ITERATION AND PRIMITIVE RECURSION IN CATEGORICAL TERMS

FOR HENK BARENDREGT'S 60TH BIRTHDAY. THANKS FOR ALL INSPIRING DISCUSSIONS

HERMAN GEUVERS AND ERIK POLL

ICIS, Radboud University Nijmegen, the Netherlands

Abstract. We study various well-known schemes for defining inductive and co-inductive types from a categorical perspective. Categorically, an inductive type is just an initial algebra and a coinductive type is just a terminal co-algebra. However, in category theory these notions are quite strong, requiring the existence of a certain map and its uniqueness. In a formal system like type theory one usually only enforces the existence, because uniqueness complicates the computational model. (Equality becomes undecidable.) It is then more difficult to show the existence of maps defined by primitive recursion, so one introduces separate notions e.g. *primitive recursive types*, etc. The interdefinability of these various notions has been studied by various authors.

It is well-known that also the categorical notions can be weakened, removing the uniqueness requirement. In the present paper we study various weakened versions of the notion of initial algebra (and its dual, terminal co-algebra), and we show in categorical terms how these notions relate to each other. In that sense, this paper can be seen as a categorical recast of type theoretic constructions of [4].

1. Introduction

There has been a lot of research into the formalizations of inductive and coinductive types in systems of typed λ-calculus, mostly (extensions of) simply typed and polymorphic lambda calculus. It is well-known that in polymorphic lambda calculus, many (co)inductive data types can be encoded. For inductive data types such encodings are given in for instance [2] and [5]. For coinductive types, as described in [6], an encoding is given in [13]. Two ways of using the inductive building up of a type to define functions on that type can be distinguished, the *iterative* way and the *primitive recursive* way. An *iterative* function is defined by induction on the building up of the type by defining the function value in terms of the previous values. A *primitive recursive* function is also defined by induction, but now by defining the function value in terms of the previous values *and the previous inputs*. For functions on the natural numbers that is $h : \mathsf{Nat} \to A$, with $h(0) = c, h(n+1) = f(h(n))$ (for $c : A, f : A \to A$) is *iterative* and $h : \mathsf{Nat} \to A$, with $h(0) = c, h(n+1) = g(h(n), n)$ (for $c : A, g : A \times \mathsf{Nat} \to A$) is *primitive recursive*. If one has pairing, the recursive functions can be defined using just iteration, which was essentially already shown by [7][1] encoding. But if

[1] As told in [3], Kleene came up with the idea while he was at the dentist having two wisdom teeth pulled.

REFLECTIONS ON TYPE THEORY, λ-CALCULUS, AND THE MIND

Essays Dedicated to Henk Barendregt on the Occasion of his 60th Birthday

we work in a typed lambda calculus this translation of recursion in terms of iteration only works for certain inputs and becomes inefficient.

This asks for an explicit scheme for recursion in typed lambda calculus, which yields for, say, the natural numbers the scheme of Gödel's T. Various proposals have been made for this, the oldest ones appearing in [10], [11] and [4]. The last paper also gives an overview and a comparison of various schemes, showing how schemes for defining primitive (co-)recursive types can be defined in terms of each other. An interesting additional – and often very useful – feature in these translations is that in the polymorphic λ calculus a weak form of initial algebras (and dually a weak form of terminal co-algebras) is definable. Other results extending this work and presenting other translations can be found in [9] and [12]. The type schemes that are discussed in these papers are inspired by categorical diagrams, but the translations are given in type theoretic terms. Also [1] presents and studies various type theoretic schemes for inductive and coinductive types.

In the present paper we cast everything in categorical terms. We present various notions of "weak" initial algebras and we show how they relate to each other by giving categorical constructions. Everything can be dualized, so this extends to "weak" notions of terminal co-algebra. The main result in this paper is a categorical recast of a result that was stated in type theoretical terms in [4], showing how so called *primitive recursive algebras* can be constructed in a category that has *iterative algebras* and *case constructions*. These notions will be explained in the paper.

2. Initial Algebras

As said, we shall get our intuitions about inductive and coinductive types from the field of category theory. The first definitions and examples are completely standard.

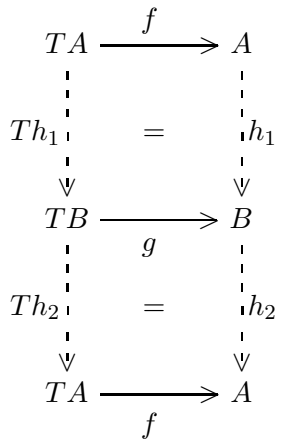
Definition 2.1. Let C be a category, and T a functor from C to C.

(1) A *T-algebra in C* is a pair (A, f), with A an object and $f : TA \to A$.
(2) If (A, f) and (B, g) are T-algebra's, *a morphism from (A, f) to (B, g)* is a morphism $h : A \to B$ such that the following diagram commutes.

$$\begin{array}{ccc} TA & \xrightarrow{f} & A \\ {\scriptstyle Th}\downarrow & = & \downarrow{\scriptstyle h} \\ TB & \xrightarrow[g]{} & B \end{array}$$

(3) A T-algebra (A, f) is *initial* if it is initial in the category of T-algebras, i.e. for every T-algebra (B, g) there is a unique h which makes the diagram above commute.

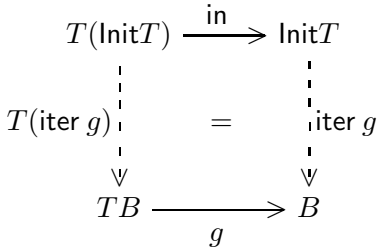
An initial T-algebra is unique up to isomorphism, as can be observed from the following diagram.



Suppose both (A, f) and (B, g) are initial T-algebras. Then $h_2 \circ h_1 \circ f = f \circ T(h_2 \circ h_1)$, so $h_2 \circ h_1 = \mathsf{id}_A$ Similarly $h_1 \circ h_2 = \mathsf{id}_B$.

We introduce some special notation for denoting the initial T-algebra.

Notation 2.2. For T a functor we denote the initial T-algebra (if it exists) by $(\mathsf{Init}T, \mathsf{in}, \mathsf{iter})$, where $(\mathsf{Init}T, \mathsf{in})$ denotes the algebra and $\mathsf{iter}\ g$ denotes the unique morphism that makes the diagram commute. In a diagram:



Let C be a category with products, coproducts and terminal object 1. The initial algebra of the functor $NAT(X) = 1 + X$ is a natural numbers object, which we denote by $(\mathsf{Nat}, \big[\mathsf{Z}, \mathsf{S}\big], \mathsf{iter})$. (If confusion may arise we use subscripts to distinguish one iter from another.) This will be our pet-example of an initial algebra, which will be used to illustrate the properties we are interested in. First we take a look at how morphisms can be defined on Nat by iteration and primitive recursion. The examples immediately generalize to arbitrary initial algebras.

Example 2.3 (Iteration)**.** Suppose $g_1 : 1 \to B$ and $g_2 : B \to B$. Then $\big[g_1, g_2\big] : 1+B \to B$ and $h := \mathsf{iter}\big[g_1, g_2\big] : \mathsf{Nat} \to B$ is the unique morphism such that $h \circ \big[\mathsf{Z}, \mathsf{S}\big] = \big[g_1, g_2\big] \circ NAT(h) = \big[g_1, g_2\big] \circ (\mathsf{id} + h)$. So h satisfies the following recursion equations:

$$\begin{cases} h \circ \mathsf{Z} &= g_1 \\ h \circ \mathsf{S} &= g_2 \circ h \end{cases}$$

This recursion scheme is known as *iteration*.

Example 2.4 (Primitive recursion)**.** (1) The most familiar recursion pattern for the natural numbers is *primitive recursion*. A morphism $h : \mathsf{Nat} \to B$ is then defined by

equations

$$\begin{cases} h \circ \mathsf{Z} & = & g_1 \\ h \circ (\mathsf{S} \circ n) & = & g_2 \circ \langle h \circ n, n \rangle \text{ for all } n : 1 \to \mathsf{Nat} \end{cases}$$

where $g_1 : 1 \to B$ and $g_2 : B \times \mathsf{Nat} \to B$. The difference with the iterative scheme is that g_2 now gives the value of h at $\mathsf{S} \circ n$ in terms of the value of $h \circ n$ *and* n itself. The standard example of a morphism that is easier to define with the primitive recursive scheme than with the iterative scheme is the predecessor $\mathsf{P} : \mathsf{Nat} \to \mathsf{Nat}$ with $\mathsf{P} \circ \mathsf{Z} = \mathsf{Z}$ and $\mathsf{P} \circ (\mathsf{S} \circ n) = n$ (take $g_1 = \mathsf{Z}$ and $g_2 = \pi_2$).

(2) Another example of a morphism defined by primitive recursion (and easier to generalize to arbitrary initial algebras) is the *inverse* of $\left[\mathsf{Z}, \mathsf{S}\right]$, i.e. a morphism $\mathsf{out} : \mathsf{Nat} \to 1 + \mathsf{Nat}$ such that $\mathit{out} \circ \left[\mathsf{Z}, \mathsf{S}\right] = \mathsf{id}$ (take $g_1 = \mathsf{inl}$ and $g_2 = \mathsf{inr} \circ \pi_2$).

It is well-known that, if one has pairing, primitive recursive morphisms can be defined using just iteration. To obtain a morphism $h : \mathsf{Nat} \to B$ satisfying the equations above, we then first define a morphism $H : \mathsf{Nat} \to B \times \mathsf{Nat}$ such that $H = \langle h, \mathsf{id} \rangle$ using the iterative scheme, and take $\pi_1 \circ H$ as h. This trick can be generalized for arbitrary initial algebras, resulting in the following folk result:

Lemma 2.5 (Primitive recursion on initial algebras)**.** Let $(A, \mathsf{in}, \mathsf{iter})$ be an initial T-algebra and $g : T(B \times A) \to B$. Then there is a unique morphism h such that

$$\begin{array}{ccc} TA & \xrightarrow{\mathsf{in}} & A \\ {\scriptstyle T\langle h, \mathsf{id}\rangle}\downarrow & = & \downarrow{\scriptstyle h} \\ T(B \times A) & \xrightarrow[g]{} & B \end{array}$$

commutes, namely

$$h = \pi_1 \circ \mathsf{iter}\langle g, \mathsf{in} \circ T(\pi_2)\rangle.$$

If the initial algebra (A, in) is clear from the context, we refer to this unique arrow h as $\mathsf{rec}\, g$.

Proof. Let $(A, \mathsf{in}, \mathsf{iter})$ be an initial T-algebra and $g : T(B \times A) \to B$. Now $\langle g, \mathsf{in} \circ T(\pi_2)\rangle : T(B \times A) \to B \times A$:

$$\begin{array}{ccc} TA & \xrightarrow{\mathsf{in}} & A \\ {\scriptstyle T(\mathsf{iter}\langle g, \mathsf{in} \circ T(\pi_2)\rangle)}\downarrow & = & \downarrow{\scriptstyle \mathsf{iter}\langle g, \mathsf{in} \circ T(\pi_2)\rangle} \\ T(B \times A) & \xrightarrow[\langle g, \mathsf{in} \circ T(\pi_2)\rangle]{} & B \times A \\ {\scriptstyle T(\pi_2)}\downarrow & & \downarrow{\scriptstyle \pi_2} \\ TA & \xrightarrow[\mathsf{in}]{} & A \end{array}$$

Now, $\pi_2 \circ \mathsf{iter}\langle g, \mathsf{in} \circ T(\pi_2)\rangle \circ \mathsf{in} = \mathsf{in} \circ T(\pi_2 \circ \mathsf{iter}\langle g, \mathsf{in} \circ T(\pi_2)\rangle)$. So due to uniqueness, $\pi_2 \circ \mathsf{iter}\langle g, \mathsf{in} \circ T(\pi_2)\rangle = \mathsf{id}_A$. We now find that

$$\begin{array}{rcll} h \circ \mathsf{in} & = & \pi_1 \circ \mathsf{iter}\langle g, \mathsf{in} \circ T(\pi_2)\rangle \circ \mathsf{in} & \text{by definition of } h \\ & = & \pi_1 \circ \langle g, \mathsf{in} \circ T(\pi_2)\rangle \circ T(\mathsf{iter}\langle g, \mathsf{in} \circ T(\pi_2)\rangle) & \text{by top half of the diagram above} \\ & = & g \circ T(\mathsf{iter}\langle g, \mathsf{in} \circ T(\pi_2)\rangle) & \\ & = & g \circ T\langle \pi_1 \circ \mathsf{iter}\langle g, \mathsf{in} \circ T(\pi_2)\rangle, \pi_2 \circ \mathsf{iter}\langle g, \mathsf{in} \circ T(\pi_2)\rangle\rangle & \\ & = & g \circ T\langle \pi_1 \circ \mathsf{iter}\langle g, \mathsf{in} \circ T(\pi_2)\rangle, \mathsf{id}_A\rangle & \text{since } \pi_2 \circ \mathsf{iter}\langle g, \mathsf{in} \circ T(\pi_2)\rangle = \mathsf{id}_A \\ & = & g \circ T\langle h, \mathsf{id}_A\rangle & \text{by definition of } h. \end{array}$$

To prove uniqueness, assume that $h' : A \to B$ also makes the diagram commute, i.e. $h' \circ \mathsf{in} = g \circ T\langle h', \mathsf{id}\rangle$. Then

$$\begin{array}{rcl} \langle h', \mathsf{id}_A\rangle \circ \mathsf{in} & = & \langle h' \circ \mathsf{in}, \mathsf{in}\rangle \\ & = & \langle g \circ T\langle h', \mathsf{id}\rangle, \mathsf{in} \circ \mathsf{id}_{TA}\rangle \quad \text{by assumption} \\ & = & \langle g \circ T\langle h', \mathsf{id}_A\rangle, \mathsf{in} \circ T(\pi_2) \circ T\langle h', \mathsf{id}_A\rangle\rangle \\ & & \quad \text{since } \mathsf{id}_{TA} = T(\mathsf{id}_A) = T(\pi_2 \circ \langle h', \mathsf{id}_A\rangle) = T(\pi_2) \circ T\langle h', \mathsf{id}_A\rangle \\ & = & \langle g, \mathsf{in} \circ T(\pi_2)\rangle \circ T\langle h', \mathsf{id}_A\rangle, \end{array}$$

but then by uniqueness $\langle h', \mathsf{id}_A\rangle = \langle h, \mathsf{id}_A\rangle$ and hence $h' = h$. □

Note that for $(A, \mathsf{in}) = (\mathsf{Nat}, \big[\mathsf{Z}, \mathsf{S}\big])$ and $g = \big[g_1, g_2\big]$ the equation given by the diagram above – $h \circ \big[\mathsf{Z}, \mathsf{S}\big] = \big[g_1, g_2\big] \circ NAT\langle h, \mathsf{id}\rangle = \big[g_1, g_2\big] \circ (\mathsf{id} + \langle h, \mathsf{id}\rangle)$ – indeed implies the equations for primitive recursion given in Example 2.4.

We state a general (easy) property of initial T-algebras $(\mathsf{Init}T, \mathsf{in}, \mathsf{iter})$: $T(\mathsf{Init}T)$ is isomorphic to $\mathsf{Init}T$ via in, also know as Lambek's Lemma [8]:

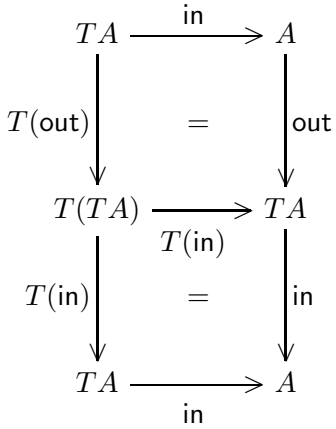
Lemma 2.6. If $(\mathsf{Init}T, \mathsf{in}, \mathsf{iter})$ is an initial T-algebra, then $\mathsf{Init}T$ is a fixed point of T via in. That is, there is a morphism $\mathsf{out} : \mathsf{Init}T \to T(\mathsf{Init}T)$ such that

$$\begin{array}{rcl} \mathsf{out} \circ \mathsf{in} & = & \mathsf{id}_{T(\mathsf{Init}T)} \\ \mathsf{in} \circ \mathsf{out} & = & \mathsf{id}_{\mathsf{Init}T} \end{array}$$

Proof. Let $(\mathsf{Init}T, \mathsf{in}, \mathsf{iter})$ be an initial T-algebra and abbreviate $\mathsf{Init}T$ by A. Note that $T(\mathsf{in}) : T(TA) \to TA$, so $(T(TA), T(\mathsf{in}))$ is a T-algebra and $\mathsf{iter}T(\mathsf{in}) : A \to TA$. Define

$$\mathsf{out} := \mathsf{iter}T(\mathsf{in}) : A \to TA$$

and consider the following commuting diagrams



As the outside diagram commutes, we find that $\mathsf{in} \circ \mathsf{out} = \mathsf{id}_A$ by uniqueness. But then $\mathsf{out} \circ \mathsf{in} = T(\mathsf{in}) \circ T(\mathsf{out}) = \mathsf{id}_{TA}$. □

Example 2.7 (Case/Pattern-matching)**.** A simple (the simplest) pattern for defining a function on the natural numbers is by pattern matching (or case analysis). A morphism $h : \mathsf{Nat} \to B$ is then defined by equations

$$\begin{cases} h \circ \mathsf{Z} &= g_1 \\ h \circ \mathsf{S} &= g_2 \end{cases}$$

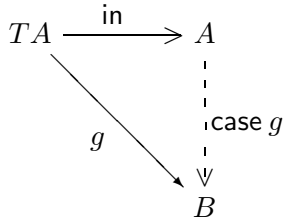
where $g_1 : 1 \to B$ and $g_2 : \mathsf{Nat} \to B$ This is the standard pattern matching construction that we know from functional programming, but without any recursion.

We state as a general lemma that over initial algebras, functions can be defined by pattern matching.

Lemma 2.8. Let $(A, \mathsf{in}, \mathsf{iter})$ be an initial T-algebra. Then, for every morphism $g : T(A) \to B$, there exists a unique morphism $h : A \to B$ such that

$$h \circ \mathsf{in} = g.$$

We denote this h by $\mathsf{case}\, g$. So, the following diagram commutes.



Proof. Let $(A, \mathsf{in}, \mathsf{iter})$ be an initial T-algebra. From Lemma 2.6 we know that there is a morphism $\mathsf{out} : A \to TA$ such that $\mathsf{in} \circ \mathsf{out} = \mathsf{id}_A$. Now take

$$\mathsf{case}\, g := g \circ \mathsf{out}.$$

Then $\mathsf{case}\, g \circ \mathsf{in} = g \circ \mathsf{out} \circ \mathsf{in} = g$. Furthermore, if $h \circ \mathsf{in} = g$, then $h = h \circ \mathsf{in} \circ \mathsf{out} = g \circ \mathsf{out}$. □

3. Weakening the notion of initial algebra

In the previous section we have seen what we can do with initial algebras in terms of defining functions on them. Specific patterns for defining functions (known from e.g. functional programming) have been distinguished. Now, in practice, especially in the practice of formal languages, the requirements on an initial algebra are quite strong: in syntax we are usually not dealing with an initial algebra, but with a weaker variant of it. This is mainly because in a formal language, we want type checking to be decidable, which conflicts with the strong requirement of *uniqueness* (which corresponds to an *extensional* equality). We therefore present here a number of weakened forms of initial algebra and we compare them with each other. The weakenings we encounter here are all related (in some way) to weakening of the uniqueness condition in the definition of initial algebra.

The first weakening is just obtained by lifting the uniqueness requirement in the definition of initial algebra.

Definition 3.1. An *iterative T-algebra* (also known as a weakly initial T-algebra) is a triple $(A, \mathsf{in}, \mathsf{iter})$ such that (A, in) is a T-algebra and for every morphism $g : TB \to B$ there exists a morphism $\mathsf{iter}\, g : A \to B$ such that the following diagram commutes.

$$\begin{array}{ccc} TA & \xrightarrow{\mathsf{in}} & A \\ {\scriptstyle T(\mathsf{iter}\, g)}\downarrow & = & \downarrow{\scriptstyle \mathsf{iter}\, g} \\ TB & \xrightarrow[g]{} & B \end{array}$$

We will denote an iterative T-algebra as $(\mathsf{Iter}T, \mathsf{in}_T, \mathsf{iter}_T)$, sometimes leaving the subscript T implicit. It should be noted that iterative T-algebras are not unique in any sense, so this notation should not obscure that there is no such thing as *the* iterative T-algebra.

Obviously, every initial T-algebra is an iterative T-algebra.

Remark 3.2. The notion of iterative algebra is really weaker than that of initial algebra. For example in the category **Set**, $(2\omega, [\mathsf{Z}, \mathsf{S}])$ is an iterative NAT-algebra, but also $(2\omega, [\mathsf{Z}, \mathsf{S}'])$, with $\mathsf{S}'(n) = \mathsf{S}(n)$, $\mathsf{S}'(\omega + n) = \mathsf{S}(n)$ is. On iterative algebras, the behavior of morphisms is only determined on the standard part (the finite elements) of the algebra, that is, in set-theoretic terms, those elements that are constructed by finitely many times applying the constructor in.

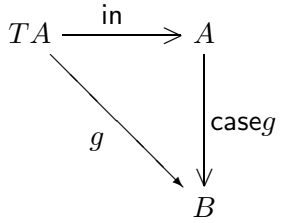
Definition 3.3. A triple $(A, \mathsf{in}, \mathsf{out})$ is a *T-fixed-point* if $\mathsf{in} : TA \to A$, $\mathsf{out} : A \to TA$ and

$$\begin{aligned} \mathsf{out} \circ \mathsf{in} &= \mathsf{id}_{T(A)}, \\ \mathsf{in} \circ \mathsf{out} &= \mathsf{id}_A. \end{aligned}$$

As a notation for a T-fixed point we use $(\mathsf{Fix}_T, \mathsf{in}_T, \mathsf{out}_T)$, where the same remarks as in Definition 3.1 apply.

Note that a T-fixed-point is both a T-algebra and a T-co-algebra. Due to Lemma 2.6, every initial T-algebra is a T-fixed point. As a matter of fact, the initiality states that $(\mathsf{Init}T, \mathsf{in}, \mathsf{iter})$ is the smallest T-fixed-point.

Definition 3.4. A triple $(A, \mathsf{in}, \mathsf{case})$ is a *T-algebra with case* if $\mathsf{in} : TA \to A$ and for every morphism $g : T(A) \to B$, there is a morphism $\mathsf{case}\, g : A \to B$ such that the following diagram commutes.



If the morphism $\mathsf{case}\, g$ is unique , $(A, \mathsf{in}, \mathsf{case})$ is a *T-algebra with unique case.*

Due to Lemma 2.8, every initial T-algebra $(A, \mathsf{in}, \mathsf{iter})$ is a T-algebra with unique case. Looking at the proof of Lemma 2.8, we observe that this implication already follows from the fact that $(A, \mathsf{in}, \mathsf{iter})$ is a T-fixed point. This implication holds also in the reverse direction, which yields the following lemma.

Lemma 3.5. A T-algebra with unique case is a T-fixed-point and vice versa.

Proof. That every T-fixed-point is a T-algebra with unique case follows immediately by inspection of the proof of Lemma 2.8.
For the reverse, let $(A, \mathsf{in}, \mathsf{case})$ be a T-algebra with unique case. Define $\mathsf{out} := \mathsf{case}\mathsf{id}_{TA} : A \to TA$. Then $\mathsf{out} \circ \mathsf{in} = \mathsf{id}_{TA}$. Furthermore, $\mathsf{in} \circ \mathsf{out} \circ \mathsf{in} = \mathsf{in}$, so $\mathsf{in} \circ \mathsf{out}$ makes the case-diagram commute for in, so $\mathsf{in} \circ \mathsf{out} = \mathsf{case}\, \mathsf{in} = \mathsf{id}_A$, due to uniqueness. □

In the proof above we observe that one of the equalities in the definition of T-fixed point 'defines' the case and the other one 'proves its uniqueness'. We therefore come to the following definition.

Definition 3.6. A *T-algebra with left-inverse* is a triple $(A, \mathsf{in}, \mathsf{out})$ such that (A, in) is a T-algebra and

$$\mathsf{out} \circ \mathsf{in} = \mathsf{id}_{TA}.$$

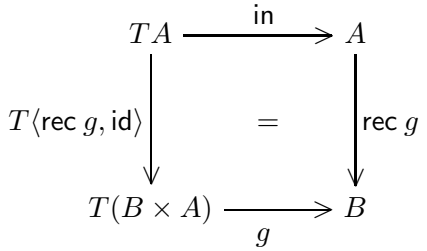
We denote a T-algebra with left-inverse by $(\mathsf{Linv}T, \mathsf{in}_G, \mathsf{out}_G)$ and drop the subscript G if clear from the context. The same remarks as in Definition 3.1 apply.

A T-algebra with left-inverse is also called a *retract*. Note that a T-algebra with left-inverse is the same as a T-co-algebra with right-inverse.

Lemma 3.7. A T-algebra with left-inverse is a T-algebra with case and vice versa.

Proof. That every T-algebra with left-inverse is a T-algebra with case follows immediately by inspection of the proof of Lemma 3.5. □

Definition 3.8 ([4])**.** A *primitive recursive T-algebra* is a triple $(A, \mathsf{in}, \mathsf{rec})$ where $\mathsf{in} : TA \to A$ and for every $g : T(B \times A) \to B$ there is a morphism $\mathsf{rec}\, g : A \to B$ such that the following diagram commutes.



Lemma 3.9. A primitive recursive T-algebra is an iterative T-algebra with case.

Proof. Let $(A, \mathsf{in}, \mathsf{rec})$ be a primitive recursive T-algebra. To show that it is an iterative T-algebra, let $g : TB \to B$. Then $g \circ T(\pi_1) : T(B \times A) \to B$. Define $\mathsf{iter} g := \mathsf{rec}(g \circ T(\pi_1)) : A \to B$, then

$$\begin{aligned} \mathsf{iter}\, g \circ \mathsf{in} &= \mathsf{rec}(g \circ T(\pi_1)) \circ \mathsf{in} \\ &= g \circ T(\pi_1) \circ T\langle \mathsf{rec}(g \circ T(\pi_1)), \mathsf{id}\rangle \\ &= g \circ T(\mathsf{rec}(g \circ T(\pi_1))) \\ &= g \circ T(\mathsf{iter}\, g). \end{aligned}$$

To define a case construct on $(A, \mathsf{in}, \mathsf{rec})$, let $g : TA \to B$. Then $g \circ T(\pi_2) : T(B \times A) \to B$. Define $\mathsf{case} g := \mathsf{rec}(g \circ T(\pi_2)) : A \to B$, then

$$\begin{aligned} \mathsf{case} g \circ \mathsf{in} &= \mathsf{rec}(g \circ T(\pi_2)) \circ \mathsf{in} \\ &= g \circ T(\pi_2) \circ T\langle \mathsf{rec}(g \circ T(\pi_2)), \mathsf{id}\rangle \\ &= g \circ T(\mathsf{id}) \\ &= g. \end{aligned}$$

□

We can summarize the results obtained so far in a diagram. This is given in Figure 4 and includes also the dual notions that we briefly introduce in the next section.

4. Dualizing

All the results given so far can be dualized, resulting in a notion of *terminal co*algebra:

Definition 4.1. (1) A *T-co-algebra in C* is a pair (A, f), with A an object and $f : A \to T(A)$.

(2) If (A, f) and (B, g) are T-co-algebras, *a morphism from (B, g) to (A, f)* is a morphism $h : B \to A$ such that the following diagram commutes.

$$\begin{array}{ccc} B & \xrightarrow{\;g\;} & TB \\ {\scriptstyle h}\downarrow & = & \downarrow{\scriptstyle Th} \\ A & \xrightarrow[\;f\;]{} & TA \end{array}$$

(3) A T-co-algebra (A, f) is *terminal* if it is terminal in the category of T-co-algebras, i.e. for every co-algebra (B, g) there's a unique h which makes the diagram above commute. If the terminal co-algebra (A, f) is clear from the context, we refer to this unique arrow h as $\mathsf{Intro}\, g$.

Our pet example for terminal co-algebras is the one for $STR(X) = \mathsf{Nat} \times X$, an object of infinite lists – or streams – of natural numbers, for which we write $(\mathsf{Str}, \langle \mathsf{H}, \mathsf{T} \rangle)$. Dualizing the notion of iterative morphisms we get *co-iterative* morphisms *to* Str.

Example 4.2 (Co-iteration for Str)**.** Suppose $\langle g_1, g_2\rangle : B \to \mathsf{Nat}\times B$. Then $h := \mathsf{Intro}\langle g_1, g_2\rangle : B \to \mathsf{Str}$ is the unique morphism such that such that $\langle \mathsf{H}, \mathsf{T}\rangle \circ h = STR(h) \circ \langle g_1, g_2\rangle = (\mathsf{id} \times h) \circ \langle g_1, g_2\rangle$. Then h satisfies the following co-recursion equations:

$$\begin{cases} \mathsf{H} \circ h &= g_1 \\ \mathsf{T} \circ h &= h \circ g_2 \end{cases}$$

So $h \circ b$ is the stream "$g_1 \circ b,\ g_1 \circ g_2 \circ b,\ g_1 \circ g_2^2 \circ b, \ldots$" .

Dualizing lemma 2.5 produces the following notion of *co-recursion*:

Lemma 4.3 (Primitive co-recursion on terminal co-algebra)**.** Let (A, f) be a terminal T-co-algebra and $g : B \to T(B + A)$. Then there is a unique morphism h such that

$$\begin{array}{ccc} B & \xrightarrow{\quad g \quad} & T(B+A) \\ {\scriptstyle h}\,\vdots & = & \vdots\,{\scriptstyle T[h,\mathsf{id}]} \\ \vee & & \vee \\ A & \xrightarrow[\quad f \quad]{} & TA \end{array}$$

commutes, namely $h = \mathsf{Intro}\big[g, T(\mathsf{inr}) \circ f\big] \circ \mathsf{inl}$. If the initial algebra (A, f) is clear from the context, we refer to this unique arrow h as $\mathsf{Corec}\, g$.

Example 4.4 (Primitive co-recursion for Str)**.** Suppose $\langle g_1, g_2\rangle : B \to STR(B + \mathsf{Str})$, i.e. $g_1 : B \to \mathsf{Nat}$ and $g_2 : B \to B + \mathsf{Str}$. Then $h := \mathsf{Corec}\langle g_1, g_2\rangle : B \to \mathsf{Str}$ satisfies the equation $\langle \mathsf{H}, \mathsf{T}\rangle \circ h = STR\big[h, \mathsf{id}\big] \circ \langle g_1, g_2\rangle = \mathsf{id} \times \big[h, \mathsf{id}\big] \circ \langle g_1, g_2\rangle$, so

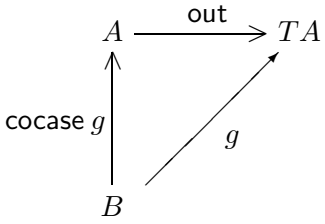
$$\begin{cases} \mathsf{H} \circ h &= g_1 \\ \mathsf{T} \circ h &= \big[h, \mathsf{id}\big] \circ g_2 \end{cases}$$

An example of a morphism that is easier to define with the primitive co-recursive scheme than with the co-iterative scheme is the inverse of $\langle \mathsf{H}, \mathsf{T}\rangle : \mathsf{Str} \to \mathsf{Nat} \times \mathsf{Str}$, i.e. a morphism $\mathsf{Cons} : \mathsf{Nat} \times \mathsf{Str} \to \mathsf{Str}$ such that $\langle \mathsf{H}, \mathsf{T}\rangle \circ \mathsf{Cons} = \mathsf{id}$ (take $g_1 = \pi_1$ and $g_2 = \mathsf{inr} \circ \pi_2$).

Just as for initial algebras, also the definition of terminal co-algebra splits up in two parts, the 'existence part' (there's an h such that...) and the 'uniqueness part' (the h is unique). Just as we have done in Section 3, we also weaken the notion of terminal co-algebra to *co-iterative algebra*, *co-algebra with co-case*, *co-algebra with unique co-case*, *co-recursive co-algebra* and *co-algebra with right-inverse*. These definitions are obtained by dualizing. Their usefulness is suggested by Definition 4.3 and Example 4.4.

Definition 4.5.
(1) A T-co-algebra, (A, f) is *co-iterative* (or weakly final), if for every T-co-algebra, (B, g) there exists an arrow h that makes the diagram in Definition 4.1 commute. The arrow h is sometimes denoted as $\mathsf{coiter}_T g$ and the co-iterative T-co-algebra as $(\mathsf{Coiter}T, \mathsf{out}_T, \mathsf{coiter}_T)$, where $\mathsf{out}_T : \mathsf{Coiter}T \to T(\mathsf{Coiter}T)$.
(2) A *primitive co-recursive T-co-algebra*, is a triple $(\mathsf{Corec}T, \mathsf{out}_T, \mathsf{corec}_T)$ that satisfies the properties for (A, f) in Lemma 4.3. That is, if we replace A by $\mathsf{Corec}T$ and f by out_T in the diagram, then $\mathsf{corec}_T g$ is the map h that makes the diagram commute for every $g : B \to T(B + A)$.

(3) A *T-co-algebra with co-case* is a triple $(A, \mathsf{out}, \mathsf{cocase})$ with $\mathsf{out} : A \to T(A)$ and for every morphism $g : B \to T(A)$, there is a morphism $\mathsf{cocase}\, g : B \to A$ such that the following diagram commutes.



If the morphism $\mathsf{cocase}\, g$ is unique, $(A, \mathsf{out}, \mathsf{cocase})$ is a *T-co-algebra with unique co-case.*

(4) A *T-co-algebra with right-inverse* is a triple $(A, \mathsf{out}, \mathsf{in})$ such that $\mathsf{out} : A \to T(A)$, $\mathsf{in} : T(A) \to A$ and $\mathsf{out} \circ \mathsf{in} = \mathsf{id}_{TA}$.

Concerning these dual notions about co-algebras we have the same lemmas as for algebras, so lemmas 2.6, 2.8, 3.5, 3.7 and 3.9 continue to hold after dualizing. Furthermore, it should be noted that a T-algebra with a left-inverse is the same as a T-co-algebra with a right-inverse. As an overview we have the diagram in Figure 4 that relates the definitions. An arrow from concept X to concept Y indicates that if we have a structure satisfying X, we can construct a structure satisfying Y. The numbers under the arrows indicate in which lemma this result has been proven. In case there is no number on an arrow in the left part of the diagram, the relation follows trivially from the definition. In the right part of the diagram, the arrows follow from the ones on the left by dualization.

One may wonder whether the reverse of Lemma 3.9 also holds: if a T-algebra is iterative and it has a case construction, is it a primitive recursive T-algebra?

We prove something weaker: if a category has 'enough' iterative algebras and algebras with left-inverse, then the category has a primitive recursive T-algebra.

Lemma 4.6. Let T be a functor and consider the indexed functor F_X (X an object of the category) defined by

$$F_X : Y \mapsto T(Y \times X).$$

Suppose that there is an iterative F_X-algebra $(\mathsf{Iter}F_X, \mathsf{in}_{F_X}, \mathsf{iter}_{F_X})$ for every X. Assume that

$$G : X \mapsto \mathsf{Iter}F_X$$

behaves functorially and has an algebra with left-inverse $(\mathsf{Linv}G, \mathsf{in}_G, \mathsf{out}_G)$.
Then there is a primitive recursive T-algebra $(A, \mathsf{in}_T, \mathsf{rec}_T)$, given by

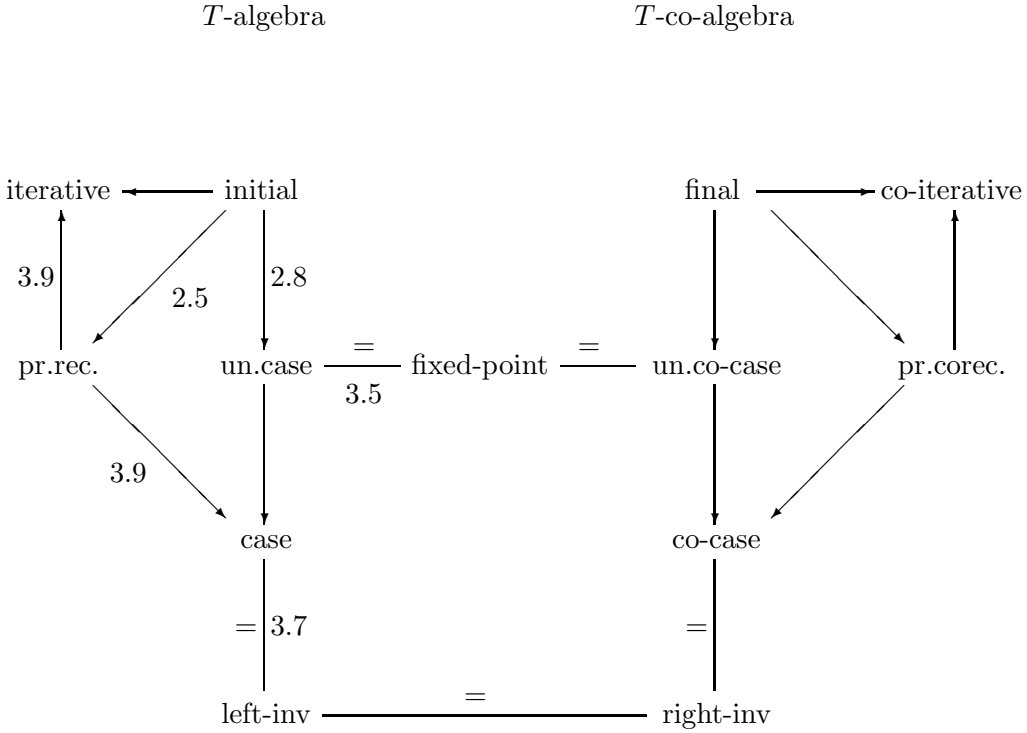
$$\begin{aligned} A &:= \mathsf{Linv}G, \\ \mathsf{in}_T &:= \mathsf{in}_G \circ \mathsf{in}_{F_A} \circ T\langle \mathsf{out}_G, \mathsf{id}_A \rangle, \\ \mathsf{rec}_T g &:= \mathsf{iter}_{F_A} g \circ \mathsf{out}_G. \end{aligned}$$

for $g : T(B \times A) \to B$.

Proof. Let T be the functor for which we want to define the primitive recursive T-algebra and define A, in_T and rec_T as in the lemma.

The fact that A is a G-algebra with left-inverse gives us

$$GA \xrightarrow{\mathsf{in}_G} A \xrightarrow{\mathsf{out}_G} GA$$



with $\mathsf{out}_G \circ \mathsf{in}_G = \mathsf{id}_{GA}$.

The fact that $GA = \mathsf{Iter}_{F_A}$ is an iterative F_A-algebra and $F_A(Y) = T(Y \times A)$ gives us the following equations and diagram.

$$\begin{aligned} T(GA \times A) &= F_A(\mathsf{Iter}_{F_A}) \\ GA &= \mathsf{Iter}_{F_A} \end{aligned}$$

$$\begin{array}{ccccccc} T(GA \times A) & \overset{=}{\longrightarrow} & F_A(GA) & \overset{=}{\longrightarrow} & F_A(\mathsf{Iter}F_A) & \xrightarrow{\mathsf{in}_{F_A}} & \mathsf{Iter}F_A \\ \downarrow{\scriptstyle T(\mathsf{iter}_A g \times \mathsf{id})} & & & & \downarrow{\scriptstyle F_A(\mathsf{iter}_{F_A} g)} & & \downarrow{\scriptstyle \mathsf{iter}_{F_A} g} \\ T(B \times A) & & \underset{=}{\longrightarrow} & & F_A(B) & \xrightarrow[g]{} & B \end{array}$$

So, $\mathsf{rec}_T g := \mathsf{iter}_{F_A} g \circ \mathsf{out}_G : A \to B$ for $g : T(B \times A) \to B$.
Also $\mathsf{in}_T := \mathsf{in}_G \circ \mathsf{in}_{F_A} \circ T\langle \mathsf{out}_G, \mathsf{id}_A \rangle : TA \to A$, because

$$TA \xrightarrow{T\langle \mathsf{out}_G, \mathsf{id}_A \rangle} T(GA \times A) = F_A(\mathsf{Iter}_{F_A}) \xrightarrow{\mathsf{in}_{F_A}} \mathsf{Iter}(F_A) = GA \xrightarrow{\mathsf{in}_G} A$$

so the maps rec_T and in_T have the right domain and range.

We now verify that the equality for primitive recursive T-algebras,

$$\mathsf{rec}_T g \circ \mathsf{in}_T = g \circ T\langle \mathsf{rec}_T g, \mathsf{id}\rangle$$

holds:

$$\begin{aligned}
\mathsf{rec}_T g \circ \mathsf{in}_T &= \underbrace{\mathsf{iter}_{F_A} g \circ \underbrace{\mathsf{out}_G \circ \mathsf{in}_G}_{\mathsf{id}_{GA}} \circ \mathsf{in}_{F_A}}_{g \circ T(\mathsf{iter}_{F_A} g \times \mathsf{id}_A)} \circ T\langle \mathsf{out}_G, \mathsf{id}_A\rangle \\
&= g \circ T\langle \mathsf{iter}_{F_A} g \circ \mathsf{out}_G, \mathsf{id}_A\rangle \\
&= g \circ T\langle \mathsf{rec}_T g, \mathsf{id}_A\rangle
\end{aligned}$$

□

Lemma 4.6 can also be dualized, obtaining a technique for defining primitive co-recursive co-algebras in a category that has "enough" co-iterative co-algebras and co-algebras with right-inverse. We state the lemma without proof.

Lemma 4.7. Let T be a functor and consider the indexed functor F_X (X an object of the category) defined by

$$F_X : Y \mapsto T(Y + X).$$

Suppose that there is an co-iterative F_X-co-algebra $(\mathsf{Coiter}F_X, \mathsf{out}_{F_X}, \mathsf{coiter}_{F_X})$ for every X. Assume that

$$G : X \mapsto \mathsf{Coiter}F_X$$

behaves functorially and has a co-algebra with right-inverse $(\mathsf{Rinv}G, \mathsf{out}_G, \mathsf{in}_G)$.
Then there is a primitive co-recursive T-co-algebra $(A, \mathsf{out}_T, \mathsf{corec}_T)$.

As an application of the results given here, we find that in system F (polymorphic λ-calculus [5]), we can define a primitive recursive T-algebra for T a positive type scheme, if we have left-inverses to all positive type schemes. This was first discussed in [4], where a left-inverse for the type scheme $T(\alpha)$ is called a *retract types* and denoted $\rho\alpha.T(\alpha)$. The syntactic (type theoretic) construction of [4] is here cast in a categorical framework, but Lemma 4.6 presents the same construction. That we do not need to assume the existence of iterative T-algebras in system F is just because they are already definable. In system F with retract types, the definition of the type Lemma 4.6 is just

$$A := \rho\beta.\forall\alpha.(T(\beta \times \alpha)\to\alpha)\to\alpha$$

and we remark that $G(\beta) := \forall\alpha.(T(\beta\times\alpha)\to\alpha)\to\alpha$ is just the definition of the iterative algebra $\mathsf{Iter}F_\beta$ in system F.

References

[1] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[2] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[3] J.N. Crossley. Reminiscences of logicians. In *Algebra and Logic. Papers from the 1974 Summer Research Institute of the Australian Mathematical Society, Monash University, Australia*, volume 450 of *Lecture Notes in Mathematics*, pages 1–62. Springer-Verlag, 1975.

[4] Herman Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Pettersson, and G. Plotkin, editors, *Informal Proceedings Workshop on Types for Proofs and Programs, Båstad, Sweden, 8–12 June 1992*, pages 193–217. Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., 1992.
[5] J.-Y. Girard et al. *Proofs and types*, volume 7 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, Cambridge, 1989.
[6] T. Hagino. A typed lambda calculus with categorical type constructions. In D.H. Pitt, A. Poigné, and D.E. Rydeheard, editors, *Category Theory and Computer Science*, volume 283 of *LNCS*, pages 140–157. Springer-Verlag, 1987.
[7] S.C. Kleene. λ-definability and recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936.
[8] J. Lambek. A fixed point theorem for complete categories. *Mathematisches Zeitschrift*, 103:151–161, 1968.
[9] Ralph Matthes. Monotone fixed-point types and strong normalization. In Georg Gottlob, Etienne Grandjean, and Katrin Seyr, editors, *Proceedings 12th Int. Workshop on Computer Science Logic, CSL'98, Brno, Czech Republic, 24–28 Aug 1998*, volume 1584 of *LNCS*, pages 298–312. Springer-Verlag, Berlin, 1999.
[10] N. P. Mendler. Recursive types and type constraints in second order lambda calculus. In *Proceeding 2nd IEEE Symposium on Logic in Computer Science, Ithaca*, pages 30–36, 1987.
[11] N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Ann. Pure Appl. Logic*, 51(1-2):159–172, 1991.
[12] Tarmo Uustalu and Varmo Vene. Least and greatest fixed points in intuitionistic natural deduction. *Theoretical Computer Science*, 272(1–2):315–339, 2002.
[13] G.C. Wraith. A note on categorical datatypes. In D.H. Pitt, A. Poigné, and D.E. Rydeheard, editors, *Category Theory and Computer Science*, volume 389 of *LNCS*, pages 118–127. Springer-Verlag, 1989.