A Data Type of Partial Recursive Functions in Martin-Löf Type Theory

Anton Setzer*

November 12, 2006

Abstract

In this article we investigate how to represent partial recursive functions in Martin-Löf's type theory. Our representation will be based on the approach by Bove and Capretta, which makes use of indexed inductive-recursive definitions (IIRD). We will show how to restrict the IIRD used so that we obtain directly executable partial recursive functions, Then we introduce a data type of partial recursive functions. We show how to evaluate elements of this data type inside Martin-Löf's type theory, and that therefore the functions defined by this data type are in fact partial recursive. The data type formulates a very general schema for defining functions recursively in dependent type theory. The initial version of this data type, for which we introduce an induction principle, needs to be expanded, in order to obtain closure under composition. We will obtain two versions of this expanded data type, and prove that they define the same set of partial recursive functions. Both versions will be large types. Next we prove a Kleene-style normal form theorem. Using it we will show how to obtain a data type of partial recursive functions which is a small set. Finally, we show how to define self-evaluation as a partial recursive function. We obtain a version of this evaluation function, which not only computes recursively a result, but as well a proof that the result is correct.

Keywords: Martin-Löf type theory, computability theory, recursion theory, Kleene index, Kleene brackets, Kleene's normal form theorem, partial recursive functions, inductive-recursive definitions, indexed induction-recursion, self-evaluation.

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

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

*Dept. of Computer Science, University of Wales Swansea,Singleton Park, Swansea SA2 8PP, UK. Tel.: +44 1792 513368, Fax: +44 1792 295651, Email: a.g.setzer@swan.ac.uk, webpage: http://www.cs.swan.ac.uk/∼csetzer/. This research was supported by EPSRC grant GR/S30450/01. Part of this work was done while the author was an invited research follow at the Graduate School of Science and Technology, Kobe University, Japan.

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

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

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

$$\begin{array}{lll}
\mathrm{defined}^0_f & : & f(0)\downarrow \ , \qquad f(0)[\mathrm{defined}^0_f] = 0 \ , \\
\mathrm{defined}^{\mathrm{S}}_f & : & (n : \mathbb{N}, p : f(n)\downarrow, q : f(f(n)[p])\downarrow) \to f(n+1)\downarrow \ , \\
\multicolumn{3}{l}{f(n+1)[\mathrm{defined}^{\mathrm{S}}_f(n,p,q)] = f(f(n)[p])[q] \ .}
\end{array}$$

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 $f(n+1)[p'] = f(m)[q]$. We observe that $\mathrm{defined}^{\mathrm{S}}_f$ refers to $f(n)[p]$, so we have to define simultaneously $f(\cdot)\downarrow$ inductively, while defining $f(\cdot)[\cdot]$ 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)

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

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 obtain 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.

Overview. We start by introducing in Sect. 2 our basic notations and type theoretic assumptions. Then we review briefly in Sect. 3 indexed inductive-recursive definitions (IIRD), and show, which restrictions need to be applied in order to obtain IIRD which correspond to directly executable partial recursive functions. In Sect. 4 we formalise a data type of IIRD consisting of all those IIRD which are restricted in the sense of the previous section. In order to prove properties of this data type and to introduce transformations, we need an induction principle, which will be introduced in Sect. 5. In Sect. 6 we show how to compute recursively the functions obtained by this data type inside type theory. This shows that the partial recursive functions are in fact directly executable. In Sect. 7 we prove that this direct execution obtains the correct result. Sect. 8 is a slightly more technical section, in which we prove that we can replace, when computing a partial recursive function for a given argument, recursive references to the function by a reference to a finite approximation of it. In Sect. 9 we show how to combine several partial recursive functions into one. The combination will be called the sum of those functions. In Sect. 10 we investigate the problem that our data type is not closed under reference to other partial recursive functions. We will show two alternative solutions for overcoming this problem, the data types Rec_0^+ and Rec_1^+. Rec_0^+ can be obtained directly from Rec, while Rec_1^+ involves an induction definition which forms true types. The advantage of Rec_1^+ is that it will be much easier to define recursive functions in it. In order to get the advantages of both, we show in Sect. 11, that both data types define the same set of partial recursive functions. In Sect. 12 we will prove that the partial recursive functions definable in Rec_0^+ and Rec_1^+ are closed under the standard operations for forming partial recursive functions, including closure under total functions. We will show as well that for every index e for a partial recursive function we can introduce an index e' for another partial recursive functions, which not only computes the same result, but as well a proof that this result is correct. In the final Sect. 13 we show a Kleene style normal form function, namely that every function can be evaluated by finding the least n s.t. a certain total function returns the Boolean value true, and then computing from it by using another total function the result. Using this observation we obtain another data type of partial recursive function Rec_2^+, which is equivalent to Rec_0^+ and Rec_1^+, but a small set. We show then how to evaluate elements of Rec_2^+ inside Rec_2^+ itself. We introduce as well a variant of this self-evaluation function, which as well computes a proof that the result is correct. We finish with some concluding remarks (Sect. 14).

Related Work Martin-Löf has suggested to add an itertion type Ω to type theory plus an element $\omega : \Omega$, which is a fixed point of the successor function. This type theory has been studied intensively by Palmgren and Stoltenberg-Hansen in [PSH90, PSH92, Pal91]. The problem of this approach is that the resulting type theory is inconsistent, and can therefore be only be used as a programming language, but no longer as a system for proving theorems. Various researchers, e.g. Paulson [Pau93] and Nordström [Nor88], have used accessibility predicates in order to deal with partial functions. However these approaches do not lead to a Turing complete data type of partial recursive functions. Constable and Scott Smith [Con83, CS87] have developed an approach in which they have a type theory with both partial and total types. This is a very elaborate approach, which is implemented in NuPrl, but requires a substantial modification of the underlying type theory. Geuvers, Poll and Zwanenburg studied the addition of the Y-combinator to the type theory and prove a conservativity result. However, their approach is limited, since the result applies only to proofs which can be converted into proofs not using Y – this excludes proofs which reasoning about partial functions in general. The current article is heavily based on the approach by Bove and Capretta [BC05a, BC05b, Bov02a, Bov02b]. Capretta [Cap05] has recently proposed the use of the delay monad in order to represent partial objects. This seems to be a very interesting approach, but requires the extension of type theory by coinductive data types. There are many more approaches. Excellent overviews over the literature can be found in [Cap05, BC05a].

2 Notations and Type theoretic assumptions

2.1 Type Theoretic Assumptions

Definition 2.1 (Basic Logic Notations) *(a) We use $a[x_1 := b_1, \ldots, x_n := b_n]$ for the result of simultaneously substituting in a the variable x_i by b_i ($i = 1, \ldots, n$).*

(b) Often arguments of functions will be written as subscripts. How to write those arguments will be clear from its use.

(c) We will frequently overload notations, i.e. use the same notation for different operations. This overloading can always be resolved by the reader. When formally carrying out proofs in type theory, one would need to use different notations for the overloaded variants of a notation.

(d) We will often omit parameters, which can be inferred from the context. When working formally in type theory, those parameters need to be reintroduced. When writing them as indices, they will usually simply be omitted, otherwise we sometimes use $\_$ for a suppressed parameter.

Assumption 2.2 (Type Theoretic Assumptions) *(a) We work in extensional type theory. We assume however that many theorems can be, possibly with some restrictions, shown in intensional type theory as well.*

(b) We assume the logical framework. So we have a collection of small types called Set *and a collection of large types called* Type*. We have* $\mathrm{Set} : \mathrm{Type}$ *and if* $A : \mathrm{Set}$ *then* $A : \mathrm{Type}$*. Both* Set *and* Type *will be closed under the dependent function type (Π) and the dependent product (Σ) of the logical framework.*

(c) We do not distinguish in this article between the logical framework dependent function type and product and the set constructions Π and Σ. All of them will be equipped with the η-rule.

Definition 2.3 (Basic Type Theoretic Notations) *(a) We usually write $(x : A) \to B$ for the dependent function type, which in type theory is often denoted by $\Pi x : A.B$. $(x : A, y : B) \to C := (x : A) \to (y : B) \to C$, $(x : A, B) \to C := (x : A, y : B) \to C$ for some fresh y. Related abbreviations are to be understood in the same way.*

(b) We write $\Sigma x : A.B$ for the dependent sum type, which is sometimes denoted by $(x : A) \times B$.

(c) We use $\forall x : A.B$ and $(x : A) \to B$ as two notations for the same type, where $\forall x : A.B$ is used when considering it as a proposition, and $(x : A) \to B$ is used when considering it as a data type.

Similarly $\Sigma x : A.B$ and $\exists x : A.B$ denote the same objects.

(d) $A \times B$ and $A \wedge B$ are two notations for $\Sigma x : A.B$, where B does not depend on x.

(e) The canonical elements of $\Sigma x : A.B$ are denoted by $\langle a, b\rangle$. We write for elements of iterated Σ-types $\Sigma a : A.\Sigma b : B(a).C(a, b)$ short $\langle a, b, c\rangle$ for $\langle a, \langle b, c\rangle\rangle$, similarly for longer sequences.

(f) Assume $I : \mathrm{Set}$, $A : I \to \mathrm{Set}$ *written as A_i for $A(i)$.*

(i) When we want to stress that $\Sigma i : I.A_i$ is the disjoint union of $(A_i)_{i:I}$, we write sometimes $\Sigma_{i:I} A_i$ for $\Sigma i : I.A_i$.

(ii) For $i : I, a : A_i$, let $\mathrm{in}_i(a) := \langle i, a\rangle$, when considering it as the injection of a into $\Sigma_{i:I} A_i$.

(g) We write $\pi_0(a)$ and $\pi_1(a)$ for the first and second projection of an element $a : (\Sigma x : A.B)$.

(h) We write $\lambda\langle a, b\rangle.r$ for $\lambda x.r[a := \pi_0(x), b := \pi_1(x)]$ for some fresh x. Similar notations (e.g. $\lambda\langle a, b, c\rangle.r$, $\Sigma\langle a, b\rangle : A.B$ etc.) are to be understood in the same way.

(i) We write application as in mathematics, not in functional style. So f applied to a is written as $f(a)$. $f(a_1, \ldots, a_n) = f(a_1)(a_2)\cdots(a_n)$. Sometimes, we identify a function $f : (\langle a, b\rangle : \Sigma a : A.B(a)) \to C(a, b)$ with it's Curried form $\lambda a, b.f(\langle a, b\rangle) : (a : A, b : B(a)) \to C(a, b)$. Then $f(a, b)$ stands for $f(\langle a, b\rangle)$. Whether Currying or iterated application is meant will always be clear from the context.

Definition 2.4 (Finite Sets) *(a) We write $\{a_1, \ldots, a_n\}$ for the set with n elements, i.e. the set A having constructors $a_i : A$ and the usual elimination rules for a set with n elements. In case $n = 0$ we write $\emptyset$, or $\perp$, when considering it as the false proposition, for the set with no elements.*

(b) Ex falsum quodlibet (latin for "from falsity follows everything") creates an element of any set from a (non-existing) proof of falsity:

$$\mathrm{efq} \quad : \quad (B : \mathrm{Set}, p : \perp) \to B \ , \quad \mathrm{efq}_B(p) \quad = \quad \mathrm{case}\ p \ \ \mathrm{of}\ \{\ \ \} \ .$$

We sometimes omit the parameter B.

(c) $\top := \{\mathrm{true}\}$, the always true proposition.

(d) We define the set of Booleans is defined as $\mathrm{Bool} := \{\mathrm{tt}, \mathrm{ff}\} : \mathrm{Set}$.

(e) We define atom as the operation, which converts a Boolean value into a proposition, which is inhabited (i.e. provable), in case the value is true, and not inhabited (i.e. unprovable), in case the value is false:

$$\begin{array}{rcl} \mathrm{atom} & : & \mathrm{Bool} \to \mathrm{Set} \ , \\ \mathrm{atom}(\mathrm{tt}) & := & \top \ , \quad \mathrm{atom}(\mathrm{ff}) \quad := \quad \perp \ . \end{array}$$

Definition 2.5 (Natural Numbers) *The set of natural numbers is denoted by $\mathbb{N}$. We have the usual notations like 0, $n + m$, $n < m$.*

Definition 2.6 (The set $\mathbb{N}_n$) *(a) $\mathbb{N}_n$ is the finite set with n elements, and by overloading the notation for the natural numbers, we write $\mathbb{N}_n = \{0, \ldots, n-1\}$. We use $\mathbb{N}_n$ as being defined by induction on $\mathbb{N}$, which can be done by defining $\mathbb{N}_0 = \emptyset$, $\mathbb{N}_{n+1} = \{0\} + \mathbb{N}_n$. This means that the element i stands more formally for $\underbrace{\mathrm{inr}(\mathrm{inr}(\cdots \mathrm{inr}(}_{i\ \mathit{times}}\mathrm{inl}(0))\cdots))$.*

(b) If $n : \mathbb{N}$, then by $(i < n) \to D[i]$ we mean $(i : \mathbb{N}_n) \to D[i]$, and $\lambda i < n.r[i]$ stands for $\lambda i : \mathbb{N}_n.r[i]$. Although sometimes i occurring in $D[i]$ is actually a natural number, it can in our applications always be replaced by an element $\mathbb{N}_n$.

Definition 2.7 (Lists) *(a) $\mathrm{List}(A)$ is the set of lists of elements of type A (where $A : \mathrm{Set}$).*

(b) *For a list we have* $\mathrm{nil} : \mathrm{List}(A)$, $\mathrm{cons} : A \to \mathrm{List}(A) \to \mathrm{List}(A)$.

(c) *We have the usual definition of* $\mathrm{length}(l)$ *for* $l : \mathrm{List}(A)$*, and for* $i < \mathrm{length}(l)$ *we have the* i*th element of the list denoted by* $(l)_i$.

(d) *If* $l : \mathrm{List}(A)$*, then* $(x \in l) \to D[x]$ *stands for* $(i < \mathrm{length}(l)) \to D[(x)_i]$*, and* $\lambda x \in l.s$*, which is supposed to be an element of* $(x \in l) \to D[x]$*, stands for* $\lambda i < \mathrm{length}(l).r[(x)_i]$.

(e) *The formula (or type)* $x \in l$ *is defined as* $(i < \mathrm{length}(l)) \times (x =_A (l)_i)$*. Similar notations (e.g.* $\exists x \in l$, $l \subseteq l'$*) are to be understood correspondingly.*

(f) *We write sometimes* $\langle a_1, \ldots, a_n \rangle$ *for the list with elements* $a_1, \ldots, a_n$ *(in that order). Note that in case of* $n = 2$ *this overloads the notation* $\langle a, b \rangle$ *(used as well for elements of* $\Sigma a : A.B(a)$*).*

(g) map *is the usual map function, so if* $l = \langle a_1, \ldots, a_n \rangle : \mathrm{List}(A)$, $f : A \to B$*, then* $\mathrm{map}(f, l) := \langle f(a_1), \ldots, f(a_n) \rangle : \mathrm{List}(B)$.

Remark 2.8 *Note that when actually carrying out proofs on the machine, it is usually easier to replace lists by a pair consisting of* $n : \mathbb{N}$ *and* $f : \mathbb{N}_n \to A$*. Then the formula* $(x \in l) \to D[x]$*, with* l *corresponding to* n, f *stands for* $(i < n) \to D[f(i)]$.

Definition 2.9 ($A + B$, Equality) (a) *If* $A, B : \mathrm{Set}$*, then* $A + B : \mathrm{Set}$ *is the disjoint union of* A, B *with constructors* $\mathrm{inl} : A \to (A + B)$,
$\mathrm{inr} : B : B \to (A + B)$ *(for in-left and in-right).*

(b) *The equality set is denoted by* $a =_A b$ *for* $a, b : A$*. We sometimes omit the index* A*, and assume that it is usually clear whether the equality set or definitional equality is meant.*

Definition 2.10 *If* $f : A \to A'$*, then* $\mathrm{inj}(f) := \forall a, a' : A.f(a) =_{A'} f(a') \to a =_A a'$ *(*f *is injective).*

Definition 2.11 (Families of Sets) (a) *If* X *is a type,* $\mathrm{Fam}(X) := \Sigma A : \mathrm{Set}.A \to X$.

(b) *If* $C : \mathrm{Fam}(X)$, $C = \langle A, B \rangle$*, then* $C.A := A$, $C.B := B$.

(c) $\emptyset_{\mathrm{Fam}} := \langle \emptyset, \lambda x.\mathrm{efq}(x) \rangle$*, the empty family.*

(d) *If* $A, A' : \mathrm{Set}$, $B : A \to \mathrm{Set}$, $B' : A' \to \mathrm{Set}$*. Then define* $[B, B'] : (A + B) \to \mathrm{Set}$, $[B, B'](\mathrm{inl}(a)) = B(a)$, $[B, B'](\mathrm{inr}(a')) = B'(a')$.

(e) *Assume* $C : \mathrm{Fam}(\mathrm{Set})$, $C' : \mathrm{Fam}(\mathrm{Set})$*. Then* $C \oplus C' := \langle C.A + C'.A, [C.B, C'.B] \rangle : \mathrm{Fam}(\mathrm{Set})$.

(f) *Assume* $A : I \to \mathrm{Set}$, $B : (i : I, A_i) \to \mathrm{Set}$*, written as* $B_i(a)$ *for* $B(i, a)$*. Then* $[\,]_{i:I} B_i : (\Sigma i : I.A_i) \to \mathrm{Set}$, $[\,]_{i:I} B_i(\langle i, a \rangle) := B_i(a)$.

(g) *Assume* $C : I \to \mathrm{Fam}(\mathrm{Set})$, $C_i = \langle A_i, B_i \rangle$*. Then* $\oplus_{i:I} C_i := \langle \Sigma_{i:I} A_i, [\,]_{i:I} B_i \rangle$.

(h) *Assume an operation like* $\mathrm{Rec}, \mathrm{Rec}'$ *as introduced later having arguments written as indices* $A : \mathrm{Set}$, $B : A \to \mathrm{Set}$*. If we write such an operations with an index* $C : \mathrm{Fam}(\mathrm{Set})$*, we mean the indices* $C.A$, $C.B$*. E.g.* $\mathrm{Rec}_C := \mathrm{Rec}_{C.A, C.B}$, $\mathrm{Rec}'_{C,a} := \mathrm{Rec}_{C.A, C.B, a}$*, etc.*

2.2 Partial Objects

In this article we will introduce various partial objects of some set B, which we usually denote by $\{e\}(a)$ or $\{a, e'\}_e$, possibly with some subscripts or superscripts. They will all be elements of $\mathrm{PrePar}(B(a))$ or $\mathrm{Par}(B(a))$, as defined in the next definition.

Definition 2.12 *(a) The set of pre-partial objects of the set B is defined as $\mathrm{PrePar} := \mathrm{Fam}(B)$. If $e : \mathrm{PrePar}$, we define $e{\downarrow} := \pi_0(e)$, and for $p : e{\downarrow}$, $e[p] := \pi_1(e)(p)$. (So $p : e{\downarrow}$ means that e is defined, and $e[p]$ is the value of e depending on the proof p.)*

(b) If $e : \mathrm{PrePar}$, the consistency of e, $\mathrm{Consistent}(e)$ is defined as $\forall p, p' : e{\downarrow}.e[p] =_B e[p']$.

(c) The set of partial objects of the set B is defined as $\mathrm{Par}(B) := \Sigma e : \mathrm{PrePar}(B).\mathrm{Consistent}(e)$.
Overloading the notation we define for $e : \mathrm{Par}(B)$
$e{\downarrow} := \pi_0(e){\downarrow}$ and $e[p] := \pi_0(e)[p]$.

We will usually introduce elements $e : \mathrm{Par}(B)$ by first defining $e{\downarrow}$, $e[p]$, and then later showing its consistency. Note that elements of $\mathrm{Par}(B)$ are usually not directly executable.

Definition 2.13 *(a) Let $B : \mathrm{Set}$, $e, e' : \mathrm{PrePar}(B)$ (or $: \mathrm{Par}(B)$), $b : B$.*

$$e \simeq b \quad :\Leftrightarrow \quad \exists p : e{\downarrow}.e[p] = b \ , \qquad e \sqsubseteq e' \quad :\Leftrightarrow \quad \forall b : B.e \simeq b \to e' \simeq b \ ,$$
$$e \simeq e' \quad :\Leftrightarrow \quad e \sqsubseteq e' \wedge e' \sqsubseteq e \ .$$

(b) Assume $f, g : (a : A) \to \mathrm{Par}(B(a))$ (or $f, g : (a : A) \to \mathrm{PrePar}(B(a))$. Then we define
$f \subseteq g :\Leftrightarrow \forall a : A.f(a) \sqsubseteq g(a)$, $\quad f \cong g :\Leftrightarrow f \subseteq g \wedge g \subseteq f$.

When in the above situation, f or g is of the form $\{e\}$ (possibly with some subscripts or superscripts; such kind of notations will be introduced at various places in this article), we sometimes write e instead of $\{e\}$, e.g. $e \subseteq e'$ instead of $\{e\} \subseteq \{e'\}$.

2.3 Finite Functions

We introduce the set of multivalued finite functions $(a : A) \rightharpoonup^{\mathrm{multi}}_{\mathrm{fin}} B(a)$ as lists of pairs $\langle a, b\rangle$ for $a : A$, $b : B(a)$. Then we define finite functions $(a : A) \rightharpoonup_{\mathrm{fin}} B(a)$ as consistent elements of $(a : A) \rightharpoonup^{\mathrm{multi}}_{\mathrm{fin}} B(a)$.

Definition 2.14 (Multivalued Finite Functions) *Assume $C = \langle A, B\rangle : \mathrm{Fam}(\mathrm{Set})$, which we keep fixed.*

(a) $(a : A) \rightharpoonup^{\mathrm{multi}}_{\mathrm{fin}} B := \mathrm{List}(\Sigma a : A.B)$.

(b) Let $g : (a : A) \rightharpoonup^{\mathrm{multi}}_{\mathrm{fin}} B(a)$. We define $\{g\}^{\mathrm{finfun}} : A \to \mathrm{PrePar}(B(a))$ by

$$\{g\}^{\mathrm{finfun}}(a){\downarrow} \quad :\Leftrightarrow \quad \exists b : B(a).\langle a, b\rangle \in g \ , \quad \{g\}^{\mathrm{finfun}}(a)[\langle b, q\rangle] \quad := \quad b \ .$$

(c) $\emptyset_{\mathrm{finfun}} := \mathrm{nil} : (a : A) \rightharpoonup^{\mathrm{multi}}_{\mathrm{fin}} B(a)$. We sometimes omit the subscript finfun.

(d) Assume $g : (a : A) \rightharpoonup^{\mathrm{multi}}_{\mathrm{fin}} B(a)$, $a : A$, $b : B(a)$. Then

$$g[a \mapsto b] := \mathrm{cons}(\langle a, b\rangle, g) : (a : A) \rightharpoonup^{\mathrm{multi}}_{\mathrm{fin}} B(a) \ .$$

(e) $(\langle a,b\rangle)_{\mathrm{finfun}} := \emptyset_{\mathrm{finfun}}[a \mapsto b]$.

(f) *Assume* $g : (a : A) \rightharpoonup^{\mathrm{multi}}_{\mathrm{fin}} B(a)$, $B' : A \to \mathrm{Set}$, $h : (a : A, B(a)) \to B'(a)$. *Then* $h \circ_{\mathrm{finfun,left}} g := k$, *where* $k := \mathrm{map}(\lambda\langle a,b\rangle.\langle a, h(a,b)\rangle, g)$.
We usually omit the subscripts finfun, left *of* $\circ$.

(g) *Assume* $A, A' : \mathrm{Set}$, $B' : A' \to \mathrm{Set}$, $h : A \to A'$, $B : A \to \mathrm{Set}$, $B(a) := B'(h(a))$. *Assume* $g : (a : A) \rightharpoonup^{\mathrm{multi}}_{\mathrm{fin}} B(a)$. *Then*

$$g \circ_{\mathrm{finfun,right}} h^{-1} := k \ ,$$

where $k := \mathrm{map}(\lambda\langle a,b\rangle.\langle h(a), b\rangle, g)$. *We usually omit the indices* finfun, right *from* $\circ$.

Definition 2.15 (Finite Functions) *Assume* $C = \langle A, B\rangle : \mathrm{Fam}(\mathrm{Set})$, *which we keep fixed.*

(a) $(a : A) \rightharpoonup_{\mathrm{fin}} B := \Sigma f : ((a : A) \rightharpoonup^{\mathrm{multi}}_{\mathrm{fin}} B(a)).\forall a : A.\mathrm{Consistent}(\{f\}^{\mathrm{finfun}}(a))$.

If $f : (a : A) \rightharpoonup^{\mathrm{multi}}_{\mathrm{fin}} B(a)$, *and we later determine* $p : \mathrm{Consistent}(f)$, *then we will then consider* f *as an element of* $(a : A) \rightharpoonup_{\mathrm{fin}} B(a)$, *even so the element of* $(a : A) \rightharpoonup_{\mathrm{fin}} B(a)$ *is more precisely* $\langle f, p\rangle$.

(b) *We can lift in an obvious way* $\{\cdot\}^{\mathrm{finfun}}$, $\emptyset_{\mathrm{finfun}}$, $(\langle a,b\rangle)_{\mathrm{finfun}}$, $h \circ_{\mathrm{finfun,left}} g$ *to operations on* $(a : A) \to_{\mathrm{finfun}} B(a)$ *to* $(a : A) \to_{\mathrm{finfun}} B(a)$. *This overloads the corresponding notations. We usually omit indices as when dealing with multivalued finite functions. In case of* $\{\cdot\}^{\mathrm{finfun}}$ *we obtain in fact* $\{g\}^{\mathrm{finfun}}(a) : \mathrm{Par}(B(a))$.

(c) *Assume* $g : (a : A) \rightharpoonup_{\mathrm{fin}} B(a)$, $g = \langle f, p\rangle$, $a : A$, $b : B(a)$,
$p' : \forall b' : B(a).\{g\}^{\mathrm{finfun}}(a) \simeq b' \to b' =_{B(a)} b$. *Then*

$$g[a \mapsto b]_{p'} := \langle f[a \mapsto b], p'''\rangle \ ,$$

where $p''' : \mathrm{Consistent}(f[a \mapsto b])$ *is obtained from* p *and* p'. *If* p' *is obvious, the subscript* p' *will be omitted.*

(d) *Assume* $A, A' : \mathrm{Set}$, $B' : A' \to \mathrm{Set}$, $h : A \to A'$, $B : A \to \mathrm{Set}$, $B(a) := B'(h(a))$. *Assume* $g = \langle f, p\rangle : (a : A) \rightharpoonup_{\mathrm{fin}} B(a)$, $A' : \mathrm{Set}$, $p' : \mathrm{inj}(h)$. *Then*

$$g \circ_{\mathrm{finfun,right},p'} h^{-1} := \langle f \circ_{\mathrm{finfun,right}} h^{-1}, p''\rangle \ ,$$

and $p'' : \mathrm{Consistent}(f \circ_{\mathrm{finfun,right}} h^{-1})$ *is obtained from* p *and* p'.

We usually omit the indices finfun, right, *and, if it is obvious,* p' *from* $\circ$.

(e) *Assume* $g : (a : A) \rightharpoonup_{\mathrm{fin}} B(a)$, $g = \langle f, p\rangle$. *Then* $\mathrm{length}(g) := \mathrm{length}(f)$. *Assume furthermore* $h : (a : A, b : B(a)) \to \mathbb{N}$. *Then* $\sum_{a:\mathrm{dom}(g)} .h(a, g(a)) := \sum_{i<\mathrm{length}(l)} h(\pi_0((f)_i), h_1((f)_i))$. *Note that* $g \cong g' \not\Rightarrow \mathrm{length}(g) = \mathrm{length}(g')$, *and* $g \cong g' \not\Rightarrow \sum_{a:\mathrm{dom}(g)} h(a, g(a)) = \sum_{a:\mathrm{dom}(g')} h(a, g'(a))$, *since in* f *an element* $\langle a, b\rangle$ *might occur more than once.*

3 Inductive-Recursive Definitions and Partial Recursive Functions

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 $D : \mathrm{Type}$. IRD 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 was 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 $I : \mathrm{Set}$, and $i : I \Rightarrow D[i] : \mathrm{Type}$. 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 $A : \mathrm{Set}$, $B : A \to \mathrm{Set}$. Such functions will be translated into IIRD with index set A and $D[x] := B(x)$. Note that $D[x] : \mathrm{Set}$. In [DS06] Dybjer and the author have shown that IIRD with $D[x] : \mathrm{Set}$ can always be reduced to indexed inductive definitions. We sketch the idea briefly by taking the example from the introduction. Remember that $\mathrm{defined}_f^{\mathrm{S}}$ had type

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

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

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

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

$$\begin{aligned}
f(0)^{\mathrm{aux}}[\mathrm{defined}_f^{0,\mathrm{aux}}] &:= 0 \ , \\
f(n+1)^{\mathrm{aux}}[\mathrm{defined}_f^{\mathrm{S},\mathrm{aux}}(n, p, m, q)] &:= f(m)^{\mathrm{aux}}[q] \ .
\end{aligned}$$

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

$$
\begin{aligned}
&\mathrm{Corr}_f \colon (n : \mathbb{N}, p : f(n)^{\mathrm{aux}}{\downarrow}) \to \mathrm{Set}\ , \qquad \mathrm{Corr}_f(0, \mathrm{defined}_f^{0,\mathrm{aux}}) := \top\ ,\\
&\mathrm{Corr}_f(n+1, \mathrm{defined}_f^{\mathrm{S,aux}}(n, p, m, q) :=\\
&\qquad\qquad \mathrm{Corr}_f(n, p) \wedge \mathrm{Corr}_f(m, q) \wedge m =_{\mathbb{N}} f(n)^{\mathrm{aux}}[p]\ .
\end{aligned}
$$

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

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

and simulate $f(\cdot)[\cdot]$ by

$$
f(\cdot)'[\cdot] \ :\ (n : \mathbb{N}, p : f(n){\downarrow}') \to \mathbb{N}\ ,\quad f(n)'[\langle p, q\rangle] \qquad := f(n)^{\mathrm{aux}}[p]\ .
$$

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 functions, which can be computed partial-recursively. Here by "compute partial-recursively" we mean that we can consider f as a partial recursive function and in this way compute the value of $f(n)$, where the computation terminates if and only if $f(n){\downarrow}$, and, if $f(n){\downarrow}$, returns the value of f.

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.

   $f(n+1)[\mathrm{defined}_f^{\mathrm{S},'}(n)]$ returns a different value, e.g. 5. This corresponds to adding contradictory rewrite rules, such as $f(n+1) \longrightarrow 5$.

   Furthermore, allowing multiple constructors would prevent us in general to partial-recursively compute the functions: In order to partial-recursively compute $f(n)$, one has to find the arguments of the constructor in such a way that we obtain an element of $f(n){\downarrow}$. This requires in general a search process. In case the elements of the index set of the set of constructors are finitely representable (e.g. it is a finite set or $\mathbb{N}$, or, more generally, an element of an inductive data type without infinite branching degrees), we could search for such an index. Otherwise, because of the incompleteness of type theory, we might have an index in a consistent extension of the type theory used, which does not exist in the current type theory. (Or, looking at it from a non-constructive perspective, we might have an index in the "standard" model of type theory, but not in the type theory). So a search process for a proof in the current type theory might not find the index needed.

In order to obtain partial-recursively computable functions we therefore require that we are able to determine for each index its constructor and which arguments are needed. 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:

$$\begin{array}{rcll} f(n)\downarrow & := & \text{case } n \text{ of } 0 & \rightarrow \text{ data } \mathrm{C}^0_f \\ & & \mathrm{S}(n') & \rightarrow \text{ data } \mathrm{C}^{\mathrm{S}}_f(p : f(n')\downarrow,\ \ q : f(f(n')[p])\downarrow) \ . \end{array}$$

So $f(0)\downarrow$ has constructor C^0_f, and $f(\mathrm{S}(n'))\downarrow$ has constructor $\mathrm{C}^{\mathrm{S}}_f$ with arguments $p : f(n')\downarrow$ and $q : f(f(n')[p])\downarrow$.

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 $f(n)[p]$). Another problem with non-trivial non-inductive arguments is that when computing the function partial-recursively, one needs to search for instances of these non-inductive argument. Therefore, as argued before, one does not obtain in general partial-recursively computable functions. We could allow non-indexed arguments indexed over the one-element set 1 and the empty set $\emptyset$, since this does not require a search process. 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. As we indicated before (in this case the incompleteness is a problem even in simple situations, if A is infinite), this does in general not result in partial-recursively computable functions.

   So in order to obtain partial-recursively computable functions, we will restrict inductive arguments to single valued ones, i.e. in the above situation to arguments of the form $p : f(i)\downarrow$.

4 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 : Set, $B : A \to$ Set fixed. Unless explicitly needed, we suppress in the following dependencies on A, B (which would be added as initial indices). 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:

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

(Without suppressing the dependency on A, B this set would be called $\mathrm{Rec}_{A,B}$; similarly for later operations defined in this section).
For e : Rec we define

$$\{e\} : A \to \mathrm{PrePar}(B(a)) \ \ .$$

(We will later after the definition of Rec is finished show the consistency of $\{e\}(a)$, and can extend it therefore to an element of $\mathrm{Par}(B(a))$). In fact we define for e : Rec inductively

$$\{e\}(\cdot)\downarrow : A \to \mathrm{Set} \ \ ,$$

while recursively defining for $a : A$ $p : \{e\}(a)\downarrow$

$$\{e\}(a)[p] : B(a) \ \ .$$

$\{e\}(a)\downarrow$ means that the function with index e is defined for argument a, and if $p : \{e\}(a)\downarrow$, then $\{e\}(a)[p]$ is result of evaluating $\{e\}(a)$ using this proof.
Rec is a restricted IIRD, which means that for each $a : A$ we can determine the type of arguments for the constructor with result $\{e\}(a)\downarrow$. Let Rec'_a be the type of codes for possible arguments for the constructor of an IIRD e with result $\{e\}(a)\downarrow$. 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 $\{e\}(a)\downarrow$ and the result of $\{e\}(a)[p]$ for the constructed element p will depend on $\{e\}(\cdot)\downarrow$ and $\{e\}(\cdot)[\cdot]$. Since, when introducing Rec'_a, $\{e\}(\cdot)\downarrow$ and $\{e\}(\cdot)[\cdot]$ are not available, we define more generally, depending on $a : A$, $e : \mathrm{Rec}'_a$, for general X and Y having the types of $\{e\}(\cdot)\downarrow$, $\{e\}(\cdot)[\cdot]$, respectively, the following operations:

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

Let for $e' : \mathrm{Rec}'_a$

$$\begin{array}{rcl} \{a, e'\}^{\mathrm{aux}}_e & : & \mathrm{PrePar}(B(a)) \ \ , \\ \{a, e'\}^{\mathrm{aux}}_e\downarrow & := & \mathrm{Arg}_{a,e'}(\{e\}(\cdot)\downarrow, \{e\}(\cdot)[\cdot]) : \mathrm{Set} \ \ , \\ \{a, e'\}^{\mathrm{aux}}_e[p] & := & \mathrm{Eval}_{a,e'}(\{e\}(\cdot)\downarrow, \{e\}(\cdot)[\cdot], p) : B(a) \quad (\text{where } p : \{a, e'\}^{\mathrm{aux}}_e\downarrow) \ \ . \end{array}$$

(We will later show the consistency of $\{a, e'\}^{\mathrm{aux}}_e$, and therefore can consider it as an element of $\mathrm{Par}(B(a))$.) Then

$$\{a, e(a)\}^{\mathrm{aux}}_e\downarrow$$

will be the type of the arguments of the constructor of $\{e\}(a)\downarrow$. If using arguments p we have constructed $q : \{e\}(a)\downarrow$, then

$$\{e\}(a)[q] = \{a, e(a)\}^{\mathrm{aux}}_e[p] \ .$$

If we call the constructor for $\{e\}(a)\downarrow$ $\mathrm{tot}_{e,a}$, then the introduction and equality rules for $\{e\}(\cdot)\downarrow$ and $\{e\}(\cdot)[\cdot]$ are as follows:

$$\begin{array}{c}\mathrm{tot}_{e,a} : \{a, e(a)\}^{\mathrm{aux}}_e\downarrow \to \{e\}(a)\downarrow \ , \\ \{e\}(a)[\mathrm{tot}_{e,a}(p)] \ = \ \{a, e(a)\}^{\mathrm{aux}}_e[p] \ .\end{array}$$

Essentially, $\{e\}(a)\downarrow$ is defined as $\{a, e(a)\}^{\mathrm{aux}}_e\downarrow$ (we only put a constructor around this). $\{a, e'\}^{\mathrm{aux}}_e\downarrow$ is an auxiliary definition, which defines, whether the subcomputation for argument a given by e' is defined, provided that the complete function has index e. $\{a, e'\}^{\mathrm{aux}}_e[p]$ defines, how to compute the value for this subcomputation, depending on a proof $p : \{a, e'\}^{\mathrm{aux}}_e\downarrow$.
Rec'_a has 2 constructors:

1. Initial (constant) case: the constructor for $\{e\}(a)\downarrow$ has no arguments (or more precisely the trivial argument $x : \{*\}$ for the one-element set $\{*\}$). $\{e\}(a)[p]$ returns, independently of p, a fixed element $b : B(a)$ (we use the notation return, since it makes writing of codes for partial recursive functions more readable: when we reach return, we return a result obtained):

$$\begin{array}{c}\mathrm{return} : B(a) \to \mathrm{Rec}'_a \ , \qquad \mathrm{Arg}_{a,\mathrm{return}(b)}(X, Y) = \{*\} \ , \\ \mathrm{Eval}_{a,\mathrm{return}(b)}(X, Y, *) = b \ .\end{array}$$

   So we obtain, if $e' = \mathrm{return}(b)$,

$$\begin{array}{rcl}\{a, e'\}^{\mathrm{aux}}_e\downarrow & = & \{*\} \ , \\ \{a, e'\}^{\mathrm{aux}}_e[p] & = & b \ .\end{array}$$

   $\{a, e'\}^{\mathrm{aux}}_e\downarrow$ is always defined, and the result of evaluating it returns b. This means that $\{a, e'\}^{\mathrm{aux}}_e\downarrow$ is the constant function, which returns always b.

2. A single inductive argument. The constructor of $\{e\}(a)\downarrow$ has as an inductive argument $p : \{e\}(a')\downarrow$, and depending on $b := \{e\}(a')[p]$ later arguments. As a partial recursive function this means that we make a recursive call to $\{e\}(a')\downarrow$. Depending on the result b, we choose further steps. So the constructor rec 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 $\{e\}(a')[p]$ the later arguments of the constructor. We obtain

$$\begin{array}{rcl}\multicolumn{3}{c}{\mathrm{rec} : (a' : A, g : B(a') \to \mathrm{Rec}'_a) \to \mathrm{Rec}'_a} \ , \\ \mathrm{Arg}_{a,\mathrm{rec}(a',g)}(X, Y) & = & \Sigma x : X(a').\mathrm{Arg}_{a,g(Y(a',x))}(X, Y) \ , \\ \mathrm{Eval}_{a,\mathrm{rec}(a',g)}(X, Y, \langle x, y\rangle) & = & \mathrm{Eval}_{a,g(Y(a',x))}(X, Y, y) \ .\end{array}$$

   So we obtain, if $e = \mathrm{rec}(a', g)$,

$$\begin{array}{rcl}\{a, e'\}^{\mathrm{aux}}_e\downarrow & = & \Sigma p : \{e\}(a')\downarrow.\{a, g(\{e\}(a')[p])\}^{\mathrm{aux}}_e\downarrow \ , \\ \{a, e'\}^{\mathrm{aux}}_e[\langle p, q\rangle] & = & \{a, g(\{e\}(a')[p])\}^{\mathrm{aux}}_e[q] \ .\end{array}$$

   So $\{a, e'\}^{\mathrm{aux}}_e\downarrow$ holds, if we have a proof $p : \{e\}(a')\downarrow$, and if $\{e\}(a')[p] = b'$, we have for the subcomputation $e'' := g(b')$ of e' a proof $q : \{a, e''\}^{\mathrm{aux}}_e\downarrow$. Then $\{e'\}(a)[\langle p, q\rangle] \simeq \{a, e''\}^{\mathrm{aux}}_e[q]$. This means that $\{a, e'\}^{\mathrm{aux}}_e$ is the function, which first computes $\{e\}(a')$, and if that computation terminates with result b, then computes $\{a, g(b)\}^{\mathrm{aux}}_e$ and returns its result.

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 $\mathrm{return}(b)$. Arg and Eval can be defined by recursion on Rec'_a, after the definition of Rec'_a is finished. $\{e\}(\cdot)\downarrow$ and $\{e\}(\cdot)[\cdot]$ are given by an IIRD which is determined by e.

5 An Induction Principle for the Data Type of Partial Recursive Functions.

In order to prove properties of elements of Rec, we need an induction principle for it corresponding to the fact that it is the least set introduced by the introduction rules. In case of general IIRD (see especially [DS06], but as well [DS01, DS03]) we had introduced this induction principle. Adapted to our setting, this principle is as follows: Assume as before $A : \mathrm{Set}$, $B : A \to \mathrm{Set}$ (these arguments will be suppressed), $a : A$, $e : \mathrm{Rec}'_a$. Depending on $X : A \to \mathrm{Set}$, $Y : (a' : A, x : X(a')) \to B(a')$ and $b : \mathrm{Arg}_{a,e}(X, Y)$ we defined a set of pairs $\langle a', p \rangle$ s.t. $\mathrm{rec}(a', g)$ occurred in the part of e used by b and s.t. b contained the element $p : X(a)$. These correspond in IIRD to the inductive arguments used there, or more precisely to the pairs $\langle i, x \rangle$ consisting of the index i for the inductive argument and the inductive argument x used itself.

In our setting we have only single-valued inductive arguments, and therefore only finitely many of them. Therefore the set of inductive arguments is finite, and we can represent the inductive arguments as a finite function, namely an element of $(a : A) \rightharpoonup_{\mathrm{fin}} X(a)$. If we specialise this to the situation where $X = \{e\}(\cdot)\downarrow$ and $Y = \{e\}(\cdot)[\cdot]$, then the inductive arguments are given by a function $h := \mathrm{calls}_e(a, e', p) : (a : A) \rightharpoonup_{\mathrm{fin}} \{e\}(a)\downarrow$ (where $e : \mathrm{Rec}$, $a : A$, $e' : \mathrm{Rec}'_a$, $p : \{a, e'\}^{\mathrm{aux}}_e\downarrow$), which will be defined in the following. The domain of h will be the set of a' s.t. $\mathrm{rec}(a', g)$ (i.e. an inductive argument with index a') occurs in the part of e' used by p. Then $\{h\}^{\mathrm{finfun}}(a)[\_] = p'$ for the proof p' used in p (i.e. the actual inductive argument). For convenience we define as well a finite function callresults which returns $\{e\}(a')[p]$ rather than $p : \{e\}(a')\downarrow$.

Definition 5.1 *(a) Assume $e : \mathrm{Rec}$, $a : A$, $e' : \mathrm{Rec}'_a$, $p : \{a, e'\}^{\mathrm{aux}}_e\downarrow$. We define*

$$\mathrm{calls}_e(a, e', p) : (a' : A) \rightharpoonup^{\mathrm{multi}}_{\mathrm{fin}} \{e\}(a)\downarrow \ ,$$

by recursion on e' as follows:

$$\begin{array}{rcl} \mathrm{calls}_e(a, \mathrm{return}(b), *) & := & \emptyset_{\mathrm{finfun}} \ , \\ \mathrm{calls}_e(a, \mathrm{rec}(a', g), \langle p, p' \rangle) & := & \mathrm{calls}_e(a, g(\{e\}(a')[p]), p')[a' \mapsto p] \ . \end{array}$$

We will later see that there is at most one proof of $\{e\}(a)\downarrow$ and therefore $\mathrm{Consistent}(\{\mathrm{calls}_e(a, e', p)\}^{\mathrm{finfun}}(a'))$,

$$\mathrm{calls}_e(a, e', p) : (a' : A) \rightharpoonup_{\mathrm{fin}} \{e\}(a)\downarrow \ .$$

(b) We define assuming e, a, e', p as before

$$\begin{array}{rcl} \mathrm{callresults}(a, e', p) & : & (a' : A) \rightharpoonup^{\mathrm{multi}}_{\mathrm{fin}} B(a') \ , \\ \mathrm{callresults}(a, e', p) & := & (\lambda a, p.\{e\}(a)[p]) \circ \mathrm{calls}_e(a, e', p) \ . \end{array}$$

When we have obtained $\mathrm{calls}_e(a, e', p) : (a' : A) \rightharpoonup_{\mathrm{fin}} \{e\}(a)\downarrow$*, then we obtain as well*

$$\mathrm{callresults}_e(a, e', p) : (a' : A) \rightharpoonup_{\mathrm{fin}} B(a) \ .$$

Lemma 5.2 *Assume $a, a' : A$, $e : \mathrm{Rec}'_a$, $g : B(a') \to \mathrm{Rec}'_a$, $p : \{e\}(a')\downarrow$, $b := \{e\}(a')[p]$, $p' : \{a, g(b)\}^{\mathrm{aux}}_e\downarrow$. Let $h := \mathrm{calls}_e(a, \mathrm{rec}(a', g), \langle p, p' \rangle)$.*

(a) $\{h\}^{\mathrm{finfun}}(a') \simeq p$.

(b) $\mathrm{calls}_e(a, g(b)) \subseteq h$.

Proof: Immediate.

Assume, we want to prove, assuming $D : (a : A, p : \{e\}(a)\downarrow) \to \mathrm{Set}$,
that the following holds:

$$(a : A, p : \{e\}(a)\downarrow) \to D(a, p) \ .$$

(This corresponds to the formula $\forall a : A.\forall p : \{e\}(a)\downarrow .D(a, p)$.)
Let for $a : A$, $p : \{a, e(a)\}^{\mathrm{aux}}_{e}\downarrow$, $h(a, p) := \mathrm{calls}_{e}(a, e(a), p) : (a : A) \rightharpoonup_{\mathrm{fin}} \{e\}(a)\downarrow$.
Then the step function is

$$\begin{aligned} \mathrm{step} : (&a : A, \\ &p : \{a, e(a)\}^{\mathrm{aux}}_{e}\downarrow, \\ &ih : (a' : A, p : \{h(a,p)\}^{\mathrm{finfun}}(a')\downarrow)) \to D(a', \{h(a,p)\}^{\mathrm{finfun}}(a')[p]) \\ &\to D(a, \mathrm{tot}_{e,a}(p)) \ . \end{aligned}$$

The induction principle for $\{e\}(\cdot)\downarrow$ says that if we have such a step function, then we obtain a function

$$g : (a : A, p : \{e\}(a)\downarrow) \to D(a, p) \ ,$$

s.t. we have

$$g(a, \mathrm{tot}_{e,a}(p)) = \mathrm{step}(a, p, \lambda a' : A.\lambda p : \{h(a,p)\}^{\mathrm{finfun}}(a')\downarrow .g(a', \{h(a,p)\}^{\mathrm{finfun}}(a')[p])) \ .$$

The elimination and equality rules for $\{e\}(\cdot)\downarrow$ is the principle, that for any step function (possibly depending on some context) there exist a function g as above satisfying the equality just mentioned. This principle can be introduced as well by introducing a suitable recursion operator – we refrain from doing so in order not to avoid introducing too many unnecessary notations.
We leave it to the reader to observe that this induction principle is equivalent to the one introduced in [DS06], which corresponds to the fact that a family of sets defined by an IIRD is the least family closed under the corresponding operator. See as well [DS03] for an analysis of the relationship to initial algebras.
The following lemma follows immediately using the induction principle:

Lemma 5.3 *If $p : \{e\}(a)\downarrow$. Then there exist $p' : \{a, e(a)\}^{\mathrm{aux}}_{e}\downarrow$ s.t. $p =_{\{e\}(a)\downarrow} \mathrm{tot}_{e,a}(p')$.*

The length of elements of $\{e\}(a)\downarrow$, $\{a, e'\}^{\mathrm{aux}}_{e}\downarrow$. We introduce a length function for these elements. Most proofs carried out in this article will be carried out by induction on this length, which makes it easier to argue then by using directly using the induction principle. When we say later induction on e we will often mean more precisely induction on $\mathrm{length}(e)$.

Lemma and Definition 5.4 *Assume $A : \mathrm{Set}$, $B : A \to \mathrm{Set}$, $e : \mathrm{Rec}_{A,B}$ fixed. There exist functions (we overload the name* length *here, which is already used for lists; we omit as well the additionally needed subscripts A, B, e)*

$$\begin{aligned} \mathrm{length} &: (a : A, p : \{e\}(a)\downarrow) \to \mathbb{N} \ , \\ \mathrm{length}^{\mathrm{aux}} &: (a : A, e' : \mathrm{Rec}'_{a}, p : \{a, e'\}^{\mathrm{aux}}_{e}\downarrow) \to \mathbb{N} \ . \end{aligned}$$

s.t. the following holds

(a) $\mathrm{length}(a, \mathrm{tot}_{e,a}(p)) = \mathrm{length}^{\mathrm{aux}}(a, e(a), p)$.

(b) Let $h := \mathrm{calls}_{e}(a, e', p)$. If $a' : A$, $p : \{h\}^{\mathrm{finfun}}(a')\downarrow$, $q := \{h\}^{\mathrm{finfun}}(a')[p]$, then $\mathrm{length}(a', q) < \mathrm{length}^{\mathrm{aux}}(a, e', p)$.

(c) *If* $\langle p,p'\rangle : \{a, \mathrm{rec}(a',g)\}_e^{\mathrm{aux}}\downarrow$, *then*

$$\begin{array}{rcl} \mathrm{length}(a,p) & < & \mathrm{length}^{\mathrm{aux}}(a, \mathrm{rec}(a',g), \langle p,p'\rangle) \ , \\ \mathrm{length}^{\mathrm{aux}}(a,p',g(\{e\}(a')[p])) & < & \mathrm{length}^{\mathrm{aux}}(a, \mathrm{rec}(a',g), \langle p,p'\rangle) \ . \end{array}$$

We will usually omit the superscript aux of $\mathrm{length}^{\mathrm{aux}}$*.*

Proof: We define $\mathrm{length}(a,p)$ by induction on a,p as

$$\mathrm{length}(a, \mathrm{tot}_{e,a}(p)) := \mathrm{length}^{\mathrm{aux}}(a, e(a), p)$$

where

$$\mathrm{length}^{\mathrm{aux}}(a,e',p) := \sum_{a' \in \mathrm{dom}(\mathrm{calls}_e(a,e',p))} (\mathrm{length}(a', \mathrm{calls}_e(a,e',p)(a')) + 1) \ .$$

The conditions follow obviously.

Theorem 5.5 (Consistency of $\{e\}(a)$, $\{a,e'\}_e^{\mathrm{aux}}$) *(a) If* $p,p' : \{e\}(a)\downarrow$, *then* $p =_{\{e\}(a)\downarrow} p'$.

(b) *If* $p,p' : \{a,e'\}_e^{\mathrm{aux}}\downarrow$ *then* $p =_{\{a,e'\}_e^{\mathrm{aux}}\downarrow} p'$.

(c) $\mathrm{Consistent}(\{e\}(a))$, $\mathrm{Consistent}(\{a,e'\}_e^{\mathrm{aux}})$.

(d) $\mathrm{calls}_e(a,e',p) : (a' : A) \rightharpoonup_{\mathrm{fin}} \{e\}(a)\downarrow$,
$\mathrm{callresults}_e(a,e',p) : (a' : A) \rightharpoonup_{\mathrm{fin}} B(a)$.

Proof: We will write $=$ instead of $=_A$ for some set A. We show (a), (b) simultaneously by induction on $\mathrm{length}(p)$. Then (c), (d) follow by (a), (b). (a) reduces to (b), since by Lem. 5.3 we can assume $p = \mathrm{tot}_{e,a}(p_0)$, $p' = \mathrm{tot}_{e,a}(p'_0)$, $\mathrm{length}(p) = \mathrm{length}(p_0)$, $\mathrm{length}(p') = \mathrm{length}(p'_0)$. So we need only to show (b). Case $e' = \mathrm{return}(b)$. Then $p = * = p'$. Case $e' = \mathrm{rec}(a',g)$. Then $p = \langle p_0, p_1\rangle$, $p' = \langle p'_0, p'_1\rangle$, and we have by IH and Lemma 5.4 (c) $p_0 = p'_0$, $p_1 = p'_1$, therefore $p = p'$.

6 Evaluation of Elements of Rec

In [Set06] we introduced the evaluation of elements of Rec_e applied to arguments $a : A$ as an external operation. We are going to show how to introduce this as an internal operation. This will be done by introducing a reduction relation, s.t., if iterated reduction terminates (or more precisely reduces to itself), we can extract from the resulting element a $b : B(a)$, and have $b = \{e\}(a)[\_]$. The reductions will be carried on elements of a data structure which is close to a monad. The data structure is Rec'_a (as before we keep $A : \mathrm{Set}$ and $B : A \to \mathrm{Set}$ fixed).
We define the following monad like operations:

$$\begin{array}{rcl} \mathrm{return}' & : & (a : A, b : B(a)) \to \mathrm{Rec}'_a \ , \\ \mathrm{return}'_a(b) & := & \mathrm{return}(b) \ , \\ (*) & : & (a : A, a' : A, e : \mathrm{Rec}'_a, h : B(a) \to \mathrm{Rec}'_{a'}) \to \mathrm{Rec}'_{a'} \ , \\ & & \text{written as } e *_{a,a'} h \text{ and defined by induction on } e\text{:} \\ \mathrm{return}(b) *_{a,a'} h & = & h(b) \ , \\ \mathrm{rec}(a'',h') *_{a,a'} h & = & \mathrm{rec}(a'', \lambda b'' : B(a'').h'(b'') *_{a,a'} h) \ . \end{array}$$

Now define

$$
\begin{array}{rcl}
\text{defined} & : & (a : A, \text{Rec}'_{A,B,a}) \to \text{Bool} \ , \\
\text{defined}_a(\text{return}(b)) & := & \text{tt} \ , \\
\text{defined}_a(\text{rec}(a,h)) & := & \text{ff} \ , \\
\text{Defined} & : & (a : A, e : \text{Rec}'_{A,B,a}) \to \text{Set} \ , \\
\text{Defined}_a(e) & := & \text{atom}(\text{defined}_a(e)) \ .
\end{array}
$$

$\text{Defined}_a(e)$ is true, if the computation of e is finished.
We define the function which extracts the resulting element of $B(a)$, if the computation has terminated

$$
\begin{array}{rcl}
\text{Extract} & : & (a : A, e : \text{Rec}'_a, p : \text{Defined}_a(e)) \to B(a) \ , \\
\text{Extract}_a(\text{return}(b), p) & = & b \ , \\
\text{Extract}_a(\text{rec}(a', h), p) & = & \text{efq}_{B(a)}(p) \ .
\end{array}
$$

Next we define, the one step reduction of an element of Rec'_a, assuming an element $e : \text{Rec}$ corresponding to the global function:

$$
\begin{array}{rcl}
\text{Red} & : & (e : \text{Rec}, a : A, e' : \text{Rec}'_a) \to \text{Rec}'_a \ , \\
\text{Red}_{e,a}(\text{return}(b)) & := & \text{return}(b) \ , \\
\text{Red}_{e,a}(\text{rec}(a', h)) & := & e(a') *_{a',a} h \ .
\end{array}
$$

So $\text{return}(b)$ does not evaluate any further. $\text{rec}(a', h)$ reduces to a computation which consists of first a computation of $e(a')$, followed, if the result was b', by a computation of $h(b')$.
Let

$$
\text{Red}^n_{e,a}(b) := \underbrace{\text{Red}^n_{e,a}(\text{Red}^n_{e,a}(\cdots \text{Red}^n_{e,a}(}_{n \text{ times}} b) \cdots)) \ .
$$

Then one can easily see

$$
\text{Defined}_a(\text{Red}^n_{e,a}(b)) \to \forall m \geq n . \text{Red}^m_{e,a}(b) = \text{Red}^n_{e,a}(b) \ . \qquad (*)
$$

We can now define for $a : A$, $e' : \text{Rec}_a$, $e : \text{Rec}$ the partial object $\{a, e'\}^{\text{comp,aux}}_e : \text{Par}(B(a))$ by

$$
\begin{array}{rcl}
\{a, e'\}^{\text{comp,aux}}_e \downarrow & := & \Sigma n : \mathbb{N}.\text{Defined}_a(\text{Red}^n_{e,a}(e')) \ , \\
\{a, e'\}^{\text{comp,aux}}_e [\langle n, p \rangle] & := & \text{Extract}_a(\text{Red}^n_{e,a}(e'), p) : B(a) \ ,
\end{array}
$$

where $\text{Consistent}(\{a, e'\}^{\text{comp,aux}}_e)$ follows by $(*)$. Then we define for $e : \text{Rec}$, $a : A$ $\{e\}^{\text{comp}}(a) : \text{Par}(B(a))$ by

$$
\{e\}^{\text{comp}}(a) := \{a, e(a)\}^{\text{comp,aux}}_e \ .
$$

Assuming $e : \text{Rec}$, $a : A$, $b : B(a)$ has been derived under an empty context, and that the underlying type theory is normalising w.r.t. weak head normal form for closed term, then $\{a, e'\}^{\text{comp,aux}}_e$ and therefore as well $\{e\}^{\text{comp}}(a)$ are externally computable: All we need is to iteratively apply $\text{Red}^n_{e,a}$ iteratively to e' until we obtain $\text{defined}_a(\text{Red}^n_{e,a}(e')) = \text{tt}$, and then compute $\text{Extract}_a(\text{Red}^n_{e,a}(e', \text{true})$. So we see that for every $e : \text{Rec}$ we obtain a partial recursive function $\{e\}^{\text{comp}}(a)$.

7 Correctness of the Evaluation of Elements of Rec

Theorem 7.1 *Assume* $e : \text{Rec}, a : A$, $e' : \text{Rec}'_a$. *Then* *(a)* $\{a, e'\}_e^{\mathrm{comp,aux}} \simeq \{a, e'\}_e^{\mathrm{aux}}$.

(b) $\{e\}^{\mathrm{comp}} \cong \{e\}$.

In order to prove Theorem 7.1, we show first the following lemma:

Lemma 7.2 *Assume* $e : \mathrm{Rec}, a : A, e' : \mathrm{Rec}'_a, a' : A, e'' : B(a) \to \mathrm{Rec}'_{a'}$. *Then we have:*

(a) There exists $q : \mathrm{Defined}'_a(\mathrm{Red}^n_{e,a'}(e' *_{a,a'} e''))$, *if for some* $k, m \leq n$ *s.t.* $k+m = n$ *we have some* $p : \mathrm{Defined}_a(\mathrm{Red}^k_{e,a}(e'))$ *and for* $b := \mathrm{Extract}_a(\mathrm{Red}^k_{e,a}(e'), p)$ *we have some* $p' : \mathrm{Defined}_{a'}(\mathrm{Red}^m_{e,a'}(e''(b)))$.

Furthermore, if such q, k, m, p, p' *exists, then*

$$\mathrm{Extract}_{a'}(\mathrm{Red}^n_{e,a'}(e' *_{a,a'} e'', q) = \mathrm{Extract}_{a'}(\mathrm{Red}^m_{e,a'}(e''(b)), p') \enspace .$$

(b) $\{a, e'\}_e^{\mathrm{aux}}\downarrow \Leftrightarrow \exists n : \mathbb{N}.\mathrm{Defined}(\mathrm{Red}^n_{e,a}(e'))$

(c) For all $p : \{a, e'\}_e^{\mathrm{aux}}\downarrow$, $n : \mathbb{N}$, $q : \mathrm{Defined}(\mathrm{Red}^n_{e,a}(e'))$ *we have*

$$\{a, e'\}_e^{\mathrm{aux}}[p] = \mathrm{Extract}(\mathrm{Red}^n_{e,a}(e'), q) \enspace .$$

If we have shown this lemma, then Theorem 7.1 follows: 7.1 (a) follows immediately by (b) and (c). 7.1 (b) follows by 7.1 (a).

Proof of Lemma 7.2: We will omit indices of Extract, Defined, Red and $*$.

- Proof of *(a)* by induction on n:
  - Case $e' = \mathrm{return}(b)$: Then $\mathrm{Defined}'(\mathrm{Red}^k(e'))$ holds for all k, $\mathrm{Extract}(\mathrm{Red}^k(e'), p) = b$, $\mathrm{Red}^n(e' * e'') = \mathrm{Red}^n(e''(b))$, therefore the assertion holds (with $k := 0, m := n$).
  - Case $e' = \mathrm{rec}(a'', h)$. We have $e' * e'' = \mathrm{rec}(a'', \lambda b'' : B(a'').h(b'') * e'')$. We have $\neg\mathrm{Defined}(\mathrm{Red}^0(e' * e''))$, $\neg\mathrm{Defined}(\mathrm{Red}^0(e'))$. Furthermore

    $$\begin{array}{rcl} \mathrm{Red}^{n+1}(e' * e'') & = & \mathrm{Red}^n(e(a'') * \lambda b'' : B(a'').h(b'') * e'') \enspace , \\ \mathrm{Red}^{k+1}(e') & = & \mathrm{Red}^k(e(a'') * h) \enspace . \end{array}$$

    $\mathrm{Defined}(\mathrm{Red}^{n+1}(e' * e''))$ if and only if
    $\mathrm{Defined}(\mathrm{Red}^n(e(a'') * \lambda b'' : B(a'').h(b'') * e''))$,
    which holds by applying the IH twice if and only if
    there exists $k, l, m \leq n$ s.t. $k+l+m = n$ and there exist $p'' : \mathrm{Defined}(\mathrm{Red}^k(e(a'')))$,
    for $b'' := \mathrm{Extract}(\mathrm{Red}^k(e(a'')), p'')$ there exists $p : \mathrm{Defined}(\mathrm{Red}^l(h(b'')))$,
    and for $b := \mathrm{Extract}(\mathrm{Red}^l(h(b'')), p)$ we have $\mathrm{Defined}(\mathrm{Red}^m(e''(b)))$.
    Again by IH this holds if
    there exists $k', k'' \leq n$ s.t. $k' + k'' = n$ and $p : \mathrm{Defined}(\mathrm{Red}^{k'}(e(a'') * h))$
    and with $b := \mathrm{Extract}(\mathrm{Red}^{k'}(e(a'')*h), p)$ we have $\mathrm{Defined}(\mathrm{Red}^m(e''(b)))$.
    Since $\mathrm{Red}^{k'}(e(a'') * h) = \mathrm{Red}^{k'+1}(\mathrm{rec}(a'', h))$ we obtain the equivalence in (a). The equality statement of (a) follows similarly.
- We show direction "$\Leftarrow$" of (b) and statement (c) simultaneously by induction on n.
  - Case $e' = \mathrm{return}(b)$. We have $\{a, e'\}_e^{\mathrm{aux}}\downarrow$, $\mathrm{Red}^n(e') = e'$ and $\mathrm{Defined}(\mathrm{Red}^n(e'))$ for all n, $\{a, e'\}_e^{\mathrm{aux}}[p] = b = \mathrm{Extract}(\mathrm{Red}^n(e'), p')$.

- Case $e' = \mathrm{rec}(a', h)$. Considering the left sides of the equivalence and equation, we obtain
$\{a, \mathrm{rec}(a', h)\}^{\mathrm{aux}}_e\downarrow = \Sigma p : \{e\}(a')\downarrow.\{a, h(\{e\}(a')[p])\}^{\mathrm{aux}}_e\downarrow$,
therefore $\{a, e'\}^{\mathrm{aux}}_e\downarrow$ iff there exists $p : \{e\}(a')\downarrow$ s.t. with $b := \{e\}(a')[p]$ we get $q : \{a, h(b)\}^{\mathrm{aux}}_e\downarrow$. Furthermore for p, q as above we get
$\{a, e'\}^{\mathrm{aux}}_e[\langle p, q\rangle] = \{a, h(b)\}^{\mathrm{aux}}_e[q]$.
If we consider the right sides of the equivalence and equation we get $\neg(\mathrm{Defined}(\mathrm{Red}^0_{e,a}(e'))$, $\mathrm{Red}^{n+1}_{e,a}(e') = \mathrm{Red}^n_{e,a}(e(a') * h)$, therefore
$\mathrm{Defined}(\mathrm{Red}^{n+1}_{e,a}(e'))$ if and only if there exist $m, m' \leq n.m + m' = n$, $p : \mathrm{Defined}(\mathrm{Red}^{m'}_{e,a'}(e(a'))$ s.t. with $b := \mathrm{Extract}(\mathrm{Red}^{m}_{e,a'}(e(a')), p)$ we get $q : \mathrm{Defined}(\mathrm{Red}^{m'}_{e,a}(h(b)))$.
Furthermore for n, m, m' and a proof $p' : \mathrm{Defined}(\mathrm{Red}^{n+1}_{e,a}(e'))$ obtained from p, q as above we obtain
$\mathrm{Extract}(\mathrm{Red}^{n+1}_{e,a}(e'), p') = \mathrm{Extract}(\mathrm{Red}^{m'}_{e,a}(h(b), q))$.
The statements follow now by the IH.

The statement (b), direction "⇒" follows by a similar argument, but by induction on $p : \{a, e'\}^{\mathrm{aux}}_e\downarrow$.

□

8 Finite Approximations of $\{e\}(a)\downarrow$

We are later going to translate elements of $e : \mathrm{Rec}'_{C,a}$ into an element $e' : \mathrm{Rec}'_{C',a'}$ for suitable C, C', a, a'. We need to establish a correctness statement concerning the relationship between $\{a, e\}^{\mathrm{aux}}_{e_0}$ and $\{a', e'\}^{\mathrm{aux}}_{e'_0}$. The problem is that $e_0 : \mathrm{Rec}_C$ has to be replaced by some $e'_0 : \mathrm{Rec}_{C'}$, and in general it is difficult to define such e'_0 from e_0 (or vice versa). Therefore we introduce a variant of $\{\cdot,\cdot\}^{\mathrm{aux}}_{\cdot}$, where we refer to finite approximations of functions $g : (a : C.A) \to C.B$ rather than elements of Rec_C. Then it will usually be easy to define from g corresponding to e a corresponding function g' corresponding to the translated index e'.

Definition 8.1 ($\{a, e\}^{\mathrm{fin}}_g$) *(a) For $g : (a : A) \rightharpoonup_{\mathrm{fin}} B(a)$, $a : A$, $e : \mathrm{Rec}'_{A,B,a}$ we define $\{a, e\}^{\mathrm{fin}}_g : \mathrm{PrePar}(B(a))$ as follows (by Lemma 8.2 (a) it will follow that $\{a, e\}^{\mathrm{fin}}_g : \mathrm{Par}(B(a))$):*

- *If $e = \mathrm{return}(b)$, then*

$$\begin{array}{rcl} \{a, e\}^{\mathrm{fin}}_g\downarrow & := & \{*\} : \mathrm{Set} \ , \\ \{a, e\}^{\mathrm{fin}}_g[*] & := & b : B(a) \ . \end{array}$$

- *If $e = \mathrm{rec}(a, h)$, then*

$$\begin{array}{l} \{a, e\}^{\mathrm{fin}}_g\downarrow := \Sigma p : \{g\}^{\mathrm{finfun}}(a)\downarrow.\{a, h(\{g\}^{\mathrm{finfun}}(a)[p])\}^{\mathrm{fin}}_g\downarrow : \mathrm{Set} \ , \\ \{a, e\}^{\mathrm{fin}}_g[\langle p, p'\rangle] \ := \ \{a, h(\{g\}^{\mathrm{finfun}}(a)[p])\}^{\mathrm{fin}}_g[p'] \ . \end{array}$$

(b) Assume $g : (a : A) \rightharpoonup_{\mathrm{fin}} B(a)$, $a : A$, $e : \mathrm{Rec}'_{A,B,a}$, $p : \{a, e\}^{\mathrm{fin}}_g\downarrow$. We define $\mathrm{callresults}_g(a, e, p) : (a : A) \rightharpoonup_{\mathrm{fin}} B(a)$ (which will be the finite approximation of g used in p) s.t. $\mathrm{callresults}_g(a, e, p) \subseteq g$ as follows:

$$\begin{array}{rcl} \mathrm{callresults}_g(a, \mathrm{return}(b), *) & := & \emptyset_{\mathrm{finfun}} \ , \\ \mathrm{callresults}_g(a, \mathrm{rec}(a', k), \langle p, p'\rangle) & = & \mathrm{callresults}_g(a, k(b), p')[a' \mapsto b]_q \ , \\ & & \quad \textit{where } b = \{g\}^{\mathrm{finfun}}(a')[p]. \end{array}$$

Here in the second clause q is determined by the fact that $\mathrm{callresults}_g(a, k(b), p') \subseteq g$.

Lemma 8.2 *(a)* Consistent($\{a, e\}^{\text{fin}}_g$).

(b) Assume $g \subseteq g'$. Then $\{a, e\}^{\text{fin}}_g \sqsubseteq \{a, e\}^{\text{fin}}_{g'}$.

(c) Assume $p : \{a, e'\}^{\text{aux}}_e\downarrow$, $g := \text{callresults}_e(a, e', p)$ (which is the finite approximation of e which is used in p). Then $\{a, e'\}^{\text{aux}}_e \simeq \{a, e'\}^{\text{fin}}_g$.

(d) **($\text{callresults}_g(a, e, p)$ is the minimal finite approximation of g used in p)** *Assume $g : (a : A) \rightharpoonup_{\text{fin}} B(a)$, $e : \text{Rec}_{A,B}$, $p : \{a, e'\}^{\text{fin}}_g\downarrow$, and let $g' := \text{callresults}_g(a, e', p)$. Then $\{a, e'\}^{\text{fin}}_g \simeq \{a, e'\}^{\text{fin}}_{g'}$. Furthermore, if $g \subseteq g''$, $p'' : \{a, e\}^{\text{fin}}_{g''}\downarrow$, then $\text{callresults}_g(a, e, p) \cong \text{callresults}_{g''}(a, e, p'')$.*

(e) **($\text{callresults}_e(a, e', p)$ is the minimal finite approximation of e used in p)** *Assume $g : (a : A) \rightharpoonup_{\text{fin}} B(a)$, $e : \text{Rec}_{A,B}$, $g \subseteq \{e\}$. Assume $p : \{a, e'\}^{\text{fin}}_g\downarrow$. Then $\{a, e'\}^{\text{fin}}_g \simeq \{a, e'\}^{\text{aux}}_e$. Furthermore, if $p' : \{a, e'\}^{\text{aux}}_e\downarrow$, then $\text{callresults}_e(a, e', p') \cong \text{callresults}_g(a, e', p)$.*

Proof: Easy induction on e or e'.

9 Sum of Elements of Rec

In this section, we are going to introduce various operations, which allow to combine several recursive functions into one.
We first determine an embedding from elements $e' : \text{Rec}'_{C,a}$ into $e'_0 : \text{Rec}'_{C',g(a)}$ for an homomorphism $g : C \to C'$ (i.e. $g : A \to A'$ s.t. $B(a) = B'(g(a))$). The correctness statement will refer to finite approximations of elements of Rec_e rather than elements of Rec_e. The reason is that $\{a, e'\}^{\text{aux}}_e\downarrow$ and $\{g(a), e'_0\}^{\text{aux}}_{e_0}\downarrow$ refer to different e and e_0, namely $e : \text{Rec}_C$ and $e_0 : \text{Rec}_{C'}$. We do not know how to define e_0, since this function will be defined after the embedding of e' to e'_0 has been completed. However, the relationship between the finite approximations can easily be defined.

Lemma and Definition 9.1 *Assume $A : \text{Set}$ and $C' = \langle A', B' \rangle : \text{Fam(Set)}$, Assume $g : A \to A'$. Let $B := \lambda a : A.B'(g(a)) : A \to \text{Set}$, $C := \langle A, B \rangle : \text{Fam(Set)}$. (When applying this lemma, usually B, B' are given first and we have $a : A \Rightarrow B(a) = B'(g(a))$). Then there exist*

$$\text{emb}_{C,C',g} \quad : \quad (a : A, e : \text{Rec}'_{C,a}) \to \text{Rec}'_{C',g(a)} \quad ,$$

s.t. the following holds:
Assume $inj_g : \text{inj}(g)$. Assume $a : A$, $e : \text{Rec}'_{C,a}$. Let $a_0 := g(a)$, $e_0 := \text{emb}_{C,C',g}(a, e)$.

(a) Assume $h : (a : A) \rightharpoonup_{\text{fin}} B(a)$, $h_0 := h \circ_{inj_g} g^{-1} : (a : A') \rightharpoonup_{\text{fin}} B'(a)$. Then $\{a, e\}^{\text{fin}}_h \sqsubseteq \{a_0, e_0\}^{\text{fin}}_{h_0}$.

(b) Assume $h_0 : (a : A') \rightharpoonup_{\text{fin}} B'(a)$, $p_0 : \{a_0, e_0\}^{\text{fin}}_{h_0}\downarrow$. Let $r_0 := \text{callresults}_{h_0}(a_0, e_0, p_0)$. Then for all $a'_0 : A$ s.t. $\{r_0\}^{\text{finfun}}(a'_0)\downarrow$, we have $a'_0 = g(a')$ for some a'. Let $r : (a : A) \to B(a)$ s.t. $r \circ g^{-1} = r_0$. Then $\{a, e\}^{\text{fin}}_r \simeq \{a_0, e_0\}^{\text{fin}}_{r_0} \simeq \{a_0, e_0\}^{\text{fin}}_{h_0}$.

We usually omit the indices C, C' of $\text{emb}_{C,C',g}$.

Proof: We define $\text{emb}_g(a, e)$ by induction on e and show the statements (a), (b). Note that in (b) we only have to show $\{a, e\}^{\text{fin}}_r\downarrow$. Once this is established, then the remaining equalities follow by (a) and Lem. 8.2 (d).

- Case $e = \text{return}(b)$. Then $\text{emb}_g(a, e) := \text{return}(b)$. The correctness statements are trivial.

- Case $e = \mathrm{rec}(a', k)$. $\mathrm{emb}_g(a, e) := \mathrm{rec}(g(a'), \lambda b : B(a').\mathrm{emb}_g(k(b)))$. For statement (a) assume $p = \langle p', p''\rangle : \{a, e\}^{\mathrm{fin}}_h\downarrow$, and let $e_0 := \mathrm{emb}_g(a, e)$. We have $p' : \{h\}^{\mathrm{finfun}}(a')\downarrow$, and with $b := \{h\}^{\mathrm{finfun}}(a')[p'] : B(a')$, $p'' : \{a, k(b)\}^{\mathrm{fin}}_h\downarrow$. There exists $p'_0 : \{h_0\}^{\mathrm{finfun}}(a'_0)\downarrow$, and we have $\{h_0\}^{\mathrm{finfun}}(a'_0)[p'_0] = b$. By IH for $k(b)$ there exists $p''_0 : \{a_0, \mathrm{emb}_g(k(b))\}^{\mathrm{fin}}_{h_0}\downarrow$. Therefore $\langle p'_0, p''_0\rangle : \{a_0, e_0\}^{\mathrm{fin}}_{h_0}\downarrow$, and we get $\{a, e\}^{\mathrm{fin}}_h[\_] = \{a, k(b)\}^{\mathrm{fin}}_h[\_] = \{a_0, \mathrm{emb}_g(k(b))\}^{\mathrm{fin}}_{h_0}[\_] = \{a_0, e_0\}^{\mathrm{fin}}_{h_0}[\_]$. For assertion (b) assume $p_0 = \langle p'_0, p''_0\rangle : \{a_0, h_0\}^{\mathrm{fin}}_{e_0}\downarrow$. We have $p'_0 : \{h_0\}^{\mathrm{finfun}}(a'_0)\downarrow$, and with $b := \{h_0\}^{\mathrm{finfun}}(a'_0)[p'_0] : B(a')$, $p''_0 : \{a_0, \mathrm{emb}(k(b))\}^{\mathrm{fin}}_{h_0}\downarrow$. Let $r'_0 := \mathrm{callresults}_{h_0}(\mathrm{emb}(k(b)), p''_0)$. Let $r_0 := r'_0[a'_0 \mapsto b]$. By IH there exists r' s.t. $r' \circ g^{-1} = r'_0$, $p''_1 : \{a, k(b)\}^{\mathrm{fin}}_{r'}\downarrow$. Let $r := r'[a' \mapsto b]$, which is consistent since r_0 is consistent and g is injective. $r \circ g^{-1} = r_0$. There exist $p'' : \{a, k(b)\}^{\mathrm{fin}}_r\downarrow$, $p' : \{r\}^{\mathrm{finfun}}(a')\downarrow$, and we get therefore $\langle p', p''\rangle : \{a, e\}^{\mathrm{fin}}_r\downarrow$.

$\square$

Definition 9.2 ((Embedding from C_i into the disjoint union of C_i) *(a) For $C, C' : \mathrm{Fam}(\mathrm{Set})$, $C = \langle A, B\rangle$, $C' = \langle A', B'\rangle$*

$$\begin{array}{rcl} \mathrm{emb}_{\mathrm{inl},C,C'} & := & \mathrm{emb}_{C,C\oplus C',\lambda x.\mathrm{inl}(x)} \\ & : & (a : A, \mathrm{Rec}'_{C,a}) \to \mathrm{Rec}'_{C\oplus C',\mathrm{inl}(a)} \ , \\ \mathrm{emb}_{\mathrm{inr},C,C'} & := & \mathrm{emb}_{C',C\oplus C',\lambda x.\mathrm{inr}(x)} \\ & : & (a : A', \mathrm{Rec}'_{C',a}) \to \mathrm{Rec}'_{C\oplus C',\mathrm{inr}(a)} \ . \end{array}$$

(b) For $I : \mathrm{Set}$, $C : I \to \mathrm{Fam}(\mathrm{Set})$, $C_i = \langle A_i, B_i\rangle$, $i : I$ let

$$\begin{array}{rcl} \mathrm{emb}_{i,C} & := & \mathrm{emb}_{C_i,\oplus_{i:I}C_i,\lambda x.\mathrm{in}_i(x)} \\ & : & (a : A_i, e : \mathrm{Rec}'_{C_i,a}) \to \mathrm{Rec}'_{\oplus_{i:I}C_i,\mathrm{in}_i(a)} \ . \end{array}$$

Lemma and Definition 9.3 *Assume $C = \langle A, B\rangle : \mathrm{Fam}(\mathrm{Set})$, $C' = \langle A', B'\rangle : \mathrm{Fam}(\mathrm{Set})$.*

(a) Assume $e : \mathrm{Rec}_C$, $e' : \mathrm{Rec}_{C'}$. Then we can define $e \oplus_{C,C'} e' : \mathrm{Rec}_{C\oplus C'}$ s.t. the following holds (we usually omit the indices C, C' of $\oplus$):

(i) For $a : A$, $\{e\}(a) \simeq \{e \oplus e'\}(\mathrm{inl}(a))$.

(ii) For $a : A'$, $\{e'\}(a) \simeq \{e \oplus e'\}(\mathrm{inr}(a))$.

Note that we overload the notation $\oplus$.

(b) Assume $e : (a : A) \to \mathrm{Rec}'_{C\oplus C',\mathrm{inl}(a)}$, $e' : \mathrm{Rec}_{C'}$. Then we can define $e \oplus^{\mathrm{inr}}_{C,C'} e'$ s.t. the following holds (we usually omit the indices C, C' of $\oplus^{\mathrm{inr}}$):

(i) For $a : A$, $\{e \oplus e'\}(\mathrm{inl}(a)) \simeq \{\mathrm{inl}(a), e(a)\}^{\mathrm{aux}}_{e\oplus e'}$.

(ii) For $a : A'$, $\{e'\}(a) \simeq \{e \oplus e'\}(\mathrm{inr}(a))$.

Again we usually omit the indices C, C' of $\oplus$. Note that we overload the symbol $\oplus$.

(c) Assume $e : \mathrm{Rec}_C$, $e' : (a : A') \to \mathrm{Rec}'_{C\oplus C',\mathrm{inr}(a)}$, Then we can define $e \oplus^{\mathrm{inl}}_{C,C'} e'$ s.t. a similar statement as in (b) holds. We usually omit the indices C, C' of $\oplus^{\mathrm{inl}}$.

Proof: *(a)*:

$$\begin{array}{lrcl} e \oplus e' := \lambda x.\mathrm{case}\ x\ \mathrm{of} & & & \\ & \mathrm{inl}(a) & \to & \mathrm{emb}_{\mathrm{inl}}(a, e(a)) \\ & \mathrm{inr}(a) & \to & \mathrm{emb}_{\mathrm{inr}}(a, e'(a)) \ . \end{array}$$

We show

(1) $\forall a : A.\forall p : \{e\}(a)\downarrow.\exists p' : \{e \oplus e'\}(\mathrm{inl}(a))\downarrow.\{e\}(a)[p] = \{e \oplus e'\}(\mathrm{inl}(a))[p']$.

(2) $\forall a : A.\forall p : \{e \oplus e'\}(\mathrm{inl}(a))\downarrow.\exists p' : \{e\}(a)\downarrow.\{e\}(a)[p'] = \{e \oplus e'\}(\mathrm{inl}(a))[p]$.

by induction on length(p). Then we obtain (i) (using uniqueness of elements of $\{e\}(a)\downarrow$). Assertion (ii) follow similarly.
Proof of (1): Assume $a : A$, $p : \{e\}(a)\downarrow$, $k := \mathrm{callresults}_e(a, e(a), p)$. There exists $p' : \{a, e(a)\}^{\mathrm{fin}}_k\downarrow$, and therefore
$p'_0 : \{\mathrm{inl}(a), \mathrm{emb}_{\mathrm{inl}}(a, e(a))\}^{\mathrm{fin}}_{k\circ\mathrm{inl}^{-1}}\downarrow = \{\mathrm{inl}(a), (e \oplus e')(\mathrm{inl}(a))\}^{\mathrm{fin}}_{k\circ\mathrm{inl}^{-1}}\downarrow$.
By IH we have for $a' : A$ s.t. $\{k\}^{\mathrm{finfun}}(a')\downarrow$
$\{k \circ \mathrm{inl}^{-1}\}^{\mathrm{finfun}}(\mathrm{inl}(a')) \simeq \{k\}^{\mathrm{finfun}}(a') \simeq \{e\}(a') \simeq \{e \oplus e'\}(\mathrm{inl}(a'))$.
Therefore we obtain from p'_0 some $p_0 : \{\mathrm{inl}(a), (e \oplus e')(\mathrm{inl}(a))\}^{\mathrm{aux}}_{e\oplus e'}\downarrow$ and
$\{e\}(a)[p] = \{a, e(a)\}^{\mathrm{fin}}_k[p'] = \{\mathrm{inl}(a), (e \oplus e')(\mathrm{inl}(a))\}^{\mathrm{fin}}_{k\circ\mathrm{inl}^{-1}}[p'_0]$
$= \{\mathrm{inl}(a), (e \oplus e')(\mathrm{inl}(a))\}^{\mathrm{aux}}_{e\oplus e'}[p_0] = \{e \oplus e'\}(\mathrm{inl}(a))[\mathrm{tot}_{e\oplus e',a_0}(p_0)]$.
Statement (2) can be shown similarly.
(b):

$$\begin{array}{lll} e \oplus e' := \lambda x.\mathrm{case}\ x\ \mathrm{of} & & \\ \qquad \mathrm{inl}(a) & \to & e(a) \\ \qquad \mathrm{inr}(a) & \to & \mathrm{emb}_{\mathrm{inr}}(a, e'(a)) \ . \end{array}$$

(i) follows by definition and (ii) follows as for (a).
(c) follows similarly.

$\square$

We introduce now similar operations for the potentially infinite disjoint union of families of sets.

Lemma and Definition 9.4 *Assume* I : Set*,* $C : I \to \mathrm{Fam(Set)}$*, written as* C_i *for* $C(i)$*. Let* $C_i = \langle A_i, B_i\rangle$*. Assume* $e : (i : I) \to \mathrm{Rec}_{C_i}$*. Then we can define* $\oplus_{i:I} e_i : \mathrm{Rec}_{\oplus_{i:I} C_i}$ *s.t.*

$$\{e_i\}(a) \simeq \{\oplus_{i:I} e_i\}(\mathrm{in}_i(a)) \ .$$

Proof: Straightforward generalisation of Lemma and Definition 9.3 (a).

10 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 $\langle C, D\rangle$: Fam(Set). If we want to allow reference to arbitrary such functions, we will end up with $\mathrm{Rec}_{A,B}$: Type instead of $\mathrm{Rec}_{A,B}$: 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. There are two solutions for this problems.

- One solution is to restrict the domain and codomain of partial recursive functions used to elements of a universe. Then $\mathrm{Rec}_{A,B}$ will 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. $\mathrm{U} := \mathbb{N}$: Set and for $n : \mathbb{N}$ $\mathrm{T}(n) := \mathbb{N}^n$: Set.

- The other solution will be given in Sect. 13, where we will show that we can always restrict the domain and codomain of other functions used to specific sets, and that therefore Rec can be replaced by a set.

There are two alternative ways of dealing with references 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 $f : (a : A) \rightharpoonup B(a)$ which makes use of $g : (a : A') \rightharpoonup B'(a)$, we combine f, g into one function $fg : (a : A'') \rightharpoonup B''(a)$, where $C := \langle A, B\rangle$, $C' := \langle A', B'\rangle$, $C'' := C \oplus C'$, $C'' = \langle A'', B''\rangle$. Let f be given as an element of $(a : A) \to \mathrm{Rec}'_{C'',\mathrm{inl}(a)}$, g be given as an element of $\mathrm{Rec}_{C'}$. Then we can represent f and g as one function $fg := f \oplus^{\mathrm{inr}} g$. In general we define

$$\mathrm{Rec}^{+}_{0,C} := \Sigma C' : \mathrm{Fam}(\mathrm{Set}).\mathrm{Rec}_{C\oplus C'} \ ,$$

   and for $e = \langle C', e'\rangle : \mathrm{Rec}^{+}_{0,A,B}$ we define

$$\begin{array}{rcl} \{e\}_0 & : & A \to \mathrm{PrePar}(B(a)) \ , \\ \{e\}_0(a)\downarrow & := & \{e'\}_0(\mathrm{inl}(a))\downarrow \ , \\ \{e\}_0(a)[p] & := & \{e'\}_0(\mathrm{inl}(a))[p] \ . \end{array}$$

2. The approach which is makes it easier to define partial recursive functions is to extend Rec by a constructor which calls a partial recursive function having an arbitrary type. So we extend Rec_C, $\mathrm{Rec}'_{C,a}$ to types $\mathrm{Rec}^{+}_{1,C}$, $\mathrm{Rec}^{+,\prime}_{1,C,a}$ with an additional constructor call as follows (We write Arg_1, Eval_1, $\{\cdot\}_1$ for the variants of Arg, Eval, $\{\cdot\}$, respectively, which refer to Rec^{+}_{1}):

$$\begin{array}{l} \mathrm{call} : (C' : \mathrm{Fam}(\mathrm{Set}), e : \mathrm{Rec}^{+}_{1,C'}, a' : C'.A, g : C'.B(a') \to \mathrm{Rec}^{+,\prime}_{1,C,a}) \\ \qquad \to \mathrm{Rec}^{+,\prime}_{1,C,a} \ , \\ \mathrm{Arg}_{1,a,\mathrm{call}(C',e,a',g)}(X,Y) = \Sigma p : \{e\}_{1,C'}(a')\downarrow.\mathrm{Arg}_{1,a,g(\{e\}_{1,C'}(a')[p])}(X,Y) \ , \\ \mathrm{Eval}_{1,a,\mathrm{call}(C',e,a',g)}(X,Y,\langle p,q\rangle) = \mathrm{Eval}_{1,a,g(\{e\}_{1,C'}(a')[p])}(X,Y,q) \ . \end{array}$$

   If we extend $\{a,e'\}^{\mathrm{aux}}_{e}$ in a straightforward way to an operation $\{a,e'\}^{\mathrm{aux}}_{1,e}$ having arguments $e : \mathrm{Rec}^{+}_{1}$, $a : A$, $e' : \mathrm{Rec}^{+,\prime}_{1,a}$, then we obtain in the special case $e' = \mathrm{call}(C', e'', a', g)$

$$\begin{array}{rcl} \{a,e'\}^{\mathrm{aux}}_{1,e}\downarrow & = & \Sigma p : \{e''\}_{1,C'}(a')\downarrow.(\{a, g(\{e''\}_{1,C'}(a')[p])\}^{\mathrm{aux}}_{1,e}\downarrow \ , \\ \{a,e'\}^{\mathrm{aux}}_{1,e}[\langle p,q\rangle] & = & \{a, g(\{e''\}_{1,C'}(a')[p])\}^{\mathrm{aux}}_{1,e}[q] \ . \end{array}$$

   We defined as well the variant $\{\cdot,\cdot\}^{\mathrm{fin}}_{1,\cdot}$ of $\{\cdot,\cdot\}^{\mathrm{aux}}_{\cdot}$ using finitary approximations of functions $(a : A) \to B(a)$. In case of $e = \mathrm{return}(b)$ or $e = \mathrm{rec}(a,g)$, it is defined as for $\{\cdot,\cdot\}^{\mathrm{fin}}_{\cdot}$. In case of $e = \mathrm{call}(C', e'', a', g)$, the definition is as follows:

$$\begin{array}{rcl} \{a,e'\}^{\mathrm{fin}}_{1,h}\downarrow & = & \Sigma p : \{e''\}_{1,C'}(a')\downarrow.(\{a, g(\{e''\}_{1,C'}(a')[p])\}^{\mathrm{fin}}_{1,h}\downarrow) \ , \\ \{a,e'\}^{\mathrm{fin}}_{1,h}[\langle p,q\rangle] & = & \{a, g(\{e''\}_{1,C'}(a')[p])\}^{\mathrm{fin}}_{1,h}[q] \ . \end{array}$$

   Furthermore, $\mathrm{calls}_e(a,e',p)$ is extended to $\mathrm{calls}_{1,e}(a,e',p)$ for $e : \mathrm{Rec}^{+}_{1}$, $a : A$, $e' : \mathrm{Rec}^{+,\prime}_{1,a}$, $p : \{a,e'\}^{\mathrm{aux}}_{1,e}\downarrow$ by extending the previous definition and defining

$$\mathrm{calls}_{1,e}(a, \mathrm{call}(C', e'', a', g'), \langle p, p'\rangle) := \mathrm{calls}_{1,e}(g'(\{e''\}_1(a')[p]), p') \ .$$

   Note that with this approach $\mathrm{Rec}^{+}_{1,C}$: Type are defined simultaneously for all C : Fam(Set) and simultaneously with $\{\cdot\}_{1,C}(\cdot)\downarrow$, $\{\cdot\}_{1,C}(\cdot)[\cdot]$, Arg_1, Eval_1.

   The induction principle for $\mathrm{Rec}^{+,\prime}_{1,C,a}$ is now simultaneously for all $\langle C, a, e\rangle$ s.t. C : Fam(Set), $a : C.A$, and $e : \mathrm{Rec}^{+,\prime}_{1,C,a}$. In case of $\mathrm{call}(C', e'', a', g')$:

$\mathrm{Rec}^{+,'}_{1,C,a}$ the induction hypothesis is for $\langle C', e''(a''), a'\rangle$ (for all $a'' : C'.A$) and $\langle C, g'(b), a\rangle$ for $b : C'.B(a')$.

The induction principle for $p : \{e\}_1(a)\downarrow$ will not refer to $\mathrm{calls}_{1,e}(a, e', p)$, but to a finite set (which we will not define explicitly here; more precisely it is a type rather than a set), which determines all $\langle C', e', a', p''\rangle$ s.t. $e' : \mathrm{Rec}^{+}_{1,C'}$, $a' : C'.A$, $p'' : \{e'\}_{1,C'}(a')\downarrow$ are referred to in p. So in case of $\langle \mathrm{call}(C', e'', a', g'), \langle p, p'\rangle\rangle$, this contains $\langle C', e'', a', p\rangle$ and the elements referred to in p'.

We can define for $e : \mathrm{Rec}^{+}_{1,C}$ a course of value recursion principle: Define for $e : \mathrm{Rec}^{+,'}_{1,C,a}$ all pairs $\langle C', e'\rangle$ s.t. $C' : \mathrm{Fam}(\mathrm{Set})$, $e' : \mathrm{Rec}^{+,'}_{1,C'}$ occurs in e, and call them subterms of e. For instance in case of $\mathrm{call}(C', e', a', g')$, these subterms would be $\langle C', e'\rangle$, all subterms of $e'(a'')$ (where $a'' : C'.A$), and all subterms of $\langle C, g(b)\rangle$ (where $b : C'.B(a')$). We sometimes talk more loosely as e' instead of $\langle C', e'\rangle$ as being a subterm of e. The subterms of $e : \mathrm{Rec}^{+}_{1,C}$ are the subterms of $e(a)$ for $a : C.A$. Then course of value induction states that a property holds if from the property holding for all subterms of e we can infer the property for e itself.

Remark 10.1 (Evaluation of elements of Rec^{+}_{0}, Rec^{+}_{1}) *(a) We can easily lift the evaluation of elements of* $e : \mathrm{Rec}_{A,B}$ *to elements of* $e : \mathrm{Rec}^{+}_{0,A,B}$ *by defining* $\{\cdot\}^{\mathrm{comp}}_{0} : (e : \mathrm{Rec}^{+}_{0,A,B}, a : A) \to \mathrm{Par}(B(a))$, $\{e\}^{\mathrm{comp}}_{0}(a) := \{\pi_1(e)\}^{\mathrm{comp}}(\mathrm{inl}(a))$. *From the evaluation method for* $\{\cdot\}^{\mathrm{comp}}$ *we obtain therefore an evaluation method for* $\{\cdot\}^{\mathrm{comp}}_{0}$ *and therefore for* $\{\cdot\}_0$.

(b) In case of $\mathrm{Rec}^{+}_{1,A,B}$*, to define such an evaluation in a direct way would require a bit more work, although it is not too difficult. However, since in the next section we show how to translate elements of* Rec^{+}_{1} *into elements of* Rec^{+}_{0}*, we obtain via this translation and (a) an evaluation method for* Rec^{+}_{1}.

11 Proof of the Equivalence of the Two Versions of Rec^{+}.

We show that Rec^{+}_{0} and Rec^{+}_{1} are equivalent.

Theorem 11.1 *There exist functions* $\mathrm{trans}^{+}_{0,1} : \mathrm{Rec}^{+}_{0} \to \mathrm{Rec}^{+}_{1}$ *and* $\mathrm{trans}^{+}_{1,0} : \mathrm{Rec}^{+}_{1} \to \mathrm{Rec}^{+}_{0}$ *s.t. for* $e : \mathrm{Rec}^{+}_{0}$, $e' : \mathrm{Rec}^{+}_{1}$, $a : A$, *the following holds:*

(a) $\{e\}_0 \cong \{\mathrm{trans}^{+}_{0,1}(e)\}_1$.

(b) $\{e'\}_1 \cong \{\mathrm{trans}^{+}_{1,0}(e')\}_0$.

Definition of $\mathrm{trans}^{+}_{0,1}$ and proof of (a): First note that Rec can be embedded in a straightforward way into Rec^{+}_{1} (by induction on Rec) s.t. $\{e\}(\cdot)\downarrow$ and $\{e\}(\cdot)[\cdot]$ are preserved in the usual way (this proof is by induction on the length of the element of $\{e\}(a)\downarrow$ or $\{e\}_1(a)\downarrow$ respectively). We identify Rec with the subset of Rec^{+}_{1} and do not mention in the following the embedding operation.
We first define $\mathrm{trans}^{+}_{0,1}$: Assume $e : \mathrm{Rec}^{+}_{0,C}$, $e = \langle C', e'\rangle$, where $C' : \mathrm{Fam}(\mathrm{Set})$, $e' : \mathrm{Rec}_{C\oplus C'}$. Assume $C = \langle A, B\rangle$, $C' = \langle A', B'\rangle$. Let $\mathrm{trans}^{+}_{0,1}(e) := e_0 : \mathrm{Rec}^{+}_{1,C}$, where

$$e_0 := \lambda a : A.\mathrm{call}(C \oplus C', e', \mathrm{inl}(a), \lambda b : B(a).\mathrm{return}(b)) \ .$$

We have (using the identification of Rec with a subset of Rec^{+}_{0}) that

$$\{e_0\}_1(a)\downarrow = \{e'\}_1(\mathrm{inl}(a))\downarrow \times \{*\} = \{e\}_0(a)\downarrow \times \{*\} \ ,$$
$$\{e_0\}_1(a)[\langle p, *\rangle] = \{e'\}_1(\mathrm{inl}(a))[p] = \{e\}_0(a)[p] \ .$$

Therefore the assertion follows.

Definition of $\mathrm{trans}^+_{1,0}$ and proof of (b): The idea for the translation is to to first collect all $\langle C'_i, e'_i\rangle$, s.t. $\mathrm{call}(C'_i, e'_i, \ldots)$ occurs in a given element of Rec^+_1, into one function $\langle \oplus_{i:I} C'_i, \oplus_{i:I} e'_i\rangle$. Then replace, using the IH, e'_i by $e''_i : \mathrm{Rec}^+_0$ which correspond to some $e'''_i : \mathrm{Rec}_{C'''}$. Now construct from e and e'''_i an $\widetilde{e} : \mathrm{Rec}_{C \oplus C'''}$, s.t. $\{\widetilde{e}\}(\mathrm{inl}(a)) \simeq \{e\}_1(a)$, where references to some $\{e'_i\}_1(a')$ in e by using the constructor call are replaced by recursive calls to $\mathrm{inr}(\langle i, \mathrm{inl}(a')\rangle)$, and $\{\widetilde{e}\}(\mathrm{inr}(\langle i, a\rangle)) \simeq \{e'''_i\}(a)$ is obtained by lifting of e'''_i.

We first define a translation of elements of $\mathrm{Rec}^{+,'}_{1,C,a}$. For this we need a corresponding notation for Rec^+_0. In this notation, we refer to a family of functions $e'' : (i : D.I) \to \mathrm{Rec}^+_{1,D.C(i)}$, which are the functions $e''(i)$ s.t. $\mathrm{call}(D.C(i), e''(i), \ldots)$ occurs in the code e for the partial recursive function to be translated. Note that $e''(i) : \mathrm{Rec}^+_1$, and not $e''(i) : \mathrm{Rec}^+_0$: at this stage we do not know yet how to translate $e''(i)$ into an element of Rec^+_0. This will then be done in the proof of the theorem by using the IH (for which we need the fact that $e''(i)$ is a subterm of e).

Definition 11.2 *(a) Assume* $D = \langle I, C\rangle : \mathrm{Fam}(\mathrm{Fam}(\mathrm{Set}))$, *i.e.* $I : \mathrm{Set}$, $C : I \to \mathrm{Fam}(\mathrm{Set})$. *Then* $D.I := I$, $D.C := C$, $\oplus D := \oplus_{i:I} C(i)$.

(b) If $C : \mathrm{Fam}(\mathrm{Set})$, $D : \mathrm{Fam}(\mathrm{Fam}(\mathrm{Set}))$, *then* $\oplus(C, D) := C \oplus (\oplus D)$*)*

(c) Assume $C = \langle A, B\rangle : \mathrm{Fam}(\mathrm{Set})$, $a : A$.

$$\mathrm{Rec}^{+,'}_{0,C,a} := \Sigma D : \mathrm{Fam}(\mathrm{Fam}(\mathrm{Set})).\mathrm{Rec}'_{\oplus(C,D),\mathrm{inl}(a)} \times (i : D.I) \to \mathrm{Rec}^+_{1,D.C(i)} \ .$$

The translation from $\mathrm{Rec}^{+,'}_{1,C,a}$ to $\mathrm{Rec}^{+,'}_{0,C,a}$ is as follows:

Lemma and Definition 11.3 *Assume* $C = \langle A, B\rangle : \mathrm{Fam}(\mathrm{Set})$, $a : A$. *We can define*

$$\mathrm{trans}'_{1,0} : (a : A, e : \mathrm{Rec}^{+,'}_{1,C,a}) \to \mathrm{Rec}^{+,'}_{0,C,a} \ ,$$

s.t. the following holds: Assume $\mathrm{trans}'_{1,0}(a, e) = \langle D, e', e''\rangle$. *Let* $\widetilde{C}' := \oplus(D)$, $\widetilde{C}' = \langle \widetilde{A}', \widetilde{B}'\rangle$, $\widetilde{C} := \oplus(C, D)$, $\widetilde{C} = \langle \widetilde{A}, \widetilde{B}\rangle$.

(a) Assume $g : (a : A) \rightharpoonup_{\mathrm{fin}} B(a)$ *s.t.* $p : \{a, e\}^{\mathrm{fin}}_{1,g}\downarrow$. *Then there exists* $g' : (a : \widetilde{A}) \rightharpoonup_{\mathrm{fin}} \widetilde{B}(a)$ *s.t. the following holds:*

- $\{g'\}^{\mathrm{finfun}}(\mathrm{inl}(a')) \sqsubseteq \{g\}^{\mathrm{finfun}}(a')$.
- $\{g'\}^{\mathrm{finfun}}(\mathrm{inr}(\mathrm{in}_i(a'))) \sqsubseteq \{e''(i)\}_1(a')$.
- $\{a, e\}^{\mathrm{fin}}_g \simeq \{\mathrm{inl}(a), e'\}^{\mathrm{fin}}_{1,g'}$.

(b) Assume $g' : (a : \widetilde{A}) \rightharpoonup_{\mathrm{fin}} \widetilde{B}(a)$ *s.t.* $p' : \{\mathrm{inl}(a), e'\}^{\mathrm{fin}}_{1,g'}\downarrow$.
Assume that $\{g'\}^{\mathrm{finfun}}(\mathrm{inr}(\mathrm{in}_i(a'))) \sqsubseteq \{e''(i)\}_1(a')$. *Then there exists* $g : (a : A) \rightharpoonup_{\mathrm{fin}} B(a)$ *s.t. the following holds:*

- $\{g\}^{\mathrm{finfun}}(a') \sqsubseteq \{g'\}^{\mathrm{finfun}}(\mathrm{inl}(a'))$.
- $\{a, e\}^{\mathrm{fin}}_g \simeq \{\mathrm{inl}(a), e'\}^{\mathrm{fin}}_{1,g'}$.

(c) $e''(i)$ *are subterms of* e.

Proof of Lemma and Definition 11.3: We write briefly trans' instead of $\mathrm{trans}'_{1,0}$ in this proof, which is carried out by induction on e.

We note that if g' is defined depending on g s.t. the conditions of statement (a) hold, or g is defined depending on g' s.t. the conditions of statement (b) hold, then it is automatically consistent, even if g, g' was introduced only as a multivalued finite function rather than a finite function. So we do not have to care about giving proofs of consistency.

- Case $e = \mathrm{return}(b)$: $\mathrm{trans}'(a,e) := \langle \emptyset, \mathrm{return}(b), \lambda x.\mathrm{efq}(x)\rangle$. The assertions follow trivially.

- Case $e = \mathrm{rec}(a', h)$: We make the following definitions

$$
\begin{array}{rcl}
\mathrm{trans}'(a,h(b)) & =: & \langle D_b, e'_b, e''_b \rangle \qquad\qquad (\text{for } b : B(a'); \text{ given by IH}) \ , \\
D_b & =: & \langle I_b, C'_b \rangle \ , \qquad\qquad\qquad\qquad C'_b \quad =: \quad \langle A'_b, B'_b \rangle \ , \\
I_0 & := & \Sigma_{b:B(a)} I_b : \mathrm{Set} \ , \\
C'_0 & : & I_0 \to \mathrm{Fam}(\mathrm{Set}) \ , \quad C'_0(\langle b,i \rangle) \quad := \quad C'_b(i) \ , \\
e''_0 & : & (i : I_0) \to \mathrm{Rec}^+_{1,C'_0(i)} \ , \\
e''_0(\langle b,i\rangle) & := & e''_b(i) \ , \\
D_0 & := & \langle I_0, C'_0 \rangle : \mathrm{Fam}(\mathrm{Set}) \ , \\
\widetilde{C}' & := & \oplus(D_0) \ , \qquad\qquad\qquad\qquad \widetilde{C}' \quad =: \quad \langle \widetilde{A}', \widetilde{B}' \rangle \ , \\
\widetilde{C} & := & \oplus(C, D_0) \ , \qquad\qquad\qquad \widetilde{C} \quad =: \quad \langle \widetilde{A}, \widetilde{B} \rangle \ , \\
\widetilde{C}'_b & := & \oplus(D_b) \ , \qquad\qquad\qquad\qquad \widetilde{C}'_b \quad =: \quad \langle \widetilde{A}'_b, \widetilde{B}'_b \rangle \ , \\
\widetilde{C}_b & := & \oplus(C, D_b) \ , \qquad\qquad\qquad \widetilde{C}_b \quad =: \quad \langle \widetilde{A}_b, \widetilde{B}_b \rangle \ , \\
k_b & : & \widetilde{A}_b \to \widetilde{A} \ , \qquad\qquad k_b(\mathrm{inl}(a)) \quad := \quad \mathrm{inl}(a) \ , \\
k_b(\mathrm{inr}(\mathrm{in}_i(a))) & := & \mathrm{inr}(\mathrm{in}_{\langle b,i\rangle}(a)) \ , \qquad (\text{note that } k_b \text{ is injective}) \ , \\
e'_0 & := & \mathrm{rec}(\mathrm{inl}(a'), \lambda b : B(a').\mathrm{emb}_{k_b}(e'_b)) \ .
\end{array}
$$

Then

$$\mathrm{trans}'(a,e) := \langle D_0, e'_0, e''_0 \rangle \ .$$

Condition (c) holds by IH.

We show (a): Assume $g : (a : A) \rightharpoonup_{\mathrm{fin}} B(a)$, $\langle p_0, p_1 \rangle : \{a, e\}^{\mathrm{fin}}_{1,g}\downarrow$, i.e. $p_0 : \{g\}^{\mathrm{finfun}}(a')\downarrow$, and with $b := \{g\}^{\mathrm{finfun}}(\mathrm{inl}(a'))[p_0]$, $p_1 : \{a, h(b)\}^{\mathrm{fin}}_{1,g}\downarrow$. By IH we have $g'_1 : (a : \widetilde{A}_b) \rightharpoonup_{\mathrm{fin}} \widetilde{B}_b(a)$ s.t. the assertion holds for $k(b), g, g'_1$. Let $g'_2 := g'_1 \circ k^{-1}$, $g' := g'_2[\mathrm{inl}(a') \mapsto b]$. One can easily see that g' is in relationship to g according to (a) and therefore consistent. By IH we have $p'_1 : \{\mathrm{inl}(a), e'_b\}^{\mathrm{fin}}_{g'_1}\downarrow$, therefore $p''_1 : \{\mathrm{inl}(a), \mathrm{emb}_{k_b}(e'_b)\}^{\mathrm{fin}}_{g'_2}\downarrow$, therefore $p'''_1 : \{\mathrm{inl}(a), \mathrm{emb}_{k_b}(e'_b)\}^{\mathrm{fin}}_{g'}\downarrow$. Furthermore, $p'''_0 : \{g'\}^{\mathrm{finfun}}(\mathrm{inl}(a'))\downarrow$. Therefore $p''' := \langle p'''_0, p'''_1\rangle : \{\mathrm{inl}(a), e'_0\}^{\mathrm{fin}}_{g'}\downarrow$, and we have $\{\mathrm{inl}(a), e'_0\}^{\mathrm{fin}}_{g'}[p'''] = \{\mathrm{inl}(a), \mathrm{emb}_{k_b}(e'_b)\}^{\mathrm{fin}}_{g'}[p'''_1] = \{\mathrm{inl}(a), \mathrm{emb}_{k_b}(e'_b)\}^{\mathrm{fin}}_{g'_2}[p''_1] = \{\mathrm{inl}(a), e'_b\}^{\mathrm{fin}}_{g'_1}[p'_1] = \{a, h(b)\}^{\mathrm{fin}}_{1,g}[p_1] = \{a, e\}^{\mathrm{fin}}_{1,g}[p]$.

(b) follows similarly.

Case $e = \mathrm{call}(\widehat{C}, \widehat{e}, \widehat{a}, h)$. Let $\widehat{C} = \langle \widehat{A}, \widehat{B} \rangle$, and for $b : \widehat{B}(\widehat{a})$, $\mathrm{trans}'(a, h(b)) = \langle D_b, e'_b, e''_b \rangle$, $D_b = \langle I_b, C'_b \rangle$. Let $D_0 := \langle I_0, C'_0 \rangle$ where $I_0 := \{*\} + \Sigma_{b:B(a)}.I_b$, $C'_0(\mathrm{inl}(*)) = \widehat{C}$, $C'_0(\mathrm{inr}(\langle b, i\rangle)) = C_b(i)$. Let $e''_0 : (i : I_0) \to \mathrm{Rec}^+_{1,C'_0(i)}$, $e''_0(\mathrm{inl}(*)) := \widehat{e}$, $e''_0(\mathrm{inr}(\mathrm{in}(\langle b,i\rangle))) = e''_b(i)$. Let $\widetilde{C}' = \langle \widetilde{A}', \widetilde{B}' \rangle$, $\widetilde{C} = \langle \widetilde{A}, \widetilde{B} \rangle$, $\widetilde{C}'_b = \langle \widetilde{A}'_b, \widetilde{B}'_b \rangle$, $\widetilde{C}_b = \langle \widetilde{A}_b, \widetilde{B}_b \rangle$ as before. Let $k_b : \widetilde{A}_b \to \widetilde{A}$, $k_b(\mathrm{inl}(a)) = \mathrm{inl}(a)$, $k_b(\mathrm{inr}(\mathrm{in}_i(a))) = \mathrm{inr}(\mathrm{in}_{\mathrm{inr}(\langle b,i\rangle)}(a))$. k_b is injective. Then define

$$
\begin{array}{rcl}
e'_0 & := & \mathrm{rec}(\mathrm{inr}(\mathrm{inl}(*)), \widehat{a}, \lambda b : \widehat{B}(a).\mathrm{emb}_{k_b}(e'_b)) \ , \\
\mathrm{trans}'(a,e) & := & \langle D_0, e'_0, e''_0 \rangle \ ,
\end{array}
$$

and the assertions (a) - (c) follow as in the previous case.

Proof of Theorem 11.1, direction $\mathrm{trans}^+_{1,0} : \mathrm{Rec}^+_1 \to \mathrm{Rec}^+_0$. We write in this part of the proof trans for $\mathrm{trans}^+_{1,0}$, trans$'$ for $\mathrm{trans}'_{1,0}$. If we do not suppress the parameter C, we write trans_C, trans'_C.
We first define $\mathrm{trans}_C(e)$ by induction on e, simultaneously for all $C : \mathrm{Fam}(\mathrm{Set})$:
Assume $a : A$,

$$\mathrm{trans}'(a, e(a)) = \langle \langle I_a, C'_a \rangle, e'_a, e''_a \rangle \ ,$$

where, with
$$\widetilde{C}'_a := C \oplus \oplus_{i:I_a} C'_a(i) = \langle \widetilde{A}'_a, \widetilde{B}'_a \rangle \ ,$$
we have
$$e'_a \quad : \quad \mathrm{Rec}_{\widetilde{C}'_a,\mathrm{inl}(a)} \ , \qquad e''_a \quad : \quad (i : I_a) \to \mathrm{Rec}^{+}_{1,C'_a(i)} \ .$$
By IH
$$\mathrm{trans}_{C'_a(i)}(e''_a(i)) = \langle C''_a(i), e'''_a(i) \rangle \ .$$
where, with
$$C'''_a(i) := C'_a(i) \oplus C''_a(i) = \langle A'''_a(i), B'''_a(i) \rangle \ ,$$
we have $e'''_a(i) : \mathrm{Rec}_{C'''_a(i)}$. Let

$$\begin{array}{rcccl} \widetilde{C}''_a & := & C \oplus \oplus_{i:I_a} C'''_a(i) & = & \langle \widetilde{A}''_a, \widetilde{B}''_a \rangle \ , \\ C'''' & := & \oplus_{a:A} \oplus_{i:I_a} C'''_a(i) & = & \langle A'''', B'''' \rangle \ , \\ \widetilde{C} & := & C \oplus C'''' & = & \langle \widetilde{A}, \widetilde{B} \rangle \ . \end{array}$$

Let
$$h_{\widetilde{C}'_a,\widetilde{C}} : \widetilde{A}'_a \to \widetilde{A} \ ,$$
$h_{\widetilde{C}'_a,\widetilde{C}}(\mathrm{inl}(a')) = \mathrm{inl}(a')$, $h_{\widetilde{C}'_a,\widetilde{C}}(\mathrm{inr}(\langle i, a' \rangle)) = \mathrm{inr}(\langle a, \langle i, \mathrm{inl}(a') \rangle \rangle)$. Let

$$h_{C'''_a(i),\widetilde{C}} : C'''_a(i) \to \widetilde{C} \ ,$$

$h_{C'''_a(i),\widetilde{C}}(a') = \mathrm{inr}(\langle a, \langle i, a' \rangle \rangle)$.
Define $\mathrm{trans}(e) := \langle C'''', \widetilde{e} \rangle : \mathrm{Rec}^{+}_{0,C}$, where $\widetilde{e} : \mathrm{Rec}_{\widetilde{C}}$,

$$\begin{array}{rcl} \widetilde{e}(\mathrm{inl}(a)) & = & \mathrm{emb}_{h_{\widetilde{C}'_a,\widetilde{C}}}(e'_a) \ , \\ \widetilde{e}(\mathrm{inr}(\langle a, \langle i, a' \rangle \rangle)) & = & \mathrm{emb}_{h_{C'''_a(i),\widetilde{C}}}(e'''_a(i)) \ . \end{array}$$

We show

$$\forall C : \mathrm{Fam}(\mathrm{Set}).\forall e : \mathrm{Rec}^{+}_{1,C}.\forall a : C.A.\{e\}_1(a) \sqsubseteq \{\mathrm{trans}_C(e)\}_0(a) \ ,$$

i.e.

$$\begin{array}{l} \forall C : \mathrm{Fam}(\mathrm{Set}).\forall e : \mathrm{Rec}^{+}_{1,C}.\forall a : C.A.\forall p : \{e\}_1(a)\downarrow.\exists p' : \{\mathrm{trans}_C(e)\}_0(a)\downarrow. \\ \quad \{e\}_1(a)[p] = \{\mathrm{trans}_C(e)\}_0(a)[p'] \ . \end{array}$$

This statement will be shown by induction on p, simultaneously for all C, e. Assume C, e, a, p as in the assertion, and all the variables as used in the definition of $\mathrm{trans}_C(e)$. Let $k := \mathrm{calls}_{1,C,e}(a,p) : (a : A) \rightharpoonup_{\mathrm{fin}} B(a)$. We get $p' : \{a, e(a)\}_{1,C,k}\downarrow$. By Lem. 11.3 there exists $k' : (a' : \widetilde{A}_a) \rightharpoonup_{\mathrm{fin}} \widetilde{B}_a(a')$ s.t.

$$\begin{array}{rcl} \{k'\}^{\mathrm{finfun}}(\mathrm{inl}(a')) & \sqsubseteq & \{k\}^{\mathrm{finfun}}(a') \ , \\ \{k'\}^{\mathrm{finfun}}(\mathrm{inr}(\langle i, a' \rangle)) & \sqsubseteq & \{e''_a(i)\}_1(a') \ , \\ \{\mathrm{inl}(a), e'_a\}^{\mathrm{fin}}_{k'}\downarrow & & . \end{array}$$

Let
$$k'' := k' \circ (h_{\widetilde{C}'_a,\widetilde{C}})^{-1} : (a' : \widetilde{A}) \rightharpoonup_{\mathrm{fin}} \widetilde{B}(a') \ .$$
Then we get $\{\mathrm{inl}(a), \mathrm{emb}_{h_{\widetilde{C}'_a,\widetilde{C}}}(e'_a)\}^{\mathrm{fin}}_{k''}\downarrow$ and therefore $\{\mathrm{inl}(a), \widetilde{e}(\mathrm{inl}(a))\}^{\mathrm{fin}}_{k''}\downarrow$.
If $\{k''\}^{\mathrm{finfun}}(\mathrm{inl}(a'))\downarrow$, then
$\{k''\}^{\mathrm{finfun}}(\mathrm{inl}(a')) \simeq \{k'\}^{\mathrm{finfun}}(\mathrm{inl}(a')) \simeq \{k\}^{\mathrm{finfun}}(a') \simeq \{e\}_1(a') \stackrel{\mathrm{IH}}{\simeq} \{\widetilde{e}\}(\mathrm{inl}(a'))$.
If $\{k''\}^{\mathrm{finfun}}(\mathrm{inr}(a'))\downarrow$, then $a' = \langle a, \langle i, \mathrm{inl}(a'') \rangle \rangle$,
$\{k''\}^{\mathrm{finfun}}(\mathrm{inr}(a')) \simeq \{k'\}^{\mathrm{finfun}}(\mathrm{inr}(\langle i, a'' \rangle)) \simeq \{e''_a(i)\}_1(a'')$

$\overset{\text{IH}}{\simeq} \{e'''_a(i)\}(\mathrm{inl}(a'')) \simeq \{\widetilde{e}\}(\mathrm{inr}(\langle a, \langle i, \mathrm{inl}(a'')\rangle\rangle)) \simeq \{\widetilde{e}\}(\mathrm{inr}(a'))$.
So we obtain $k'' \subseteq \{\widetilde{e}\}$. By $\{\mathrm{inl}(a), \widetilde{e}(\mathrm{inl}(a))\}^{\mathrm{fin}}_{k''}\downarrow$ therefore $\{\mathrm{inl}(a), \widetilde{e}(\mathrm{inl}(a))\}^{\mathrm{aux}}_{\widetilde{e}}\downarrow$,

$$\{\widetilde{e}\}(\mathrm{inl}(a))\downarrow \ .$$

Similarly we obtain $\{e\}_1(a)[\_] = \{\widetilde{e}\}(\mathrm{inl}(a))[\_]$.
The other direction, i.e.

$$\forall C : \mathrm{Fam}(\mathrm{Set}).\forall e : \mathrm{Rec}^+_{1,C}.\forall a : C.A.\{\mathrm{trans}_C(e)\}_0(a) \sqsubseteq \{e\}_1(a) \ ,$$

can be shown in the same way.

12 Closure Properties of Representable Partial Recursive Functions

Notation 12.1 *(a) In this section we consider partial objects constructed from other partial objects in the usual way. For instance $\{e_0\}(\langle a, \{e_1\}(b)\rangle)$ is the partial object e s.t.*

$$\begin{array}{rcl} e\downarrow & := & \Sigma p : \{e_1\}(b)\downarrow.\{e_0\}(\langle a, \{e_1\}(b)[p]\rangle)\downarrow \ , \\ e[\langle p, q\rangle] & := & \{e_0\}(\langle a, \{e_1\}(b)[p]\rangle)[q] \ . \end{array}$$

Other composed partial objects are to be understood similarly.

(b) Let in this section $(a : A) \rightharpoonup B(a) := \mathrm{Rec}_{A,B}$, $(a : A) \rightharpoonup_0 B(a) := \mathrm{Rec}^+_{0,A,B}$, $(a : A) \rightharpoonup_1 B(a) := \mathrm{Rec}^+_{1,A,B}$. Furthermore $(a : A) \rightharpoonup (b : B) \rightharpoonup C(a,b) := (\langle a, b\rangle : \Sigma a : A.B(a)) \rightharpoonup C(a,b)$, similarly for longer versions and $\rightharpoonup_0$, $\rightharpoonup_1$. When e is such a Curried version, we write $\{e\}(a,b)$ instead of $\{e\}(\langle a, b\rangle)$, similarly for longer versions.

We could deal with more complex partial objects as mentioned in Notation 12.1 (a) by defining operations like $\langle\cdot,\cdot\rangle_{\mathrm{right}} : A \to \mathrm{PrePar}(B) \to \mathrm{PrePar}(A \times B)$, or $\mathrm{compose} : (A \to \mathrm{PrePar}(B)) \to \mathrm{PrePar}(A) \to \mathrm{PrePar}(B)$, and then replacing $\langle a, \{e_1\}(b)\rangle$ as in the example by $e_2 := \langle a, \{e_1\}(b)\rangle'$, and $\{e_0\}(e_2)$ by $\mathrm{compose}(\lambda a.\{e_0\}(a), e_2)$. We leave it to the reader to make this more formal.

Remark 12.2 *We prove in the following closure properties of partial recursive functions, using the version of $\rightharpoonup_1$, since in that version they are most easily expressed. By the last statement (h), these closure properties hold as well for $\rightharpoonup_0$. One could as well prove these closure properties directly in $\rightharpoonup_0$, with a little bit of more effort. This means that the more complex induction principle used for defining $\rightharpoonup_1$ is not needed for proving those closure properties.*

Theorem and Definition 12.3 (Closure Properties of Partial Recursive Functions)

(a) **(Closure under total functions)** *Assume $\langle A, B\rangle : \mathrm{Fam}(\mathrm{Set})$, $f : (a : A) \to B(a)$. Then we can define $\mathrm{totToPar}(f) : (a : A) \rightharpoonup_1 B(a)$ s.t. $\{\mathrm{totToPar}(f)\}(a) \simeq b \leftrightarrow f(a) = b$.*

(b) **(Composition)** *Assume $\langle A, B\rangle : \mathrm{Fam}(\mathrm{Set})$, $D : A \to \mathrm{Set}$. If $e' : (a : A) \rightharpoonup_1 B(a)$ and $e : (a : A) \rightharpoonup_1 B(a) \rightharpoonup_1 D(a)$, then there we can define $e \circ e' : (a : A) \rightharpoonup_1 D(a)$ s.t. $\{e \circ e'\}(a) \simeq \{e\}(a, \{e'\}(a))$. We define as well the following special forms of composition:*

- *If* A, B, D : Set, $e' : A \rightharpoonup_1 B$, $e'' : B \rightharpoonup_1 D$ *then* $e' \circ_1 e : A \rightharpoonup_1 D$ *s.t.* $\{e' \circ_1 e\}(a) \simeq \{e'\}(\{e\}(a))$.
- *If* A, B : Set, $D : A \to B \to \mathrm{Set}$, $E : B \to \mathrm{Set}$, $F : A \to \mathrm{Set}$,
  $e_2 : (a : A) \rightharpoonup_1 \Sigma b : B.D(a, b)$,
  $e_1 : (b : B) \rightharpoonup_1 E(b)$,
  $e_0 : (a : A) \rightharpoonup_1 (b : B(a)) \rightharpoonup_1 (d : D(a, b)) \rightharpoonup_1 (d' : E(b)) \rightharpoonup_1 F(a)$.
  Then we have $e_0 \circ_2 e_1 \circ_2 e_2 : (a : A) \rightharpoonup_1 F(a)$ *s.t.*
  $\{e_0 \circ_2 e_1 \circ_2 e_2\}(a) \simeq \{e_0\}(a, \pi_0(\{e_2\}(a)), \pi_1(\{e_2\}(a)), \{e_1\}(a, \pi_0(\{e_2\}(a))))$.

Furthermore when writing compositions we often use total functions f *and then write* f *instead of* $\mathrm{totToPar}(f)$. *We will often as well Curry* f *e.g. use instead of* $f : (\langle a, b\rangle : \Sigma a : A.B(a)) \to C(a, b)$ *simply* $f : (a : A, b : B(a)) \to C(a, b)$.

(c) *We can define*

$$\begin{array}{rcl} \mathrm{evalrec}^{\mathrm{aux}} & : & (a : A) \rightharpoonup (e : \mathrm{Rec}'_{A,B,a}) \rightharpoonup (e_0 : (a : A) \rightharpoonup B(a)) \\ & & \rightharpoonup \Sigma b : B(a).\{a, e\}^{\mathrm{aux}}_{e_0} \simeq b \ , \end{array}$$

s.t. $\{\mathrm{evalrec}^{\mathrm{aux}}\}(\langle a, e, e_0\rangle) \simeq \langle b, p\rangle \leftrightarrow \{a, e\}^{\mathrm{aux}}_{e_0} \simeq b$.
Note that we are referring to $\rightharpoonup$ *instead of* $\rightharpoonup_1$.

(d) **(Correct self-evaluation for elements of** $(a : A) \rightharpoonup B(a)$**)** *We can define*

$$\begin{array}{rcl} \mathrm{evalrec} & : & (e : (a : A) \rightharpoonup B(a)) \rightharpoonup_1 (a : A) \rightharpoonup_1 B(a) \ , \\ \mathrm{evalcorrect} & : & (e : (a : A) \rightharpoonup B(a)) \rightharpoonup_1 (a : A) \\ & & \rightharpoonup_1 \Sigma b : B(a).\{e\}(a) \simeq b \ , \end{array}$$

s.t. $\{\mathrm{evalcorrect}\}_1(e, a) \simeq \langle b, p\rangle \leftrightarrow \{\mathrm{evalrec}\}_1(e, a) \simeq b \leftrightarrow \{e\}(a) \simeq b$.
Note the use of $\rightharpoonup$ *and* $\rightharpoonup_1$.

(e) **Partial recursive functions with correctness proofs.**
If $e : (a : A) \rightharpoonup_1 B(a)$, *then there exists*

$$e' : (a : A) \rightharpoonup_1 \Sigma b : B(a).\{e\}(a) \simeq b$$

s.t. $\{e'\}(a) \simeq \langle b, p\rangle \leftrightarrow \{e\}(a) \simeq b$.

(f) **(Primitive recursion)** *Assume* A : Set, $B : (a : A) \to \mathbb{N} \to \mathrm{Set}$,

$$\begin{array}{rcl} e_0 & : & (a : A) \rightharpoonup_1 B(a, 0) \ , \\ e_1 & : & (a : A) \rightharpoonup_1 (n : \mathbb{N}) \rightharpoonup_1 B(a, n) \rightharpoonup_1 B(a, n+1) \ . \end{array}$$

Then there exists $e := \mathrm{primrec}(e_0, e_1) : (a : A) \rightharpoonup_1 (n : \mathbb{N}) \rightharpoonup B(a, n)$ *s.t.* $\{e\}(a, 0) \simeq \{e_0\}(a)$, $\{e\}(a, n+1) \simeq \{e_1\}(a, n, \{e\}(a, n))$.

(g) **(Closure under** μ**)** *Assume* A : Set, $e : A \rightharpoonup \mathbb{N} \rightharpoonup \mathrm{Bool}$. *Then we can define*

$$\mu(e) \quad : \quad A \rightharpoonup_1 \mathbb{N} \ ,$$

s.t.

$$\{\mu(e)\}(a) \simeq n \leftrightarrow \{e\}(a, n) \simeq \mathrm{tt} \wedge \forall k < n.\{e\}(a, n) \simeq \mathrm{ff} \ .$$

We write $\mu(f)$ *instead of* $\mu(\mathrm{totToPar}(f))$. *Furthermore we can define in case of* $f : A \to \mathbb{N} \to \mathrm{Bool}$

$$\mu^+(f) : (a : A) \to \Sigma n : \mathbb{N}.\mathrm{atom}(f(a, n))$$

s.t. $\mu^+(f) \simeq \langle n, q\rangle \leftrightarrow \mu(f) \simeq n$.
Furthermore we can define an internalised version
$\mu^+_0 : (f : (a : A) \to \mathbb{N} \to B(a)) \rightharpoonup_1 (a : A) \rightharpoonup_1 \Sigma n : \mathbb{N}.\mathrm{atom}(f(a, n))$,
s.t. $\{\mu^+_0\}_1(f, a) \simeq \{\mu^+(f)\}_1(a)$.

(h) *Except for (c) all the above closure properties hold as well with* $\rightharpoonup_1$ *replaced by* $\rightharpoonup_0$.

Proof: First (h) holds, since we can first translate all ingredients which are elements of $\_ \rightharpoonup_0 \_$ into elements of $\_ \rightharpoonup_1 \_$, then apply the operation for $\_ \rightharpoonup_1 \_$ and then translate it back into $\_ \rightharpoonup_0 \_$. Then one can easily see that the desired properties hold.
In order to prove (a) - (g), we will usually only give the definition of the resulting partial recursive function – that the given property holds is straightforward. We will usually suppress the parameter C in $\mathrm{call}(C, e, a, k)$.
(a): $\mathrm{totToPar}(e) := \lambda a.\mathrm{return}(f(a))$.
(b):

$$\begin{aligned} e \circ e' := \lambda a : A.\mathrm{call}(\_, e', a, \lambda b : B(a).\\ \mathrm{call}(\_, e, \langle a, b\rangle, \lambda d : D(a).\mathrm{return}(d))) \ . \end{aligned}$$

The other composition operations can be defined similarly.
(c):

$$\begin{array}{lcl} \mathrm{evalrec}^{\mathrm{aux}}(a, \mathrm{return}(b), e_0) & := & \mathrm{return}(\langle b, q\rangle) \ , \\ \mathrm{evalrec}^{\mathrm{aux}}(a, \mathrm{rec}(a', h), e_0) & := & \mathrm{rec}(\langle a', e_0(a'), e_0\rangle, \\ & & \quad \lambda\langle b, p\rangle : \Sigma b : B(a').\{a', e_0(a')\}^{\mathrm{aux}}_{e_0} \simeq b. \\ & & \quad \mathrm{rec}(\langle a, h(b), e_0\rangle \\ & & \qquad \lambda\langle b', p'\rangle : \Sigma b : B(a).\{a, h(b)\}^{\mathrm{aux}}_{e_0} \simeq b'. \\ & & \qquad\quad \mathrm{return}(\langle b', q'\rangle))) \ , \end{array}$$

where the proofs q, q' are obtained from the parameters in an obvious way. The correctness condition follows now easily.
(d): $\mathrm{evalcorrect} := \lambda\langle e, a\rangle.\mathrm{call}(\_, \mathrm{evalrec}^{\mathrm{aux}}, \langle a, e(a), e\rangle, \lambda\langle b, p\rangle.\mathrm{return}(\langle b, p'\rangle))$, where $p' : \{e\}(a) \simeq b$ is obtained from $p : \{a, e(a)\}^{\mathrm{aux}}_e \simeq b$. evalrec is obtained by composing the first projection with evalcorrect.
(e): We can define $e_0 : (a : A) \rightharpoonup_0 B(a)$ s.t. $\forall a : A.\{e\}_1(a) \simeq \{e_0\}_0(a)$. Let $e_0 = \langle C_1, e_1\rangle$ with $C_1 = \langle A_1, B_1\rangle$, $e_1 : (a : A + A_1) \rightharpoonup [B, B_1](a)$ and $\{e_0\}_0(a) \simeq \{e_1\}(\mathrm{inl}(a))$. Now define

$$\begin{aligned} e' := \lambda a : A.\mathrm{call}(\_, \mathrm{evalcorrect}, \langle e_1, \mathrm{inl}(a)\rangle, \\ \lambda\langle b, p\rangle : (\Sigma b : B(a).\{e_1\}(\mathrm{inl}(a)) \simeq b). \\ \mathrm{return}(\langle b, p'\rangle)) \ , \end{aligned}$$

where $p' : \{e\}_1(a) \simeq b$ is obtained from p. The assertion follows now easily.
(f): $\mathrm{primrec}(e_0, e_1) := e$ where

$$\begin{array}{rcl} e(a, 0) & := & \mathrm{call}(\_, e_0, a, \lambda b : B(\langle a, 0\rangle).\mathrm{return}(b)) \ , \\ e(a, n+1) & := & \mathrm{rec}(\langle a, n\rangle, \\ & & \quad \lambda b : B(\langle a, n\rangle). \\ & & \quad \mathrm{call}(\_, e_1, \langle a, n, b\rangle, \lambda b : B(\langle a, n+1\rangle).\mathrm{return}(b))) \ . \end{array}$$

(g): We define first $\mu'(e) : A \rightharpoonup_1 \mathbb{N} \rightharpoonup_1 \mathbb{N}$ s.t.

$$\{\mu'(e)\}_1(a, n) \simeq m \leftrightarrow m \geq n \wedge \{e\}_1(a, m) \simeq \mathrm{tt} \wedge \forall k < n.m \leq k \rightarrow \{e\}_1(a, m) \simeq \mathrm{ff} \ .$$

$$\begin{aligned} \mu'(e) := \lambda\langle a, m\rangle.\mathrm{call}(\_, e, \langle a, m\rangle, \\ \lambda b : \mathrm{Bool}.\mathrm{if}\ b \\ \mathrm{then}\ \mathrm{return}(m) \\ \mathrm{else}\ \mathrm{rec}(\langle a, m+1\rangle, \lambda k.\mathrm{return}(k))) \ . \end{aligned}$$

It is easy to see that the desired property holds. Then $\mu(e) := \mu'(e) \circ (\lambda a.\langle a, 0\rangle)$.

Concerning $\mu^+(f)$ we obtain from $\mu(f)$ and part (e) of the theorem an

$$e' : (a : A) \rightharpoonup_1 \Sigma n : \mathbb{N}.\{\mu(f)\}_1(a) \simeq n \ ,$$

s.t. $\{e'\}_1(a) \simeq \langle n, p\rangle \leftrightarrow \{\mu(f)\}_1(a) \simeq n$.
We have $\{\mu(f)\}_1(a) \simeq n \to \mathrm{atom}(f(a))$ and obtain therefore a function $h : (\Sigma n : \mathbb{N}.\{\mu(f)\}_1(a) \simeq n) \to (\Sigma n : \mathbb{N}.\mathrm{atom}(f(n)))$. Now $\mu^+(f) := h \circ_1 e'$.
The definition of μ_0^+ is similar to that of $\mu^+(f)$ using the same steps, where in the first step we define

$$\begin{array}{rcl} \mu_0' & : & (f : (a : A) \to B(a)) \rightharpoonup_1 (a : A) \rightharpoonup_1 (n : \mathbb{N}) \rightharpoonup \mathbb{N} \ , \\ \mu_0'(f, a, n) & := & \text{if } f(a, n) \text{ then return}(m) \text{ else rec}(\langle f, a, m+1\rangle, \lambda k.\text{return}(k)) \ . \end{array}$$

Note that because f is total we do not need to use the self-evaluation function evalrec.

13 Normal Form Theorem, Size Problem, Self-Evaluation

Based on the evaluation of elements of Rec introduced in Sect. 6 and its correctness proof given in Sect. 7, we can derive the normal form theorem 13.1, which is similar to Kleene's normal form theorem for partial recursive functions. This will allow to define a new type Rec_2^+ of partial recursive functions (Theorem and Definition 13.3), which is equivalent to Rec_0^+, Rec_1^+, but an element of Set. This solves the size problem (that Rec_0^+, Rec_1^+ are true types). Furthermore, using evalcorrect as defined in Theorem and Definition 12.3 (d) we can define self-evaluation for elements of Rec_2^+, including a version which computes as well a correctness proof.

13.1 The Normal Form Theorem for Rec_1^+

Theorem 13.1 *(a) For* $e : \mathrm{Rec}_{1,A,B}^+$ *there exists* $f : A \to \mathbb{N} \to \mathrm{Bool}$ *and* $g : (a : A, n : \mathbb{N}, \mathrm{atom}(f(a, n))) \to B(a)$ *s.t. for all* $a : A, b : B(a)$

$$\{e\}_1(a) \simeq b \Leftrightarrow \exists n : \mathbb{N}.\exists p : \mathrm{atom}(f(a, n)).g(a, n, p) = b \ .$$

(b) **(Normal Form Theorem)** *Depending on* f, g *as defined in (a) we can define a function* $h_{f,g} : (\Sigma a : A.\Sigma n : \mathbb{N}.\mathrm{atom}(f(a, n))) \to B(a)$*, s.t. for all* $e : \mathrm{Rec}_{1,A,B}^+$

$$\{e\}_1(a) \simeq \{h_{f,g} \circ \mu^+(f)\}_1(a) \ .$$

(c) Let $A_0 : \mathrm{Set}$*,*
$A_0 := \Sigma f : (A \to \mathbb{N} \to \mathrm{Bool}).\Sigma g : (a : A, n : \mathbb{N}, \mathrm{atom}(f(a, n))) \to B(a).A$.
Let $B_0 : A_0 \to \mathrm{Set}$*,* $B_0(\langle f, g, a\rangle := B(a)$.
Then we can define $k : \mathrm{Rec}_{1,A_0,B_0}^+$ *s.t.* $\{k\}_1(\langle f, g, a\rangle) \simeq \{h_{f,g} \circ \mu^+(f)\}_1(a)$.

Proof: *(a):* We have $e' : \mathrm{Rec}_1^+$ s.t. $\{e\}_1(a) \simeq \{e'\}_0(a)$. Let $e' = \langle C'', e''\rangle$, $C'' = \langle A'', B''\rangle$. Then $\{e'\}_0(a) \simeq \{e''\}(\mathrm{inl}(a))$. By Theorem 7.1 $\{e''\}(\mathrm{inl}(a)) \simeq \{e''\}^{\mathrm{comp}}(\mathrm{inl}(a))$. Let $f(a, n) := \mathrm{defined}_a(\mathrm{Red}^n_{e'',\mathrm{inl}(a)}(e''(\mathrm{inl}(a))))$, $g(a, n, p) := \mathrm{Extract}_a(\mathrm{Red}^n_{e'',\mathrm{inl}(a)}(e''(\mathrm{inl}(a))), p)$. We have by definition $\{e''\}^{\mathrm{comp}}(\mathrm{inl}(a)) \simeq b \leftrightarrow \exists n : \mathbb{N}.\exists p : \mathrm{atom}(f(a, n)).g(a, n, p) = b$, i.e the assertion.
(b): We define $h_{f,g}(\langle a, n, p\rangle) := g(a, n, p)$. Then the assertion follows immediately.
(c):

$$\begin{array}{l} k(\langle f, g, a\rangle) := \\ \ \mathrm{call}(\_, \mu_0^+, \langle f, a\rangle, \lambda\langle n, p\rangle : \Sigma n : \mathbb{N}.\Sigma p : \mathrm{atom}(f(a, n)).\mathrm{return}(h_{f,g}(\langle a, n, p\rangle)))\ . \end{array}$$

13.2 Solving the Size Problem

Theorem 13.2 *For every* $C : \mathrm{Fam}(\mathrm{Set})$ *there exists* $C' : \mathrm{Fam}(\mathrm{Set})$ *s.t.*

$$\forall e : \mathrm{Rec}^{+}_{1,C}.\exists e' : \mathrm{Rec}_{C\oplus C'}.\forall a : C.A.\{e\}_1(a) \simeq \{e'\}(\mathrm{inl}(a)) \ .$$

Remark on the proof: The solution given in the following will use a lot of theory developed before and will result in a rather complex family C'. One could carry out this proof directly by implementing the function used in the normal form theory directly in Rec^{+}_{0}, and obtain a much simpler family C'.

Proof: Let k as in Theorem 13.1 (c). Then for every e there exists f, g s.t. $\{e\}_1(a) \simeq \{k\}_1(\langle f, g, a\rangle)$. Let $k' : \mathrm{Rec}^{+}_{0,C''}$ s.t. $\{k\}_1(b) \simeq \{k'\}_0(\mathrm{inl}(b))$ for some C. Let $k' = \langle C''', k''\rangle$, $k'' : \mathrm{Rec}_{C''\oplus C'''}$. Let $C' := C'' \oplus C'''$. Assume $e : \mathrm{Rec}^{+}_{1,C}$ and let f, g be chosen as above. Let $e_0 : (a : A) \to \mathrm{Rec}'_{C\oplus C',\mathrm{inl}(a)}$, $e_0(a) := \mathrm{rec}(\mathrm{inr}(\mathrm{inl}(\langle f, g, a\rangle)), \lambda b : B(a).\mathrm{return}(b))$. Let $e_1 := e_0 \oplus^{\mathrm{inr}} k'' : \mathrm{Rec}_{C\oplus C'}$. Then $\{e_1\}(\mathrm{inl}(a)) \simeq \{k''\}(\mathrm{inl}(\langle f, g, a\rangle)) \simeq \{k\}_1(\langle f, g, a\rangle) \simeq \{e\}_1(a)$.

Theorem and Definition 13.3 ((Solution of the size problem for Rec_C) *Assume the following definitions:*

(a) $\mathrm{Rec}^{+}_{2,C} := \mathrm{Rec}_{C\oplus C'} : \mathrm{Set}$*, where* C' *is defined depending on* C *as in Theorem 13.2.*

(b) For $e : \mathrm{Rec}^{+}_{2,C}$*,* $a : A$ *let* $\{e\}_2(a) : \mathrm{PrePar}(B(a))$*,* $\{e\}_2(a){\downarrow} := \{e\}(\mathrm{inl}(a)){\downarrow}$*,* $\{e\}_2(a)[p] := \{e\}(\mathrm{inl}(a))[p]$.

Then for $e : \mathrm{Rec}^{+}_{1,C}$ *there exists* $e' : \mathrm{Rec}^{+}_{2,C}$ *s.t.* $\{e\}_1 \cong \{e\}_2$*, and vice versa.*

Proof: By the above.

13.3 Self-Evaluation for Rec^{+}_{2}

We write in this section $(a : A) \rightharpoonup B(a)$, $(a : A) \rightharpoonup_0 B(a)$, $(a : A) \rightharpoonup_1 B(a)$, $(a : A) \rightharpoonup_2 B(a)$ for $\mathrm{Rec}_{A,B}$, $\mathrm{Rec}^{+}_{0,A,B}$, $\mathrm{Rec}^{+}_{1,A,B}$, $\mathrm{Rec}^{+}_{2,A,B}$, respectively.

Theorem and Definition 13.4 *For* $C = \langle A, B\rangle : \mathrm{Fam}(\mathrm{Set})$ *There exists*

(i) $\mathrm{eval}_{2,C} : (e : (a : A) \rightharpoonup_2 B(a)) \rightharpoonup_2 (a : A) \rightharpoonup_2 B(a)$*,*

(ii) $\mathrm{evalcorrect}_{2,C} : (e : (a : A) \rightharpoonup_2 B(a)) \rightharpoonup_2 (a : A) \rightharpoonup_2 \Sigma b : B(a).\{e\}_2(a) \simeq b$.

s.t.

(i) $\{\mathrm{eval}_2\}_2(\langle e, a\rangle) \simeq \{e\}_2(a)$*,*

(ii) $\{\mathrm{evalcorrect}_2\}_2(\langle e, a\rangle) \simeq \langle b, p\rangle \leftrightarrow \{e\}_2(a) \simeq b$.

Proof: Let $((a : A) \rightharpoonup_2 B(a)) = ((a : \widetilde{A}) \rightharpoonup \widetilde{B}(a))$ where $\langle \widetilde{A}, \widetilde{B}\rangle = \langle A, B\rangle \oplus \langle A', B'\rangle$ for some A', B'. There exists

$$\mathrm{evalcorrect} : (e : (a : \widetilde{A}) \rightharpoonup \widetilde{B}(a)) \rightharpoonup_1 (a : \widetilde{A}) \rightharpoonup_1 \Sigma b : \widetilde{B}(a).\{e\}(a) \simeq b \ ,$$

s.t. $\{\mathrm{evalcorrect}\}_1(e, a) \simeq \langle b, p\rangle \leftrightarrow \{e\}(a) \simeq b$.
By composing it with $\mathrm{inl} : A \to \widetilde{A}$, and transferring $p : \{e\}(\mathrm{inl}(a)) \simeq b$ into $p : \{e\}_2(a) \simeq b$, and using $((a : A) \rightharpoonup_2 B(a)) = ((a : \widetilde{A}) \rightharpoonup \widetilde{B}(a))$, we obtain some

$$e' : (e : (a : A) \rightharpoonup_2 B(a)) \rightharpoonup_1 (a : A) \rightharpoonup_1 \Sigma b : B(a).\{e\}_2(a) \simeq b \ .$$

Now let $\mathrm{evalcorrect}_2$ be s.t. $\{\mathrm{evalcorrect}_2\}_2(d) \simeq \{e'\}_1(d)$. eval_2 can be defined by composing $\mathrm{evalcorrect}_2$ with a projection. (Note that because of the equivalence of $\{\cdot\}_1$ and $\{\cdot\}_2$, Rec^{+}_{2} is closed under the standard operations for forming partial recursive functions.)

14 Conclusion

In this paper we have explored the representation of partial recursive function by indexed inductive-recursive definitions. We have introduced a data type of partial recursive functions and we have seen how to (partial) recursively evaluate elements of this data type inside type theory. We have seen that reference to other partial recursive functions requires the extension of this data type and we have obtained two data types Rec_0^+ and Rec_1^+, both of which are large types. Then we have shown that they are equivalent, and have proved a normal form theorem. This allowed to show that we can restrict the size of additional types needed, and therefore to introduce a fourth data type Rec_2^+, which is equivalent to Rec_0^+ and Rec_1^+, but a small set. We have seen as well how to define self-evaluation inside Rec_2^+ and how to find apart from results of the computed function as well proofs that it is correct. The data type Rec_1^+ provided a very general recursion schema which allows to easily define general recursive functions while referring to the full type hierarchy. We believe that this general schema can be exploited further, for instance in generic programming – we could take codes for partial recursive functions and compute from them new codes.

14.1 Future Work

A fifth data type of partial recursive functions and the delay monad. We could define a fifth data type of partial recursive functions by referring to the normal form theorem and observing that for every partial recursive function $e : \mathrm{Rec}_1^+$ there exist functions f and g s.t. $\{e\}_1(a) \simeq \{h_{f,g} \circ \mu^+(f)\}_1(a)$. Since f and g can be encoded into one function $h : (a : A) \to \mathbb{N} \to (\{*\} + B(a))$, we could define

$$\begin{array}{rcl} \mathrm{Rec}_3^+ & := & (a : A) \to \mathbb{N} \to (\{*\} + B(a)) \ , \\ \{\cdot\}_3 & : & \mathrm{Rec}_3^+ \to (a : A) \to \mathrm{Par}(B(a)) \ , \\ \{h\}_3(a){\downarrow} & := & \Sigma n : \mathbb{N}.\Sigma b : B(a).f(a,n) = \mathrm{inr}(b) \wedge \forall m < n.f(a,n) = \mathrm{inl}(*) \\ \{h\}_3(a)[\langle n, b, p, q\rangle] & := & b \ . \end{array}$$

By searching for the least n s.t. $h(a,n)$ is of the form $\mathrm{inr}(b)$, we can compute $\{\langle f, g\rangle\}_3(a)$ inside type theory partial-recursively. It goes beyond this article to investigate this idea further (e.g. to show closure under standard operations for forming partial recursive functions).
Note that this data type is closely related to the delay monad used in [Cap05]. Elements h of the functions from A into the delay monad for B, i.e. in $(a : A) \to B(a)^\nu$ correspond directly to $f : \mathrm{Rec}_3^+$ s.t. we have $n < m \to (f(a,n) = \mathrm{inr}(b)) \to f(a,m) = \mathrm{inr}(b)$: Assume h, and define f. We first unpack $h(a)$ n times, e.g. replace $\mathrm{step}(b)$ by b and leave $\mathrm{return}(b)$ as it is. If we obtain $\mathrm{return}(b)$, $f(a,n) = \mathrm{inr}(b)$, otherwise $f(a,n) = \mathrm{inl}(*)$. On the other hand, from f with the property before we can in an obvious way obtain an element $h : (a : A) \to B(a)^\nu$ by using guarded recursion: First define $f' : (a : A, n : \mathbb{N}) \to B(a)^\nu$ by $f'(a,n) = \mathrm{return}(b)$, if $f(a,n) = \mathrm{inr}(b)$, and $f'(a,n) = \mathrm{step}(f'(a,n+1))$, if $f(a,n) = \mathrm{inl}(*)$. Now define $h(a) := f'(a,0)$. Using extensional type theory we can see that the two translations are inverse to each other. We will leave it to a future paper to investigate this idea further.

Higher type recursion. The original motivation for this article was to be able to define functions which refer recursively to themselves as a whole. There was no room in this paper to investigate this idea further. However, we have some promising ideas how to pursue in this direction.

Provably correct programs. One of the main goals of developing the theory of dependent type theory is its use in program verification, which requires some suitable notion of partial recursive functions. It will be interesting to see in which sense the work of this article can be used in the verification of recursively defined programs. We belief that the idea that from a partial recursive function we were able to obtain a partial recursive function which computes the same result plus a correctness proof is of particular use in the area of program verification.

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.

[Bov02a] A. Bove. General recursion in type theory. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, International Workshop TYPES 2002, The Netherlands*, LNCS 2646, pages 39–58, April 2002.

[Bov02b] A. Bove. *General Recursion in Type Theory.* PhD thesis, Department of Computing Science, Chalmers University of Technology, November 2002.

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

[Con83] Robert L. Constable. Partial functions in constructive formal theories. In *Theoretical Computer Science*, pages 1–18, 1983.

[Coq94] Thierry Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Selected Papers 1st Int. Workshop on Types for Proofs and Programs, TYPES'93, Nijmegen, The Netherlands, 24–28 May 1993*, volume 806, pages 62–78. Springer-Verlag, Berlin, 1994.

[CS87] Robert L. Constable and Scott F. Smith. Partial objects in constructive type theory. In *Proceedings, Symposium on Logic in Computer Science, 22-25 June 1987, Ithaca, New York, USA*, pages 183–193. IEEE Computer Society, 1987.

[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.

[GPZ99] Herman Geuvers, Erik Poll, and Jan Zwanenburg. Safe proof checking in type theory with y. In *CSL '99: Proceedings of the 13th International Workshop and 8th Annual Conference of the EACSL on Computer Science Logic*, pages 439–452. Springer-Verlag, 1999.

[ML84] Per Martin-Löf. *Intuitionistic type theory.* Bibliopolis, Naples, 1984.

[Nor88] Bengt Nordström. Terminating recursion. *BIT*, 28:605 – 619, 1988.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's type theory. An Introduction.* Oxford University-Press, Oxford, 1990.

[Pal91] Erik Palmgren. A construction of type: Type in Martin-Löf's partial type theory with one universe. *J. Symb. Log.*, 56(3):1012–1015, 1991.

[Pau93] Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *CoRR*, cs.LO/9301102, 1993.

[PSH90] Erik Palmgren and Viggo Stoltenberg-Hansen. Domain interpretations of Martin-Löf's partial type theory. *Ann. Pure Appl. Logic*, 48(2):135–196, 1990.

[PSH92] Erik Palmgren and Viggo Stoltenberg-Hansen. Remarks on Martin-Löf's partial type theory. *BIT*, 32(1):70–83, 1992.

[Set06] Anton Setzer. Partial recursive functions in Martin-Löf Type Theory. In Arnold Beckmann, Ulrich Berger, Benedikt Löwe, and John V. Tucker, editors, *Logical Approaches to Computational Barriers: Second Conference on Computability in Europe, CiE 2006, Swansea, UK, June 30-July 5, 2006. Proceedings.*, pages 505 – 515. Springer Lecture Notes in Computer Science 3988, 2006.