsf{dom}_{\mathsf{Map}}\ f\ xs)\\ &\qquad\qquad (\mathsf{mapAcc}\ f\ \mathsf{cons}(x, xs))\end{aligned}$$

$$\begin{aligned}&\mathsf{map}\colon (f\colon \gamma \rightharpoonup \delta; ys\colon \mathsf{List}\ \gamma; \mathsf{mapAcc}\ f\ ys) \to \mathsf{List}\ \delta\\ &\quad \mathsf{map}\ f\ \mathsf{nil}\ (\mathsf{mapacc}_{\mathsf{nil}}\ f) = \mathsf{nil}\\ &\quad \mathsf{map}\ f\ \mathsf{cons}(x, xs)\ (\mathsf{mapacc}_{\mathsf{cons}}\ f\ x\ xs\ h\ h_1) = \mathsf{cons}(f_{[h]}(x), \mathsf{Map}_{[h_1]}(f, xs))\end{aligned}$$

Recall that $(\mathsf{dom}_{\mathsf{Map}}\ f\ xs)$ and $\mathsf{Map}_{[h_1]}(f, xs)$ reduce to $(\mathsf{mapAcc}\ f\ xs)$ and $(\mathsf{map}\ f\ xs\ h_1)$, respectively. In addition, when map is applied to a concrete partial function $\langle \mathsf{fAcc}, \mathsf{f}\rangle$, $(\mathsf{dom}_f\ x)$ and $f_{[h]}(x)$ reduce to $(\mathsf{fAcc}\ x)$ and $(\mathsf{f}\ x\ h)$, respectively. When checking the validity of the inductive-recursive definitions, we have to expand such terms.

Abstractions: `sumdel`. Here, the functions +, `sum` and `delete` are all structurally recursive. The function + is defined as expected, and `sum` and `delete` are as in the Haskell prelude. Below, we assume that the mentioned functions have already been translated into type theory. Moreover, for simplicity reasons, we use the structurally recursive versions of these functions rather than the formal translations we would obtain with our new method.

Functional version:

$$\begin{array}{l}\texttt{fix sumdel}\colon \texttt{List N} \Rightarrow \texttt{N}\\ \quad \texttt{sumdel}(\texttt{nil}) = 0\\ \quad \texttt{sumdel}(\texttt{cons}(n, l)) = n + \texttt{sum}(\texttt{map}([x]\texttt{sumdel}(\texttt{cons}(n, \texttt{delete}(x, l))), l))\end{array}$$

The actual value computed by the function is $\texttt{sumdel}(n_1, \ldots, n_k) = n_1\texttt{sd}(k)$ where $\texttt{sd}(0) = 0$ and $\texttt{sd}(k+1) = k\texttt{sd}(k) + 1$.

Type-theoretic version: $\mathsf{Sumdel} = \langle \mathsf{sumdelAcc}, \mathsf{sumdel}\rangle\colon \mathsf{List\ N} \rightharpoonup \mathsf{N}$ with:

$$\begin{array}{l}\mathsf{sumdelAcc}\colon \mathsf{List\ N} \to \mathsf{Prop}\\ \quad \mathsf{sumdelacc_{nil}}\colon (\mathsf{sumdelAcc\ nil})\\ \quad \mathsf{sumdelacc_{cons}}\colon (n\colon \mathsf{N}; l\colon \mathsf{List\ N}; h\colon \mathsf{mapAcc\ G}\ l)(\mathsf{sumdelAcc\ cons}(n, l))\\ \\ \mathsf{sumdel}\colon (l\colon \mathsf{List\ N}; \mathsf{sumdelAcc}\ l) \to \mathsf{N}\\ \quad \mathsf{sumdel\ nil\ sumdelacc_{nil}} = 0\\ \quad \mathsf{sumdel\ cons}(n, l)\ (\mathsf{sumdelacc_{cons}}\ n\ l\ h) = n + \mathsf{sum}\ \ \mathsf{Map}_{[h]}(\mathsf{G}, l)\end{array}$$

with $\mathsf{G} \equiv \langle \mathsf{gAcc}, \mathsf{g}\rangle\colon \mathtt{N} \rightharpoonup \mathtt{N}$ where

$$\begin{array}{l}\mathsf{gAcc} = [x\colon \mathtt{N}](\mathsf{sumdelAcc\ cons}(n, (\mathsf{delete}\ x\ l)))\colon \mathsf{N} \to \mathsf{Prop}\\ \mathsf{g} = [x\colon \mathtt{N}; h\colon \mathsf{gAcc}\ x]\mathsf{Sumdel}_{[h]}(\mathsf{cons}(n, (\mathsf{delete}\ x\ l)))\colon (x\colon \mathsf{N}; \mathsf{gAcc}\ x) \to \mathsf{N}\end{array}$$

Notice that G is local to Sumdel and hence, n and l are known while defining G.

The important feature of this translation, not possible in the old one, is that we can assign a precise domain predicate gAcc to the local function generated by the λ-abstraction. Notice that in the body of the main function, the local function is applied only to arguments that satisfy gAcc, so termination is ensured.

Partial Application: `is_div2`. Functional version:

$$\begin{array}{l}\texttt{fix is\_div2}\colon \texttt{List N} \Rightarrow \texttt{List N}\\ \qquad \texttt{is\_div2}(xs) = \texttt{map}(\texttt{my\_mod}(2), xs)\end{array}$$

where the function `my_mod` is the one defined at the end of Section 2. Given a list of numbers, this function returns a list of 0's and 1's depending on whether the numbers in the list are divisible by 2 or not, respectively.

Observe that the type-theoretic version of `my_mod` is the same as before since this function does not present any of the problematic aspects on which the method has been changed. Let $\mathsf{My\_Mod} = \langle \mathsf{my\_modAcc}, \mathsf{my\_mod}\rangle : \mathsf{N}, \mathsf{N} \rightharpoonup \mathsf{N}$ with $\mathsf{my\_modAcc}$ and $\mathsf{my\_mod}$ as defined on page 122.

Type-theoretic version: $\mathsf{Is\_Div2} = \langle \mathsf{is\_div2Acc}, \mathsf{is\_div2}\rangle : \mathsf{List\ N} \rightharpoonup \mathsf{List\ N}$ with:

$$\begin{array}{l}\mathsf{is\_div2Acc} : \mathsf{List\ N} \to \mathsf{Prop}\\ \quad \mathsf{is\_div2acc} : (xs : \mathsf{List\ N}; h : \mathsf{mapAcc\ My\_Mod}(2)\ xs)(\mathsf{is\_div2Acc}\ xs)\end{array}$$

$$\begin{array}{l}\mathsf{is\_div2} : (xs : \mathsf{List\ N}; \mathsf{is\_div2Acc}\ xs) \to \mathsf{List\ N}\\ \quad \mathsf{is\_div2}\ xs\ (\mathsf{is\_div2acc}\ xs\ h) = \mathsf{Map}_{[h]}(\mathsf{My\_Mod}(2), xs)\end{array}$$

Notice that the translation of the partial application $\mathtt{my\_mod}(2)$ does not introduce any domain constrains in the type of the constructor $\mathsf{is\_div2acc}$. Given an element x in the list xs, the application of the function $\mathsf{My\_Mod}(2)$ to the argument x will only be possible if we can find a proof of $(\mathsf{my\_modAcc}\ 2\ x)$. This is taken care of in the definition of mapAcc.

Necessity of Impredicativity: `itz`. The following example shows that it is necessary to have an impredicative type theory if we want our method to apply to every functional program. Functional version:

$$\begin{array}{l}\mathtt{fix\ itz} : \mathtt{N} \to \mathtt{N}, \mathtt{N} \Rightarrow \mathtt{N}\\ \qquad \mathtt{itz}(f, 0) = f\ 0\\ \qquad \mathtt{itz}(f, \mathtt{succ}(n)) = f\ \ \mathtt{itz}(\mathtt{itz}(f), n)\end{array}$$

Type-theoretic version: $\mathsf{Itz} = \langle \mathsf{itzAcc}, \mathsf{itz}\rangle : (\mathsf{N} \rightharpoonup \mathsf{N}), \mathsf{N} \rightharpoonup \mathsf{N}$ with:

$$\begin{array}{l}\mathsf{itzAcc} : (\mathsf{N} \rightharpoonup \mathsf{N}) \to \mathsf{N} \to \mathsf{Prop}\\ \quad \mathsf{itzacc}_0 : (f : \mathsf{N} \rightharpoonup \mathsf{N}; h : \mathsf{dom}_f\ 0)(\mathsf{itzAcc}\ f\ 0)\\ \quad \mathsf{itzacc}_{\mathsf{succ}} : (f : \mathsf{N} \rightharpoonup \mathsf{N}; n : \mathsf{N}; h_1 : \mathsf{itzAcc\ Itz}(f)\ n;\\ \qquad\qquad h_2 : \mathsf{dom}_f\ \mathsf{Itz}_{[h_1]}(\mathsf{Itz}(f), n))(\mathsf{itzAcc}\ f\ \mathsf{succ}(n))\end{array}$$

$$\begin{array}{l}\mathsf{itz} : (f : \mathsf{N} \rightharpoonup \mathsf{N}; n : \mathsf{N}; \mathsf{itzAcc}\ f\ n) \to \mathsf{N}\\ \quad \mathsf{itz}\ f\ 0\ (\mathsf{itzacc}_0\ f\ h) = f_{[h]}(0)\\ \quad \mathsf{itz}\ f\ \mathsf{succ}(n)\ (\mathsf{itzacc}_{\mathsf{succ}}\ f\ n\ h_1\ h_2) = f_{[h_2]}(\mathsf{Itz}_{[h1]}(\mathsf{Itz}(f), n))\end{array}$$

We see that impredicativity is essential when we follow our method to formalise this example. When defining itzAcc, the constructor $\mathsf{itzacc}_{\mathsf{succ}}$ quantifies over all partial functions $f : \mathsf{N} \rightharpoonup \mathsf{N}$ (and, therefore, over the domain predicates of all those functions) and in the body of the constructor $\mathsf{itzacc}_{\mathsf{succ}}$ the function $\mathsf{Itz}(f)$ is itself an argument of itzAcc.

The alternative to use a predicative hierarchy of type universes $\mathsf{U}_0, \mathsf{U}_1, \mathsf{U}_2, \ldots$ does not work in this example. Function spaces would need to be stratified too, according to the universe in which the domain predicate lives, so we would have

$$A \rightharpoonup_i B = \Sigma P : A \to \mathsf{U}_i.(x : A; P\ x) \to B$$

Since we are quantifying over $A \to \mathsf{U}_i$, predicatively it must be $A \rightharpoonup_i B : \mathsf{U}_j$ with $j > i$. That is, we have at least $A \rightharpoonup_i B : \mathsf{U}_{i+1}$.

In the case of Itz, if we try to assign universe levels, that is, if we try to give it the type $\mathsf{Itz}\colon (\mathsf{N} \rightharpoonup_i \mathsf{N}), \mathsf{N} \rightharpoonup_j \mathsf{N}$ for some i and j, we reach a contradiction regardless of what i and j are. To start with, we have $\mathsf{itzAcc}\colon (\mathsf{N} \rightharpoonup_i \mathsf{N}) \to \mathsf{N} \to \mathsf{U}_j$. The first constructor itzacc_0 contains a quantification on $\mathsf{N} \rightharpoonup_i \mathsf{N}$, so its type must be at least in U_{i+1}. Thus, also the universe of itzAcc must be at least U_{i+1}, that is, $j \geqslant i+1$. Now, in the constructor $\mathsf{itzacc}_{\mathsf{succ}}$ we have the subterm $\mathsf{Itz}(\mathsf{Itz}(f), n)$, but this term does not type-check because $\mathsf{Itz}\colon (\mathsf{N} \rightharpoonup_i \mathsf{N}), \mathsf{N} \rightharpoonup_j \mathsf{N}$ and $\mathsf{Itz}(f)\colon \mathsf{N} \rightharpoonup_j \mathsf{N}$ with $j \geqslant i+1$. Hence, Itz cannot be applied to $\mathsf{Itz}(f)$ because the type is not correct: it expects an argument of type $\mathsf{N} \rightharpoonup_i \mathsf{N}$ and it gets one of type $\mathsf{N} \rightharpoonup_j \mathsf{N}$ with $j \geqslant i+1$.

5 Conclusions

This article presents a method to translate functional programs into type theory based on the one previously presented in [BC04]. The new approach relies on a type of partial functions whose elements are pairs consisting of a domain predicate and a function depending on a proof of the predicate. The problems that were left open in [BC04] are now solved: functional arguments are dealt with by lifting their domain conditions to the main call; λ-abstractions denote partial functions with domain condition generated locally; partial application is interpreted by just fixing the given parameters both in the domain predicate and the function.

These results are obtained at the cost of two disadvantages. First of all, we need an impredicative type theory for the method to be applicable to all functional programs (see example `itz`). This is indispensable to obtain a general result, but in most practical cases the method could be adapted to work on a predicative type theory with type universes.

As we have already mentioned, this paper is the last in a series of articles [Bov01, BC01, Bov02a, Bov02b, Bov03, BC04] aimed at representing general recursive functions in type theory. See the section on related work in [BC04] for a thorough discussion of the literature regarding representations of recursive functions in logical frameworks.

Acknowledgements. We would like to thank Thierry Coquand for his constructive criticism on earlier versions of this article and for useful comments on the type of partial functions and on the use of impredicativity. We are also grateful to Christine Paulin-Mohring for giving us a clarifying explanation of her method to extract computational content from the accessibility predicates.

References

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

[BC04] A. Bove and V. Capretta. Modelling general recursion in type theory. To appear in *Mathematical Structures in Computer Science*. Available on the WWW: `http://www.cs.chalmers.se/~bove/Papers/general_presentation.ps.gz`, May 2004.

[Bla03] Frédéric Blanqui. Inductive types in the calculus of algebraic constructions. In M. Hofmann, editor, *Typed Lambda Calculi and Applications: 6th International Conference, TLCA 2003*, volume 2701 of *LNCS*, pages 46–59. Springer-Verlag, 2003.

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

[Bov02a] A. Bove. Mutual general recursion in type theory. Technical Report, Chalmers University of Technology. Available on the WWW: `http://www.cs.chalmers.se/~bove/Papers/mutual_rec.ps.gz`, May 2002.

[Bov02b] Ana Bove. *General Recursion in Type Theory*. PhD thesis, Chalmers University of Technology, Department of Computing Science, November 2002. Available on the WWW: `http://cs.chalmers.se/~bove/Papers/phd_thesis.ps.gz`.

[Bov03] A. Bove. General recursion in type theory. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, International Workshop TYPES 2002, The Netherlands*, number 2646 in Lecture Notes in Computer Science, pages 39–58, March 2003.

[Cap04] Venanzio Capretta. A polymorphic representation of induction-recursion. Draft paper. Available from `http://www.science.uottawa.ca/~vcapr396/`, 2004.

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

[CM85] R. L. Constable and N. P. Mendler. Recursive definitions in type theory. In Rohit Parikh, editor, *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 1985.

[Coq02] Coq Development Team. LogiCal Project. *The Coq Proof Assistant. Reference Manual. Version 7.4*. INRIA, 2002.

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

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

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

[Jon03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries. The Revised Report*. Cambridge University Press, April 2003.

[Pau] Christine Paulin. How widely applicable is Coq? Contribution to the Coq mailing list, 19 Aug 2002, `http://pauillac.inria.fr/bin/wilma_hiliter/coq-club/200208/msg00003.html`.