Computation by Prophecy

Ana Bove[1] and Venanzio Capretta[2]

[1] Department of Computer Science and Engineering
Chalmers University of Technology,
412 96 Göteborg, Sweden
Tel.: +46-31-7721020, Fax: +46-31-7723663
bove@chalmers.se
[2] Computer Science Institute (iCIS),
Radboud University Nijmegen,
Tel.: +31-24-3652631, Fax: +31-24-3652525
venanzio@cs.ru.nl

Abstract. We describe a new method to represent (partial) recursive functions in type theory. For every recursive definition, we define a co-inductive type of *prophecies* that characterises the traces of the computation of the function. The structure of a prophecy is a possibly infinite tree, which is coerced by linearisation to a type of partial results defined by applying the delay monad to the co-domain of the function. Using induction on a weight relation defined on the prophecies, we can reason about them and prove that the formal type-theoretic version of the recursive function, resulting from the present method, satisfies the recursive equations of the original function. The advantages of this technique over the method previously developed by the authors via a special-purpose accessibility (domain) predicate are: there is no need of extra logical arguments in the definition of the recursive function; the function can be applied to any element in its domain, regardless of termination properties; we obtain a type of partial recursive functions between any two given types; and composition of recursive functions can be easily defined.

1 Introduction

The implementation of general recursive functions in type theory has received wide attention in the last decade, and several methods to implement recursive algorithms and reason about them have been described in the literature.

We give a survey of different approaches in the *related work* section of a previous article [6]. After the publication of that paper, the type-theory based proof assistant *Coq* [14,3] has been extended with a new feature to define total recursive functions that expands the native structural recursion. The feature `Function` (based on the work by Balaa and Bertot [1]) facilitates the definition of total functions that have a well-founded relation associated to them. In addition, the tactic `functional induction` (based on the work by Barthe and Courtieu [2]) has been added to the system, providing induction principles that follow the definition of structurally recursive functions. These contributions enlarge

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 70–83, 2007.

the class of total recursive functions that can be studied in *Coq*. There are, however, functions that lay outside the reach of these new features, for example, all strictly partial recursive functions. Our main goal in this work is to provide a good general type-theoretic treatment of recursive computations (partial or not).

We have previously worked on two methods to tackle this issue. The first method [6,4,7] consists in characterising the inputs on which the function terminates by an inductive (domain) predicate easily defined from the recursive equations, and then defining the function itself by recursion on the proof that the input argument satisfies this domain predicate. The second method [8] consists in defining co-inductive types of partial elements and implementing general recursive functions by co-recursion on these types.

Both methods have pros and cons.

The first method has two main advantages. First, the type-theoretic equations defining the function are almost identical to the recursive equations in a functional programming language, except that the former require proof terms for the domain predicate as additional arguments. Second, it is easy to reason about a function formalised with this method by induction on its domain predicate. A disadvantage is that the application of a function to a certain input always requires a proof that the input satisfies the domain predicate defined for the function; as a consequence, the function can only be applied to arguments for which we know how to construct such a proof.

On the other hand, a function formalised with the second method requires no additional logical arguments in its definition and, hence, the function can be applied even to arguments for which it might not terminate. The main disadvantage is that reasoning about functions formalised in this way is much more involved than with the first method.

Here, we show a new way of representing general recursive functions in intensional type theory, where we adapt some of the ideas of the first method to facilitate the definition of functions with the second one and the subsequent reasoning about their formalisation. In other words, we propose a co-inductive version of the Bove/Capretta method.

Throughout this paper we will use a generalisation of the Fibonacci function as a running example to illustrate the notions we are presenting. This algorithm is defined as follows:

$$\begin{aligned}&\mathsf{F}\colon \mathbb{N} \to \mathbb{N}\\ &\quad \mathsf{F}\ 0 = \mathsf{a}\\ &\quad \mathsf{F}\ 1 = \mathsf{b} + \mathsf{c} * \mathsf{F}\ (g\ 0)\\ &\quad \mathsf{F}\ (\mathsf{S}\ (\mathsf{S}\ n)) = \mathsf{F}\ (g\ n) + \mathsf{F}\ (g\ (\mathsf{S}\ n)).\end{aligned}$$

where a, b and c are natural numbers and $g\colon \mathbb{N} \to \mathbb{N}$. We can get the standard Fibonacci sequence by letting $\mathsf{a} = 1, \mathsf{b} = 1, \mathsf{c} = 0$, and letting g be the identity function.

Observe that the totality of F depends on the definition of g: with the same choices of $\mathsf{a}, \mathsf{b}, \mathsf{c}$, but choosing for g the successor function, we obtain a function that is defined in 0 but diverges for any other value.

This paper is organised as follows. In Section 2, we recall how to define a recursive function with the inductive Bove/Capretta method. Sections 3 and 4 present, respectively, the *prophecies*, that is, the co-inductive version of the Bove/Capretta method, and their evaluation procedure. In Section 5, we show the validity of this new method. Finally, Section 6 presents some conclusions.

We have formalised the running example in the proof assistant *Coq*. As an additional example, we have also formalised the *quicksort* algorithm using the method we present here. The files of both formalisations are available on the web at the address: `http://www.cs.ru.nl/~venanzio/Coq/prophecy.html`.

2 Overview of the Bove/Capretta Method

We outline the general steps of the inductive Bove/Capretta method by showing how an algorithm defined by general recursive equations can be formalised in type theory.

Let f be a recursive function defined by a series of (non-overlapping) equations. We assume that the informal definition of the function has the following form:

$$\begin{array}{l} f\colon A \to B \\ \quad \cdots \\ \quad f\ p = e[(f\ p_1), \ldots, (f\ p_n)] \\ \quad \cdots \end{array}$$

where "$f\ p = \cdots$" is one of the recursive equations defining f. The term p is a pattern, possibly containing variables that can occur in the right-hand side of the equation. The right-hand side is an expression e, recursively calling f on the arguments $p_1, \ldots, p_n$.

For an argument a matching the pattern p, there are three phases in the computation of $(f\ a)$: first, from the argument a, the recursive arguments $a_1, \ldots, a_n$ are computed; then, the program f is recursively applied to these arguments; and finally, the results of the recursive calls are fed into the operator e to obtain the final result. This three-steps process is general and can be used to give a very abstract notion of computable function [9].

We recall that the Bove/Capretta method consists in characterising the domain of a function by an inductive predicate with (in principle) one constructor for each of the equations defining the function[1]. The constructor corresponding to each equation takes as parameters assumptions stating that the recursive arguments in the equation satisfy the domain predicate. The general form of the domain predicate for the function f above is as follows:

$$\begin{array}{l} \mathsf{D}_f\colon A \to \mathsf{Prop} \\ \quad \cdots \\ \quad \mathsf{d}_p\colon (\Gamma_p)(\mathsf{D}_f\ p_1) \to \cdots \to (\mathsf{D}_f\ p_n) \to (\mathsf{D}_f\ p) \\ \quad \cdots \end{array}$$

[1] Equations with a case-expression on their right-hand side can give raise to several constructors. See [6] for a more detailed explanation on how to handle these equations.

where Γ_p is a context local to the equation, comprising the variables that occur in p.

For our example we have

$$\begin{array}{l}\mathsf{D_F}\colon \mathbb{N} \to \mathsf{Prop}\\ \quad \mathsf{d_0}\colon (\mathsf{D_F}\ 0)\\ \quad \mathsf{d_1}\colon (\mathsf{D_F}\ (g\ 0)) \to (\mathsf{D_F}\ 1)\\ \quad \mathsf{d_S}\colon (n\colon \mathbb{N})(\mathsf{D_F}\ (g\ n)) \to (\mathsf{D_F}\ (g\ (\mathsf{S}\ n))) \to (\mathsf{D_F}\ (\mathsf{S}\ (\mathsf{S}\ n)))\end{array}$$

The type-theoretic version of f takes as an extra argument a proof that the input satisfies the domain predicate, and it is defined by structural recursion on this extra argument:

$$\begin{array}{l}f\colon (y\colon A)(\mathsf{D}_f\ y) \to B\\ \quad \cdots\\ \quad f\ p\ (\mathsf{d}_p\ \overrightarrow{x}\ h_1\ \cdots\ h_n) = e[(f\ p_1\ h_1), \ldots, (f\ p_n\ h_n)]\\ \quad \cdots\end{array}$$

For the F function we get:

$$\begin{array}{l}\mathsf{F}\colon (m\colon \mathbb{N})(\mathsf{D_F}\ m) \to \mathbb{N}\\ \quad \mathsf{F}\ 0\ \mathsf{d_0} = \mathsf{a}\\ \quad \mathsf{F}\ 1\ (\mathsf{d_1}\ h) = \mathsf{b} + \mathsf{c} * (\mathsf{F}\ (g\ 0)\ h)\\ \quad \mathsf{F}\ (\mathsf{S}\ (\mathsf{S}\ n))\ (\mathsf{d_S}\ n\ h_1\ h_2) = \mathsf{F}\ (g\ n)\ h_1 + \mathsf{F}\ (g\ (\mathsf{S}\ n))\ h_2\end{array}$$

For a complete description of this method and for more examples of its application, the reader is referred to [6,4,7].

3 Views and Prophecies

We can consider an alternative representation of the domain D_f of f where we ignore the elements of A altogether. Below we present the general form of this new domain type which we call A_f, and its corresponding instance for the F function which we call $\mathbb{N}_\mathsf{F}$.

$$\begin{array}{l}A_f\colon \mathsf{Type}\\ \quad \cdots\\ \quad \mathsf{c}_p\colon \Gamma_p \to \underbrace{A_f \to \cdots \to A_f}_{n \text{ times}} \to A_f\\ \quad \cdots\end{array}$$

$$\begin{array}{l}\mathbb{N}_\mathsf{F}\colon \mathsf{Type}\\ \quad \mathsf{c_0}\colon \mathbb{N}_\mathsf{F}\\ \quad \mathsf{c_1}\colon \mathbb{N}_\mathsf{F} \to \mathbb{N}_\mathsf{F}\\ \quad \mathsf{c_S}\colon \mathbb{N} \to \mathbb{N}_\mathsf{F} \to \mathbb{N}_\mathsf{F} \to \mathbb{N}_\mathsf{F}\end{array}$$

Using the terminology of [12], we call A_f a *view* of the domain.

We now formalise f as a function on this new type:

$$\begin{array}{l}f_\star\colon A_f \to B\\ \quad \cdots\\ \quad f_\star\ (\mathsf{c}_p\ \overrightarrow{x}\ t_1\ \cdots\ t_n) = e[(f_\star\ t_1), \ldots, (f_\star\ t_n)]\\ \quad \cdots\end{array}$$

The variables in Γ_p are still required as parameters of the constructor c_p, even if they do not occur in the arguments of the branches, since they may occur in the operator e.

For our example we obtain

$$\begin{array}{l}\mathsf{F}_\star : \mathbb{N}_\mathsf{F} \to \mathbb{N}\\ \quad \mathsf{F}_\star\ \mathsf{c}_0 = \mathsf{a}\\ \quad \mathsf{F}_\star\ (\mathsf{c}_1\ h) = \mathsf{b} + \mathsf{c} * (\mathsf{F}_\star\ h)\\ \quad \mathsf{F}_\star\ (\mathsf{c}_\mathsf{S}\ n\ h_1\ h_2) = \mathsf{F}_\star\ h_1 + \mathsf{F}_\star\ h_2\end{array}$$

We can see A_f as the type of *abstract inputs* for the function f. It is easy to obtain the elements in A_f from the elements of A satisfying D_f:

$$\begin{array}{l}\imath_f : (y\colon A)(\mathsf{D}_f\ y) \to A_f\\ \quad \cdots\\ \quad \imath_f\ p\ (\mathsf{d}_p\ \overrightarrow{x}\ h_1 \cdots h_n) = \mathsf{c}_p\ \overrightarrow{x}\ (\imath_f\ p_1\ h_1) \cdots (\imath_f\ p_n\ h_n)\\ \quad \cdots\end{array}$$

The reverse direction is however not possible, since elements in A_f can be constructed in an arbitrary way, completely disconnected from the behaviour of the function f. The possible projections of those elements in A have absolutely no reason to satisfy D_f. Consider for example the element $(\mathsf{c}_\mathsf{S}\ 10\ \mathsf{c}_0\ \mathsf{c}_0)$ for our example of the function F.

We now try to dualise the inductive Bove/Capretta method by using a co-inductive approach. The idea is that, instead of defining an inductive characterisation of the domain, we define a co-inductive characterisation of the co-domain of the function:

$$\begin{array}{l}\mathsf{CoInductive}\ B^f : \mathsf{Type}\\ \qquad \cdots\\ \qquad \mathsf{b}_e : \Gamma_e \to \underbrace{B^f \to \cdots \to B^f}_{n \text{ times}} \to B^f\\ \qquad \cdots\end{array}$$

where Γ_e is the context comprising the variables occurring free in e. In principle, Γ_e could coincide with Γ_p; however, some of the variables in the pattern p may not be needed for the computation of e and hence, they could be omitted in Γ_e[2].

Observe that the definition of B^f is identical to that of A_f except for the contexts appearing in the equations of both definitions, and for the fact that it is a co-inductive definition instead of an inductive one.

The co-inductive characterisation of the co-domain of F is defined as

$$\begin{array}{l}\mathsf{CoInductive}\ \mathbb{N}^\mathsf{F} : \mathsf{Type}\\ \qquad \mathsf{f}_0 : \mathbb{N}^\mathsf{F}\\ \qquad \mathsf{f}_1 : \mathbb{N}^\mathsf{F} \to \mathbb{N}^\mathsf{F}\\ \qquad \mathsf{f}_\mathsf{S} : \mathbb{N}^\mathsf{F} \to \mathbb{N}^\mathsf{F} \to \mathbb{N}^\mathsf{F}.\end{array}$$

[2] Actually, in some cases, not even all the variables that are free in e need to be included in Γ_e; for example, the Natural argument n is omitted from the constructor f_S in the definition of $\mathbb{N}^\mathsf{F}$.

We now give a version of f that has B^f as co-domain:

$$\begin{array}{l} f^\star \colon A \to B^f \\ \quad \cdots \\ \quad f^\star\ p = \mathsf{b}_e\ \overrightarrow{x^f}\ (f^\star\ p_1) \cdots (f^\star\ p_n) \\ \quad \cdots \end{array}$$

This definition is sound because the co-recursive calls $(f^\star\ p_1), \ldots, (f^\star\ p_n)$ are guarded by the constructor b_e (see [11] for further reading on this issue).

We can see B^f as the type of *abstract outputs* in the same way we saw A_f as the type of abstract inputs. In a certain sense, the elements of B^f are a prediction of the structure of the result. For this reason we will call them *prophecies*.

The version of F using a co-inductive co-domain is

$$\begin{array}{l} \mathsf{F}^\star \colon \mathbb{N} \to \mathbb{N}^{\mathsf{F}} \\ \quad \mathsf{F}^\star\ 0 = \mathsf{f}_0 \\ \quad \mathsf{F}^\star\ 1 = \mathsf{f}_1\ (\mathsf{F}^\star\ (g\ 0)) \\ \quad \mathsf{F}^\star\ (\mathsf{S}\ (\mathsf{S}\ n)) = \mathsf{f}_{\mathsf{S}}\ (\mathsf{F}^\star\ (g\ n))\ (\mathsf{F}^\star\ (g\ (\mathsf{S}\ n))). \end{array}$$

4 Evaluating the Prophecies

The relation between B^f and B is not very clear: since the elements of B^f may represent infinite computations, they do not always correspond to elements of B. Instead, we can find a correspondence between B^f and the type of partial elements of B defined in [8] as:

$$\begin{array}{l} \mathsf{CoInductive}\ B^\nu \colon \mathsf{Type} \\ \qquad\qquad\qquad \mathsf{return} \colon B \to B^\nu \\ \qquad\qquad\qquad \mathsf{step} \colon B^\nu \to B^\nu \end{array}$$

Following [8], we use the notations $\ulcorner b \urcorner$ for $(\mathsf{return}\ b)$ and $\triangleright\, x$ for $(\mathsf{step}\ x)$. The definition above means that an element of B^ν can be either a finite sequence of $\triangleright$ steps followed by the return of a value of B, or an infinite sequence of $\triangleright$ steps. So B^ν represents the partial elements of B.

Our goal is then to represent the program f in type theory as a function with type $A \to B^\nu$. To accomplish this, we need to define an *evaluation* operator $\mathsf{evaluate}_f \colon B^f \to B^\nu$. Notice that B^f has a tree structure, since elements constructed by b_e correspond to nodes of branching degree n, while B^ν has a linear structure given by a sequence of $\triangleright$ steps. The problem consists in linearising a tree or, in computational terms, sequentialising a parallel computation. To achieve this, we create a stack of calls and we execute them sequentially. Every call, when evaluated, can in turn generate new calls that are added to the stack. The stack is represented by a vector. The empty vector is denoted by $\langle\ \rangle$. We use the notation $\langle x_1, \ldots, x_n; \overrightarrow{v} \rangle$ to denote the vector whose first n elements are $x_1, \ldots, x_n$ and whose elements from the $(n+1)$th on are given by the vector $\overrightarrow{v}$. In the case where $\overrightarrow{v}$ is the empty vector, we simply write $\langle x_1, \ldots, x_n \rangle$. In what follows, A^k represents the type of vectors of elements in A with length k.

The evaluation operator also has an extra parameter: a function that will compute the final result from the results of the recursive calls on the elements in the stack. This is similar to continuation-passing programming [13].

The co-recursive evaluation function θ_f, defined below, performs the sequentialisation we just mentioned. The function is defined by cases on the length of the vector and, when the vector is not empty, by cases on its first element.

$$
\begin{array}{l}
\theta_f \colon (k \colon \mathbb{N})(B^f)^k \to (B^k \to B) \to B^\nu \\
\quad \theta_f\ 0\ \langle\ \rangle\ h = \ulcorner h\ \langle\ \rangle \urcorner \\
\quad \cdots \\
\quad \theta_f\ (\mathsf{S}\ k)\ \langle(\mathsf{b}_e\ \overrightarrow{x'}\ y_1 \cdots y_n); \overrightarrow{v}\rangle\ h = \triangleright(\theta_f\ (n+k)\ \langle y_1, \ldots, y_n; \overrightarrow{v}\rangle\ h') \\
\qquad \text{where } h'\ \langle z_1, \ldots, z_n; \overrightarrow{u}\rangle = h\ \langle e[z_1, \ldots, z_n]; \overrightarrow{u}\rangle \\
\quad \cdots
\end{array}
$$

Notice that the recursive call in the function θ_f are guarded by the constructor $\triangleright$, hence this is a valid co-fixpoint definition.

In our running example for the function F we obtain:

$$
\begin{array}{l}
\theta_{\mathsf{F}} \colon (k \colon \mathbb{N})(\mathbb{N}^{\mathsf{F}})^k \to (\mathbb{N}^k \to \mathbb{N}) \to \mathbb{N}^\nu \\
\theta_{\mathsf{F}}\ 0\ \langle\ \rangle\ h = \ulcorner h\ \langle\ \rangle \urcorner \\
\theta_{\mathsf{F}}\ (\mathsf{S}\ k)\ \langle \mathsf{f}_0; \overrightarrow{v}\rangle\ h = \triangleright(\theta_{\mathsf{F}}\ k\ \overrightarrow{v}\ h') \\
\qquad \text{where } h'\ \overrightarrow{u} = h\ \langle \mathsf{a}; \overrightarrow{u}\rangle \\
\theta_{\mathsf{F}}\ (\mathsf{S}\ k)\ \langle(\mathsf{f}_1\ y); \overrightarrow{v}\rangle\ h = \triangleright(\theta_{\mathsf{F}}\ (\mathsf{S}\ k)\ \langle y; \overrightarrow{v}\rangle\ h') \\
\qquad \text{where } h'\ \langle z; \overrightarrow{u}\rangle = h\ \langle(\mathsf{b} + \mathsf{c} * z); \overrightarrow{u}\rangle \\
\theta_{\mathsf{F}}\ (\mathsf{S}\ k)\ \langle(\mathsf{f}_{\mathsf{S}}\ y_1\ y_2); \overrightarrow{v}\rangle\ h = \triangleright(\theta_{\mathsf{F}}\ (2+k)\ \langle y_1, y_2; \overrightarrow{v}\rangle\ h') \\
\qquad \text{where } h'\ \langle z_1, z_2; \overrightarrow{u}\rangle = h\ \langle(z_1 + z_2); \overrightarrow{u}\rangle
\end{array}
$$

The evaluation function, both in its general form and its instantiation for our example, is defined as follows:

$$
\begin{array}{ll}
\mathsf{evaluate}_f \colon B^f \to B^\nu & \mathsf{evaluate}_{\mathsf{F}} \colon \mathbb{N}^{\mathsf{F}} \to \mathbb{N}^\nu \\
\quad \mathsf{evaluate}_f\ y = \theta_f\ 1\ \langle y\rangle\ (\lambda\langle z\rangle.z) & \quad \mathsf{evaluate}_{\mathsf{F}}\ y = \theta_{\mathsf{F}}\ 1\ \langle y\rangle\ (\lambda\langle z\rangle.z)
\end{array}
$$

where $\lambda\langle z\rangle.z$ denotes the function giving the only element of a vector of length one.

Finally, we define the desired functions f and F:

$$
\begin{array}{ll}
f \colon A \to B^\nu & \mathsf{F} \colon \mathbb{N} \to \mathbb{N}^\nu \\
\quad f\ x = \mathsf{evaluate}_f\ (f^\star\ x) & \quad \mathsf{F}\ n = \mathsf{evaluate}_{\mathsf{F}}\ (\mathsf{F}^\star\ n).
\end{array}
$$

5 Validity of the Prophecy Method

We want to prove that the formal version of the function f defined with the prophecy method is a correct implementation of the informal recursive function. Specifically, we want to prove the validity of the equations defining the function.

Remember that f may return an element consisting of infinite computation steps when applied to certain inputs. The inductive relation (defined in [8])

$\mathsf{Value}\colon A^\nu \to A \to \mathsf{Prop}$, for which we use the notation $(\_ \downarrow \_)$, characterises terminating computations. The expression $(x \downarrow a)$ states that the element x of A^ν converges to the value a in A. Its inductive definition has two rules:

$$\frac{}{\ulcorner a \urcorner \downarrow a} \qquad \frac{x \downarrow a}{\triangleright\, x \downarrow a}.$$

We now formulate the recursive equations in terms of $(\_ \downarrow \_)$ and we prove that our implementation of the function satisfies them.

For each non-recursive equation $f\ p = a$ in the informal definition of f, where a may contain occurrences of the variables in Γ but no recursive calls to f, we would like to show that the formal version of f satisfies

$$\forall \overrightarrow{x} : \Gamma, f\ p \downarrow a;$$

where $\overrightarrow{x}$ are the variables defined in the context Γ, and for each recursive equation of the form $f\ p = e[(f\ p_1), \ldots, (f\ p_n)]$ in the informal definition of f, we would like to show that its formalisation is such that

$$\forall \overrightarrow{x} : \Gamma, \forall r_1, \ldots, r_n : B, (f\ p_1) \downarrow r_1 \to \cdots \to (f\ p_n) \downarrow r_n \to (f\ p) \downarrow e[r_1, \ldots, r_n].$$

For our example function F, we want to prove the following three statements

$$(\mathsf{F}\ 0) \downarrow \mathsf{a},$$

$$\forall m\colon \mathbb{N}, (\mathsf{F}\ (g\ 0)) \downarrow m \to (\mathsf{F}\ 1) \downarrow (\mathsf{b} + \mathsf{c} * m),$$

$$\forall n, m_1, m_2\colon \mathbb{N}, (\mathsf{F}\ (g\ n)) \downarrow m_1 \to (\mathsf{F}\ (g\ (\mathsf{S}\ n))) \downarrow m_2 \to (\mathsf{F}\ (\mathsf{S}\ (\mathsf{S}\ n)))) \downarrow (m_1 + m_2).$$

On the road to proving these results, let us consider more closely the meaning of prophecies. A prophecy can be seen as the tree representation of the computation of the result of an expression. That is, it would be the computation trace, if parallel evaluation were allowed. For example the prophecy

$$\mathsf{b}_e\ \overrightarrow{x'}\ y_1 \cdots y_n$$

specifies a parallel computation in which we first evaluate the subtrees $y_1, \ldots, y_n$ and, if all these computations terminate giving $r_1, \ldots, r_n$ as result, respectively, then we obtain the output by computing the expression $e[r_1, \ldots, r_n]$.

Recall that, since types of partial elements like B^ν represent computations in a sequential model, we could not directly define the evaluation of a prophecy following the above intuition, but we needed to use the sequentialising operator θ_f.

We can characterise the behaviour of $(\theta_f\ k\ \overrightarrow{v}\ h)$ as follows:

> $(\mathsf{evaluate}_f\ v_i)$ converges for every prophecy v_i in the vector $\overrightarrow{v}\colon (B^f)^k$ if and only if $(\theta_f\ k\ \overrightarrow{v}\ h)$ converges; moreover, in case they converge, if $z_i\colon B$ is the value of $(\mathsf{evaluate}_f\ v_i)$ for $0 \leqslant i \leqslant k$, then $(h\ \langle z_1, \ldots, z_k \rangle)$ is the value of $(\theta_f\ k\ \overrightarrow{v}\ h)$.

The characterisation of θ_f is described in the following lemma, where the inductive relation $(\overrightarrow{v} \downarrow\downarrow \overrightarrow{u})$ expresses that $(\mathsf{evaluate}_f\ v_i) \downarrow u_i$ for $0 \leqslant i \leqslant k$, where v_i is the ith element of $\overrightarrow{v} : (B^f)^k$ and u_i is the ith element of $\overrightarrow{u} : B^k$.

Lemma 1

$$\forall k : \mathbb{N}, \forall \overrightarrow{v} : (B^f)^k, \forall h : B^k \to B, \forall b : B,$$
$$(\theta_f\ k\ \overrightarrow{v}\ h) \downarrow b \longleftrightarrow \exists \overrightarrow{u} : B^k, (\overrightarrow{v} \downarrow\downarrow \overrightarrow{u}) \wedge (h\ \overrightarrow{u} = b).$$

In order to prove Lemma 1 we define a *weight* on prophecies and vectors of prophecies. Intuitively, the weight of a finite prophecy indicates the size of the prophecy. The weight of a prophecy, when defined, must be a positive natural number. Moreover, the weight of a tree-structured prophecy y will only be defined if the weights of all its children are defined. In addition, the weight of y has to be strictly greater than the sum of the weights of its children.

Since prophecies need not be well-founded trees, it is not possible to define their weights by a total function. Instead, the weight of a prophecy is defined as an inductive relation $\mathsf{Wght} : B^f \to \mathbb{N} \to \mathsf{Prop}$ with a constructor for each constructor in the set of prophecies. For each constant constructor $\mathsf{b} : B^f$, there is a weight constructor of the form

$$\frac{}{\mathsf{wght_b} : \mathsf{Wght\ b\ 1}},$$

and for each non-constant constructor $\mathsf{b}_e : \Gamma_e \to \overbrace{B^f \to \cdots \to B^f}^{n \text{ times}} \to B^f$, there is a weight constructor of the form

$$\frac{\overrightarrow{x'} : \Gamma_e \qquad h_1 : \mathsf{Wght}\ y_1\ w_1 \quad \cdots \quad h_n : \mathsf{Wght}\ y_n\ w_n}{\mathsf{wght}_{\mathsf{b}_e}\ \overrightarrow{x'}\ h_1 \cdots h_n : \mathsf{Wght}\ (\mathsf{b}_e\ \overrightarrow{x'}\ y_1\ \cdots y_n)\ (\mathsf{S}\ (w_1 + \cdots + w_n))}.$$

The weight of a vector of prophecies is the sum of the weights of its elements plus its length, and it is given by an inductive relation $\mathsf{Weight} : (k : \mathbb{N})(B^f)^k \to \mathbb{N} \to \mathsf{Prop}$ defined as follows:

$$\frac{}{\mathsf{weight}_0 : \mathsf{Weight}\ 0\ \langle\rangle\ 0} \qquad \frac{h_y : \mathsf{Wght}\ y\ w_y \qquad h_v : \mathsf{Weight}\ k\ \overrightarrow{v}\ w_v}{\mathsf{weight}_{\mathsf{S}}\ k\ h_y\ h_v : \mathsf{Weight}\ (\mathsf{S}\ k)\ \langle y; \overrightarrow{v}\rangle\ (\mathsf{S}\ (w_y + w_v))}.$$

The statement of Lemma 1 can now be restrained to those vectors of prophecies that have a weight. This makes it easier to prove the lemma by applying course-of-value induction on the weight of the vector. Later, in lemmas 3 and 5, we show that the restriction can be relaxed because all converging prophecies have a weight.

Lemma 2

$$\forall\ w : \mathbb{N}, \forall k : \mathbb{N}, \forall \overrightarrow{v} : (B^f)^k, \mathsf{Weight}\ k\ \overrightarrow{v}\ w \to$$
$$\forall h : B^k \to B, \forall b : B, (\theta_f\ k\ \overrightarrow{v}\ h) \downarrow b \longleftrightarrow \exists \overrightarrow{u} : B^k, (\overrightarrow{v} \downarrow\downarrow \overrightarrow{u}) \wedge (h\ \overrightarrow{u} = b).$$

Proof. By course-of-value induction on the weight w and cases on k. Recall that the value of k determines the structure of the vector v. When v is not empty, we also perform cases on its first element.

If $k = 0$, then $\overrightarrow{v} = \langle\,\rangle$. By definition of θ_f, we have $(\theta_f\ 0\ \langle\,\rangle\ h) = \ulcorner h\ \langle\,\rangle \urcorner$ and, therefore, it must be that $b = (h\ \langle\,\rangle)$ and $\overrightarrow{u} = \langle\,\rangle$. Hence, the statement is true.

If $k = (\mathsf{S}\ k')$ then $\overrightarrow{v} = \langle y; \overrightarrow{v'}\rangle$. We now perform case analysis on y.

Let y be a leaf. We know that the vector $\overrightarrow{v'}$ must have a weight w' and moreover, w' must be strictly smaller than w, $w' < w$. Both directions of the lemma can now be easily proved by induction hypothesis on the weight w', and by definition of the functions θ_f and $\mathsf{evaluate}_f$.

Let y have a tree structure, that is, $y = (\mathsf{b}_e\ \overrightarrow{x'}\ y_1 \cdots y_n)$. Given h, by definition of θ_f we get

$$\theta_f\ (\mathsf{S}\ k')\ \langle(\mathsf{b}_e\ \overrightarrow{x'}\ y_1 \cdots y_n); \overrightarrow{v'}\rangle\ h = \triangleright(\theta_f\ (n + k')\ \langle y_1, \ldots, y_n; \overrightarrow{v'}\rangle\ h')$$

with $(h'\ \langle z_1, \ldots, z_n; \overrightarrow{u}\rangle) = h\ \langle e[z_1, \ldots, z_n]; \overrightarrow{u}\rangle$. Hence, for any given b, we have the equivalence

$$(\theta_f\ (\mathsf{S}\ k')\ \overrightarrow{v}\ h) \downarrow b \longleftrightarrow (\theta_f\ (n + k')\ \langle y_1, \ldots, y_n; \overrightarrow{v'}\rangle\ h') \downarrow b. \tag{1}$$

The new vector of prophecies has a smaller weight than the original one, since we replaced the first element in the vector by all its children. That is, there is a weight w' such that $(\mathsf{Weight}\ (n + k')\ \langle y_1, \ldots, y_n; \overrightarrow{v'}\rangle\ w')$ and $w' < w$. Therefore we can apply the induction hypothesis to w', h' and b and obtain that

$$\begin{aligned}&(\theta_f\ (n + k')\ \langle y_1, \ldots, y_n; \overrightarrow{v'}\rangle\ h') \downarrow b \longleftrightarrow \\ &\exists\langle z_1, \ldots, z_n; \overrightarrow{u}\rangle \colon B^{n+k'}, (\langle y_1, \ldots, y_n; \overrightarrow{v'}\rangle \twoheadrightarrow \langle z_1, \ldots, z_n; \overrightarrow{u}\rangle) \wedge \\ &\qquad\qquad (h'\ \langle z_1, \ldots, z_n; \overrightarrow{u}\rangle = b).\end{aligned} \tag{2}$$

In addition, the weight of $\langle y_1, \ldots, y_n\rangle$ is strictly smaller than the weight of the vector $\overrightarrow{v}$, so we can apply the inductive hypothesis again with the continuation $h = \lambda\langle z_1, \ldots, z_n\rangle.e[z_1, \ldots, z_n]$ to obtain

$$\begin{aligned}&\forall b \colon B, (\theta_f\ n\ \langle y_1, \ldots, y_n\rangle\ (\lambda\langle z_1, \ldots, z_n\rangle.e[z_1, \ldots, z_n])) \downarrow b \longleftrightarrow \\ &\quad \exists\overrightarrow{z} \colon B^n, (\langle y_1, \ldots, y_n\rangle \twoheadrightarrow \overrightarrow{z}) \wedge ((\lambda\langle z_1, \ldots, z_n\rangle.e[z_1, \ldots, z_n])\ \overrightarrow{z} = b).\end{aligned} \tag{3}$$

Now we prove the main statement.

In the direction from left to right, let us assume $(\theta_f\ (\mathsf{S}\ k')\ \overrightarrow{v}\ h) \downarrow b$. By the equivalence in (1), we have that $(\theta_f\ (n + k')\ \langle y_1, \ldots, y_n; \overrightarrow{v'}\rangle\ h') \downarrow b$. Now, by the instantiated induction hypothesis in (2), we know that there exists a vector $\langle z_1, \ldots, z_n; \overrightarrow{u}\rangle$ with the stated properties. In particular, $(\mathsf{evaluate}_f\ y_i) \downarrow z_i$ for $0 \leqslant i \leqslant n$, and hence $\langle y_1, \ldots, y_n\rangle \twoheadrightarrow \langle z_1, \ldots, z_n\rangle$.

Let $z = e[z_1 \cdots z_n]$; we claim that $\langle z; \overrightarrow{u}\rangle$ is the vector satisfying the conclusion of the the lemma. We need to prove that

$$(\langle y; \overrightarrow{v'}\rangle \twoheadrightarrow \langle z; \overrightarrow{u}\rangle) \wedge (h\ \langle z; \overrightarrow{u}\rangle = b),$$

which amounts to proving that $(\mathsf{evaluate}_f\ y) \downarrow z$, since the rest follows from the equivalence in (2). By definition of $\mathsf{evaluate}_f$, we need to prove that

$$(\theta_f\ 1\ \langle(\mathsf{b}_e\ \overrightarrow{x}\ y_1 \cdots y_n)\rangle\ (\lambda\langle z'\rangle.z')) \downarrow e[z_1 \cdots z_n].$$

By unfolding the definition of θ_f, we obtain that this statement is equivalent to the following:

$$(\theta_f\ n\ \langle y_1, \ldots, y_n\rangle\ (\lambda\langle z_1, \ldots, z_n\rangle.e[z_1, \ldots, z_n])) \downarrow e[z_1 \cdots z_n]. \tag{4}$$

We now instantiate (3) with $b = e[z_1 \cdots z_n]$, and we apply the right-to-left direction of the resulting equivalence with the vector $\langle z_1, \ldots, z_n\rangle$ and the corresponding proofs of the needed hypotheses. We obtain then a proof of the claim in (4).

In the direction from right to left, assume that

$$\exists\langle z; \overrightarrow{u}\rangle \colon B^k, (\langle y; \overrightarrow{v'}\rangle \downdownarrows \langle z; \overrightarrow{u}\rangle) \wedge (h\ \langle z; \overrightarrow{u}\rangle = b).$$

Then $(\mathsf{evaluate}_f\ y) \downarrow z$ and, since $y = (b_e\ \overrightarrow{x}\ y_1 \cdots y_n)$, by the definitions of $\mathsf{evaluate}_f$ and θ_f, we can apply the left-to-right direction of (3) with $b = z$. Thus, there exists a vector $\langle z_1, \ldots, z_n\rangle$ such that $(\mathsf{evaluate}_f\ y_i) \downarrow z_i$ for $0 \leqslant i \leqslant n$, and $e[z_1 \cdots z_n] = z$.

We can now use the right-to-left direction of (2) with the vector $\langle z_1, \ldots, z_n; \overrightarrow{u}\rangle$ and the corresponding proofs of the needed hypotheses.

Finally, the equivalence in (1) allows us to conclude that $(\theta_f\ (\mathsf{S}\ k')\ \langle y; \overrightarrow{v}\rangle\ h) \downarrow b$. □

In the next three lemmas, we show that all converging prophecies have a weight. This allows us to eliminate the weight constraint in Lemma 2, which in turn, easily allows us to obtain a proof of Lemma 1.

Lemma 3

$$\forall k \colon \mathbb{N}, \forall \overrightarrow{v} \colon (B^f)^k, \forall h \colon B^k \to B, \forall b \colon B,\\ (\theta_f\ k\ \overrightarrow{v}\ h \downarrow b) \to \exists\ w, \mathsf{Weight}\ k\ \overrightarrow{v}\ w.$$

Proof. By induction on the structure of the proof of $(\theta_f\ k\ \overrightarrow{v}\ h) \downarrow b$, and by cases on the vector and, when the vector is not empty, on its first element. □

Lemma 4

$$\forall y \colon B^f, \forall b \colon B, (\mathsf{evaluate}_f\ y) \downarrow b \to \exists w, \mathsf{Wght}\ y\ w.$$

Proof. By definition of the operator $\mathsf{evaluate}_f$ and Lemma 3. □

Lemma 5

$$\forall k \colon \mathbb{N}, \forall \overrightarrow{v} \colon (B^f)^k, \forall \overrightarrow{u} \colon (B)^k, (\overrightarrow{v} \downdownarrows \overrightarrow{u}) \to \exists\ w, \mathsf{Weight}\ k\ \overrightarrow{v}\ w.$$

Proof. By induction on k, using Lemma 4 on each of the elements in the vector $\overrightarrow{v}$. □

Finally, we are in the position of proving the validity of the equations defining the function f. Proving the validity of non-recursive equations is immediate: we only need to reduce the functions f and then $\mathsf{evaluate}_f$ to obtain the desired result. The validity of the recursive equations can be proved by using Lemma 1.

Theorem 1 (Validity of Recursive Equations)

$$\forall \overrightarrow{x} : \Gamma, \forall r_1, \ldots, r_n : B, (f\ p_1) \downarrow r_1 \rightarrow \cdots \rightarrow (f\ p_n) \downarrow r_n \rightarrow (f\ p) \downarrow e[r_1, \ldots, r_n].$$

Proof. Assume that $(f\ p_i) \downarrow r_i$, for $1 \leqslant i \leqslant n$. By definition of f this means that $\mathsf{evaluate}_f\ (f^\star\ p_i) \downarrow r_i$. Unfolding definitions, we have that:

$$\begin{aligned} f\ p &= \mathsf{evaluate}_f\ (f^\star\ p) \\ &= \theta_f\ 1\ \langle (f^\star\ p) \rangle\ (\lambda \langle z \rangle . z) \\ &= \theta_f\ 1\ \langle \mathsf{b}_e\ \overrightarrow{x'}\ (f^\star\ p_1) \cdots (f^\star\ p_n) \rangle\ (\lambda \langle z \rangle . z) \\ &= \triangleright (\theta_f\ n\ \langle (f^\star\ p_1), \cdots, (f^\star\ p_n) \rangle\ \lambda \langle z_1, \ldots, z_n \rangle . e[z_1, \ldots, z_n]). \end{aligned}$$

The conclusion easily follows now by applying the second constructor of the inductive relation $(\_ \downarrow \_)$ and the right-to-left direction of Lemma 1. □

In the specific case of the function F, Theorem 1 gives the validity of the equations presented in page 77.

When reasoning about recursive functions, an inversion principle given by the converse of Theorem 1 may be useful.

Theorem 2 (Inversion Principle for Recursive Equations)

$$\begin{aligned} &\forall \overrightarrow{x} : \Gamma, \forall b : B, (f\ p) \downarrow b \rightarrow \\ &\exists r_1, \ldots, r_n : B, (f\ p_1) \downarrow r_1 \wedge \cdots \wedge (f\ p_n) \downarrow r_n \wedge b = e[r_1, \ldots, r_n]. \end{aligned}$$

Proof. By the left-to-right direction of Lemma 1. □

6 Conclusions

This article describes a new method to represent (partial) recursive functions in type theory. It combines ideas from our previous work on the subject, namely, the one characterising the inputs on which a function terminates by an inductive (domain) predicate [6,4,7], and the one implementing recursive functions by co-recursion on co-inductive types of partial elements [8].

Given the recursive equations for a computable function $f : A \rightarrow B$, we define a co-inductive set B^f of *prophecies* representing the traces of the computation of the function. This set is the dual of the predicate defining the domain of the function as described in [6]. It is easy to define a formal recursive function $f^\star : A \rightarrow B^f$ returning prophecies as output. The type-theoretic version of the

original function is then a function whose co-domain is the set B^ν of partial elements as studied in [8]. This function is defined by *linearising* the prophecy obtained from the formal recursive function $f^\star$. The linearisation is done by an operator θ_f with two parameters: a stack of recursive calls and a continuation function that will compute, when possible, the final result.

We prove that a function formalised in type theory with this new method satisfies all the equations (recursive or not) of its informal version.

We illustrate the method on a toy example, a generalisation F of the Fibonacci function. The development for F has been fully formalised in the proof assistant *Coq* [14,3]. In addition, we also performed a complete formalisation of the *quicksort* algorithm following this method, and of the proofs stating the validity of the equations for the algorithm. The files of both formalisations are available on the web at the address: `http://www.cs.ru.nl/~venanzio/Coq/prophecy.html`.

At the moment, the method has been tested just on simple recursive programs, that is, not nested or mutually recursive. The formalisation of mutually recursive functions usually presents no major problems in systems like *Coq*, but on the other hand, nested functions are not trivial to formalise. Already our previous work (using the domain predicates) on nested recursive functions [5] could not directly be translated into *Coq* because the proof assistant lacks support for inductive-recursive definitions, as described by Dybjer in [10].

As can be seen from the proofs, the method is general and can be adapted to every simple recursive program. However, we have yet to formalise the mechanisation process and therefore, the user must go through the tedious but trivial process of adapting definitions and proofs. It would be desirable, in a future stage, to fully automatise the definitions of the set of prophecies, of the function θ_f, and the related proofs from the set of (recursive) equations given by the user.

As mentioned before, this technique has several advantages over the method we have previously developed via a special-purpose accessibility (domain) predicate [6,4,7]; namely, there is no need of extra logical arguments in the definition of the recursive function; the function can be applied to any element in its domain, regardless of termination properties; we obtain a type of partial recursive functions between any two given types; and composition of recursive functions can be easily defined through the usual composition of monadic function (see [8] for further reading on this point).

References

1. Balaa, A., Bertot, Y.: Fonctions récursives générales par itération en théorie des types. Journées Francophones des Langages Applicatifs - JFLA02, INRIA (January 2002)
2. Barthe, G., Courtieu, P.: Efficient reasoning about executable specifications in Coq. In: Carreño, V.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2002. LNCS, vol. 2410, pp. 20–23. Springer, Heidelberg (2002)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer, Berlin Heidelberg (2004)

4. Bove, A.: General recursion in type theory. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 39–58. Springer, Heidelberg (2003)
5. Bove, A., Capretta, V.: Nested general recursion and partiality in type theory. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 121–135. Springer, Heidelberg (2001)
6. Bove, A., Capretta, V.: Modelling general recursion in type theory. Mathematical Structures in Computer Science 15(4), 671–708 (2005)
7. Bove, A., Capretta, V.: Recursive functions with higher order domains. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 116–130. Springer, Heidelberg (2005)
8. Capretta, V.: General recursion via coinductive types. Logical Methods in Computer Science 1(2), 1–18 (2005)
9. Capretta, V., Uustalu, T., Vene, V.: Recursive coalgebras from comonads. Information and Computation 204(4), 437–468 (2006)
10. Dybjer, P.: A general formulation of simultaneous inductive-recursive definitions in type theory. Journal of Symbolic Logic, vol. 65(2) (2000)
11. Giménez, E.: Codifying guarded definitions with recursive schemes. In: Smith, J., Dybjer, P., Nordström, B. (eds.) TYPES 1994. LNCS, vol. 996, pp. 39–59. Springer, Heidelberg (1995)
12. McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming 14(1), 69–111 (2004)
13. Reynolds, J.C.: The discoveries of continuations. Lisp. and Symbolic Computation 6(3–4), 233–248 (1993)
14. The Coq Development Team. LogiCal Project. The Coq Proof Assistant. Reference Manual. Version 8. INRIA (2004) Available at the web page `http://pauillac.inria.fr/coq/coq-eng.html`