Partial Recursive Functions in Martin-Löf Type Theory

Anton Setzer*

Dept. of Computing Science, University of Wales Swansea,
Singleton Park, Swansea SA2 8PP, UK.
Tel.: +44 1792 513368, Fax: +44 1792 295651,
a.g.setzer@swan.ac.uk,
http://www.cs.swan.ac.uk/~csetzer/

Abstract. In this article we revisit the approach by Bove and Capretta for formulating partial recursive functions in Martin-Löf Type Theory by indexed inductive-recursive definitions. We will show that all inductive-recursive definitions used there can be replaced by inductive definitions. However, this encoding results in an additional technical overhead. In order to obtain directly executable partial recursive functions, we introduce restrictions on the indexed inductive-recursive definitions used. Then we introduce a data type of partial recursive functions. This allows to define higher order partial recursive functions like the map functional, which depend on other partial recursive functions. This data type will be based on the closed formalisation of indexed inductive-recursive definitions introduced by Dybjer and the author. All elements of this data type will represent partial recursive functions, and the set of partial recursive functions will be closed under the standard operations for forming partial recursive functions, and under the total functions.
Keywords: Martin-Löf type theory, computability theory, recursion theory, Kleene index, Kleene brackets, partial recursive functions, inductive-recursive definitions, indexed induction-recursion.

1 Introduction

A problem when developing computability theory in Martin-Löf type theory is that the function types only contain total functions, therefore partial recursive functions are not first class objects. One approach to overcome this problem has been taken by Bove and Capretta (e.g. [BC05a, BC05b]), who have shown how to represent partial recursive functions by indexed inductive-recursive definitions (IIRD), and in this article we will investigate their approach. In order to illustrate it, we make use of a toy example. We choose a notation which is closer to that used in computability theory.

Assume the partial recursive function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ defined by

$$f(0) :\simeq 0 \ , \qquad f(n+1) :\simeq f(f(n)) \ .$$

* Supported by EPSRC grant GR/S30450/01.

This function is constantly zero, but we want to represent it directly in Martin-Löf type theory, so that we can prove for instance, that it is in fact constantly zero. In order to do this, Bove and Capretta introduce

$$f(\cdot)\downarrow : \mathbb{N} \to \mathrm{Set} \ , \quad \mathrm{eval}_f : (n : \mathbb{N}, p : f(n)\downarrow) \to \mathbb{N} \ .$$

Here $f(n)\downarrow$ expresses that $f(n)$ is defined and $\mathrm{eval}_f(n,p)$ computes, depending on $n : \mathbb{N}$ and a proof $p : f(n)\downarrow$, the value $f(n)$.

In the literature, $f(\cdot)\downarrow$ is often referred to as the accessibility predicate for f. If we define for arguments a, b of f that $a \prec b$ if and only if the call of $f(b)$ recursively calls $f(a)$, then $f(a)\downarrow$ if and only if a is in the accessible part of $\prec$. The approach by Bove/Capretta can be seen as a general method of determining the accessible part of $\prec$ for a large class of recursively defined functions.

If we take the definition of f as it stands, we see that the definitions of $f(\cdot)\downarrow$ and eval_f refer to each other. $f(\cdot)\downarrow$ has two constructors $\mathrm{defined}^0_f$, $\mathrm{defined}^{\mathrm{S}}_f$ corresponding to the two rewrite rules, and we obtain the following introduction and equality rules:

$$\begin{array}{l} \mathrm{defined}^0_f : f(0)\downarrow \ , \qquad \mathrm{eval}_f(0, \mathrm{defined}^0_f) = 0 \ , \\ \mathrm{defined}^{\mathrm{S}}_f : (n : \mathbb{N}, p : f(n)\downarrow, q : f(\mathrm{eval}_f(n,p))\downarrow) \to f(n+1)\downarrow \ , \\ \mathrm{eval}_f(n+1, \mathrm{defined}^{\mathrm{S}}_f(n,p,q)) = \mathrm{eval}_f(\mathrm{eval}_f(n,p), q) \ . \end{array}$$

The constructor $\mathrm{defined}^{\mathrm{S}}_f$ has arguments $n : \mathbb{N}$, $p : f(n)\downarrow$, and if $f(n) \simeq m$, a proof $q : f(m)\downarrow$. Then $p' := \mathrm{defined}^{\mathrm{S}}_f(n,p,q)$ proves $f(n+1)\downarrow$ and we have $\mathrm{eval}_f(n+1, p') = \mathrm{eval}_f(m,q)$. We observe that $\mathrm{defined}^{\mathrm{S}}_f$ refers to $\mathrm{eval}_f(n,p)$, so we have to define simultaneously $f(\cdot)\downarrow$ inductively, while defining eval_f recursively. This is an instance of an IIRD, as introduced by Dybjer [Dyb00, Dyb94]. We will see below that such kind of IIRD can be reduced to inductive definitions.

Bove and Capretta face the problem that they cannot define a data type of partial recursive functions (unless using impredicative type theory) and therefore cannot deal with partial recursive functions depending on other partial recursive functions as an argument. A simple example would be to define depending on a partial recursive function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ (e.g. f as above)

$$g : \mathrm{List}(\mathbb{N}) \rightharpoonup \mathrm{List}(\mathbb{N}) \ , \qquad g(l) :\simeq \mathrm{map}(f, l) \ .$$

Here $\mathrm{map}(f, [n_0, \ldots, n_k]) :\simeq [f(n_0), \ldots, f(n_k)]$. In order to define the above directly, we need to define map, depending on an arbitrary partial recursive function f. More complex examples of this kind are discussed in [BC05a].

In this article we will show how to overcome this restriction by introducing a data type of partial recursive functions. This will be based on the closed formulation of IIRD, as developed by P. Dybjer and the author. In order to have that all functions represented by an IIRD correspond directly to a partial recursive function, without using search functions, we will impose restrictions on the set of IIRD used. The data type given in this article will define exactly those restricted IIRD. We will then show that the functions given by those indices are all partial

recursive, and that they are closed under the standard constructions for defining partial recursive functions, and under the total functions.

Future work. With the above research we will be able to define functions referring to indices of other partial recursive functions. However, we will not yet be able to deal with mutually recursive functions, in which one function refers to another as a whole. In order to deal with this, we will in a follow-up paper introduce partial recursive functions with function arguments represented as oracles. We will then obtain a recursion theorem stating that recursion equations defined using this principle can always be solved.

2 Inductive-Recursive Definitions

Before formulating the data type of partial recursive definitions as a data type of IIRD let us sketch briefly Dybjer's notion of IIRD.

Dybjer introduced first the simpler notion of inductive-recursive definitions (IRD). In IRD one defines a set $\mathrm{U} : \mathrm{Set}$ together with a function $\mathrm{T} : \mathrm{U} \to D$ where $D : \mathrm{Type}$. Inductive-recursive definitions emerged first as universes, i.e. $D = \mathrm{Set}$. Universes are sets of sets, which are given by a set U of codes for sets, and a decoding function $\mathrm{T} : \mathrm{U} \to \mathrm{Set}$, which determines for every code $a : \mathrm{U}$ the set $\mathrm{T}(a) : \mathrm{Set}$ it denotes.

Strictly positive inductive definitions are given by a set U together with constructors $C : A_1 \to \cdots A_n \to \mathrm{U}$, where A_i can refer to U strictly positively: (1) Either A_i is a set, which were defined before one started to introduce U. Arguments of C referring to such sets are called *non-inductive arguments*. (2) Or A_i is of the form $(B_1 \to \cdots \to B_m \to \mathrm{U})$ for some sets B_i defined before U was introduced. Arguments referring to such sets are called *inductive arguments*. In dependent type theory, we can extend inductive definitions by allowing A_i to refer to previous arguments, i.e. we get $\mathrm{C} : (a_1 : A_1, \ldots, a_n : A_n) \to \mathrm{U}$, and A_i might depend on a_j for $j < i$. However, closer examination reveals that only dependencies on non-inductive arguments are possible: When introducing the constructor C, U has not been defined yet, so we are not able to define any sets depending on it.

In an IRD of $\mathrm{U} : \mathrm{Set}$ and $\mathrm{T} : \mathrm{U} \to D$, A_i might refer to previous inductive arguments via T: if $a_j : A_j = (B_1 \to \cdots \to B_k \to \mathrm{U})$, and $j < i$, then A_i might make use of $\mathrm{T}(a_j(b_1, \ldots, b_k))$. An example is the constructor $\widehat{\Pi}$ expressing the closure of a universe under the dependent function type (which is often written as $\Pi(A, B)$): $\widehat{\Pi}$ has type $(a : \mathrm{U}, b : \mathrm{T}(a) \to \mathrm{U}) \to \mathrm{U}$ and the type of the second argument b of $\widehat{\Pi}$ depends on $\mathrm{T}(a)$.

The intuition why this is a good predicative definition is that one defines the elements of U inductively. Whenever one introduces a new element of U, one computes recursively T applied to it. Therefore, when referring to a previous inductive argument, we can make use of T applied to it.

When applying this principle, one notices that one often needs to define several universes $(\mathrm{U}_i, \mathrm{T}_i)$ simultaneously. Indexed inductive-recursive definitions IIRD extend the principle of IRD so that it allows to define $\mathrm{U} : I \to \mathrm{Set}$ and

$\mathrm{T} : (i : I, u : \mathrm{U}(i)) \to D[i]$ simultaneously for all $i : I$. Here I : Set, and $i : I \Rightarrow D[i]$: Type. The constructors of $\mathrm{U}(i)$ might refer to $\mathrm{U}(j)$ for any j in a strictly positive way, and make use of T applied to previous inductive arguments.

Reduction to inductive definitions. When formalising the representation of partial recursive functions as IIRD in general, one wants to represent partial recursive functions $f : (x : A) \rightharpoonup B(x)$ for arbitrary A : Set, $B : A \to$ Set. Such functions will be translated into IIRD with index set A and $D[x] := B(x)$. Note that $D[x]$: Set. In [DS06] Dybjer and the author have shown that IIRD with $D[x]$: Set can always be reduced to indexed inductive definitions. We sketch the idea briefly by taking the example from the introduction. Remember that $\mathrm{defined}^{\mathrm{S}}_f$ had type

$$\mathrm{defined}^{\mathrm{S}}_f : (n : \mathbb{N}, p : f(n)\!\downarrow, q : f(\mathrm{eval}_f(n,p))\!\downarrow) \to f(n+1)\!\downarrow \ .$$

In order to avoid the use of $\mathrm{eval}_f(n,p)$ in the type of $\mathrm{defined}^{\mathrm{S}}_f$, one introduces first an inductive definition of a set $f(n)\!\downarrow^{\mathrm{aux}} : \mathbb{N} \to$ Set, with constructors

$$\mathrm{defined}^{0,\mathrm{aux}}_f : f(0)\!\downarrow^{\mathrm{aux}} \ ,$$
$$\mathrm{defined}^{\mathrm{S},\mathrm{aux}}_f : (n : \mathbb{N}, p : f(n)\!\downarrow^{\mathrm{aux}}, m : \mathbb{N}, q : f(m)\!\downarrow^{\mathrm{aux}}) \to f(n+1)\!\downarrow^{\mathrm{aux}} \ .$$

Then we compute recursively $\mathrm{eval}^{\mathrm{aux}}_f : (n : \mathbb{N}, p : f(n)\!\downarrow^{\mathrm{aux}}) \to \mathbb{N}$ (this definition is now separated from the inductive definition of $f(\cdot)\downarrow$) by

$$\mathrm{eval}^{\mathrm{aux}}_f(0, \mathrm{defined}^{0,\mathrm{aux}}_f) := 0 \ ,$$
$$\mathrm{eval}^{\mathrm{aux}}_f(n+1, \mathrm{defined}^{\mathrm{S},\mathrm{aux}}_f(n,p,m,q)) := \mathrm{eval}^{\mathrm{aux}}_f(m,q) \ .$$

$p : f(n)\!\downarrow^{\mathrm{aux}}$ proves that $f(n)$ is defined, provided that, whenever we made use of $\mathrm{defined}^{\mathrm{S},\mathrm{aux}}_f(n,p,m,q)$, we had $m = \mathrm{eval}^{\mathrm{aux}}_f(n,p)$. We introduce a correctness predicate expressing this:

$$\mathrm{Corr}_f\colon (n : \mathbb{N}, p : f(n)\!\downarrow^{\mathrm{aux}}) \to \mathrm{Set} \ , \qquad \mathrm{Corr}_f(0, \mathrm{defined}^{0,\mathrm{aux}}_f) := \mathrm{True} \ ,$$
$$\mathrm{Corr}_f(n+1, \mathrm{defined}^{\mathrm{S},\mathrm{aux}}_f(n,p,m,q) :=$$
$$\mathrm{Corr}_f(n,p) \wedge \mathrm{Corr}_f(m,q) \wedge m =_{\mathbb{N}} \mathrm{eval}^{\mathrm{aux}}_f(n,p) \ .$$

One can now simulate $f(\cdot)\!\downarrow$ by

$$f(\cdot)\!\downarrow' : \mathbb{N} \to \mathrm{Set} \ , f(n)\!\downarrow' := (p : f(n)\!\downarrow^{\mathrm{aux}}) \times \mathrm{Corr}_f(n,p) \ ,$$

and simulate eval_f by

$$\mathrm{eval}'_f : (n : \mathbb{N}, p : f(n)\!\downarrow') \to \mathbb{N} \ , \mathrm{eval}'_f \ \ (n, \langle p, q\rangle) := \mathrm{eval}^{\mathrm{aux}}_f(n,p) \ .$$

In [DS06] this reduction has been carried out in detail and there we were able to show that indeed all IIRD with target type of T being a set can be simulated by indexed inductive definitions.

Note that this reduction adds an additional overhead. So we assume when verifying the correctness of partial recursive functions on the machine one probably prefers to use the original IIRD.

Restrictions to the class of IIRD**.** Bove and Capretta have shown that the set of partial recursive functions definable this way is Turing-complete – it contains all partial recursive functions on $\mathbb{N}$ – and that a large class of recursion schemes can be represented this way. However, not all IIRD having the above types correspond to directly executable partial recursive functions.

1. We need to replace general IIRD by restricted IIRD. The concepts of general and restricted IIRD were investigated in [DS01, DS06]. In general IIRD, one introduces constructors for the inductively defined set U and then determines for each constructor C depending on its arguments $\vec{a}$ the i s.t. $C(\vec{a}) : \mathrm{U}(i)$. In the example used in the introduction we used in fact such kind of general IIRD. We have a constructor $\mathrm{defined}_f^{\mathrm{S}}$, and we state that $\mathrm{defined}_f^{\mathrm{S}}(n,p,q) : f(n+1)\downarrow$, so the index $n+1$ depends on the arguments n, p, q. The problem with this is that it allows to define multivalued functions: Nothing prevents us from adding a second constructor $\mathrm{defined}_f^{\mathrm{S},'}(n) : f(n+1)\downarrow$, s.t. $\mathrm{eval}_f(n+1, \mathrm{defined}_f^{\mathrm{S},'}(n))$ returns a different value, e.g. 5. This corresponds to adding contradictory rewrite rules, such as $f(n+1) \longrightarrow 5$.
   Furthermore this principle does not mean that the functions are directly executable, unless one has a proof of $f(n)\downarrow$ (in which case eval_f allows of course to compute the value of $f(n)$). In order to evaluate $f(n)$, one has to guess the arguments of the constructor in such a way that we obtain an element of $f(n)\downarrow$, which requires in general a search process. $f(n)$ is still partially recursive (the $f(n)$ are always computable since one can always search for a proof $p : f(n)\downarrow$, and then compute $\mathrm{eval}_f(n,p)$), but we do not regard the search for arguments as a means of directly executing a function.
   In order to obtain directly executable partial recursive functions we need to determine for each index its constructor. This corresponds to restricted IIRD as introduced in [DS06]. If one defines in restricted IIRD $\mathrm{U} : I \to \mathrm{Set}$, one needs to determine, depending on i the set of constructors having result type $\mathrm{U}(i)$. The initial example can be represented as a restricted IIRD by defining it as follows:

$$\begin{aligned} f(n)\downarrow := \text{case } n \text{ of } 0 \quad &\to \text{data } \mathrm{C}_f^0 \\ \mathrm{S}(n') &\to \text{data } \mathrm{C}_f^{\mathrm{S}}(p : f(n')\downarrow,\ \ q : f(\mathrm{eval}_f(n',p))\downarrow) \end{aligned}$$

   So $f(0)\downarrow$ has constructor C_f^0, and $f(\mathrm{S}(n'))\downarrow$ has constructor $\mathrm{C}_f^{\mathrm{S}}$ with arguments $p : f(n')\downarrow$ and $q : f(\mathrm{eval}_f(n',p))\downarrow$.
2. In order to avoid multivalued functions, for each argument $a : A$ there should be at most one constructor of type $f(a)\downarrow$. One can easily achieve that there is always exactly one constructor – if there is none, one can always add the constructor $C : (p : f(a)\downarrow) \to f(a)\downarrow$ corresponding to black hole recursion (i.e. rewrite rule $f(a) \longrightarrow f(a)$).
3. We disallow non-inductive arguments. Non-inductive arguments, except for the empty set and the one-element set might result in multivalued functions (different choices for one argument of the constructor might yield different

proofs of $f(n)\downarrow$ and therefore different values of $\mathrm{eval}_f(n,p)$). Another problem with non-trivial non-inductive arguments is that when evaluating the partial recursive function, one needs to search for instances of these non-inductive argument. Therefore one does not obtain directly executable functions. We could allow non-indexed arguments indexed over the one-element set 1 and the empty set $\emptyset$. But arguments indexed over 1 can be ignored, and arguments indexed over $\emptyset$ have the effect that $f(a)\uparrow$, which can alternatively be obtained by using black-hole recursion as above.

4. Inductive arguments should be single ones. In general IIRD of a set U, the constructor might have an inductive argument of the form $(x : A) \to \mathrm{U}(i(x))$. In our setting such an inductive argument would be of the form $(x : A) \to f(i(x))\downarrow$, which expresses $f(i(x))$ is defined for all $x : A$. Such an argument requires the evaluation of f for possibly infinitely many values $i(x)$ $(x : A)$, which is non-computable. One can search for a proof of $(x : A) \to f(i(x))$ and use this search process as a means of evaluating f. However, such a search will miss the situation where $(x : A) \to f(i(x))$ is true (even constructively), but unprovable in the type theory in question. Furthermore, we do not regard such a search process as a means of directly executing a function.

   So in order to obtain directly executable functions, we need to restrict inductive arguments to single valued ones, i.e. in the above situation to arguments of the form $p : f(i)\downarrow$.

3 A Data Type of Partial Recursive Functions

In [BC05a] it was pointed out that one of the limitations of their approach is that they cannot define partial recursive functions referring to other partial recursive functions as a whole. We have given a toy example in the introduction (the function $g : \mathrm{List}(\mathbb{N}) \rightharpoonup \mathrm{List}(\mathbb{N})$). In computability theory one overcomes this problem by introducing Kleene-indices for partial recursive functions. Then one can define for instance map by having two natural numbers as arguments, one which is a Kleene-index for a partial recursive function, and the second one a code for a list. In order to do the same using the approach by Bove/Capretta, we will introduce a data type of codes for the IIRD we were referring to above. This data type is a subtype of the data type of IIRD introduced in [DS06] (see as well [DS01, DS03, DS99]). This shows the consistency of the new rules introduced in this article.

Assume $A : \mathrm{Set}$, $B : A \to \mathrm{Set}$ fixed. Unless explicitly needed, we suppress in the following dependencies on A, B. We regard a partial recursive function $f : (a : A) \rightharpoonup B(a)$ as being given by an IIRD, and introduce the data type Rec of codes for those IIRD, which correspond according to the previous section to partial recursive functions. So the set Rec will as well be the set of codes for partial recursive functions. We have formation rule:

$$\mathrm{Rec} : \mathrm{Set} \ .$$

The set defined inductively by a code $e : \mathrm{Rec}$ is given as

$$\mathrm{f}_e(\cdot)\!\downarrow : A \to \mathrm{Set}$$

If we understand each IIRD as defining a partial recursive function, $\mathrm{f}_e(a)\!\downarrow$ means that the function with index e is defined for argument a. The function defined recursively is

$$\mathrm{eval}_e : (a : A, p : \mathrm{f}_e(a)\!\downarrow) \to B(a) \ ,$$

which, if the IIRD is interpreted as the definition of a partial recursive function, computes the result of this function.

Rec is a restricted IIRD, which means that for each $a : A$ we can determine the type of arguments for the constructor with result $f_e(a)\!\downarrow$. Let Rec'_a be the type of codes for possible arguments for the constructor of an IIRD e with result $\mathrm{f}_e(a)\!\downarrow$. Then an element of Rec is given by an element of Rec'_a for each $a : A$. So we have the following formation and equality rule:

$$\mathrm{Rec}' : A \to \mathrm{Set} \ , \qquad \mathrm{Rec} = (a : A) \to \mathrm{Rec}'_a \ .$$

The type of the arguments of the constructor of $\mathrm{f}_e(a)\!\downarrow$ and the result of $\mathrm{eval}_e(a, p)$ for the constructed element p will depend on $\mathrm{f}_e(\cdot)\!\downarrow$ and eval_e. Since, when introducing Rec'_a, $\mathrm{f}_e(\cdot)\!\downarrow$ and eval_e are not available, we define more generally, depending on $a : A$, $e : \mathrm{Rec}'_a$, for general X and Y, having the types of $\mathrm{f}_e(\cdot)\!\downarrow$, eval_e respectively, the following operations:

$$\begin{aligned}\mathrm{Arg}_{a,e} &: (X : A \to \mathrm{Set}, Y : (a' : A, x : X(a')) \to B(a')) \to \mathrm{Set}\\ \mathrm{Eval}_{a,e} &: (X : A \to \mathrm{Set}, Y : (a' : A, x : X(a')) \to B(a'), \mathrm{Arg}_{a,e}(X, Y)) \to B(a)\end{aligned}$$

$\mathrm{Arg}_{a,e(a)}(\mathrm{f}_e(\cdot)\!\downarrow, \mathrm{eval}_e)$ will be the type of the arguments of the constructor of $\mathrm{f}_e(a)\!\downarrow$. If using arguments p we have constructed $q : \mathrm{f}_e(a)\!\downarrow$, then $\mathrm{eval}_e(a, q) = \mathrm{Eval}_{a,e(a)}(\mathrm{f}_e(\cdot)\!\downarrow, \mathrm{eval}_e, p)$. If we call the constructor for $\mathrm{f}_e(a)\!\downarrow$ $\mathrm{tot}_{e,a}$, then the introduction and equality rules for $\mathrm{f}_e(\cdot)\!\downarrow$ and eval_e are as follows:

$$\begin{aligned}\mathrm{tot}_{e,a} &: \mathrm{Arg}_{a,e(a)}(\mathrm{f}_e(\cdot)\!\downarrow, \mathrm{eval}_e) \to \mathrm{f}_e(a)\!\downarrow \ ,\\ \mathrm{eval}_e(a, \mathrm{tot}_{e,a}(p)) &= \mathrm{Eval}_{a,e(a)}(\mathrm{f}_e(\cdot)\!\downarrow, \mathrm{eval}_e, p) \ .\end{aligned}$$

We define additionally outside type theory for closed $e : \mathrm{Rec}$ the partial recursive function $\{e\} : (a : A) \rightharpoonup B(a)$. $\{e\}$ will be defined in such a way that we can prove outside type theory $\{e\}(a)\!\downarrow \Leftrightarrow \exists p.p : \mathrm{f}_e(a)\!\downarrow$ and that if $p : \mathrm{f}_e(a)\!\downarrow$, then $\{e\}(a) \simeq \mathrm{eval}_e(a, p)$.

In order to define $\{e\}(a)$, we define recursively (outside type theory) an auxiliary partial recursive function

$$\mathrm{compute}^{\mathrm{aux}} : (e : \mathrm{Rec}, a : A, e' : \mathrm{Rec}'_a) \rightharpoonup B(a)$$

$\mathrm{compute}^{\mathrm{aux}}(e, a, e')$ roughly speaking computes the subcomputation of $\{e\}(a)$, where we consider the subcode e' of $e(a)$. However, in the definition we do not assume e' to be a subcode of $e(a)$. Then we define for $e : \mathrm{Rec}$

$$\{e\}(a) \simeq \mathrm{compute}^{\mathrm{aux}}(e, a, e(a))$$

Rec'_a has 2 constructors:

1. Initial (constant) case: the constructor for $\mathrm{f}_e(a)\downarrow$ has no arguments (or more precisely the trivial argument $x : \{*\}$ for the one-element set $\{*\}$). $\mathrm{eval}_e(a,p)$ returns, independently of p, a fixed element $b : B(a)$:

$$\mathrm{const}_a : B(a) \to \mathrm{Rec}'_a \qquad \mathrm{Arg}_{a,\mathrm{const}_a(b)}(X,Y) = \{*\}$$
$$\mathrm{Eval}_{a,\mathrm{const}_a(b)}(X,Y,*) = b$$
$$\mathrm{compute}^{\mathrm{aux}}(e,a,\mathrm{const}_a(b)) \simeq b$$

2. A single inductive argument. The constructor of $\mathrm{f}_e(a)\downarrow$ has as an inductive argument $p : \mathrm{f}_e(a')\downarrow$, and depending on $m : \mathrm{eval}_e(a',p)$ later arguments. As a partial recursive function this means that we make a recursive call to $\mathrm{f}_e(a')\downarrow$. Depending on the result m, we choose further steps. So the constructor rec_a of Rec'_a needs to have as arguments a' and a function $e' : B(a') \to \mathrm{Rec}'_a$, which determines depending on the result $b : B(a')$ of $\mathrm{eval}_e(a',p)$ the later arguments of the constructor. We obtain

$$\mathrm{rec}_a : (a' : A, e' : B(a') \to \mathrm{Rec}'_a) \to \mathrm{Rec}'_a$$
$$\mathrm{Arg}_{a,\mathrm{rec}_a(a',e')}(X,Y) = (x : X(a')) \times \mathrm{Arg}_{a,e'(Y(a',x))}(X,Y)$$
$$\mathrm{Eval}_{a,\mathrm{rec}_a(a',e')}(X,Y,\langle x,y\rangle) = \mathrm{Eval}_{a,e'(Y(a',x))}(X,Y,y)$$
$$\mathrm{compute}^{\mathrm{aux}}(e,a,\mathrm{rec}_a(a',g)) \simeq \begin{cases} \mathrm{compute}^{\mathrm{aux}}(e,a,g(b)), \\ \qquad\qquad \text{if } \mathrm{compute}^{\mathrm{aux}}(e,a',e(a)) \simeq b, \\ \bot, \quad \text{if } \mathrm{compute}^{\mathrm{aux}}(e,a',e(a)) \uparrow. \end{cases}$$

We usually omit the parameter a in const_a, rec_a. We observe that Rec'_a is an inductively defined set: it is like a W-type with branching degrees $(B(a))_{a:A}$, but with additional leaves $\mathrm{const}(b)$. Arg and Eval are then defined by recursion on Rec'_a. $\mathrm{f}_e(\cdot)\downarrow$ and eval_e are given by an IIRD which is determined by e.

Reference to other partial recursive functions. If one wants to show the closure of the resulting set of partial recursive functions under operations like composition, one sees that one needs the possibility to refer to other partial recursive functions, which is not available in the above calculus. In the context of dependent type theory, allowing this will cause one problem: we want to refer in the definition of partial recursive functions to other partial recursive functions g of any type $(c : C) \rightharpoonup D(c)$ where $\langle C, D\rangle : \mathrm{Fam(Set)}$. Here $\mathrm{Fam(Set)} := (X : \mathrm{Set}) \times (X \to \mathrm{Set}) : \mathrm{Type}$ is the type of families of sets. If we want to allow reference to arbitrary such functions, we will end up with $\mathrm{Rec}_{A,B} : \mathrm{Type}$ instead of $\mathrm{Rec}_{A,B} : \mathrm{Set}$. (We will no longer suppress the arguments A, B of Rec.) This causes problems when defining partial recursive functions having elements of $\mathrm{Rec}_{A,B}$ as arguments. However, one can usually restrict the domain and codomain of partial recursive functions used to elements of a universe, and therefore obtain $\mathrm{Rec}_{A,B}$ to be a set. If one is for instance interested in functions occurring in traditional computability theory only, one can restrict oneself to a universe $\{\mathbb{N}^k \mid k \in \mathbb{N}\}$, i.e. $\mathrm{U} := \mathbb{N} : \mathrm{Set}$ and for $n : \mathbb{N}$ $\mathrm{T}(n) := \mathbb{N}^n : \mathrm{Set}$.

In order to keep the notations simple we will in the following extend $\mathrm{Rec}_{A,B}$ to $\mathrm{Rec}^+_{A,B}$ in such a way that it refers to partial recursive functions of arbitrary $\langle C, D\rangle : \mathrm{Fam(Set)}$, which means it is a true type, and keep in mind that if $\langle C, D\rangle$ are restricted to elements of a universe, we obtain $\mathrm{Rec}^+_{A,B} : \mathrm{Set}$.

There are two alternative ways of dealing with reference to other partial recursive functions:

1. The conceptually easier one is to treat such references as recursive calls of simultaneously defined functions. In order to represent $f : (a : A) \rightharpoonup C(a)$ which makes use of $g : (a : A') \rightharpoonup C'(a)$, we combine f, g into one function $fg : (a : A'') \rightharpoonup C''(a)$ as follows:
   We have $\langle A, C\rangle, \langle A', C'\rangle : \mathrm{Fam(Set)}$. Define

$$\langle A, C\rangle \oplus \langle A', C'\rangle := \langle A + A', [C, C']\rangle$$

   where $A + A'$ is the disjoint union of A and A' and

$$[C, C'] : (A + A') \to \mathrm{Set} \ ,$$
$$[C, C'](\mathrm{inl}(x)) := C(x) \ , \quad [C, C'](\mathrm{inr}(x')) := C'(x') \ .$$

   Let $\langle A'', C''\rangle := \langle A, C\rangle \oplus \langle A', C'\rangle$. Then define

$$f \oplus g : (a : A'') \rightharpoonup C''(a) \ ,$$
$$(f \oplus g)(\mathrm{inl}(a)) \simeq f(a) \ , \quad (f \oplus g)(\mathrm{inr}(b)) \simeq g(b) \ .$$

   Define $\mathrm{emb}_{\mathrm{inl}} : (a : A, \mathrm{Rec}'_{A,C,a}) \to \mathrm{Rec}'_{A'',C'',\mathrm{inl}(a)}$ and
   $\mathrm{emb}_{\mathrm{inr}} : (a' : A', \mathrm{Rec}'_{A',C',a'}) \to \mathrm{Rec}'_{A'',C'',\mathrm{inr}(a')}$, by replacing all occurrences of $\mathrm{rec}(x, g)$ by $\mathrm{rec}(\mathrm{inl}(x), g)$ or $\mathrm{rec}(\mathrm{inr}(x), g)$, respectively.
   Let $fg := f \oplus g$. Assume we can define f recursively by making use of g. We obtain an index e_{fg} of fg by setting $e_{fg}(\mathrm{inr}(b)) := e_g(b)$ and defining $e_{fg}(\mathrm{inl}(a))$ like the recursive definition of f, but replacing recursive calls to $f(a')$ by recursive calls to $fg(\mathrm{inl}(a'))$ and calls of $g(b)$ by recursive calls to $fg(\mathrm{inr}(b))$.
   As an example we show how to define,
   assuming $g : (a : A) \rightharpoonup C(a)$, $h : (a : A) \to C(a) \rightharpoonup B(a)$,
   the function $\quad f : (a : A) \rightharpoonup B(a) \ , \quad f(a) \simeq h(a, g(a)) \ .$
   Let $C' := (a : A) \times C(a)$, $B' : C' \to \mathrm{Set}$, $B'(\langle a, c\rangle) := C(a)$.
   Let $h' : (c : C') \rightharpoonup B'$, $h'(\langle a, c\rangle) := h(a, c)$ be the uncurried form of h.
   Let $e_g : \mathrm{Rec}_{A,C}$, $e_{h'} : \mathrm{Rec}_{C',B'}$ be indices for g and h'.
   Let $\langle A^+, B^+\rangle := \langle A, B\rangle \oplus \langle A, C\rangle \oplus \langle C', B\rangle$.
   Let $fgh := f \oplus g \oplus h'$.
   Let in_f, in_g, in_h be the left, middle and right injection from A, A, C', respectively, into A^+.
   Let $\quad e'_g := \lambda b.\mathrm{emb}_{\mathrm{in}_g}(e_g(b)) : (b : B) \to \mathrm{Rec}'_{A^+,B^+,\mathrm{in}_g(b)} \ ,$
   $\quad e'_h := \lambda c.\mathrm{emb}_{\mathrm{in}_h}(e_h(c)) : (c : C') \to \mathrm{Rec}'_{A^+,B^+,\mathrm{in}_h(c)} \ .$
   We introduce an index $e_{fgh} : \mathrm{Rec}_{A^+,B^+}$ as follows:

$$e_{fgh}(\mathrm{in}_g(b)) := e'_g(b) \ , \qquad e_{fgh}(\mathrm{in}_h(c)) := e'_h(c)$$
$$e_{fgh}(\mathrm{in}_f(a)) := \mathrm{rec}(\mathrm{in}_g(a), \lambda c.\mathrm{rec}(\mathrm{in}_h(\langle a, c\rangle), \lambda b.\mathrm{const}(b)))$$

   Using extensional equality one can see that there is a bijection
   $g^{\cong}(a) : \mathrm{f}_{e_{fgh}}(\mathrm{in}_g(a))\downarrow \cong \mathrm{f}_{e_g}(a)\downarrow$, and we have

$\mathrm{eval}_{e_{fgh}}(\mathrm{in}_g(a),p) = \mathrm{eval}_{e_g}(a, g^{\cong}(a,p))$, similarly for $\mathrm{f}_{e_{fgh}}(\mathrm{in}_h(a))\downarrow$ and $\mathrm{f}_{e_h}(a)\downarrow$. Furthermore, the argument of the constructor $\mathrm{tot}_{e_{fgh},\mathrm{in}_f(a)}$ for $\mathrm{f}_{e_{fgh}}(\mathrm{in}_f(a))\downarrow$ has type

$$(p : \mathrm{f}_{e_{fgh}}(\mathrm{in}_g(a))\downarrow) \times (\mathrm{f}_{e_{fgh}}(\mathrm{in}_h(\langle a, \mathrm{eval}_{e_{fgh}}(\mathrm{in}_g(a),p)\rangle))\downarrow \times \{*\}) \ ,$$

and we have

$$\begin{aligned}&\mathrm{eval}_{e_{fgh}}(\mathrm{in}_f(a), \mathrm{tot}_{e_{fgh},\mathrm{in}_f(a)}(\langle p, \langle q, *\rangle\rangle)\\ &\qquad = \mathrm{eval}_{e_{fgh}}(\mathrm{in}_h(\langle a, \mathrm{eval}_{e_{fgh}}(\mathrm{in}_g(a),p)\rangle), q)\end{aligned}$$

Modulo the aforementioned isomorphisms, this means that $\mathrm{f}_{e_{fgh}}(\mathrm{in}_f(a))\downarrow$ iff there exists $p : \mathrm{f}_{e_g}(a)\downarrow$ and $q : \mathrm{f}_{e_h}(\langle a, \mathrm{eval}_{e_g}(a,p)\rangle)\downarrow$, and that $\mathrm{eval}_{e_{fgh}}(\mathrm{in}_f(a), \_) = \mathrm{eval}_{e_h}(\langle a, \mathrm{eval}_{e_g}(a,p)\rangle, q)$, i.e. the function defined by e_{fgh} composed with in_f is the composition of the functions given by e_g and e_h.
In general we define

$$\mathrm{Rec}^+_{A,B} := (C : \mathrm{Set}) \times (D : C \to \mathrm{Set}) \times \mathrm{Rec}_{C+A,[D,B]} \ ,$$

and for $e = \langle C, D, e'\rangle : \mathrm{Rec}^+_{A,B}$ we define $\mathrm{f}^+_e(\cdot)\downarrow : A \to \mathrm{Set}$,
$\mathrm{f}^+_e(a)\downarrow := \mathrm{f}_{e'}(\mathrm{inr}(a))\downarrow$ and $\mathrm{eval}^+_e : (a : A, p : \mathrm{f}^+_e(a)\downarrow) \to B(a)$,
$\mathrm{eval}^+_e(a,p) := \mathrm{eval}^+_{e'}(\mathrm{inr}(a), p)$.

2. The approach which is easier for implementing proofs is to extend Rec by a constructor which calls a partial recursive function having an arbitrary type. So we extend $\mathrm{Rec}_{A,B}$, $\mathrm{Rec}'_{A,B,a}$ to types $\mathrm{Rec}^+_{A,B}$, $\mathrm{Rec}^{+,'}_{A,B,a}$ with an additional constructor

$$\begin{aligned}&\mathrm{call}_a : (C : \mathrm{Set}, D : C \to \mathrm{Set}, e : \mathrm{Rec}^+_{C,D}, c : C, g : D(c) \to \mathrm{Rec}^{+,'}_{A,B,a})\\ &\qquad \to \mathrm{Rec}^{+,'}_{A,B,a}\\ &\mathrm{Arg}_{a,\mathrm{call}_a(C,D,e,c,g)}(X,Y) = (p : \mathrm{f}_{C,D,e}(c)\downarrow) \times \mathrm{Arg}_{a,g(\mathrm{eval}_{C,D,e}(c,p))}(X,Y)\\ &\mathrm{Eval}_{a,\mathrm{call}_a(C,D,e,c,g)}(X,Y,\langle p,q\rangle) = \mathrm{Eval}_{a,g(\mathrm{eval}_{C,D,e}(c,p))}(X,Y,q)\end{aligned}$$

Note that with this approach $\mathrm{Rec}^+_{C,D} : \mathrm{Type}$ are defined simultaneously for all $\langle C, D\rangle : \mathrm{Fam}(\mathrm{Set})$ and simultaneously with $\mathrm{f}_{C,D}(\cdot)\downarrow$, $\mathrm{eval}_{C,D}$, Arg, Eval.

With both approaches we can show the following theorem:

Theorem 3.1. *(a) The type of partial recursive functions represented by* Rec^+ *contains all total functions and is closed under composition, primitive recursion into higher types, and the* μ*-operator for partial recursive functions.*
(b) All partial recursive functions $\mathbb{N}^n \rightharpoonup \mathbb{N}$ *are represented in* $\mathrm{Rec}^+_{\mathbb{N}^n,\mathbb{N}}$*. For this the restriction of calls of other partial recursive functions to types being elements of a universe containing* $\{\mathbb{N}^k \mid k \in \mathbb{N}\}$ *suffices.*
(c) If $e : \mathrm{Rec}^+_{A,B}$ *is derived without a context, then we have* $\{e\}(a)\downarrow$ *iff* $p : \mathrm{f}_e(a)\downarrow$ *for some* p*. Furthermore, if* $p : \mathrm{f}_e(a)\downarrow$*, then* $\{e\}(a) \simeq \mathrm{eval}_e(a,p)$*, and* $\{e\}$ *is partial recursive.*

References

[BC05a] A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, August 2005.

[BC05b] A. Bove and V. Capretta. Recursive functions with higher order domains. In P. Urzyczyn, editor, *Typed Lambda Calculi and Applications.*, volume 3461 of *LNCS*, pages 116–130. Springer, 2005.

[Cap05] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18, 2005.

[DS99] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In J.-Y. Girard, editor, *Typed Lambda Calculi and Applications*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146, 1999.

[DS01] Peter Dybjer and Anton Setzer. Indexed induction-recursion. In R. Kahle, P. Schroeder-Heister, and R. Stärk, editors, *Proof Theory in Computer Science*, pages 93 – 113. LNCS 2183, 2001.

[DS03] Peter Dybjer and Anton Setzer. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, 124:1 – 47, 2003.

[DS06] Peter Dybjer and Anton Setzer. Indexed induction-recursion. *Journal of Logic and Algebraic Programming*, 66:1 – 49, 2006.

[Dyb94] Peter Dybjer. Inductive families. *Formal Aspects of Comp.*, 6:440–465, 1994.

[Dyb00] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, June 2000.