A Type of Partial Recursive Functions

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. Our goal is to define a type of partial recursive functions in constructive type theory. In a series of previous articles, we studied two different formulations of partial functions and general recursion. We could obtain a type only by extending the theory with either an impredicative universe or with coinductive definitions. Here we present a new type constructor that eludes such entities of dubious constructive credentials. We start by showing how to break down a recursive function definition into three components: the first component generates the arguments of the recursive calls, the second evaluates them, and the last computes the output from the results of the recursive calls. We use this dissection as the basis for the introduction rule of the new type constructor. Every partial recursive function is associated with an inductive domain predicate; evaluation of the function requires a proof that the input values satisfy the predicate. We give a constructive justification for the new construct by interpreting it into the base type theory. This shows that the extended theory is consistent and constructive.

1 Introduction

Our research investigates the formalisation of partial functions and general recursion in constructive type theory. In a series of previous articles, we expound two different ways of achieving that goal.

In our first approach [3,4,2,5], we define an inductive (domain) predicate that characterises the inputs on which a function terminates. The constructors of this predicate are automatically determined from the recursive equations defining the function. The function itself is formalised by structural recursion on an extra argument, a proof that the input values satisfy the domain predicate.

In our second approach [9,6], refined by Megacz [19], we associate to each data type a coinductive type of partial elements. Computations are modelled by (possibly) infinite structures. Partial and general recursive functions are implemented by corecursion on these types.

It is desirable that all partial functions with the same source and target are elements of the same type, rather than each function having an ad hoc type. In three of the articles mentioned above, we succeed in defining such a type, but there is always a cost to pay. In [5], we need to work in an impredicative type theory. Alternatively, we could work with a hierarchy of universes and accept that we are not able to formalise all recursive function definitions. In [9,6], we need support for coinductive definitions.

Other approaches to the definition of a type of partial recursive functions can be found in the literature; we summarise those we believe are more relevant.

Constable and Mendler [13] introduce a type of partial functions as a new type constructor in the Nuprl system [12]. Given a partial function, one can compute its domain predicate, which contains basically the same information as our domain predicates. A difference is that when defining a partial function in the Nuprl system, the actual definition of the partial function does not depend on its domain predicate. This would not be possible in the intuitionistic type theories in which we work. In other words, in Nuprl, a partial function from A to B maps an element a of A into an element of B provided there is some proof p that a belongs to the domain of the function. In our case, the formalisation of the partial function would map both the a and the p into B.

In [14], Constable and Smith develop a partial type theory for the Nuprl type system in which, for each type of the underlying total theory, there exists another type which might contain diverging terms. Together with this type of partial elements, a termination predicate and an induction principle to reason about partial functions are introduced.

Audebaud [1] uses the above idea [14] to define a conservative extension of the Calculus of Constructions [15] with fixed point terms and a type of partial objects. Strong normalisation still holds for terms with no fixed points. From the computational point of view, an equivalent of the Kleene theorem for partial recursive functions is obtained, but logical aspects need more examination.

Based on our work [3], Setzer [22,23] defines a type (of *codes*) of partial functions. From the code of a partial function, one can extract the domain of the function and the function itself, and one can evaluate the function on a certain argument. Nested functions and higher-order functions can also be coded as elements of the type of partial functions.

The approach illustrated in the present article is based on an analysis of a recursive definition into three components. The first component determines the arguments of the recursive calls, the second computes the recursive calls, and the third combines the results of the recursive calls to produce the output. This decomposition was described in [10] and is analogous to the separation of an hylomorphism into the composition of an anamorphism and a catamorphism in [20]. The method works in standard intuitionistic type theory and gives simple and direct formalisations, but we must pay a price: nested recursion and higher-order partiality are not possible anymore. The class of recursive functions that can be coded is still Turing-complete, but some function definitions may need to be rewritten to fit the pattern.

We formalised our approach in the proof assistant Coq [24]. The file containing the formalisation can be obtained from the following web page: `www.cs.ru.nl/~venanzio/Coq/rec_fun_type/`

The paper is organised as follows. Section 2 gives a closer look at functions and shows how a function can be split into three components, following the description in [10,11]. Section 3 gives the formal rules of the type of partial recursive functions. Section 4 formalises the signatures that describe the structure of the recursive calls in a way that ensures that the rules of the new type are predicative. Section 5 shows the consistency of the type theory extended with the new type by modelling it inside the base theory. Section 6 presents a modification of the rules to allow a general recursive pattern in which the input can also be used directly in the computation of the output, and not just in the generation of the recursive arguments. Finally, Section 7 summarises the achieved results.

2 A Closer Look at Recursive Functions

A common informal way to define a partial recursive function is by a sequence of recursive equations, as it could be implemented in a pure functional language like Haskell [17]:

$$\begin{aligned}
&f : A \to B\\
&f\ p_0 = e_0[(f\ a_{00}), \ldots, (f\ a_{0k_0})]\\
&\quad\vdots\\
&f\ p_n = e_n[(f\ a_{n0}), \ldots, (f\ a_{nk_n})].
\end{aligned}$$

Here $p_0, \ldots, p_n$ are *patterns*, that is, general terms containing variables and constructors. The notation $e[\cdots]$ denotes an expression e containing occurrences of the terms inside the square brackets. We restrict the form of the definition by requiring that the only occurrences of f are the displayed ones; specifically, f does not occur in any of the a_{ij}'s. This means that we are excluding nested recursive definitions.

When we apply a function to a concrete argument, the argument is matched against the patterns to find the equation that must be evaluated to give the result. We will not enter here into details about this process, nor about issues like exhaustiveness or overlapping patterns. What is important in what follows is that, when we give an actual input a to the function f, the system compares a with each pattern until it finds one, say p_i, that matches a. Then, it computes the new recursive arguments $a_{i0}, \ldots, a_{ik_i}$ on which to apply the function, and recursively computes f on these arguments. Finally, the system feeds the results of the recursive calls to the expression e_i in order to obtain the final output.

Next, we first illustrate with an example the issues involved in the process just described and then we present a generalisation of the example that allows a dissection of functions into three well-defined components. In the following section, we base our new type of *partial recursive functions* on this dissection.

A First Example

The example is a recursive functional program to translate Gödel's coding system for trees into Cantor's.

Kurt Gödel devised a system to encode expressions of a formal language into natural numbers by exploiting the uniqueness of prime factorisation [16]. His idea was that a complex expression can be encoded by giving the codes of its immediate n subexpressions as exponents to the first n prime numbers. As an instance of this process let us consider the coding of unlabelled well-founded trees of arbitrary branching degree. Let a tree t be represented by a node with a finite number of subtrees, that is, something of the form $\mathsf{node}(t_0, \ldots, t_k)$. Gödel's representation recursively encodes each tree t_i into a natural number g_i and then uses these g_i's as exponents of prime numbers to obtain the code of t as follows: if p_i is the $(i+1)$th prime number, the code of t is $p_0^{g_0} \cdot \cdots \cdot p_k^{g_k}$. A leaf is a node with no branches, $\mathsf{node}()$, and has code 1.

If we are interested in trees with small branching degree, high prime numbers are never used. In this case it is more convenient to adopt a different encoding that uses Cantor's pairing function. According to Cantor, a pair $\langle n_1, n_2 \rangle$ of natural numbers can be encoded by the number

$$\mathsf{pair}\ \langle n_1, n_2 \rangle = (n_1 + n_2) \cdot (n_1 + n_2 + 1)/2 + n_2.$$

This is a bijection between pairs and single numbers. Longer vectors can be encoded by repeated use of the pairing function: $\mathsf{tuple}_0\ \langle\rangle = 0$, $\mathsf{tuple}_1\ \langle n_0 \rangle = n_0$ and, recursively, $\mathsf{tuple}_{k+1}\ \langle n_0, \ldots, n_k \rangle = \mathsf{pair}\ \langle n_0, \mathsf{tuple}_k\ \langle n_1, \ldots, n_k \rangle\rangle$. We can now represent trees by an encoding similar to Gödel's: if c_i is the Cantor code of the tree t_i, then $\mathsf{pair}\ \langle k+1, \mathsf{tuple}_{k+1} \langle c_0, \ldots, c_k \rangle\rangle$ is the code of $\mathsf{node}(t_0, \ldots, t_k)$.

We adopt the convention of writing $\langle x_0, \ldots, x_{k-1} \rangle$ for a tuple of length k, understanding it to denote the empty tuple $\langle\rangle$ if $k = 0$. Similarly, a product $p_0^{i_0} \cdots p_{k-1}^{i_{k-1}}$ is understood to be equal to 1 if $k = 0$.

We define a translation function that maps Gödel's encoding of trees into Cantor's encoding:

$$\begin{array}{l}
\mathsf{trans\_code} \colon \mathbb{N} \to \mathbb{N} \\
\mathsf{trans\_code}\ x = \mathsf{pair}\ \langle k, \mathsf{tuple}_k\ \langle c_0, \ldots, c_{k-1} \rangle\rangle \\
\quad \text{where} \quad c_i = \mathsf{trans\_code}\ g_i \quad \text{for } i = 0, \ldots, k-1 \\
\qquad\qquad\quad \text{with } k, g_0, \ldots, g_{k-1} \text{ such that } x = p_0^{g_0} \cdots p_{k-1}^{g_{k-1}}.
\end{array}$$

Note that the function is undefined on 0, since 0 cannot be written as a product of powers of primes. This kind of partiality is relatively easy to deal with, since it is decidable, unlike the partiality arising from non-termination, which is the real topic of our work. We assume that one of the following solutions is adopted: state that $\mathsf{trans\_code}\ 0 = 0$, turn this kind of partiality into a non-terminating loop by adding the equation $\mathsf{trans\_code}\ 0 = \mathsf{trans\_code}\ 0$, or extend the target type to $\mathbb{N} + \mathsf{Unit}$ (the operator $+$ denotes the disjoint union on types and Unit is the type with only one element) to generate an exception on non-matchable inputs. So we brush aside this technical issue for the rest of the example.

We can analyse this translation algorithm by splitting it into three components, following the categorical description in [10,11].

The first step consists in computing, from the input x, the index $k-1$ of the largest prime divisor of x and the exponents $g_0, \ldots, g_{k-1}$ of the prime numbers

$p_0, \ldots, p_{k-1}$ in x (with $k = 0$ if $x = 1$). We give a name and a type to this function:

$$\begin{aligned}&\alpha_{\mathsf{trans\_code}} \colon \mathbb{N} \to \textstyle\sum_{k:\mathbb{N}} \mathbb{N}^k\\ &\alpha_{\mathsf{trans\_code}}\ x = \langle k, \langle g_0, \ldots, g_{k-1} \rangle\rangle\\ &\quad \text{with } k, g_0, \ldots, g_{k-1} \text{ such that } x = p_0^{g_0} \cdots p_{k-1}^{g_{k-1}}.\end{aligned}$$

The sum type $\sum_{k:\mathbb{N}} \mathbb{N}^k$ consists of elements of the form $\langle k, \boldsymbol{g}\rangle$, where k is the length of the tuple and $\boldsymbol{g}$ is a tuple of k natural numbers.

The second step consists in applying the function $\mathsf{trans\_code}$ recursively to the elements of the vector obtained in the first step. We indicate the lifting of the function to tuples by putting an arrow over it:

$$\begin{aligned}&\overrightarrow{\mathsf{trans\_code}} \colon \textstyle\sum_{k:\mathbb{N}} \mathbb{N}^k \to \sum_{k:\mathbb{N}} \mathbb{N}^k\\ &\overrightarrow{\mathsf{trans\_code}}\ \langle k, \langle g_0, \ldots, g_{k-1}\rangle\rangle = \langle k, \langle \mathsf{trans\_code}\ g_0, \ldots, \mathsf{trans\_code}\ g_{k-1}\rangle\rangle.\end{aligned}$$

The third and final step of the translation algorithm is the computation of the output from the results of the recursive calls:

$$\begin{aligned}&\beta_{\mathsf{trans\_code}} \colon \textstyle\sum_{k:\mathbb{N}} \mathbb{N}^k \to \mathbb{N}\\ &\beta_{\mathsf{trans\_code}}\ \langle k, \langle c_0, \ldots, c_{k-1}\rangle\rangle = \mathsf{pair}\ \langle k, \mathsf{tuple}_k \langle c_0, \ldots, c_{k-1}\rangle\rangle.\end{aligned}$$

The function $\mathsf{trans\_code}$ is now specified by giving the fixed point equation

$$\mathsf{trans\_code} = \beta_{\mathsf{trans\_code}} \circ \overrightarrow{\mathsf{trans\_code}} \circ \alpha_{\mathsf{trans\_code}}.$$

The General Framework

The explanation given above suggests an analysis into three steps, already studied from the categorical point of view in [10,11].

Recall the general form of the definition of a function f given by several (recursive) equations and presented at the beginning of this section. We adopt a uniform formulation that abstracts away from the actual matching algorithm: the structure of the recursive calls is given by a type operator (a functor in categorical terms) which, in the case of the function f, is as follows:

$$\begin{aligned}&F \colon \mathsf{Set} \to \mathsf{Set}\\ &FX = X^{k_0} + \cdots + X^{k_n}.\end{aligned}$$

The form of the functor may also be more complex, possibly containing type parameters and dependent families, for example, for the $\mathsf{trans\_code}$ function the functor is $F_{\mathsf{trans\_code}}\, X = \sum_{k:\mathbb{N}} X^k$.

The first step in the computation of a function is represented by a map $\alpha : A \to FA$ (in categorical terms, an F-coalgebra). Specifically, for f we have:

$$\begin{aligned}&\alpha \colon A \to A^{k_0} + \cdots + A^{k_n}\\ &\alpha\ p_0 = \mathsf{in}_0 \langle a_{00}, \ldots, a_{0k_0}\rangle\\ &\qquad\vdots\\ &\alpha\ p_n = \mathsf{in}_n \langle a_{n0}, \ldots, a_{nk_n}\rangle,\end{aligned}$$

where in_i is the ith injection: if $n = 0$ then in_0 is just the identity, and for $n > 0$, $\mathsf{in}_i\ y = \mathsf{inl}\,(\overbrace{\mathsf{inr}\,(\cdots\,(\mathsf{inr}}^{i \text{ times}}(y))\cdots))$ if $0 \leqslant i < n$ and $\mathsf{in}_n\ y = \overbrace{\mathsf{inr}\,(\cdots\,(\mathsf{inr}}^{n \text{ times}}(y))\cdots)$.

By the function α above we mean the following: when α is applied to a particular argument, this is matched against the different patterns and when the first pattern matching the argument is found, the tuple with the recursive arguments is computed and returned.

The second step in the computation of f, the evaluation of the recursive calls, is the lifting of f by the functor F, $Ff\colon FA \to FB$.

The last step is a mapping $\beta\colon FB \to B$ (in categorical jargon, an F-algebra). It is computed by applying the appropriate e_i from the recursive equations.

In short, the analysis can be expressed by the following diagram:

$$\begin{tikzcd}
A \arrow[rrr, "\alpha"] \arrow[ddd, "f"'] & & & FA \arrow[ddd, "Ff"] \\
& p_i \arrow[r, mapsto] \arrow[d, mapsto] & \langle a_{i0}, \ldots, a_{ik_i} \rangle \arrow[d, mapsto] & \\
& e_i[(f\ a_{i0}), \ldots, (f\ a_{ik_i})] & \langle (f\ a_{i0}), \ldots, (f\ a_{ik_i}) \rangle \arrow[l, mapsto] & \\
B & & & FB. \arrow[lll, "\beta"]
\end{tikzcd}$$

The function f can now be given by the single equation $f = \beta \circ Ff \circ \alpha$, in place of the set of recursive equations presented at the beginning of this section.

A Final Example

Let us illustrate this analysis again with another example: a generalisation of the Fibonacci sequence that depends on three numerical parameters a, b, and c, and on a function parameter $g\colon \mathbb{N} \to \mathbb{N}$ (the actual Fibonacci sequence is obtained for $a = b = 1$, $c = 0$, and $g = \mathsf{id}_{\mathbb{N}}$):

$$\begin{aligned}
&\mathfrak{fib} : \mathbb{N} \to \mathbb{N} \\
&\mathfrak{fib}\ 0 = a \\
&\mathfrak{fib}\ 1 = b + c \cdot \mathfrak{fib}\ (g\ 0) \\
&\mathfrak{fib}\ (m+2) = \mathfrak{fib}\ (g\ m) + \mathfrak{fib}\ (g\ (m+1)).
\end{aligned}$$

The interest in this generalisation lies in the fact that, for some choices of the parameters, $\mathfrak{fib}$ will be a total function (for example, for the choices that give the actual Fibonacci sequence), while, for other choices, $\mathfrak{fib}$ will be partial (for example, if one chooses $a = b = 1$ and $c = 0$, but $g = (+1)$, the successor function over natural numbers). For the $\mathfrak{fib}$ function, F, α, and β are as follows:

$$F_{\mathfrak{fib}}\, X = \mathsf{Unit} + X + X^2$$

$$\begin{array}{ll}
\alpha_{\mathfrak{fib}}\ 0 = \mathsf{in}_0(\mathtt{tt}) & \beta_{\mathfrak{fib}}\ \mathsf{in}_0(\mathtt{tt}) = a \\
\alpha_{\mathfrak{fib}}\ 1 = \mathsf{in}_1(g\ 0) & \beta_{\mathfrak{fib}}\ \mathsf{in}_1(x) = b + c \cdot x \\
\alpha_{\mathfrak{fib}}\ (m+2) = \mathsf{in}_2\langle g\ m,\ g\ (m+1)\rangle & \beta_{\mathfrak{fib}}\ \mathsf{in}_2\langle y, z\rangle = y + z
\end{array}$$

where $\mathtt{tt}$ is the only element of the type Unit (which we identify with X^0).

3 The Type of Partial Recursive Functions

Inspired by the previous analysis, we introduce a new type constructor for partial recursive functions in which the coalgebra-algebra pair is used in the introduction rule. For the elimination rule, we define a domain predicate similar to the one in the Bove/Capretta method [3]. For full generality, the method we used in the previous section must be adapted by defining a *lifted universal quantifier*: for every predicate $P\colon X \to \mathsf{Prop}$ and functor F, we define $\bigwedge_{F,P}\colon FX \to \mathsf{Prop}$ as the conjunction of the statement of P on every element of type X occurring in an element of FX; its formal definition will be given in the next section.

We give now the formal rules for the type of partial recursive functions:

- Formation:
$$\frac{A\colon \mathsf{Set} \quad B\colon \mathsf{Set}}{A \rightharpoonup B\colon \mathsf{Set}};$$
- Introduction:
$$\frac{\alpha\colon A \to FA \quad \beta\colon FB \to B}{\mathsf{rec}(\alpha,\beta)\colon A \rightharpoonup B}\ F \text{ a functor};$$
- Domain predicate:
$$\frac{f\colon A \rightharpoonup B}{\mathsf{Dom}_f\colon A \to \mathsf{Prop}}, \quad \frac{\alpha\colon A \to FA \quad \beta\colon FB \to B \quad a\colon A \quad h\colon \bigwedge_{F,\mathsf{Dom}_{\mathsf{rec}(\alpha,\beta)}}(\alpha\ a)}{\mathsf{dom}_{\alpha,\beta}(a,h)\colon \mathsf{Dom}_{\mathsf{rec}(\alpha,\beta)}\ a};$$
- Application:
$$\frac{f\colon A \rightharpoonup B \quad a\colon A \quad h\colon \mathsf{Dom}_f\ a}{\mathsf{app}_f\ a\ h\colon B};$$
- Reduction:
$$\mathsf{app}_{\mathsf{rec}(\alpha,\beta)}\ a\ \mathsf{dom}_{\alpha,\beta}(a,h) \rightsquigarrow \beta\,(\overline{\mathsf{rec}(\alpha,\beta)}\ (\alpha\, a)\ h)$$

  where $\overline{f}$ is the lifting of the function $f : A \rightharpoonup B$ by the functor F:
$$\overline{f}\colon \forall t\colon FA,\ \bigwedge_{F,\mathsf{Dom}_f} t \to FB;$$
  its formal definition, which is more complex than the lifting of $\mathsf{trans\_code}$ in the previous section because of the presence of the domain predicate, will also be given in the next section. We could also see $\overline{f}$ as a recursive function $\overline{f}\colon F\alpha \rightharpoonup F\beta$ by setting $\overline{f} = \mathsf{rec}(F\alpha, F\beta)$. This definition would be equivalent to the one given.

With these rules, the functions $\mathsf{trans\_code}$ and $\mathfrak{fib}$ can simply be defined as:

$$\begin{array}{ll}\mathsf{trans\_code} \colon \mathbb{N} \rightharpoonup \mathbb{N} & \mathfrak{fib} \colon \mathbb{N} \rightharpoonup \mathbb{N} \\ \mathsf{trans\_code} = \mathsf{rec}(\alpha_{\mathsf{trans\_code}}, \beta_{\mathsf{trans\_code}}) & \mathfrak{fib} = \mathsf{rec}(\alpha_{\mathfrak{fib}}, \beta_{\mathfrak{fib}}).\end{array}$$

4 A Predicative Reflection

The rules given in the previous section do not take F as an explicit parameter, but in an intensional type theory F should always be given explicitly. Since every function has its own different functor F, we should really write $\mathsf{rec}(F, \alpha, \beta)$ in place of just $\mathsf{rec}(\alpha, \beta)$.

So, how should we formalise our functors? A functor is a higher-order object of type $\mathsf{Set} \to \mathsf{Set}$. (If we want to be really formal, we should give a more complex dependent type to F, also containing a proof of functoriality.) It is therefore clear that with a solution like this the type $A \rightharpoonup B$ would be inherently impredicative. One option to circumvent this impasse consists in stratifying partial functions over the hierarchy of predicative universes $\mathsf{U}_0, \mathsf{U}_1, \mathsf{U}_2, \ldots$ by defining $A \rightharpoonup_i B \colon \mathsf{U}_{i+1}$ as the type of partial functions $\mathsf{rec}(F, \alpha, \beta)$ where $F \colon \mathsf{U}_i \to \mathsf{U}_i$.

An alternative solution, adopted here, is to be stricter about the functors we allow. Observe that each equation in the definition of a recursive function always contains a finite number of recursive calls, therefore we can limit ourselves to finitary functors. The class of these functors is small enough to be encoded by a small type, similarly to the encoding of the larger class of strictly positive functors given in [8]. In other words, a functor can be given by a *signature* in the same way that algebraic structures are defined in Universal Algebras [7].

If the functor is of the form $FX = X^{k_0} + \cdots + X^{k_n}$, then the signature may consist of just the list of exponents $[k_0, \ldots, k_n]$. However, some functions have a more general signature: they may have an *a priori* unknown number of potential cases; this is the case for $\mathsf{trans\_code}$.

Therefore, we define a signature to be a mapping $\sigma \colon \mathbb{N} \to \mathbb{N}$ that specifies, for every possible number of recursive arguments, how many equations contain that number of recursive arguments. Intuitively, for each *arity* n, the signature σ gives us the number $(\sigma\, n)$ of cases (equations) of the function that generate that number of recursive calls. In other words, the function will have $(\sigma\, 0)$ cases with no recursive calls, $(\sigma\, 1)$ cases with a single recursive call, $(\sigma\, 2)$ cases with two recursive calls, and, in general, $(\sigma\, i)$ cases with i recursive calls. In the definition of the functor, the coefficient $(\sigma\ i)$ means that we take $(\sigma\ i)$ copies of X^i.

We define an operator $\_ \star \_$ such that $m \star Y$ gives the sum of m copies of Y:

$$\begin{array}{l}\_ \star \_ \colon \mathbb{N} \to \mathsf{Set} \to \mathsf{Set} \\ 0 \star Y = \emptyset \\ 1 \star Y = Y \\ (m+1) \star Y = Y + (m \star Y).\end{array}$$

A signature σ represents the following functor F_σ:

$$F_\sigma\, X = (\sigma\, 0) \star \mathsf{Unit} + (\sigma\, 1) \star X + (\sigma\, 2) \star X^2 + \cdots$$

or, more formally,

$$F_\sigma\, X = \sum_{n:\mathbb{N}} (\sigma\ n) \star X^n.$$

The corresponding signatures in the examples $\mathsf{trans\_code}$ and $\mathfrak{fib}$ are:

$$\sigma_{\mathsf{trans\_code}}\ n = 1 \quad \text{and} \quad \sigma_{\mathfrak{fib}}\ n = \begin{cases} 1 \text{ if n=0,1,2} \\ 0 \text{ otherwise.} \end{cases}$$

When we adopt this restrained class of functors, the definitions of the α and β components of a function look a bit different from what we have seen before. For example, for the $\mathfrak{fib}$ function, we have now that:

$$F_{\mathfrak{fib}}\, X = \textstyle\sum_{n:\mathbb{N}} (\sigma_{\mathfrak{fib}}\ n) \star X^n$$

$$\begin{array}{ll} \alpha_{\mathfrak{fib}}\ 0 = \langle 0, \mathtt{tt} \rangle & \beta_{\mathfrak{fib}}\ \langle 0, \mathtt{tt} \rangle = a \\ \alpha_{\mathfrak{fib}}\ 1 = \langle 1, g\ 0 \rangle & \beta_{\mathfrak{fib}}\ \langle 1, x \rangle = b + c \cdot x \\ \alpha_{\mathfrak{fib}}\ (m+2) = \langle 2, \langle g\ m,\ g\ (m+1) \rangle \rangle & \beta_{\mathfrak{fib}}\ \langle 2, \langle x, y \rangle \rangle = y + z. \end{array}$$

This gives us a version of $\mathfrak{fib}$ equivalent to the one given in Section 2. Recall that in_0 is simply the identity, so we do not write it in the equations above.

We now revise the rules given in the previous section to use signatures as parameters in place of functors. The introduction rule for partial recursive functions becomes:

$$\frac{\sigma\colon \mathbb{N} \to \mathbb{N} \quad \alpha\colon A \to F_\sigma\, A \quad \beta\colon F_\sigma\, B \to B}{\mathsf{rec}(\sigma, \alpha, \beta)\colon A \rightharpoonup B}.$$

Similarly, in the rest of the rules, we should simply add the parameter σ and replace F with F_σ.

We can now be more precise about the definition of the operator $\bigwedge_{F,P}$ when the functor F is given by a signature. We actually substitute the functor argument F with σ and define, for $P\colon X \to \mathsf{Prop}$:

$$\begin{array}{l} \bigwedge_{\sigma,P}\colon F_\sigma\, X \to \mathsf{Prop} \\ \bigwedge_{\sigma,P} \langle 0, \mathsf{in}_i\ \mathtt{tt} \rangle = \mathsf{True} \\ \bigwedge_{\sigma,P} \langle n+1, \mathsf{in}_i\, \langle x_0, \ldots, x_n \rangle \rangle = (P\, x_0) \wedge \cdots \wedge (P\, x_n) \end{array}$$

with $0 \leqslant i < \sigma\ n$ and True the trivially true proposition (isomorphic to Unit).

The introduction rule for the domain predicate becomes:

$$\frac{\sigma\colon \mathbb{N} \to \mathbb{N} \quad \alpha\colon A \to F_\sigma\, A \quad \beta\colon F_\sigma\, B \to B \quad a\colon A \quad h\colon \bigwedge_{\sigma, \mathsf{Dom}_{\mathsf{rec}(\sigma,\alpha,\beta)}} (\alpha\ a)}{\mathsf{dom}_{\sigma,\alpha,\beta}(a, h)\colon \mathsf{Dom}_{\mathsf{rec}(\sigma,\alpha,\beta)}\ a}.$$

Finally, we specify how to lift a function $f\colon A \rightharpoonup B$ by a functor specified by a signature:

$$\begin{array}{l} \overline{f}^{\sigma}\colon \forall t\colon F_\sigma\, A, \bigwedge_{\sigma,\mathsf{Dom}_f} t \to F_\sigma\, B \\ \overline{f}^{\sigma}\ \langle 0, \mathsf{in}_i\, \mathtt{tt} \rangle\ \mathtt{tt} = \langle 0, \mathsf{in}_i\, \mathtt{tt} \rangle \\ \overline{f}^{\sigma}\ \langle n+1, \mathsf{in}_i\, \langle x_0, \ldots, x_n \rangle \rangle\ \langle h_0, \ldots, h_n \rangle = \\ \qquad \langle n+1, \mathsf{in}_i\, \langle \mathsf{app}_f\ x_0\ h_0, \ldots, \mathsf{app}_f\ x_n\ h_n \rangle \rangle \end{array}$$

with $0 \leqslant i < \sigma\ n$ and $\mathtt{tt}$ the only constructor of True.

Observe that the definitions of app_f and $\overline{f}^{\sigma}$ are mutually dependent. The recursion is well-founded, since the recursive calls are on structurally smaller arguments. Systems like Coq provide support for mutual recursion.

We mentioned before that $\mathsf{rec}(\sigma, \alpha, \beta)$ can itself be alternatively defined as a partial recursive function by $\mathsf{rec}(\sigma, F_\sigma\, \alpha, F_\sigma\, \beta)$. This possibility relies on the equivalence of $\mathsf{Dom}_{\mathsf{rec}(\sigma,F_\sigma \alpha,F_\sigma \beta)}\ t$ with $\bigwedge_{\sigma,\mathsf{Dom}_{\mathsf{rec}}(\sigma,\alpha,\beta)}\ t$, which can be shown by induction on Dom. We leave this verification to the reader and stick to our original definition for the rest of the article.

5 Consistency of the Extended Type Theory

We show here that if we extend a consistent type theory with the new type of partial functions, we obtain a consistent new theory. We achieve this goal by translating the constructors for the new type of partial recursive functions into the base theory. The base theory must be expressive enough to provide the needed operators, specifically it must have Σ-types and inductive dependent families. Our reference system is Martin-Löf's type theory [18,21], but most versions of dependent type theory will work as well.

Let TT be a consistent and normalising type system and let PTT be its extension with the type of partial functions presented in Sections 3 and 4.

We first define an interpretation function $[\![\_]\!] \colon \mathsf{PTT} \to \mathsf{TT}$. The translation is defined by recursion on the structure of the terms and types of PTT. The constructors that are already present in TT are translated into themselves; so we only need to specify how to interpret the new constructors related to the type of partial functions.

We first define a type which we will use to interpret the type of partial functions:

$$A \rightharpoonup^{*} B := \sum_{\sigma:\mathbb{N}\to\mathbb{N}} (A \to F_\sigma\, A) \times (F_\sigma\, B \to B).$$

Let now $f \colon A \rightharpoonup^{*} B$. We use the abbreviations $\sigma_f = \pi_1\, f$, $\alpha_f = \pi_1\,(\pi_2\, f)$, and $\beta_f = \pi_2\,(\pi_2\, f)$, with π_1 and π_2 the first and second projection from a pair, respectively, and we represent elements of the above Σ-type as triples $\langle\sigma, \alpha, \beta\rangle$ rather than nested pairs $\langle\sigma, \langle\alpha, \beta\rangle\rangle$.

We then define an inductive predicate which we will use to interpret the domain of f:

$$\begin{aligned}&\mathsf{Dom}^*_f \colon A \to \mathsf{Prop}\\ &\quad \mathsf{dom}^*_f \colon \forall a \colon A, \textstyle\bigwedge_{\sigma_f,\mathsf{Dom}^*_f}(\alpha_f\ a) \to \mathsf{Dom}^*_f\ a.\end{aligned}$$

Finally, we define an application operator for f by recursion on Dom^*_f:

$$\begin{aligned}&\mathsf{app}^*_f \colon \forall a \colon A, \mathsf{Dom}^*_f\ a \to B\\ &\mathsf{app}^*_f\ a\ \mathsf{dom}^*_f(a, h) = \beta_f\,(\overline{f}^{*\,\sigma_f}\ (\alpha_f\ a)\ h).\end{aligned}$$

The definition of $\overline{f}^{*\,\sigma_f}$ is exactly that of $\overline{f}^{\sigma_f}$ given at the end of Section 4 but with calls to app^* in place of calls to app, and with Dom^* in place of Dom in the type of its last argument.

The translation proceeds now formally by structural induction on the types and terms of PTT. All the elements that are already present in TT are left unchanged by the translation. Therefore, the translation of type and term variables is the identity, $[\![X]\!] = X$ and $[\![x]\!] = x$; standard type constructors, like products and sums, and their constructors, that is, abstractions and pairs, are translated straightforwardly, hence $[\![\Pi x{:}\, A.B]\!] = \Pi x{:}\, [\![A]\!].[\![B]\!]$, $[\![\Sigma_{x:A}\, B]\!] = \Sigma_{x:[\![A]\!]}\, [\![B]\!]$, $[\![\lambda x.a]\!] = \lambda x.[\![a]\!]$ and $[\![\langle x, y\rangle]\!] = \langle [\![x]\!], [\![y]\!]\rangle$; and so on for all operations already present in TT, for example, application is translated as $[\![(d\ e)]\!] = ([\![d]\!]\ [\![e]\!])$ and projections as $[\![\pi_i\ p]\!] = \pi_i\ [\![p]\!]$ for $i = 1, 2$.

The new constructions for partial recursive functions are translated using the starred definitions presented above:

$$\begin{array}{l}
[\![A \rightharpoonup B]\!] = [\![A]\!] \rightharpoonup^* [\![B]\!] \\
[\![\mathsf{Dom}_f]\!] = \mathsf{Dom}^*_{[\![f]\!]} \\
[\![\mathsf{rec}\,(\sigma, \alpha, \beta)]\!] = \langle [\![\sigma]\!], [\![\alpha]\!], [\![\beta]\!]\rangle \\
[\![\mathsf{dom}_{\sigma,\alpha,\beta}(a, h)]\!] = \mathsf{dom}^*_{\langle [\![\sigma]\!],[\![\alpha]\!],[\![\beta]\!]\rangle}\,([\![a]\!], [\![h]\!]) \\
[\![\mathsf{app}_f\ a\ h]\!] = \mathsf{app}^*_{[\![f]\!]}\ [\![a]\!]\ [\![h]\!]
\end{array}$$

It is worth noticing that any term of TT is returned unchanged from the interpretation $[\![\,\_\,]\!]$ (this can be proved by simple structural induction on the terms of TT). In addition, the functor associated to a signature, the lifted universal quantifier, and the lifting operator on functions commute with the interpretation function.

Lemma 1. *For terms of the appropriate type, we have that:*

$$[\![F_\sigma]\!] = F_{[\![\sigma]\!]}, \quad [\![\bigwedge_{\sigma,P}]\!] = \bigwedge_{[\![\sigma]\!],[\![P]\!]}, \quad \text{and} \quad [\![\overline{f}^{\sigma}]\!] = \overline{[\![f]\!]}^{*\,[\![\sigma]\!]}.$$

Proof. For the first two statements, it is sufficient to check that the definitions of these operators use only constructions from the base type theory and none of the new ones defined for partial recursive functions.

The proof of the third statement follows from from the fact that $\overline{f}^{\sigma}$ and $\overline{[\![f]\!]}^{*\,[\![\sigma]\!]}$ are defined basically in the same way. The difference between the definitions is that first one uses app while the second uses app^*, but then we have that $[\![\mathsf{app}_f\ a\ h]\!] = \mathsf{app}^*_{[\![f]\!]}\ [\![a]\!]\ [\![h]\!]$. □

In addition, the interpretation is sound, as we show in the following lemma.

Lemma 2 (Soundness of $[\![\,\_\,]\!]$). *Let Γ be a context, t a term and T a type. If $\Gamma \vdash_{\mathsf{PTT}} t{:}\,T$ then $[\![\Gamma]\!] \vdash_{\mathsf{TT}} [\![t]\!]{:}\, [\![T]\!]$.*

Proof. By induction on the derivation of $\Gamma \vdash_{\mathsf{PTT}} t{:}\,T$. We consider here only the new rules of the extended system.

– Formation:
$$\frac{A{:}\,\mathsf{Set} \quad B{:}\,\mathsf{Set}}{A \rightharpoonup B{:}\,\mathsf{Set}}.$$

By inductive hypotheses (IH), both $[\![A]\!]$ and $[\![B]\!]$ are types. Hence, it is easy to see that $[\![A \rightharpoonup B]\!] = \Sigma_{\sigma:\mathbb{N}\to\mathbb{N}} ([\![A]\!] \to F_\sigma\, [\![A]\!]) \times (F_\sigma\, [\![B]\!] \to [\![B]\!])$ is a type.

– Introduction:

$$\frac{\sigma\colon \mathbb{N}\to\mathbb{N} \quad \alpha\colon A \to F_\sigma\, A \quad \beta\colon F_\sigma\, B \to B}{\mathsf{rec}(\sigma,\alpha,\beta)\colon A \rightharpoonup B}.$$

By IH, $[\![\sigma]\!]\colon \mathbb{N}\to\mathbb{N}$, $[\![\alpha]\!]\colon [\![A \to F_\sigma\, A]\!] = [\![A]\!] \to F_{[\![\sigma]\!]}\, [\![A]\!]$ and $[\![\beta]\!]\colon F_{[\![\sigma]\!]}\, [\![B]\!] \to [\![B]\!]$. It is easy to check now that $[\![\mathsf{rec}(\sigma,\alpha,\beta)]\!] = \langle [\![\sigma]\!], [\![\alpha]\!], [\![\beta]\!]\rangle$ has type $[\![A \rightharpoonup B]\!]$.

– Domain predicate:

$$\frac{f\colon A \rightharpoonup B}{\mathsf{Dom}_f\colon A \to \mathsf{Prop}} \qquad \frac{\sigma\colon\cdots \quad \alpha\colon\cdots \quad \beta\colon\cdots \quad a\colon A \quad h\colon \bigwedge_{\sigma,\mathsf{Dom}_{\mathsf{rec}(\sigma,\alpha,\beta)}}(\alpha\ a)}{\mathsf{dom}_{\sigma,\alpha,\beta}\,(a,h)\colon \mathsf{Dom}_{\mathsf{rec}(\sigma,\alpha,\beta)}\ a}.$$

By IH we have that $[\![f]\!]\colon [\![A \rightharpoonup B]\!]$, $[\![\sigma]\!]$,$[\![\alpha]\!]$ and $[\![\beta]\!]$ have the types shown above, $[\![a]\!]\colon [\![A]\!]$, and $[\![h]\!]\colon [\![\bigwedge_{\sigma,\mathsf{Dom}_{\mathsf{rec}(\sigma.\alpha,\beta)}}(\alpha\ a)]\!] = \bigwedge_{[\![\sigma]\!],\mathsf{Dom}^*_{[\![\mathsf{rec}(\sigma,\alpha,\beta)]\!]}}([\![\alpha]\!]\ [\![a]\!])$.
By definition $[\![\mathsf{Dom}_f]\!] = \mathsf{Dom}^*_{[\![f]\!]}$ has type $[\![A]\!] \to \mathsf{Prop}$.
We know that $[\![\mathsf{dom}_{\sigma,\alpha,\beta}\,(a,h)]\!] = \mathsf{dom}^*_{[\![\mathsf{rec}(\sigma,\alpha,\beta)]\!]}([\![a]\!],[\![h]\!])$, which has type $\mathsf{Dom}^*_{[\![\mathsf{rec}(\sigma,\alpha,\beta)]\!]}\ [\![a]\!]$ as required.

– Application:

$$\frac{f\colon A \rightharpoonup B \quad a\colon A \quad h\colon \mathsf{Dom}_f\ a}{\mathsf{app}_f\ a\ h\colon B}.$$

Let $[\![f]\!]\colon [\![A \rightharpoonup B]\!]$, $[\![a]\!]\colon [\![A]\!]$, and $[\![h]\!]\colon \mathsf{Dom}^*_{[\![f]\!]}\ [\![a]\!]$ by IH. Then, by definition, $[\![\mathsf{app}_f\ a\ h]\!] = \mathsf{app}^*_{[\![f]\!]}\ [\![a]\!]\ [\![h]\!]$, which has type $[\![B]\!]$. □

We can also check that the interpretation of the reduction rule is sound. Be aware of the overloading of the symbol $\rightsquigarrow$, used to denote reduction both in PTT and TT. It should be clear from the context in which theory we are performing the reduction. We denote the reflexive-transitive closure of $\rightsquigarrow$ in TT by $\rightsquigarrow^*$.

Lemma 3 (Preservation of reductions). *Let Γ, t and T such that $\Gamma \vdash_{\mathsf{PTT}} t\colon T$. If $t \rightsquigarrow t'$ in PTT for a certain term t', then $[\![t]\!] \rightsquigarrow^* [\![t']\!]$ in TT.*

Proof. We just need to check that the new reduction rule for application of a partial recursive function is interpreted correctly:

$$\mathsf{app}_{\mathsf{rec}(\sigma,\alpha,\beta)}\ a\ \mathsf{dom}_{\sigma,\alpha,\beta}(a,h) \rightsquigarrow \beta\,(\overline{\mathsf{rec}(\sigma,\alpha,\beta)}^{\sigma}\ (\alpha\ a)\ h).$$

The interpretation of the left-hand side of the rule can be reduced using the definition of app^*:

$$\begin{aligned}[\![\mathsf{app}_{\mathsf{rec}(\sigma,\alpha,\beta)}\ a\ \mathsf{dom}_{\sigma,\alpha,\beta}(a,h))]\!] &= \mathsf{app}^*_{[\![\mathsf{rec}(\sigma,\alpha,\beta)]\!]}\ [\![a]\!]\ \mathsf{dom}^*_{[\![\mathsf{rec}(\sigma,\alpha,\beta)]\!]}([\![a]\!],[\![h]\!]) \\ &\rightsquigarrow [\![\beta]\!]\,(\overline{[\![\mathsf{rec}(\sigma,\alpha,\beta)]\!]}^{*\,[\![\sigma]\!]}\ ([\![\alpha]\!]\ [\![a]\!])\ [\![h]\!]).\end{aligned}$$

The interpretation of the right-hand side gives us the following:

$$[\![\beta\,(\overline{\mathsf{rec}(\sigma,\alpha,\beta)}^{\sigma}\ (\alpha\ a)\ h)]\!] = [\![\beta]\!]\,([\![\overline{\mathsf{rec}(\sigma,\alpha,\beta)}^{\sigma}]\!]\ ([\![\alpha]\!]\ [\![a]\!])\ [\![h]\!]).$$

Now, the correctness of the translation of the reduction rule for application is a consequence of Lemma 1. □

Conservativity and consistency follow now straightforwardly.

Theorem 1 (Conservativity of PTT). *Let Γ and T be a context and a type, respectively, in TT, and let t be a term in PTT. If $\Gamma \vdash_{\mathsf{PTT}} t{:}\,T$ then $\Gamma \vdash_{\mathsf{TT}} [\![t]\!]{:}\,T$.*

Proof. By Lemma 2 we have that $[\![\Gamma]\!] \vdash_{\mathsf{TT}} [\![t]\!]{:}\,[\![T]\!]$. Every term of TT is returned unchanged by the interpretation $[\![\_]\!]$; hence, $\Gamma = [\![\Gamma]\!]$ and $T = [\![T]\!]$. □

Theorem 2 (Consistency of PTT). PTT *is consistent.*

Proof. Let us assume that $\vdash_{\mathsf{PTT}} t{:}\,\bot$ for some term t. Then, by Theorem 1, we have that $\vdash_{\mathsf{TT}} [\![t]\!]{:}\,\bot$, which is absurd since TT is consistent. □

The results of this section show that we can implement types of partial recursive functions satisfying our formal rules in standard type theory. Our formalisation is then to be used as a useful module to develop recursion theory in a system like Coq, as we exemplified in the Coq formalisation that accompanies this article. A direct implementation of the extended system could also have its advantages, avoiding the overhead of the interpretation and improving efficiency by direct computation of the reduction rule for partial functions.

6 Full Recursion

So far we have considered recursive definitions in which the argument was used only to generate recursive calls. This is a scheme of definition by iteration. A more general scheme, giving full recursion, allows the argument to be used directly and not just inside the recursive calls. The most simple example is the recursive equation in the definition of the factorial function, $\mathsf{fact}\ (n+1) = (n+1)\cdot(\mathsf{fact}\ n)$, where the argument $(n+1)$ is used as a factor of the multiplication, beside generating the recursive call to n. The factorial is a structurally recursive function and standard type theory can be used to define it with no problem. So we generalise the factorial function as we have generalised the Fibonacci function in order to consider a function that displays the same phenomenon, but requires at the same time general recursion. Let $a{:}\,\mathbb{N}$ and $g{:}\,\mathbb{N}\to\mathbb{N}$ be fixed parameters, then we define:

$$\begin{aligned}&\mathfrak{fact}{:}\,\mathbb{N}\to\mathbb{N}\\ &\mathfrak{fact}\ 0 = a\\ &\mathfrak{fact}\ (n+1) = (n+1)\cdot\mathfrak{fact}\ (g\ n).\end{aligned}$$

If $a = 1$ and g is the identity, we get the usual factorial function, but for some choices of g, for example the successor function, $\mathfrak{fact}$ will be a partial function.

This function does not fall into the scheme described in the previous sections because we have passed the input $(n+1)$ directly as an argument of the multiplication. That is, the β component uses not only the result of the recursive call, but also the input itself. The type of β must then be changed: it involves not only the functor F_σ, but also the type of the input. The introduction rule for partial recursive functions becomes now:

$$\frac{\sigma \colon \mathbb{N} \to \mathbb{N} \quad \alpha \colon A \to F_\sigma\, A \quad \beta \colon A \times F_\sigma\, B \to B}{\mathsf{rec}(\sigma, \alpha, \beta) \colon A \rightharpoonup B}.$$

Observe that the α component remains the same: α still determines just the structure of the recursive calls and says nothing about the extra parameter. The change in the type of β should of course be made in all the other rules as well.

The reduction rule must also be modified to feed the input as an extra argument to β:

$$\mathsf{app}_{\mathsf{rec}(\sigma,\alpha,\beta)}\ a\ \mathsf{dom}_{\sigma,\alpha,\beta}(a,h) \rightsquigarrow \beta\,\langle a, \overline{\mathsf{rec}(\sigma,\alpha,\beta)}^{\sigma}\ (\alpha\ a)\ h\rangle.$$

The interpretation and the proof of soundness can be easily modified to take this generalisation into account.

In the case of the $\mathfrak{fact}$ function, the components σ, α and β are as follows:

$$\sigma_{\mathfrak{fact}}\ n = \begin{cases} 1 \text{ for } n = 0,1 \\ 0 \text{ otherwise} \end{cases} \qquad \begin{array}{l} \alpha_{\mathfrak{fact}}\ 0 = \langle 0, \mathtt{tt}\rangle \\ \alpha_{\mathfrak{fact}}\ (n+1) = \langle 1, (g\, n)\rangle \end{array} \qquad \begin{array}{l} \beta_{\mathfrak{fact}}\ \langle x, \langle 0, \mathtt{tt}\rangle\rangle = a \\ \beta_{\mathfrak{fact}}\ \langle x, \langle 1, y\rangle\rangle = x \cdot y. \end{array}$$

7 Conclusions

The work presented here is a contribution to the formalisation of general recursion in intensional type theories. For every pair of types A and B, we define a type of partial functions $A \rightharpoonup B$. Its introduction rule allows us to define a function by prescribing the structure of recursive calls and the operations that produce the output from the recursive results. Every function is coupled with a domain predicate inductively defined to be true whenever it holds for the recursive arguments. Application is allowed only for elements in the domain and the reduction rule performs recursion over the proof of the domain predicate.

The fact that all recursive functions between two given types A and B can be collected in a single type facilitates the definition of higher-order programs. Indeed, the source type A can itself be a type of partial functions, $A = X \rightharpoonup Y$. However, the mechanism does not immediately allow the inheritance of partiality from a higher-order partial argument. For example, the formalisation of the *map* function on lists (which is itself structurally recursive) raises the problem of how the partiality of the mapped argument function is reflected on the list argument. The method explained here does not directly give an answer to this problem.

Another problematic issue is that nested recursive functions are not allowed: the α component must directly identify the recursive arguments, without recursively calling the function that we are defining.

Partiality may come not only from infinite recursion but also from a sequence of recursive equations whose patterns are not exhaustive. We did not provide a direct method to deal with the fact that the α component must be total. However, as we already mentioned before, there are easy fixes for this since we can add equations to make the patterns exhaustive by doing one of three things: assign a default value to the new patterns, make the function loop on them, or add an *undefined* element to the target type B, that is, using $B + \mathsf{Unit}$ in place of B.

We proved the conservativity of the new constructors with respect to a base type theory. Thus the extended system is sound whenever the base theory is.

We maintain that this new type could be of use in the formalisation of general recursive algorithms and in the proof of their correctness. It can be readily implemented in type-theory based proof assistants like Coq.

References

1. Audebaud, P.: Partial objects in the calculus of constructions. In: Kahn, G. (ed.) Proceedings of the Sixth Annual IEEE Symp. on Logic in Computer Science, LICS 1991, July 1991, pp. 86–95. IEEE Computer Society Press, Los Alamitos (1991)
2. Bove, A.: General recursion in type theory. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 39–58. Springer, Heidelberg (2003)
3. 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)
4. Bove, A., Capretta, V.: Modelling general recursion in type theory. Mathematical Structures in Computer Science 15(4), 671–708 (2005)
5. 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)
6. Bove, A., Capretta, V.: Computation by prophecy. In: Della Rocca, S.R. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 70–83. Springer, Heidelberg (2007)
7. Capretta, V.: Universal algebra in type theory. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 131–148. Springer, Heidelberg (1999)
8. Capretta, V.: Recursive families of inductive types. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 73–89. Springer, Heidelberg (2000)
9. Capretta, V.: General recursion via coinductive types. Logical Methods in Computer Science 1(2), 1–18 (2005)
10. Capretta, V., Uustalu, T., Vene, V.: Recursive coalgebras from comonads. In: Proceedings of the Workshop on Coalgebraic Methods in Computer Science (CMCS 2004). Electronic Notes in Theoretical Computer Science, vol. 106, pp. 43–61 (2004)
11. Capretta, V., Uustalu, T., Vene, V.: Recursive coalgebras from comonads. Information and Computation 204(4), 437–468 (2006)
12. Constable, R.L.: Constructive mathematics as a programming logic I: Some principles of theory. Annals of Mathematics, vol. 24. Elsevier Science Publishers, North-Holland, Amsterdam (1985)
13. Constable, R.L., Mendler, N.P.: Recursive definitions in type theory. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 61–78. Springer, Heidelberg (1985)

14. Constable, R.L., Smith, S.F.: Partial objects in constructive type theory. In: Logic in Computer Science, Ithaca, New York, pp. 183–193. IEEE, Los Alamitos (1987)
15. Coquand, T., Huet, G.: The Calculus of Constructions. Information and Computation 76, 95–120 (1988)
16. Gödel, K.: Über formal unentscheidbare sätze der Principia Mathematica und verwandter systeme. Monatshefte für Mathematik und Physik 38, 173–198 (1931)
17. Jones, S.P.: Haskell 98 Language and Libraries: The Revised Report, April 2003. Cambridge University Press, Cambridge (2003)
18. Martin-Löf, P.: Intuitionistic Type Theory. Bibliopolis, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua (June 1980)
19. Megacz, A.: A coinductive monad for Prop-bounded recursion. In: Stump, A., Xi, H. (eds.) PLPV 2007: Proceedings of the 2007 workshop on Programming languages meets program verification, pp. 11–20. ACM Press, New York (2007)
20. Meijer, E., Fokkinga, M.M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 124–144. Springer, Heidelberg (1991)
21. Nordström, B., Petersson, K., Smith, J.M.: Programming in Martin-Löf's Type Theory. An Introduction. International Series of Monographs on Computer Scence, vol. 7. Oxford University Press, Oxford (1990)
22. Setzer, A.: Partial recursive functions in Martin-Löf Type Theory. In: Beckmann, A., Berger, U., Löwe, B., Tucker, J.V. (eds.) CiE 2006. LNCS, vol. 3988, p. 505. Springer, Heidelberg (2006)
23. Setzer, A.: A data type of partial recursive functions in Martin-Löf Type Theory, `http://www.cs.swan.ac.uk/ csetzer/articles/setzerDataTypePar RecPostProceedings.ps`
24. The Coq Development Team. LogiCal Project. The Coq Proof Assistant. Reference Manual. Version 8. INRIA (2004), `http://pauillac.inria.fr/coq/coq-eng.html`