Inductive and Coinductive types with Iteration and Recursion*

Herman Geuvers†
Faculty of Mathematics and Computer Science,
University of Nijmegen,
Toernooiveld 1,
6525 ED Nijmegen,
The Netherlands

July 1992

Abstract

We study (extensions of) simply and polymorphically typed lambda calculus from a point of view of how iterative and recursive functions on inductive types are represented. The inductive types can usually be understood as initial algebras in a certain category and then recursion can be defined in terms of iteration. However, in the syntax we often have only weak initiality, which makes the definition of recursion in terms of iteration inefficient or just impossible. We propose a categorical notion of (primitive) recursion which can easily be added as computation rule to a typed lambda calculus and gives us a clear view on what the dual of recursion, corecursion, on coinductive types is. (The same notion has, independently, been proposed by [Mendler 1991].) We look at how these syntactic notions work out in the simply typed lambda calculus and the polymorphic lambda calculus. It will turn out that in the syntax, recursion can be defined in terms of corecursion and vice versa using polymorphism: Polymorphic lambda calculus with a scheme for either recursion or corecursion suffices to be able to define the other. We compare our syntax for recursion and corecursion with that of Mendler ([Mendler 1987]) and use the latter to obtain meta properties as confluence and normalization.

1 Introduction

In this paper we want to look at formalizations of inductive and coinductive types in different typed lambda calculi, mainly extensions of the polymorphic lambda calculus. It is well-known that in polymorphic lambda calculus, many inductive data types can be defined (see e.g.[Böhm and Berarducci 1985] and [Girard et al. 1989]). In this paper we want to look at *how* functions on inductive types can be represented. Therefore, two ways of using the inductive building up of a type to define functions on that type are being distinguished, the *iterative* way and the *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 *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 $h : \mathrm{Nat} \to A$, with $h(0) = c, h(n+1) = f(h(n))$ (for $c : A, f : A \to A$) is iterative and $h : \mathrm{Nat} \to A$, with

*Extended notes of a talk given at the BRA-LF meeting in Edinburgh, May 1991

†herman@cs.kun.nl

$h(0) = c, h(n+1) = g(h(n), n)$ (for $c : A, g : A \times \mathrm{Nat} \to A$) is recursive. If one has pairing, the recursive functions can be defined using just iteration, which was essentially already shown by [Kleene 1936]. But if we work in a typed lambda calculus where pairing is not surjective, this translation of recursion in terms of iteration becomes inefficient and sometimes impossible. Moreover, if the calculus also incorporates some predicate logic, one would like to use the inductivity in doing proofs, which is not always straightforward (or just impossible.) We shall not go into the latter topic here; there is still a lot of work to be done in relating the work presented here to systems like AF2 by Krivine and Parigot (connections may be found in [Parigot 1992]) and Coq ([Dowek e.a. 1991].)

This asks for an explicit scheme for recursion in typed lambda calculus, which yields for, say, the natural numbers the scheme of Gödels T. To see how this can be done in general for inductive types, we are going to define a categorical notion of recursion (just like 'initial algebra' categorically represents the notion of iteration). One of the trade-offs is that we can dualize all this to get a notion of *corecursion* on coinductive types. These categorical notions of recursion and corecursion have independently been found by Mendler (see [Mendler 1991]) who treats these constructions in Martin-Löf type theory with predicative universes. What we define as (co)recursive (co)algebras are what Mendler calls '(co)algebras that admit simple primitive recursion'. We shall always use the term 'recursion', because, although the function-definition-scheme has a strong flavour of primitive recursion, one can define many more functions in polymorphic lambda calculus then just the primitive recursive ones. Coinductive types were first described in [Hagino 1987a] and [Hagino 1987b], with only a scheme for coiteration and without corecursion. Here we give a quite straightforward extension of simply typed lambda calculus with recursive and corecursive types.

A very surprising result is that in a polymorphic framework, if we have a notion of recursive types which reflects our notion of recursive algebra, then we can define corecursive types that correspond to corecursive coalgebras. By duality, this also works the other way around. This result will be given here syntactically: We define a polymorphic lambda calculus with recursive and corecursive types (that straightforwardly represents the categorical notions of recursive algebra and corecursive coalgebra) and show that the scheme for recursive types can be defined from the scheme for corecursive types and vice versa. We also look at a system of recursive and corecursive types defined by [Mendler 1987] and show that with either the scheme for recursive types or the scheme for corecursive types, there is a recursive Φ-algebra and a corecursive Φ-coalgebra in the syntax for every syntactic functor Φ (where syntactic functors are positive type schemes.)

2 The categorical perspective

As said, we shall get our intuitions about inductive and coinductive types from the field of category theory. The main notions in category theory related to this issue come from [Lambek 1968].

Definition 2.1 *Let C be a category, 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's a unique* h *which makes the above diagram commute.*

In a category with products, coproducts and terminal object, the initial algebra of the functor $TX = 1 + X$ is the natural numbers object, for which we write $(\mathsf{Nat}, [\mathsf{Z}, \mathsf{S}])$. The initial algebra of $TX = 1 + (A \times X)$ is the object of finite lists over A, $(\mathsf{List}_A, [\mathsf{Nil}, \mathsf{Cons}])$. In this paper our pet-example of an initial algebra will be $(\mathsf{Nat}, [\mathsf{Z}, \mathsf{S}])$, which will be used to illustrate the properties we are interested in. First take a look at how the iterative and recursive functions can be defined on Nat. (The example immediately generalizes to arbitrary initial algebras.)

Example 2.2 1. *For* $g : 1 + B \to B$ *we write* g_1 *for* $g \circ \mathsf{in}_1 : 1 \to B$ *and* g_2 *for* $g \circ \mathsf{in}_2 : B \to B$. *The* iteratively defined morphism from g_1, g_2, $\mathsf{Elim} g_1 g_2$, *is defined as the unique morphism* h *which makes the diagram commute, i.e.* $h \circ \mathsf{Z} = g_1$ *and* $h \circ \mathsf{S} = g_2 \circ h$.

2. *For* $g_1 : 1 \to B$, $g_2 : B \times \mathsf{Nat} \to B$, *the* recursively defined morphism from g_1 *and* g_2 *is constructed as follows.*
There exists a unique h *which makes the diagram*

$$\begin{array}{ccc} 1 + \mathsf{Nat} & \xrightarrow{[\mathsf{Z}, \mathsf{S}]} & \mathsf{Nat} \\ {\scriptstyle \mathsf{id} + h}\downarrow & = & \downarrow{\scriptstyle h} \\ 1 + (B \times \mathsf{Nat}) & \xrightarrow[\langle [g_1, g_2], [\mathsf{Z}, \mathsf{S} \circ \pi_2] \rangle]{} & B \times \mathsf{Nat} \end{array}$$

commute. That is $h \circ [\mathsf{Z}, \mathsf{S}] = \langle [g_1, g_2], [\mathsf{Z}, \mathsf{S} \circ \pi_2] \rangle \circ \mathsf{id} + h$. *If we write* $h_1 = \pi_1 \circ h$ *and* $h_2 = \pi_2 \circ h$ *we have the equalities*

$$\begin{aligned} h_1 \circ \mathsf{Z} &= g_1, \\ h_1 \circ \mathsf{S} &= g_2 \circ h, \\ h_2 \circ \mathsf{Z} &= \mathsf{Z}, \\ h_2 \circ \mathsf{S} &= \mathsf{S} \circ h_2. \end{aligned}$$

Now $h_2 = \mathsf{id}_{Nat}$ *by uniqueness and also* $h = \langle h_1, h_2 \rangle$, *so*

$$\begin{aligned} h_1 \circ \mathsf{Z} &= g_1, \\ h_1 \circ \mathsf{S} &= g_2 \circ \langle h_1, \mathsf{id} \rangle. \end{aligned}$$

So h_1 satisfies the recursion equalities and we define

$$\mathsf{Rec} g_1 g_2 := h_1.$$

Definition 2.3 *Let C be a category, T a functor from C to C.*

1. *A T-*coalgebra in *C is a pair (A, f), with A an object and $f : A \to TA$.*
2. *If (A, f) and (B, g) are T-coalgebras,* a morphism from (B, g) to (A, f) *is a morphism $h : B \to A$ such that the following diagram commutes.*

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

3. *A T-coalgebra (A, f) is* terminal *if it is terminal in the category of T-coalgebras, i.e. for every coalgebra (B, g) there's a unique h which makes the above diagram commute.*

Our pet example for terminal coalgebras is the one for $TX = \mathsf{Nat} \times X$, the object of infinite lists of natural numbers, for which we write $(\mathsf{Stream}, \langle \mathsf{H}, \mathsf{T} \rangle)$. We shall dualize the notions of iterative and recursive function to get *coiterative* and *corecursive* functions *to* Stream. (Again this example easily generalizes to the case for arbitrary terminal coalgebras.)

Example 2.4

1. *For $g : B \to \mathsf{Nat} \times B$, write g_1 for $\pi_1 \circ g : B \to \mathsf{Nat}$ and g_2 for $\pi_2 \circ g : B \to B$. The coiteratively defined morphism from g_1 and g_2, $\mathsf{Intro} g_1 g_2 : B \to \mathsf{Stream}$ is the (unique) morphism h for which the diagram commutes. That is, $H \circ h = g_1$ and $T \circ h = h \circ g_2$.*
   If j is a morphism from Nat *to* Nat*, one can define the morphism from* Stream *to* Stream *which applies f to every point in the stream as $\mathsf{Intro}(j \circ \mathsf{H})T$. Note that it is not so straightforward to define (coiteratively) a morphism which replaces the head of a stream by, say, zero. This, however, can easily be done using corecursion.*
2. *For $g_1 : B \to \mathsf{Nat}$, $g_2 : B \to B + \mathsf{Stream}$, the corecursively defined morphism from g_1 and g_2, $\mathsf{Corec} g_1 g_2$ is defined by $h \circ \mathsf{in}_1$, where h is the (unique) morphism which makes the diagram*

$$\begin{array}{ccc} B + \mathsf{Stream} & \xrightarrow{\quad [\langle g_1, g_2 \rangle, \langle \mathsf{H}, \mathsf{in}_2 \circ \mathsf{T} \rangle] \quad} & \mathsf{Nat} \times (B + \mathsf{Stream}) \\ {\scriptstyle h}\downarrow & = & \downarrow{\scriptstyle \mathsf{id} \times h} \\ \mathsf{Stream} & \xrightarrow[\quad \langle \mathsf{H}, \mathsf{T} \rangle \quad]{} & \mathsf{Nat} \times \mathsf{Stream} \end{array}$$

   commute. If we write $h_1 = h \circ \mathsf{in}_1, h_2 = h \circ \mathsf{in}_2$, then we have for h the following equations

$$\begin{aligned} \mathsf{H} \circ h_2 &= \mathsf{H}, \\ \mathsf{T} \circ h_2 &= h_2 \circ \mathsf{T}, \\ \mathsf{H} \circ h_1 &= g_1, \\ \mathsf{T} \circ h_1 &= h \circ g_2. \end{aligned}$$

Now $h_2 = \mathsf{id}$ *by uniqueness and also* $h = [h_1, h_2]$*, so*

$$\begin{aligned} \mathsf{H} \circ h_1 &= g_1, \\ \mathsf{T} \circ h_1 &= [h_1, \mathsf{id}] \circ g_2. \end{aligned}$$

These are the equations for corecursion; if $g_1 : B \to \mathsf{Nat}$ *and* $g_2 : B \to B + \mathsf{Stream}$*, then* $j : B \to \mathsf{Stream}$ *is* corecursively defined from g_1 and g_2 *if* $\mathsf{H} \circ j = g_1$ *and* $\mathsf{T} \circ j = [j, \mathsf{id}] \circ g_2$*. The function* $\mathsf{ZeroH} : \mathsf{Stream} \to \mathsf{Stream}$ *which changes the head of a stream into zero can now be defined as* $\mathsf{ZeroH} := \mathsf{Corec}(\mathsf{Z} \circ !)(\mathsf{in}_2 \circ \mathsf{T})$*, where* $!$ *is the unique morphism from* Stream *to* 1*. (Informally,* $\mathsf{Z} \circ !$ *is of course just* $\lambda s : \mathsf{Stream}.0$*.)*

As usual in categorical definitions, the definitions of initial algebra and terminal coalgebra split up in two parts, the 'existence part' (there's an h such that...) and the 'uniqueness part' (the h is unique.) In the following we shall sometimes refer to these two parts of the definition as the *existence property* and the *uniqueness property*.

In the typed lambda calculi that we shall consider, the inductive and coinductive types will not exactly represent initial algebras and terminal coalgebras. What the systems are lacking is the uniqueness property for the morphism h in 2.1, respectively 2.3. Algebras, respectively coalgebras, which only satisfy the existence property are called *weakly initial*, respectively *weakly terminal*.

Definition 2.5 *For* T *an endofunctor in a category* C*, The* T*-algebra, respectively* T*-coalgebra,* (A, f) *is* weakly initial*, respectively* weakly terminal*, if for every* T*-algebra, respectively* T*-coalgebra,* (B, g) *there exists an arrow* h *that makes the diagram in 2.1, respectively 2.3, commute.*

Remark 2.6 *The notion of weakly initial algebra is really weaker than that of initial algebra. For example in the category* **Set***,* $(2\omega, [\mathsf{Z}, \mathsf{S}])$ *is a weakly initial* $(\lambda X.1 + X)$*-algebra, but also* $(2\omega, [\mathsf{Z}, \mathsf{S}'])$*, with* $\mathsf{S}'(n) = \mathsf{S}(n)$*,* $\mathsf{S}'(\omega + n) = n$ *is. (On weakly initial algebras, the behaviour of morphisms is only determined on the standard part of the algebra, that is in settheoretic terms, those elements that are constructed by finitely many times applying the constructor* f*. Initiality says that the algebra is standard.)*

As we made serious use of the uniqueness property in constructing the recursive and corecursive functions, it's interesting to see how much we can do in weak initial algebras and weak terminal coalgebras. The construction of the iterative and coiterative functions of examples 2.2 and 2.4 can be done in the same way; we only loose the uniqueness property of the iteratively defined function. The construction of recursive and corecursive functions in a weak framework is not so straightforward. We shall study again the examples of natural numbers and streams of natural numbers. Fix a category C, which has weak products and coproducts. (So we *do* have e.g. $\pi_1 \circ \langle t_1, t_2 \rangle = t_1$ and $[t_1, t_2] \circ \mathsf{in}_1 = t_1$, but not $\langle \pi_1 \circ t, \pi_2 \circ t \rangle = t$ and $[t \circ \mathsf{in}_1, t \circ \mathsf{in}_2] = t$.) It will turn out that weak products and coproducts will cause some extra restrictions on the definability of functions. Therefore we shall also study what happens if product and coproduct are *semi*, that is for products $\langle f, g \rangle \circ h = \langle f \circ h, g \circ h \rangle$ and for coproducts $h \circ [f, g] = [h \circ f, h \circ g]$. The reason for not considering the strong products and coproducts in these examples is that in the syntax of typed lambda calculi product and coproduct are usually weak or semi. (The notions of semi product and semi coproduct are taken from [Hayashi 1985].)

Example 2.7 *(Recursion on a weak natural numbers object) Let* Nat *be a weakly initial* $\lambda X.1 + X$*-algebra. Consider the diagram in 2.2, where we defined recursion in terms of iteration and let* $h : \mathsf{Nat} \to B \times \mathsf{Nat}$ *be some morphism that makes the diagram commute, i.e.*

$$h \circ [\mathsf{Z}, \mathsf{S}] = \langle [g_1, g_2], [\mathsf{Z}, \mathsf{S} \circ \pi_2] \rangle \circ \mathrm{id} + h.$$

Applying projections to the left and injections to the right of the equation we obtain the following equalities (where $h_1 = \pi_1 \circ h$ *and* $h_2 = \pi_2 \circ h$*).*

$$\begin{aligned} h_1 \circ \mathsf{Z} &= g_1, \\ h_1 \circ \mathsf{S} &= g_2 \circ h, \\ h_2 \circ \mathsf{Z} &= \mathsf{Z}, \\ h_2 \circ \mathsf{S} &= \mathsf{S} \circ h_2. \end{aligned}$$

Nat *doesn't satisfy the uniqueness properties, so not necessarily* $h_2 = \mathsf{id}_{Nat}$*but only*

$$h_2 \circ \mathsf{S}^n \circ \mathsf{Z} = \mathsf{S}^n \circ \mathsf{Z}$$

for every $n \in \mathbf{N}$*, where* S^n *denotes an* n*-fold composition of* S*. Now we would like to deduce*

$$\begin{aligned} h_1 \circ \mathsf{Z} &= g_1, \\ h_1 \circ \mathsf{S}^{n+1} \circ \mathsf{Z} &= g_2 \circ \langle h_1, \mathsf{id} \rangle \circ \mathsf{S}^n \circ \mathsf{Z}, \end{aligned}$$

which says that h_1 *satisfies the recusion equations for the 'standard' natural numbers.*

- *For weak products this conclusion is only valid if* $g_2 = k \circ \pi_i$ *for some* $k : B \to B$ *or* $k : \mathsf{Nat} \to B$*. (Note that if* $g_2 = k \circ \pi_1$ *for some* $k : B \to B$*, then* h_1 *is just iteratively defined from* g_1 *and* k*, so only the case for* $g_2 = k \circ \pi_2$ *gives us really new functions, for instance the predecessor.)*
- *For semi products this conclusion is only valid for* $g_2 = k \circ \langle \pi_1, \pi_2 \rangle$ *for some* $k : B \times \mathsf{Nat} \to B$*, which is not a serious restriction: Just replace* g_2 *by* $g_2 \circ \langle \pi_1, \pi_2 \rangle$*.*

Example 2.8 *(Corecursion on a weak stream object) Let* Stream *be a weakly terminal* $\lambda X.\mathsf{Nat} \times X$*-coalgebra. Consider the diagram in 2.4, where we defined corecursion in terms of coiteration and let* $h : (B + \mathsf{Stream}) \to \mathsf{Stream}$ *be some morphism that makes the diagram commute. Write* $h_1 = h \circ \mathsf{in}_1$ *and* $h_2 = h \circ \mathsf{in}_2$*. We have the following equalities.*

$$\begin{aligned} \mathsf{H} \circ h_2 &= \mathsf{H}, \\ \mathsf{T} \circ h_2 &= h_2 \circ \mathsf{T}, \\ \mathsf{H} \circ h_1 &= g_1, \\ \mathsf{T} \circ h_1 &= h \circ g_2. \end{aligned}$$

Now we can not conclude $h \circ \mathsf{in}_2 = \mathsf{id}$*, because we don't have uniqueness, but we do have*

$$\mathsf{H} \circ \mathsf{T}^n \circ h_2 = \mathsf{H} \circ \mathsf{T}^n,$$

that is h_2 *is the identity on the 'standard' part of the stream (those points that can be obtained by finitely many applications of* H *or* T*.) Again we would like to conclude*

$$\begin{aligned} \mathsf{H} \circ h_1 &= g_1, \\ \mathsf{H} \circ \mathsf{T}^{n+1} \circ h_1 &= \mathsf{H} \circ \mathsf{T}^n \circ [h_1, \mathrm{id}] \circ g_2, \end{aligned}$$

that is h_1 *satisfies the corecursion equations for the 'standard' part of the stream.*

- *For weak coproducts this conclusion is only valid if* $g_2 = \mathsf{in}_i \circ k$ *for some* $k : B \to B$ *or* $k : B \to \mathsf{Stream}$. *(Note that if* $g_2 = \mathsf{in}_1 \circ k$ *for some* $k : B \to B$, *then* h_1 *is just coiteratively defined from* g_1 *and* k, *so only the case for* $g_2 = \mathsf{in}_2 \circ k$ *gives us really new functions, like for instance the function* ZeroH.*)*

- *For semi coproducts this conclusion is only valid if* $g_2 = [\mathsf{in}_1, \mathsf{in}_2] \circ k$ *for some* $k : B \to B + \mathsf{Stream}$. *again this is not a serious restriction: Just replace* g_2 *by* $[\mathsf{in}_1, \mathsf{in}_2] \circ g_2$.

For the morphism $\mathsf{ZeroH} : \mathsf{Stream} \to \mathsf{Stream}$ *which replaces the head by zero, defined in 2.4 by* $\mathsf{Corec}(\mathsf{Z} \circ !)(\mathsf{in}_2 \circ \mathsf{T})$, *we now have (for either weak or semi coproducts)*

$$\begin{aligned} \mathsf{H} \circ \mathsf{ZeroH} &= \mathsf{Z}, \\ \mathsf{H} \circ \mathsf{T}^{n+1} \circ \mathsf{ZeroH} &= \mathsf{H} \circ \mathsf{T}^{n+1}, \end{aligned}$$

so ZeroH *works fine on the standard part of the stream. That one can not, in general, define a morphism* ZeroH *such that* $\mathsf{T} \circ \mathsf{ZeroH} = \mathsf{T}$ *will be shown later, when we look at these examples in polymorphic lambda calculus which is an instance of a category with weakly initial algebras and weakly terminal coalgebras, semi products and weak coproducts.*

Remark 2.9 *With strong products and coproducts we would have similar problems in defining recursion and corecursion. The recursion equations would only be valid for the standard natural numbers and the corecursion equations would only be valid for the standard part of streams. The only advantage would be that the* $g_2 : B \times \mathsf{Nat} \to B$, *respectively the* $g_2 : B \to B + \mathsf{Stream}$ *can be taken arbitrarily.*

In Section 4 the polymorphic lambda calculus will be considered in which inductive and coinductive types can be defined which correspond to weakly initial algebras and weakly terminal coalgebras. It will be shown that recursion in that calculus is problematic from a point of view of efficiency. One solution could be to strengthen the reduction rules to get a stronger (extensional) equality. However, it's not possible to add some relatively easy reduction rules to the syntax to obtain the uniqueness property of initiality and terminality. (We can't say in an easy way that the only objects of a structure are the standard ones.) This is because the equality of (primitive) recursive functions can not be decided by an easy (decidable) equality. We can do something different, namely say that our functions should behave on the non-standard part as they behave on the standard part. Categorically, this can be obtained by strengthening the notion of weakly initial algebra and weakly terminal coalgebra a little bit, such that recursion 'works'. (That is for $\mathbf{N}$, for $c : A, g : A \times \mathrm{Nat} \to A$ there is a function $h : \mathrm{Nat} \to A$, with $h(0) = c$ and $h(n+1) = g(h(n), n)$.) These new notions will be called *recursive algebra* and *corecursive coalgebra*. The definitions are not difficult if one understands what makes it possible to define (co)recursion, in terms of (co)iteration.

Let in the following C be a category with weak products and weak coproducts and T a functor from C to C.

Definition 2.10 (A, f) *is a* recursive T-algebra *if* (A, f) *is a* T*-algebra and for every* $g : T(X \times A) \to X$ *there exists an* $h : A \to X$ *such that the following diagram commutes.*

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

Notice that this is the same as saying that (A, f) is weakly initial and that moreover, in the diagram for defining recursion in terms of iteration, $h_2 = \mathsf{id}$. (See 2.2)

Definition 2.11 (A, f) *is a* corecursive T-coalgebra *if* (A, f) *is a* T*-coalgebra and for every* $g : X \to T(X + A)$ *there exists an* $h : X \to A$ *such that the following diagram commutes.*

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

Again this is the same as saying that (A, f) is a weakly terminal T-coalgebra and that moreover, in the diagram for defining corecursion in terms of coiteration, $h_2 = \mathsf{id}$. (See 2.4)

When talking about weakly initial or recursive T-algebras and weakly terminal or corecursive T-coalgebras, it is convenient to denote the h that makes the diagram commute as a function of g. So we shall denote a weakly initial T-algebra by (A, f, Elim), where $\mathsf{Elim}g$ denotes a morphism h in 2.5 that makes the diagram commute. Similarly, we write (A, f, Intro) for a weakly terminal T-coalgebra, (A, f, Rec) for a recursive T-algebra and (A, f, Corec) for a corecursive T-coalgebra.

Examples 2.12 *1. If* $(\mathsf{Nat}, [\mathsf{Z}, \mathsf{S}], \mathsf{Rec})$ *is a recursive* $\lambda X.1 + X$*-algebra,* Rec *is a recursor on* Nat*: For* $[g_1, g_2] : 1 + (X \times \mathsf{Nat}) \to X$*,*

$$\begin{array}{rcl} \mathsf{Rec}[g_1, g_2] \circ \mathsf{Z} & = & g_1, \\ \mathsf{Rec}[g_1, g_2] \circ \mathsf{S} & = & g_2 \circ \langle \mathsf{Rec}[g_1, g_2], \mathsf{id}\rangle, \end{array}$$

so $\mathsf{Rec}[g_1, g_2]$ *is the recursively defined function from* g_1 *and* g_2*. We can define* $\mathsf{P} := \mathsf{Rec}[\mathsf{Z}, \pi_2]$ *and we have*

$$\begin{array}{rcl} \mathsf{P} \circ \mathsf{Z} & = & \mathsf{Z}, \\ \mathsf{P} \circ \mathsf{S} & = & \mathsf{id}. \end{array}$$

2. If $(\mathsf{Stream}, \langle \mathsf{H}, \mathsf{T}\rangle, \mathsf{Corec})$ *is a corecursive* $\lambda X.\mathsf{Nat} \times X$*-coalgebra. Then for* $\langle g_1, g_2\rangle : X \to \mathsf{Nat} \times (X + \mathsf{Stream})$*, the function* $\mathsf{Corec}\langle g_1, g_2\rangle$ *satisfies*

$$\begin{array}{rcl} H \circ \mathsf{Corec}\langle g_1, g_2\rangle & = & g_1, \\ T \circ \mathsf{Corec}\langle g_1, g_2\rangle & = & [\mathsf{Corec}\langle g_1, g_2\rangle, \mathsf{id}] \circ g_2, \end{array}$$

so $\mathsf{Corec}\langle g_1, g_2\rangle$ *is the corecursively defined function from* g_1 *and* g_2*. We can define*

$$\mathsf{ZeroH} := \mathsf{Corec}\langle \mathsf{Z}\circ!, \mathsf{in}_2 \circ \mathsf{T}\rangle$$

with

$$\begin{aligned} \mathsf{H} \circ \mathsf{ZeroH} &= \mathsf{Z}\circ!, \\ \mathsf{T} \circ \mathsf{ZeroH} &= \mathsf{T}. \end{aligned}$$

3 Extending simply typed lambda calculus with inductive and coinductive types

In his thesis ([Hagino 1987a]) Hagino derives from the notions of initial algebra and terminal coalgebra an extension of simply typed lambda calculus, which he calls categorical data types. This amounts to adding two schemes for defining a new type from a covariant functor from types to types. (In the notation of these schemes below we follow [Wraith 1989].) These new types come together with some constants and reduction rules. A covariant functor from types to types in $\lambda\!\rightarrow$ is a *positive type scheme* $\Phi(\alpha)$, that is a type Φ in which the free variable α occurs *positively*. (The type variable α occurs *positively* in the type Φ if $\alpha \notin \mathrm{FV}(\Phi)$, $\Phi \equiv \alpha$ or if $\Phi \equiv \Phi_1 \rightarrow \Phi_2$ and α occurs negatively in Φ_1, positively in Φ_2. The type variable α occurs *negatively* in Φ if $\alpha \notin \mathrm{FV}(\Phi)$, $\Phi \equiv \alpha$ or if $\Phi \equiv \Phi_1 \rightarrow \Phi_2$ and α occurs negatively in Φ_2, positively in Φ_1.) If $\Phi(\alpha)$ is a type scheme, with $\Phi(\tau)$ we mean the type Φ with τ substituted for α. If there's no ambiguity to which type variable α we're referring, we just write Φ in stead of $\Phi(\alpha)$.

A positive, respectively negative type scheme Φ can be applied to a function $f{:}\tau \rightarrow \rho$, obtaining $\Phi(f){:}\Phi(\rho) \rightarrow \Phi(\tau)$, respectively $\Phi(f){:}\Phi(\tau) \rightarrow \Phi(\rho)$ by *lifting*: $\alpha(f) \equiv f$, if $\alpha \notin \mathrm{FV}(\Phi)$, $\Phi(f) \equiv id_\Phi$ and if α occurs negatively in Φ_1, positively in Φ_2 then

$$\begin{aligned} (\Phi_1 \rightarrow \Phi_2)(f) &\equiv \lambda x{:}\Phi_1(\rho) \rightarrow \Phi_2(\rho).\lambda y{:}\Phi_1(\tau).\Phi_2(f)(x(\Phi_1(f)y)), \\ (\Phi_2 \rightarrow \Phi_1)(f) &\equiv \lambda x{:}\Phi_2(\tau) \rightarrow \Phi_1(\tau).\lambda y{:}\Phi_2(\rho).\Phi_1(f)(x(\Phi_2(f)y)). \end{aligned}$$

Definition 3.1 *Let* $\Phi_1, \Phi_2, \ldots, \Phi_n$ *be types in the simply typed lambda calculus in which the typevariable* α *occurs positively. The* sum scheme *for constructing data types is the following.*

$\sigma =$ **sum** α **with constructors**

$$\begin{array}{lcl} c_1 & : & \Phi_1 \rightarrow \alpha \\ c_2 & : & \Phi_2 \rightarrow \alpha \\ & \vdots & \\ c_n & : & \Phi_n \rightarrow \alpha \end{array}$$

end

A declaration of a type σ *using this sum scheme gives rise to an extension of the language of* $\lambda\!\rightarrow$ *with*

1. *a closed type* σ
2. *constants* $c_i{:}\Phi_i(\sigma) \rightarrow \sigma$ *for* $1 \leq i \leq n$,
3. *for every type* τ, $Elim_\tau{:}(\Phi_1(\tau) \rightarrow \tau) \rightarrow (\Phi_2(\tau) \rightarrow \tau) \rightarrow \ldots \rightarrow (\Phi_n(\tau) \rightarrow \tau) \rightarrow \sigma \rightarrow \tau$.

The reduction relation is extended with the rule

$$Elim_\tau M_1 M_2 \ldots M_n(c_i t) \longrightarrow M_i(\Phi_i(Elim_\tau M_1 \ldots M_n)t)$$

An easy example of a type defined by the sum scheme is $\sigma+\tau$ (for σ and τ types), representing the the disjoint sum of σ and τ.

$\sigma + \tau =$ **sum** α **with constructors**
inl : $\sigma \to \alpha$
inr : $\tau \to \alpha$
end

with inl : $\sigma \to \sigma + \tau$, inr : $\tau \to \sigma + \tau$ and for $M_1{:}\sigma \to \rho$, $M_2{:}\tau \to \rho$, $[M_1, M_2] : \sigma + \tau \to \rho$.

The sequence of type schemes in the sum scheme can also be empty, allowing us to define the unit type by

$1 =$ **sum** α **with constructors**
$*$: α
end

We have $* : 1$ and for any $t : \tau$, $!(t) : 1 \to \tau$ with $!(t)(*) \longrightarrow t$.

Definition 3.2 *Let $\Phi_1, \Phi_2, \ldots, \Phi_n$ be types in the simply typed lambda calculus in which the typevariable α occurs positively. The* product scheme *for constructing new data types is the following.*

$\sigma =$ **product** α **with destructors**
d_1 : $\alpha \to \Phi_1$
d_2 : $\alpha \to \Phi_2$
$\vdots$
d_n : $\alpha \to \Phi_n$
end

A declaration of a type σ using this product scheme gives rise to an extension of the language of $\lambda \to$ with

1. *a closed type σ,*
2. *constants $d_i{:}\sigma \to \Phi_i(\sigma)$, for $1 \leq i \leq n$,*
3. *for every type τ, $Intro_\tau{:}(\tau \to \Phi_1(\tau)) \to (\tau \to \Phi_2(\tau)) \to \ldots \to (\tau \to \Phi_n(\tau)) \to \tau \to \sigma$.*

The reduction relation is extended with the rule

$$d_i(Intro_\tau M_1 M_2 \ldots M_n t) \longrightarrow \Phi_i(Intro_\tau M_1 \ldots M_n)(M_i t)$$

The straightforward example of a type defined by the product scheme is $\sigma \times \tau$ (for σ and τ types), representing the the product of σ and τ.

$\sigma \times \tau =$ **product** α **with destructors**
fst : $\alpha \to \sigma$
snd : $\alpha \to \tau$
end

with fst : $\sigma \times \tau \to \sigma$, snd : $\sigma \times \tau \to \tau$ and for $M_1{:}\rho \to \sigma$, $M_2{:}\rho \to \tau$, $<M_1, M_2> : \rho \to \sigma \times \tau$.

Remark 3.3 *The type σ defined by the sum scheme from $\Phi_1(\alpha), \ldots \Phi_n(\alpha)$, will be denoted by $\mu\alpha.(\Phi_1(\alpha) + \ldots + \Phi_n(\alpha))$. The type σ defined by the product scheme from $\Phi_1(\alpha), \ldots \Phi_n(\alpha)$, will be denoted by $\nu\alpha.(\Phi_1(\alpha) \times \ldots \times \Phi_n(\alpha))$. This is also how these types should be read: as (weakly) initial algebras of $TX = \Phi_1(X) + \ldots + \Phi_n(X)$ and (weakly) terminal coalgebras of $TX = \Phi_1(X) \times \ldots \times \Phi_n(X)$, respectively. (So dualising is of course not the same as reversing all the arrows in a sum scheme to obtain a product scheme!)*

Definition 3.4 $\lambda \rightarrow^{ind}$ *is the simply typed lambda calculus extended with sum scheme and product scheme.*

Example 3.5 *The iterative functions on an inductive type can be straightforwardly defined by the Elim construct. Write Nat for* $\mu\alpha.1 + \alpha$*, then for* $c{:}\tau$ *and* $f{:}\tau\rightarrow\tau$*,* $Elim(\lambda z.c)f{:}\mathrm{Nat}\rightarrow\tau$ *is the iteratively defined function from* c *and* f*. The recursive functions can be defined by translating recursion in terms of iteration as is done in 2.2. For* $c{:}\tau$*,* $g{:}\tau\rightarrow\mathrm{Nat}\rightarrow\tau$*, define* $Recc g := \mathrm{fst}\circ\mathrm{Elim}(\lambda z.\langle c,0\rangle)(\langle\lambda x.g(\mathrm{fst}x)(\mathrm{snd}x), S\circ\mathrm{snd}\rangle)$ *and we have*

$$\begin{aligned} \mathrm{Rec}cg0 &\longrightarrow\!\!\!\!\!\rightarrow\ c, \\ \mathrm{Rec}cg(S^{n+1}(0)) &\longrightarrow\!\!\!\!\!\rightarrow\ g(\mathrm{Rec}cg(S^n0))(\mathrm{snd}(\mathrm{Elim}(\lambda z.\langle c,0\rangle)(\langle\lambda x.g(\mathrm{fst}x)(\mathrm{snd}x), S\circ\mathrm{snd}\rangle)(S^n(0))))). \end{aligned}$$

This recursor only works for terms of type Nat which are of the form S^n0, but moreover, it is quite inefficient (compared to, for instance, the recursor in Gödels T).

Proposition 3.6 *The predecessor function of type* Nat→Nat*, defined in terms of iteration in* $\lambda \rightarrow^{ind}$ *computes the predecessor of a numeral* $n+1$ *in* $3n+2$ *steps.*

Proof The predecessor function P is the $\beta\eta$-normal form of $\mathrm{Rec}0(\lambda xy.y)$, so

$$P \equiv \mathrm{fst}\circ\mathrm{Elim}_{\mathrm{Nat}\times\mathrm{Nat}}(\lambda z.\langle 0,0\rangle)(\langle\mathrm{snd}, S\circ\mathrm{snd}\rangle).$$

Now

$$P(S^{n+1}0) \longrightarrow\!\!\!\!\!\rightarrow_2 \mathrm{snd}(\mathrm{Elim}(\lambda z.\langle 0,0\rangle)(\langle\mathrm{snd}, S\circ\mathrm{snd}\rangle)(S^n(0))),$$

$$\mathrm{snd}(\mathrm{Elim}(\lambda z.\langle 0,0\rangle)(\langle\mathrm{snd}, S\circ\mathrm{snd}\rangle)(0)) \longrightarrow\!\!\!\!\!\rightarrow_3 0$$

and with induction one proves that

$$\mathrm{snd}(\mathrm{Elim}(\lambda z.\langle 0,0\rangle)(\langle\mathrm{snd}, S\circ\mathrm{snd}\rangle)(S^{n+1}(0))) \longrightarrow\!\!\!\!\!\rightarrow_3 S(\mathrm{snd}(\mathrm{Elim}(\lambda z.\langle 0,0\rangle)(\langle\mathrm{snd}, S\circ\mathrm{snd}\rangle)(S^n(0)))).$$

Proposition 3.7 *In* $\lambda \rightarrow^{ind}$ *there is no term* $P{:}\mathrm{Nat}\rightarrow\mathrm{Nat}$ *with*

$$P(S0) = 0 \text{ and}$$

$$P(Sx) = x$$

for x *a variable of type Nat.*

Proof This follows by the Church-Rosser property for reduction in $\lambda \rightarrow^{ind}$. (See [Hagino 1987b].) If $P(Sx) = x$, then $P(Sx) \longrightarrow\!\!\!\!\!\rightarrow x$. Analyzing the possible structure of P one can conclude that if $P(Sx) \longrightarrow\!\!\!\!\!\rightarrow x$, then not at the same time $P(S0) \longrightarrow\!\!\!\!\!\rightarrow 0$. This proposition is also an immediate corollary of the same proposition for system F in 4.

If one tries to do corecursion on the coinductive types in $\lambda \rightarrow^{ind}$, a similar situation occurs. For Stream $:= \nu\alpha.\mathrm{Nat}\times\alpha$, one can define ZeroH:Stream→Stream which replaces the head by 0 using the definable corecursion in weakly terminal coalgebras. We do not have $T(\mathrm{ZeroH}s) = Ts$, for s a Stream, but just $H(\mathrm{ZeroH}s) = 0$ and $H(T^{n+1}(\mathrm{ZeroH}s)) = H(T^{n+1}s)$. One can also show that there can be no term ZeroH:Stream→Stream such that $T(\mathrm{ZeroH}s) = s$ for a variable s:Stream. (Using the Church-Rosser property or as a corollary of the same proposition for system F.)

There are of course ways to strengthen the equalities of the sum and product scheme to get real recursion and corecursion. The initiality can be restored totally by adding the conditional rewrite rule

If $h(c_i t) = M_i(\sigma_i[h]t)$ for $1 \leq i \leq n$ and M_i and t of appropriate type, then $h \longrightarrow \mathrm{Elim}_\tau M_1 \ldots M_n$.

However, conditional rewrite rules are metatheoretically very complicated (the rewriting depends on the typing and on the previously generated equality.) Another alternative, which restores part of the unicity is to add a rewrite rule

$$Elim_\sigma c_1 \ldots c_n \longrightarrow Id_\sigma,$$

for

$\sigma =$ **sum** α **with constructors**

$c_1 \quad : \quad \Phi_1 \to \alpha$

$c_2 \quad : \quad \Phi_2 \to \alpha$

$\vdots$

$c_n \quad : \quad \Phi_n \to \alpha$

end

This is not enough to obtain a recursive algebra, because the Elim constructor doesn't automatically commute with pairing. One has to add

$$\mathrm{snd} \circ \mathrm{Elim}_{\tau\times\sigma}\langle g_1, c_1 \circ \Phi_1(\mathrm{snd})\rangle, \ldots, \langle g_n, c_n \circ \Phi_n(\mathrm{snd})\rangle \longrightarrow \mathrm{Elim}_\sigma c_1 \ldots c_n.$$

In the proof of Proposition 3.6, we then have that $\mathrm{Rec0}(\lambda x y.y)(S^{n+1}0) \longrightarrow\!\!\!\!\!\rightarrow S^N(0)$ in a constant number of steps. In this case it is of course better to take $\times$ (and $+$ if we add similar rules for the product scheme) as primitive type constructors. The new reduction rule is not a very pretty one.

We can also follow the categorical definitions of recursion and corecursion and strengthen the sum and product schemes themselves. (Again it is best to take $\times$ and $+$ as primitives.) For the sum scheme this would lead to the type σ with the same constructors and further

1. for every type τ, $\mathrm{Rec}_\tau{:}(\sigma_1(\tau\times\sigma)\to\tau)\to(\sigma_2(\tau\times\sigma)\to\tau)\to\ldots(\sigma_n(\tau\times\sigma)\to\tau)\to\sigma\to\tau$,
2. the reduction rule $\mathrm{Rec}_\tau M_1 M_2 \ldots M_n(c_i t) \longrightarrow M_i(\sigma_i[\langle \mathrm{Rec}_\tau M_1 \ldots M_n, \mathrm{id}\rangle]t)$

For the product scheme we would also get the same type σ with the same destructors and further

1. for every type τ, $\mathrm{Corec}_\tau{:}(\tau\to\sigma_1(\tau+\sigma))\to(\tau\to\sigma_2(\tau+\sigma))\to\ldots(\tau\to\sigma_n(\tau+\sigma))\to\tau\to\sigma$,
2. the reduction rule $d_i(\mathrm{Corec}_\tau M_1 M_2 \ldots M_n t) \longrightarrow \sigma_i[[\mathrm{id}, \mathrm{Corec}_\tau M_1 \ldots M_n]](M_i t)$.

Call the system $\lambda \to^{ind}$ with modified sum and product scheme as above $\lambda \to^{rec}$. Without proof we give the following proposition.

Proposition 3.8 *In the system $\lambda \to^{rec}$ the inductive types are recursive algebras and the coinductive types are corecursive coalgebras. (For the appropriate functors.)*

4 The polymorphic lambda calculus

We just give the rules to fix our notation and shall not go into the system further, assuming it is familiar. We write $\times$ and $+$ for the definable weak product and coproduct: $\sigma \times \tau \equiv \forall\alpha.(\sigma\to\tau\to\alpha)\to\alpha$ and $\sigma + \tau \equiv \forall\alpha.(\sigma\to\alpha)\to(\tau\to\alpha)\to\alpha$.) We could also have added $\times$ and $+$ as new type constructors with extra rules turning them into a weak product and coproduct. This however is inconvenient: The added $\times$ and $+$ would not be functorial (e.g. $\times$: Types $\times$ Types$\to$Types does not preserve identities and composition), whereas the definable $\times$ and $+$ are functorial by construction if we assume an η-reduction rule. (See Definition 4.2 and the discussion.)

Definition 4.1 *1. The set of types of F, $\mathbf{T}$, is defined by the following abstract syntax.*

$$\mathbf{T} ::= TypVar \mid \mathbf{T}\to\mathbf{T} \mid \forall TypVar.\mathbf{T}$$

2. The expressions of F, T, are defined by the following abstract syntax.

$$T ::= Var \mid TT \mid T\mathbf{T} \mid \lambda Var{:}\mathbf{T}.T \mid \Lambda TypVar.T$$

3. A context *is a sequence of* declarations *$x{:}\sigma$ ($x \in Var$ and $\sigma \in \mathbf{T}$), where it is assumed that if $x{:}\sigma$ and $y{:}\tau$ are different declarations in the same context, then $x \not\equiv y$.*

4. The typing rules for deriving judgements of the form $\Gamma \vdash M{:}\sigma$ for Γ a context, M an expression and σ a type, are the following.

- *If $x{:}\sigma$ is in Γ, then $\Gamma \vdash x{:}\sigma$,*
- $$\frac{\Gamma \vdash M{:}\sigma\to\tau \quad \Gamma \vdash N{:}\sigma}{\Gamma \vdash MN{:}\tau} \qquad \frac{\Gamma, x{:}\sigma \vdash M{:}\tau}{\Gamma \vdash \lambda x{:}\sigma.M{:}\sigma\to\tau}$$
- $$\frac{\Gamma \vdash M{:}\forall\alpha.\sigma}{\Gamma \vdash M\tau{:}\sigma[\tau/\alpha]} \text{ if } \tau \in \mathbf{T}. \qquad \frac{\Gamma \vdash M{:}\sigma}{\Gamma \vdash \Lambda\alpha.M{:}\forall\alpha.\sigma} \text{ if } \alpha \notin FTV(\Gamma).$$

FTV denotes the set of free type variables ($TypVar$.)

5. The one step reduction rules are the following.

- $(\lambda x{:}\sigma.M)N \longrightarrow_\beta M[N/x]$,
- $\lambda x{:}\sigma.Mx \longrightarrow_\eta M$ *if* $x \notin FV(M)$,
- $(\Lambda\alpha.M)\tau \longrightarrow_\beta M[\tau/\alpha]$,
- $\Lambda\alpha.M\alpha \longrightarrow_\eta M$ *if* $\alpha \notin FTV(M)$.

FV denotes the free term variables (Var.) One step reduction, $\longrightarrow$, is defined as the union of $\longrightarrow_\beta$ and $\longrightarrow_\eta$. The relations $\longrightarrow\!\!\!\!\!\rightarrow$ and $=$ are respectively defined as the transitive, reflexive and the transitive, reflexive, symmetric closure of $\longrightarrow$.

Here, $t'[t/u]$ denotes the substitution of t for the variable u in t'. Substitution is done with the usual care, renaming bound variables such that no free variable becomes bound after substitution.

Type variables will be denoted by the lower case Greek characters α, β and γ, term variables will be denoted by lower case Roman characters. The set of expressions typable in the context Γ with type σ is denoted by $\mathrm{Term}(\sigma, \Gamma)$.

We want to discuss categorical notions like weak initiality in the syntax and therefore define need a syntactic notion of functor. This will be covered by the (well-known) notion of positive or negative type scheme.

Definition 4.2 *1. A* type scheme *in F is a type $\Phi(\alpha)$ where α marks all occurrences (possibly none) of α.*

2. A type scheme $\Phi(\alpha)$ can be positive *or* negative *(but also none of the both), which is defined by induction on the structure of $\Phi(\alpha)$ as follows.*

(a) If $\alpha \notin FTV(\Phi(\alpha))$, then $\Phi(\alpha)$ is positive *and* negative,

(b) if $\Phi(\alpha) \equiv \alpha$ then $\Phi(\alpha)$ is positive,

(c) if $\Phi(\alpha) \equiv \Phi_1(\alpha) \to \Phi_2(\alpha)$, then $\Phi(\alpha)$ is positive *if $\Phi_1(\alpha)$ is negative and $\Phi_2(\alpha)$ is positive, $\Phi(\alpha)$ is* negative *if $\Phi_1(\alpha)$ is positive and $\Phi_2(\alpha)$ is negative,*

(d) if $\Phi(\alpha) \equiv \forall\beta.\Phi'(\alpha)$ then $\Phi(\alpha) \equiv \forall\beta.\Phi'(\alpha)$ is positive *(resp.* negative*) if $\Phi'(\alpha)$ is positive (resp. negative) .*

3. A positive (resp. negative) type scheme $\Phi(\alpha)$ works covariantly (resp. contravariantly) on a term $f{:}\sigma \to \tau$, obtaining a term $\Phi(f)$ of type $\Phi(\sigma) \to \Phi(\tau)$ (resp. $\Phi(\tau) \to \Phi(\sigma)$), by lifting, *defined inductively as follows. (Let $f{:}\sigma \to \tau$.)*

(a) If $\alpha \notin FTV(\Phi(\alpha))$, then $\Phi(f) := \mathrm{id}_{\Phi(\alpha)}$,

(b) if $\Phi(\alpha) \equiv \alpha$ then $\Phi(f) := f$,

(c) if $\Phi(\alpha) \equiv \Phi_1(\alpha) \to \Phi_2(\alpha)$, then, if $\Phi(\alpha)$ is positive, $\Phi(f) := \lambda x{:}\Phi_1(\sigma) \to \Phi_2(\sigma).\lambda y{:}\Phi_1(\tau).\Phi_2(f)(x(\Phi_1(f)y)))$, if $\Phi(\alpha)$ is negative, $\Phi(f) := \lambda x{:}\Phi_2(\tau) \to \Phi_1(\tau).\lambda y{:}\Phi_2(\sigma).\Phi_1(f)(x(\Phi_2(f)y)))$,

(d) if $\Phi(\alpha) \equiv \forall\beta.\Phi'(\alpha)$, then, if $\Phi(\alpha)$ is positive, $\Phi(f) := \lambda x{:}\Phi(\sigma).\lambda\beta.\Phi'(f)(x\beta)$, if $\Phi(\alpha)$ is negative, then $\Phi(f) := \lambda x{:}\Phi(\tau).\lambda\beta.\Phi'(f)(x\beta)$.

It is easy to check that the lifting preserves identity and composition: $\Phi(\mathrm{id}) = \mathrm{id}$ and if $\Phi(\alpha)$ is positive then $\Phi(f \circ g) = \Phi(f) \circ \Phi(g)$, if $\Phi(\alpha)$ is negative then $\Phi(f \circ g) = \Phi(g) \circ \Phi(f)$. This also works for type schemes containing $\times$ or $+$, if we interpret $\times$ and $+$ as the definable weak product and coproduct:

$$
\begin{array}{rcl}
\sigma \times \tau & =: & \forall\alpha.(\sigma \to \tau \to \alpha) \to \alpha, \\
\mathrm{fst} & := & \lambda x{:}\sigma \times \tau.x\sigma(\lambda y{:}\sigma.\lambda z{:}\tau.y), \\
\mathrm{snd} & := & \lambda x{:}\sigma \times \tau.x\tau(\lambda y{:}\sigma.\lambda z{:}\tau.z), \\
<f, g> & := & \lambda z{:}\rho.\lambda\alpha.\lambda k{:}\sigma \to \tau \to \rho.k(fz)(gz), \\
 & & \text{for } f{:}\sigma \to \rho, g{:}\tau \to \rho, \\
\sigma + \tau & =: & \forall\alpha.(\sigma \to \alpha) \to (\tau \to \alpha) \to \alpha, \\
\mathrm{inl} & := & \lambda x{:}\sigma.\lambda\alpha.\lambda f : \sigma \to \alpha.\lambda g{:}\tau \to \alpha.fx, \\
\mathrm{inr} & := & \lambda x{:}\sigma.\lambda\alpha.\lambda f : \sigma \to \alpha.\lambda g{:}\tau \to \alpha.gx, \\
[f, g] & := & \lambda z{:}\sigma + \tau.z\rho fg, \\
 & & \text{for } f{:}\rho \to \sigma, g{:}\rho \to \tau.
\end{array}
$$

It should be remarked here that if one lifts $f{:}\sigma{\to}\tau$ via a type scheme $\Phi(\alpha) \equiv \Phi_1(\alpha) \times \Phi_2(\alpha)$ (respectively $\Phi(\alpha) \equiv \Phi_1(\alpha) + \Phi_2(\alpha)$) according to Definition 4.2, this does not give the (expected) result $\Phi(f) = \lambda x{:}\Phi(\sigma).<\Phi_1(f)(\mathrm{fst}x), \Phi_2(f)(\mathrm{snd}x)>$ (respectively $\Phi(f) = [\mathrm{inl} \circ \Phi_1(f), \mathrm{inr} \circ \Phi_2(f)]$.) If we take the latter definition for lifting a function via a product or sum, this doesn't yield functoriality of $\times$ and $+$. We introduce some new notation to denote this lifting via $\times$ and $+$.

Definition 4.3 *Let $f{:}\sigma{\to}\tau$, $g{:}\mu{\to}\rho$.*

1. $\mathsf{times}fg : \sigma \times \mu{\to}\tau \times \rho$ *is defined by*

$$\mathsf{times}fg := \lambda z{:}\sigma \times \mu.\lambda\beta.\lambda y{:}\tau{\to}\rho{\to}\beta.z\beta(\lambda p{:}\sigma.\lambda q{:}\mu.y(fp)(gq)).$$

2. $\mathsf{plus}fg : \sigma + \mu{\to}\tau + \rho$ *is defined by*

$$\mathsf{plus}fg := \lambda z{:}\sigma + \mu.\lambda\beta.\lambda y_1{:}\tau{\to}\beta.y_2{:}\rho{\to}\beta.z\beta(y_1 \circ f)(y_2 \circ g).$$

Now for $f{:}\sigma{\to}\tau$, if $\Phi(\alpha) = \Phi_1(\alpha) \times \Phi_2(\alpha)$ then $\Phi(f) = \mathsf{times}(\Phi_1(f))(\Phi_2(f))$ and if $\Psi(\alpha) = \Psi_1(\alpha) + \Psi_2(\alpha)$ then $\Psi(f) = \mathsf{plus}(\Psi_1(f))(\Psi_2(f))$. Let's state some more easy facts about times and plus, some of which will be used later.

Fact 4.4 *For f, g, h and k of the right type we have.*

1. $\mathsf{plus}fg \circ \mathrm{inl} = \mathrm{inl} \circ f$,
2. $\mathsf{plus}fg \circ \mathrm{inr} = \mathrm{inr} \circ g$,
3. $\mathsf{plus}fg \circ \mathsf{plus}hk = \mathsf{plus}(f \circ h)(g \circ k)$,
4. $\mathsf{times}fg \circ \mathsf{times}hk = \mathsf{times}(f \circ h)(g \circ k)$,
5. $[f, g] \circ \mathsf{plus}hk = [f \circ h, g \circ k]$,
6. $\mathsf{plus}hk \circ <f, g> = <h \circ f, k \circ g>$.

(In general we don't have $\mathrm{fst} \circ \mathsf{times}fg = f \circ \mathrm{fst}$ or $\mathrm{snd} \circ \mathsf{times}fg = g \circ \mathrm{snd}$.)

Positive (negative) type schemes can really be viewed as (contravariant) functors in the syntax of polymorphic lambda calculus. (Consider a syntax with countably many variables of every type and view types as objects and terms of type $\sigma{\to}\tau$ as morphisms from σ to τ.) The positive type schemes are a syntactic version of covariant functors. Similarly we also have syntactic versions of weakly initial (terminal) (co)algebras and (co)recursive (co)algebras.

Definition 4.5 *Suppose we work in (an extension of) polymorphic lambda calculus where we have fixed a notation for weak products and coproducts (e.g. the second order definable ones.) Let $\Phi(\alpha)$ be a positive type scheme.*

1. *The triple* $(\sigma_0, M_0, \mathrm{Elim})$ *is a* syntactic weakly initial Φ-algebra *if*
   (a) $\sigma_0 \in \mathbf{T}$,
   (b) $\vdash M_0{:}\Phi(\sigma_0){\to}\sigma_0$,

(c) $\vdash \mathrm{Elim}{:}\forall\beta.(\Phi(\beta){\to}\beta){\to}\sigma_0{\to}\beta$,

such that

$$\mathrm{Elim}\tau g \circ M_0 = g \circ \Phi(\mathrm{Elim}\tau g)$$

for any $\tau \in \mathbf{T}$ *and* $\Gamma \vdash g{:}\Phi(\tau){\to}\tau$.

2. *The triple* $(\sigma_1, M_1, \mathrm{Intro})$ *is a* syntactic weakly terminal Φ-coalgebra *if*

   (a) $\sigma_1 \in \mathbf{T}$,

   (b) $\vdash f_1{:}\sigma_1{\to}\Phi(\sigma_1)$,

   (c) $\vdash \mathrm{Intro}{:}\forall\beta.(\beta{\to}\Phi(\beta)){\to}\beta{\to}\sigma_1$,

   such that

   $$M_1 \circ \mathrm{Intro}\tau g = \Phi(\mathrm{Intro}\tau g) \circ g$$

   for any $\tau \in \mathbf{T}$ *and* $\Gamma \vdash g{:}\tau{\to}\Phi(\tau)$.

3. *The triple* $(\sigma_0, M_0, \mathrm{Rec})$ *is a* syntactic recursive Φ-algebra *if*

   (a) $\sigma_0 \in \mathbf{T}$,

   (b) $\vdash M_0{:}\Phi(\sigma_0){\to}\sigma_0$,

   (c) $\vdash \mathrm{Rec}{:}\forall\beta.(\Phi(\beta \times \sigma_0){\to}\beta){\to}\sigma_0{\to}\beta$,

   such that

   $$\mathrm{Rec}\tau g \circ M_0 = g \circ \Phi(<\mathrm{Rec}\tau g, \mathrm{id}>)$$

   for any $\tau \in \mathbf{T}$ *and* $\Gamma \vdash g{:}\Phi(\tau \times \sigma_0){\to}\tau$.

4. *The triple* $(\sigma_1, M_1, \mathrm{Corec})$ *is a* syntactic corecursive Φ-coalgebra *if*

   (a) $\sigma_1 \in \mathbf{T}$,

   (b) $\vdash f_1{:}\sigma_1{\to}\Phi(\sigma_1)$,

   (c) $\vdash \mathrm{Corec}{:}\forall\beta.(\beta{\to}\Phi(\beta + \sigma_1)){\to}\beta{\to}\sigma_1$,

   such that

   $$M_1 \circ \mathrm{Corec}\tau g = \Phi([\mathrm{Corec}\tau g, \mathrm{id}]) \circ g$$

   for any $\tau \in \mathbf{T}$ *and* $\Gamma \vdash g{:}\tau{\to}\Phi(\tau + \sigma_1)$.

We have the following proposition, of which the first part is a syntactic version of a result in [Reynolds and Plotkin 1990] and the second part is a result of [Wraith 1989]. In fact, the first part of the proposition says that the algebraic inductive data types can be represented in F, which result originally goes back to [Böhm and Berarducci 1985]. Here we just want to give these representations in short; for further details one may consult [Böhm and Berarducci 1985], [Leivant 1989] or [Girard et al. 1989].

Proposition 4.6 *We work in the system* F. *Let* $\Phi(\alpha)$ *be a positive type scheme. Then*

1. *There is a syntactic weakly initial* Φ*-algebra.*

2. *There is a syntactic weakly terminal* Φ*-coalgebra.*

Proof Let $\Phi(\alpha)$ be a positive type scheme.

1. Define $\sigma_0 := \forall\alpha.(\Phi(\alpha)\to\alpha)\to\alpha$, $M_0 := \lambda x{:}\Phi(\sigma).\lambda\alpha.\lambda g{:}\Phi(\alpha)\to\alpha.g(\Phi(\mathrm{Elim}\alpha g)x)$, and Elim $:=$ $\lambda\alpha.\lambda g{:}\Phi(\alpha)\to\alpha.\lambda y{:}\sigma.y\alpha g$. Now $(\sigma_0, M_0, \mathrm{Elim})$ is a syntactic weakly initial Φ-algebra.

2. Define $\sigma_1 := \forall\alpha.(\forall\beta.(\beta\to\Phi(\beta))\to\beta\to\alpha)\to\alpha$,
   $M_1 := \lambda x{:}\sigma.x(\Phi(\sigma))(\lambda\beta.\lambda g{:}\beta\to\Phi(\beta).\lambda z{:}\beta.\Phi(\mathrm{Intro}\beta g)(gx))$, and
   Intro $:= \lambda\alpha.\lambda g{:}\alpha\to\Phi(\alpha).\lambda y{:}\alpha.\lambda\beta.\lambda h{:}\forall\gamma.(\gamma\to\Phi(\gamma))\to\gamma\to\beta.h\alpha gy$. Now $(\sigma_1, M_1, \mathrm{Intro})$ is a syntactic weakly terminal Φ-coalgebra.

We don't know whether there are syntactic recursive algebras or syntactic corecursive coalgebras in F. The answer seems to be negative. The well-known definitions of algebraic data-types in F (which are almost the ones defined in the proof above) do in general not allow recursion or corecursion, as will be illustrated by looking at the examples of natural numbers and streams of natural numbers. This means that recursion and corecursion have to be defined in terms of iteration and coiteration, using the techniques discussed in the Examples 2.7 and 2.8. As was noticed there, it makes a difference whether product and coproduct are weak or semi, so let's note the following fact.

Fact 4.7 *The definable coproduct in F is a weak coproduct, but the definable product in F is a semi product.*
(That is, $<f,g>\circ h = <f\circ h, g\circ h>$, but not $h\circ[f,g] = [h\circ f, h\circ g]$)

Example 4.8 *(See also Example 3.5 and Proposition 3.6.) We define recursive functions on the weak initial algebra of natural numbers.*
Let $(\mathrm{Nat}, M_0, \mathrm{Elim})$ *be the syntactic weak initial algebra of* $\Phi(\alpha) = 1+\alpha$, *as given in the proof of 4.6, where* 1 *and* $+$ *are the second order definable ones. (One can also take the well-known polymorphic Church numerals, which is a slight modification of our type* Nat. *The exposition is not essentially different, but we want to use our categorical understanding of recursion of 2.7.)*
So $\mathrm{Nat} = \forall\alpha((1+\alpha)\to\alpha)\to\alpha$, $M_0 = \lambda x{:}1+\mathrm{Nat}.\lambda\alpha.\lambda g.g((\mathrm{id}+\mathrm{Elim}\alpha g)x)$ *and*
$\mathrm{Elim} = \lambda\alpha.\lambda g.\lambda y{:}\mathrm{Nat}.y\alpha g$. *Now we first define* $\mathbf{Z} := M_0\circ\mathrm{inl}$ *and* $\mathbf{S} := M_0\circ\mathrm{inr}$.
Following Example 2.7, we now define $\mathrm{Rec}g = \mathrm{fst}\circ\mathrm{Elim}(\tau\times\mathrm{Nat})(<g, [\mathbf{Z}, \mathbf{S}\circ\mathrm{snd}]>)$, *for* $g{:}1+\tau\times\mathrm{Nat}\to\tau$. *If*

$$g = [g_1, k\circ<\mathrm{fst},\mathrm{snd}>]$$

for some $k{:}\tau\times\mathrm{Nat}\to\tau$, *we obtain the recursion equalities for* $\mathrm{Rec}g$*:*

$$\begin{aligned}\mathrm{Rec}g\circ\mathbf{Z} &= g_1,\\ \mathrm{Rec}g\circ\mathbf{S}^{n+1}\circ\mathbf{Z} &= k\circ<\mathrm{Rec}g,\mathrm{id}>\circ\mathbf{S}^n\circ\mathbf{Z}.\end{aligned}$$

(See 2.7 for the restriction on the form of g; the product is semi here.) The predecessor is now defined by taking $g = [\mathbf{Z},\mathrm{snd}]$, *so* $P := \mathrm{fst}\circ\mathrm{Elim}(\tau\times\mathrm{Nat})(<[\mathbf{Z},\mathrm{snd}], [\mathbf{Z},\mathbf{S}\circ\mathrm{snd}]>)$. *Notice that* $P(\mathbf{S}t) = t$ *only for standard natural numbers, i.e. for* $t = \mathbf{S}^n(\mathbf{Z}*)$, *with* $*$ *the unique (closed) term of type* 1. *Also notice that P computes the predecessor of a natural number n in a number of steps of order n.*

Example 4.9 *We define corecursive functions on streams of natural numbers. Take for* Stream *the syntactic weakly terminal Φ-coalgebra as in the proof of 4.6, for* $\Phi(\alpha) = \mathrm{Nat}\times\alpha$. *So* $\mathrm{Stream} = \forall\alpha.(\forall\beta.(\beta\to(\mathrm{Nat}\times\beta))\to\beta\to\alpha)\to\alpha$, $M_1 = \lambda x{:}\mathrm{Stream}.x(\mathrm{Nat}\times\sigma)(\lambda\beta.\lambda g{:}\beta\to\mathrm{Nat}\times$

$\beta.\lambda z{:}\beta.(\mathrm{id} \times \mathrm{Intro}\beta g)(gx))$, *and*
$\mathrm{Intro} = \lambda\alpha.\lambda g{:}\alpha{\to}\mathrm{Nat} \times \alpha.\lambda y{:}\alpha.\lambda\beta.\lambda h{:}\forall\gamma.(\gamma{\to}\mathrm{Nat} \times \gamma){\to}\gamma{\to}\beta.h\alpha gy$. *We can define head and tail functions by taking* $\mathbf{H} := \mathrm{fst} \circ M_1$ *and* $\mathbf{T} := \mathrm{snd} \circ M_1$. *Following Example 2.8, we now define for* $g{:}\tau{\to}\mathrm{Nat} \times (\tau + \mathrm{Stream})$ $\mathrm{Corec}g := \mathrm{Intro}(\tau + \mathrm{Stream})([g, <H, \mathrm{inr} \circ \mathbf{T}>]) \circ \mathrm{inl}$. *As the coproduct is not semi, but weak (see 2.8), we find that only for* $g = <g_1, \mathrm{in} \circ k>$ *for* in *is* inr *or* inl *and some* $k{:}\mathrm{Stream}{\to}B$ *or* $k{:}\mathrm{Stream}{\to}\mathrm{Stream}$ *we obtain the corecursion equations.*

$$\begin{aligned} \mathbf{H} \circ \mathrm{Corec}g &= g_1, \\ \mathbf{T} \circ \mathrm{Corec}g &= [\mathrm{Corec}g, \mathrm{id}] \circ \mathrm{snd} \circ g. \end{aligned}$$

The function that replaces the head of a stream by zero is now defined by $\mathrm{ZeroH} := \mathrm{Corec}{<}\mathbf{Z}, \mathrm{inr}{\circ}\ \mathbf{T}{>}$

It is really impossible to define a 'global' predecessor on the weakly initial natural numbers as described above (and similarly for the polymorphic Church numerals.) Also it is impossible to define a global ZeroH-function on the weakly terminal streams as described above. This is shown in the following proposition.

Proposition 4.10 *1. For* $\mathrm{Nat} = \forall\alpha((1 + \alpha){\to}\alpha){\to}\alpha$, *there is no closed term* $P{:}\mathrm{Nat}{\to}\mathrm{Nat}$ *such that* $P(\mathbf{S}x) = x$ *for* x *a variable.*

2. For $\mathrm{Stream} := \forall\beta.(\forall\gamma.(\gamma{\to}\mathrm{Nat}){\to}(\gamma{\to}\gamma){\to}\gamma{\to}\beta){\to}\beta$ *there is no closed term* $\mathrm{ZeroH}{:}\mathrm{Stream}{\to}\mathrm{Stream}$ *such that* $\mathbf{T}(\mathrm{ZeroH}y) = \mathbf{T}y$ *and* $\mathbf{H}(\mathrm{ZeroH}y) = 0$ *for* y *a variable.*

Proof Both cases immediately by the Church-Rosser property for the system F.

One can show in general that the (co)inductive types in system F as defined above do not allow (co)recursion, i.e. they are weakly initial (terminal) (co)algebras.

5 Recursive algebras and corecursive coalgebras and polymorphism

We define an extension of F which includes a syntactic formalization of recursive algebras and corecursive coalgebras. Then we show the remarkable fact that in this system one can define recursive algebras in terms of corecursive coalgebras and vice versa, so one of the two is enough to be able to define the other. This fact has a counterpart in semantics in the form that every K-model of polymorphic lambda calculus that has a recursive T-algebras for every expressible functor T, also has a corecursive T-coalgebra for every expressible functor T and vice versa. (The notion of K-model is in [Reynolds and Plotkin 1990]; it is a syntax dependent notion of model for F, described by giving a set of constraints that a structure and an interpretation function should satisfy in order to be a model. As it covers a lot of known models it serves well as a framework for stating this property semantically. Also the notion of expressible functor comes from [Reynolds and Plotkin 1990]; roughly speaking, a functor is expressible if there is a type scheme whose interpretation in the model (as a function of the free type variable) is a the functor.)
We then want to relate our extension of F with recursive (and corecursive) types to a system described by [Mendler 1987]. The latter system has a different scheme for recursive (and corecursive) types, the syntax of which is a bit too weak to define one in terms of the other.

We can, however, interpret our system with recursive and corecursive types in Mendler's (with either recursive or corecursive types.) This is done by showing that the system has syntactic recursive Φ-algebras and syntactic corecursive Φ-coalgebras for every positive type scheme Φ. (See Definition 4.5.)

Definition 5.1 *The system $F_{(co)rec}$ is the system F extended with the following.*

1. *The set of types* $\mathbf{T}$ *is extended with $\mu\alpha.\Phi(\alpha)$ and $\nu\alpha.\Phi(\alpha)$, for $\Phi(\alpha)$ a positive type scheme.*

2. *For $\mu\alpha.\Phi(\alpha)$ and $\nu\alpha.\Phi(\alpha)$ we have the extra constants*

$$\begin{array}{rlrl} \mathbf{In}_\mu & : \Phi(\mu)\to\mu & \mathrm{Rec}_\mu & : \forall\alpha.(\Phi(\alpha\times\mu)\to\alpha)\to\mu\to\alpha, \\ \mathbf{Out}_\nu & : \nu\to\Phi(\nu) & \mathrm{Corec}_\nu & : \forall\alpha.(\alpha\to\Phi(\alpha+\nu))\to\alpha\to\nu. \end{array}$$

3. *Reduction rules for μ and ν:*

$$\begin{array}{rcl} \mathrm{Rec}\tau g(\mathbf{In}x) & \longrightarrow_\mu & g(\Phi(<\mathrm{Rec}\tau g,\mathrm{id}>)x), \\ \mathbf{Out}(\mathrm{Corec}\tau gx) & \longrightarrow_\nu & \Phi([\mathrm{Corec}\tau g,\mathrm{id}])(gx). \end{array}$$

(μ abbreviates $\mu\alpha.\Phi(\alpha)$ and ν abbreviates $\nu\alpha.\Phi(\alpha)$.)

We now have the following theorem, stating that in polymorphic lambda calculus, if one has recursive types the corecursive types can be defined and vice versa.

Theorem 5.2 *In F we can define ν,* **Out** *and* Corec *in terms of μ,* **In** *and* Rec *and vice versa.*

Proof Suppose we only have the rules for μ, **In** and Rec and let Φ be a positive type scheme. Define

$$\begin{array}{rcl} \Theta(\alpha) & := & \forall\gamma.(\forall\beta.(\beta\to\Phi(\beta+\alpha)\times\beta)\to\gamma)\to\gamma, \\ \sigma & := & \mu\alpha.\Theta(\alpha), \\ \mathrm{Corec} & := & \lambda\alpha.\lambda g{:}\alpha\to\Phi(\alpha+\sigma).\lambda x{:}\alpha.\mathbf{In}_\sigma(\lambda\gamma.\lambda h{:}\forall\beta.(\beta\to\Phi(\beta+\sigma)\times\beta)\to\gamma.h\alpha<g,x>), \\ \mathbf{Out} & := & \mathrm{Rec}_\sigma(\Phi(\sigma))(\lambda z{:}\Theta(\Phi(\sigma)\times\sigma).z(\Phi(\sigma))(\lambda\gamma.\lambda p.\Phi([\mathrm{Corec}\gamma(\Phi(\mathsf{plus}(\mathrm{id})(\mathrm{snd}))\circ p_1),\mathrm{snd}])(p_1p_2))), \end{array}$$

where p_1 and p_2 abbreviate $\mathrm{fst}p$ and $\mathrm{snd}p$ (the type of p is $(\gamma\to\Phi(\gamma+(\Phi(\sigma)\times\sigma)))\times\sigma$.) We have

$$\mathrm{Rec}_\sigma\tau g(\mathbf{In}_\sigma x)\longrightarrow_\mu g(\Theta(<\mathrm{Rec}_\sigma\tau g,\mathrm{id}>)x),$$

for any g and x of appropriate types and we want to show

$$\mathbf{Out}(\mathrm{Corec}\tau gx)\longrightarrow\!\!\!\!\!\rightarrow\Phi([\mathrm{Corec}\tau g,\mathrm{id}])(gx))$$

for $g{:}\tau\to\Phi(\tau+\sigma)$ and $x{:}\tau$.

To make things easier to read we omit the type information in lambda abstractions. Let τ be a type, $g{:}\tau\to\Phi(\tau+\sigma)$ and $x{:}\tau$ and abbreviate $F\equiv\lambda\gamma p.\Phi([\mathrm{Corec}\gamma(\Phi(\mathsf{plus}(\mathrm{id})(\mathrm{snd}))\circ p_1),\mathrm{snd}])(p_1p_2)$. The lifting of an f via Θ is defined as (following 4.2)

$$\Theta(f)\equiv\lambda t\gamma k.t\gamma(\lambda\beta z.k\beta(\mathsf{times}(\lambda yx.\mathsf{plus}(\mathrm{id})f(yx))(\mathrm{id})z).$$

Now

$$\begin{aligned}
\mathbf{Out}(\mathrm{Corec}\tau g x) &\longrightarrow\!\!\!\!\!\rightarrow \mathrm{Rec}_\sigma(\Phi(\sigma))(\lambda z.z(\Phi(\sigma))F)(\mathbf{In}_\sigma(\lambda\gamma h.h\tau{<}g,x{>})) \\
&\longrightarrow\!\!\!\!\!\rightarrow \Theta({<}\mathrm{Rec}_\sigma(\Phi(\sigma))(\lambda z.z(\Phi(\sigma))F),\mathrm{id}{>})(\lambda\gamma h.h\tau{<}g,x{>})(\Phi(\sigma))F \\
&\longrightarrow\!\!\!\!\!\rightarrow (\lambda\beta z.F\beta(\mathsf{times}(\lambda yx.\Phi(\mathsf{plus}(\mathrm{id})F)(yx)(\mathrm{id})z))\tau{<}g,x{>} \\
&\longrightarrow\!\!\!\!\!\rightarrow \Phi([\mathrm{Corec}\tau(\Phi(\mathsf{plus}(\mathrm{id})(\mathrm{snd}))\circ\Phi(\mathsf{plus}(\mathrm{id})F)\circ g,\mathrm{snd}])(\Phi(\mathsf{plus}(\mathrm{id})F)(gx)) \\
&\longrightarrow\!\!\!\!\!\rightarrow \Phi([\mathrm{Corec}\tau g,\mathrm{snd}]\circ\mathsf{plus}(\mathrm{id})F))(gx) \\
&\longrightarrow\!\!\!\!\!\rightarrow \Phi([\mathrm{Corec}\tau g,\mathrm{id}])(gx).
\end{aligned}$$

The other way around, suppose we only have rules for ν, **Out** and Corec and let Φ be a positive type scheme. Define

$$\begin{aligned}
\Theta(\alpha) &= \forall\beta.(\Phi(\beta\times\alpha)\to\beta)\to\beta, \\
\sigma &:= \nu\alpha.\Theta(\alpha), \\
\mathrm{Rec} &:= \lambda\alpha\lambda g{:}\Phi(\alpha\times\sigma)\to\alpha.\lambda x{:}\sigma.\mathbf{Out}_\sigma x\alpha g, \\
\mathbf{In} &:= \mathrm{Corec}_\sigma(\Phi(\sigma))(\lambda z{:}\Phi(\sigma).\lambda\beta.\lambda h{:}\Phi(\beta\times\Phi(\sigma))\to\beta.h(\Phi({<}\mathrm{Rec}\beta(h\circ\Phi(\mathrm{id}\times\mathrm{inr})),\mathrm{inr}{>})z)).
\end{aligned}$$

We have

$$\mathbf{Out}_\sigma(\mathrm{Corec}_\sigma\tau g x)\longrightarrow\!\!\!\!\!\rightarrow\Theta([\mathrm{Corec}_\sigma\tau g,\mathrm{id}])(gx))$$

and we want to prove

$$\mathrm{Rec}\tau g(\mathbf{In}x)\longrightarrow\!\!\!\!\!\rightarrow g(\Phi({<}\mathrm{Rec}\tau g,\mathrm{id}{>})x).$$

We omit again type information in lambda abstractions and abbreviate $F\equiv\lambda z\beta h.h(\Phi({<}\mathrm{Rec}\beta(h\circ\Phi(\mathrm{id}\times\mathrm{inr})),\mathrm{inr}{>})z)$. According to Definition 4.2,

$$\Theta(f)\equiv\lambda t\beta k.t\beta(\lambda z.k(\Phi(\mathsf{times}(\mathrm{id})f)z)).$$

Now

$$\begin{aligned}
\mathrm{Rec}\tau g(\mathbf{In}x) &\longrightarrow\!\!\!\!\!\rightarrow \mathbf{Out}_\sigma(\mathrm{Corec}_\sigma(\Phi(\sigma))Fx)\tau g \\
&\longrightarrow\!\!\!\!\!\rightarrow \Theta([\mathrm{Corec}_\sigma(\Phi(\sigma))F,\mathrm{id}])(\lambda\beta h.h(\Phi({<}\mathrm{Rec}\beta(h\circ\Phi(\mathrm{id}\times\mathrm{inr})),\mathrm{inr}{>})x)\tau g \\
&\longrightarrow\!\!\!\!\!\rightarrow (\lambda\beta h.h(\Phi({<}\mathrm{Rec}\beta(h\circ\Phi(\mathrm{id}\times\mathrm{inr})),\mathrm{inr}{>})x)\tau(\lambda z.g(\Phi(\mathsf{times}(\mathrm{id})F)z)) \\
&\longrightarrow\!\!\!\!\!\rightarrow g(\Phi(\mathsf{times}(\mathrm{id})F)(\Phi({<}\mathrm{Rec}\tau(\lambda z.g(\Phi(\mathsf{times}(\mathrm{id})F)z)\circ\Phi(\mathsf{times}(\mathrm{id})(\mathrm{inr})),\mathrm{inr}{>})x) \\
&\longrightarrow\!\!\!\!\!\rightarrow g(\Phi({<}\mathrm{Rec}\tau(g\circ\Phi(\mathsf{times}(\mathrm{id})F)\circ\Phi(\mathsf{times}(\mathrm{id})(\mathrm{inr}),\mathrm{id}{>})x) \\
&\longrightarrow\!\!\!\!\!\rightarrow g(\Phi({<}\mathrm{Rec}\tau g,\mathrm{id}{>})x)
\end{aligned}$$

We now want to look at the system of recursive types, as defined by [Mendler 1987], let's call it $F_{(CO)REC}$. (The system also has corecursive types.)

Definition 5.3 ([Mendler 1987]) *The system $F_{(CO)REC}$ is defined by adding to the polymorphic lambda calculus the following.*

1. *The set of types* **T** *is extended with $\mu\alpha.\Phi(\alpha)$ and $\nu\alpha.\Phi(\alpha)$, for $\Phi(\alpha)$ a positive type scheme.*
2. *For $\mu\alpha.\Phi(\alpha)$ and $\nu\alpha.\Phi(\alpha)$ we have the extra constants*

$$\begin{aligned}
\mathbf{in}_\mu &: \Phi(\mu)\to\mu & \mathrm{R}_\mu &: \forall\beta.(\forall\gamma.(\gamma\to\mu)\to(\gamma\to\beta)\to\Phi(\gamma)\to\beta)\to\mu\to\beta, \\
\mathbf{out}_\nu &: \nu\to\Phi(\nu) & \Omega_\nu &: \forall\beta.(\forall\gamma.(\nu\to\gamma)\to(\beta\to\gamma)\to\beta\to\Phi(\gamma))\to\beta\to\nu.
\end{aligned}$$

3. *Reduction rules for μ and ν:*

$$\begin{aligned} \mathrm{R}_\mu \tau g(\mathbf{in}_\mu x) &\longrightarrow_\mu g\mu(\mathrm{id}_\mu)(\mathrm{R}_\mu \tau g)x, \\ \mathbf{out}_\nu(\Omega_\nu \tau g x) &\longrightarrow_\nu g\nu(\mathrm{id}_\nu)(\Omega_\nu \tau g)x. \end{aligned}$$

(μ abbreviates $\mu\alpha.\Phi(\alpha)$ and ν abbreviates $\nu\alpha.\Phi(\alpha)$.)

In [Mendler 1987] it is shown that this system satisfies a lot of nice meta-properties, like strong normalization and confluence of the reduction relation.

Definition 5.4 *The system $F_{(CO)REC}$ with only the rules for μ will be called F_{REC} and similarly, $F_{(CO)REC}$ with only the rules for ν will be called F_{COREC}.*

We show that the system $F_{(co)rec}$ can be defined in both F_{REC} and F_{COREC}, so both systems have all syntactic recursive algebras and all syntactic corecursive coalgebras.

Proposition 5.5 1. *The μ-types of $F_{(co)rec}$ can be defined in F_{REC}.*

2. *The ν-types of $F_{(co)rec}$ can be defined in F_{COREC}.*

Proof Let $\Phi(\alpha)$ be a positive type scheme.

1. Write μ for $\mu\alpha.\Phi(\alpha)$ and take $\mathbf{In} = \mathbf{in}$,
   $\mathrm{Rec}_\mu = \lambda\beta.\lambda g{:}\Phi(\beta\times\mu)\to\beta.\mathrm{R}_\mu\beta(\lambda\alpha.\lambda f{:}\alpha\to\mu.\lambda k{:}\alpha\to\beta.g\circ\Phi(<k,f>))$. Then μ, $\mathbf{In}$ and Rec_μ together define the μ-type of $F_{(co)rec}$ in the sense that $\mathrm{Rec}_\mu\tau g(\mathbf{In}x) = g(\Phi(<\mathrm{Rec}_\mu\tau g, \mathrm{id}>)x)$.

2. Write ν for $\nu\alpha.\Phi(\alpha)$ and take $\mathbf{Out} = \mathbf{out}$,
   $\mathrm{Corec}_\nu = \lambda\beta.\lambda g{:}\beta\to\Phi(\beta+\nu).\Omega\beta(\lambda\alpha.\lambda f{:}\nu\to\alpha.\lambda k{:}\beta\to\alpha.\Phi([k,f])\circ g)$. Then ν, $\mathbf{Out}$ and Corec_ν together define the ν-type of $F_{(co)rec}$ because $\mathbf{Out}(\mathrm{Corec}_\nu\tau g x) = \Phi([\mathrm{Corec}_\nu\tau g, \mathrm{id}])(gx)$.

Corollary 5.6 1. *$F_{(co)rec}$ can be defined in both F_{REC} and F_{COREC}.*

2. *For every positive type scheme $\Phi(\alpha)$ both F_{REC} and F_{COREC} have a syntactic recursive Φ-algebra and a syntactic corecursive Φ-coalgebra.*

Proof Both immediately by the proposition and Theorem 5.2.

As a corollary of the translation of $F_{(co)rec}$ in to $F_{(CO)REC}$ we find that $F_{(co)rec}$ is strongly normalizing.

Proposition 5.7 *The reduction relation of the system $F_{(co)rec}$ is strongly normalizing and confluent.*

Proof In order to prove strong normalization we define a mapping $[-]$ from the terms of $F_{(co)rec}$ to the term of $F_{(CO)REC}$ that preserves infinite reduction paths. Then $F_{(co)rec}$ is strongly normalizing by the fact that $F_{(CO)REC}$ is strongly normalizing (see [Mendler 1987].) One easily verifies that the system is weakly confluent (i.e. if $M \longrightarrow N$ and $M \longrightarrow \mathrm{P}$, then $\exists Q[N \longrightarrow\!\!\!\!\!\rightarrow Q \,\&\, P \longrightarrow\!\!\!\!\!\rightarrow Q]$.) The confluence then follows from Newman's lemma ([Newman 1942]), stating that strong normalization and weak confluence together imply confluence.

The definition of $[-]$ is very similar to the mapping defined in the proof of Proposition 5.5. (As the types do not in any way interfere with the reduction process we omit the types in abstractions.) Define $[-]$ by

$$\begin{array}{rcl}
[\mathrm{Rec}_\mu] & := & \lambda\beta g.\mathrm{R}_\mu\beta(\lambda\alpha fk.g \circ \Phi(<k,f>)),\\
[\mathrm{Rec}_\mu\tau] & := & \lambda g.\mathrm{R}_\mu\tau(\lambda\alpha fk.g \circ \Phi(<k,f>)),\\
[\mathrm{Rec}_\mu\tau g] & := & \mathrm{R}_\mu\tau(\lambda\alpha fk.g \circ \Phi(<k,f>)),\\
[\mathbf{In}] & := & \mathbf{in},\\
[\mathrm{Corec}_\nu] & := & \lambda\beta g.\Omega\beta(\lambda\alpha fk.\Phi([k,f]) \circ g),\\
[\mathrm{Corec}_\nu\tau] & := & \lambda g.\Omega\tau(\lambda\alpha fk.\Phi([k,f]) \circ g),\\
[\mathrm{Corec}_\nu\tau g] & := & \Omega\tau(\lambda\alpha fk.\Phi([k,f]) \circ g),\\
[\mathbf{Out}] & := & \mathbf{out},
\end{array}$$

and further by induction on the structure of the terms. Then

$$\begin{array}{ll}
M \longrightarrow_\beta N & \Rightarrow [M] \longrightarrow\!\!\!\!\!\rightarrow^+_\beta [N],\\
M \longrightarrow_\eta N & \Rightarrow [M] \longrightarrow\!\!\!\!\!\rightarrow_\eta [N],\\
M \longrightarrow_\mu N & \Rightarrow [M] \longrightarrow_\mu [N],\\
M \longrightarrow_\nu N & \Rightarrow [M] \longrightarrow_\nu [N],
\end{array}$$

where $\longrightarrow\!\!\!\!\!\rightarrow^+_\beta$ denotes a reduction in at least one step. As there is no infinite η-reduction in $F_{(co)rec}$, the mapping $[-]$ maps an infinite reduction path in $F_{(co)rec}$ to an infinite reduction path in $F_{(CO)REC}$, so we are done.

It doesn't seem possible to define the μ-types in terms of the ν-types in $F_{(CO)REC}$, nor to define the system F_{COREC} in the system F_{corec}. When one attempts to do so, some extra equalities seem to be required.

6 Discussion

As pointed out by Christine Paulin ([Paulin 1992]) the technique in the proof of Theorem 5.2 also applies to a polymorphic lambda calculus with a kind of 'retract types' that we shall describe now. We give the syntax as it has been communicated to us by Christine Paulin; it is implicit in papers by Parigot ([Parigot 1988] and [Parigot 1992]), where extensions of the system AF2 with recursive types are studied. (AF2 is a system of second oder predicate logic with an interpretation of proofs as untyped lambda terms.) The connections between our system with recursive types and the (extensions) of AF2 is a subject which needs further investigation; we feel that this is not the place to do so.

Definition 6.1 *The system F_{ret} is the extension of system F with the following.*

1. *The set of types $\mathbf{T}$ is extended with $\rho\alpha.\Phi(\alpha)$, for $\Phi(\alpha)$ a positive type scheme.*
2. *For $\rho\alpha.\Phi(\alpha)$ we have the extra constants*

$$\mathbf{i}_\rho : \Phi(\rho)\rightarrow\rho, \mathbf{o}_\rho : \rho\rightarrow\Phi(\rho).$$

3. Reduction rule for ρ:

$$\mathbf{o}_\rho(\mathbf{i}_\rho x) \longrightarrow_\rho x.$$

(ρ abbreviates $\rho\alpha.\Phi(\alpha)$.)

In this system one can construct for $\Phi(\alpha)$ a positive type scheme a type ρ with $\Phi(\rho) < \rho$ ($\Phi(\rho)$ is a retract of ρ.) As pointed out to us by [Paulin 1992], the technique of 5.2 can be applied to obtain that both the systems F_{rec} and F_{corec} can be defined in F_{ret}. Also the reverse holds: F_{ret} can be defined in both F_{rec} and F_{corec}. It would be an interesting subject for further investigations to see how the retract types relate to the recursive and corecursive types on the categorical level.

Theorem 6.2 *The systems F_{rec} and F_{corec} can be defined in F_{ret} and vice versa.*

Proof To define F_{rec} in F_{ret} take

$$\begin{array}{rcl}
\mu\alpha.\Phi(\alpha) & := & \rho\alpha.\forall\gamma.(\Phi(\gamma\times\alpha)\to\alpha)\to\alpha, \\
\mathrm{Rec}_\mu & := & \lambda\gamma.\lambda f{:}\Phi(\gamma\times\mu)\to\gamma.\lambda x{:}\mu.\mathbf{o}x\gamma f, \\
\mathbf{In}_\mu & := & \lambda x{:}\Phi(\mu).\mathbf{i}(\lambda\gamma.\lambda f{:}\Phi(\gamma\times\mu)\to\mu.f(\Phi(<\mathrm{Rec}_\mu\gamma f,\mathrm{id}>))x)),
\end{array}$$

and $\mathrm{Rec}_\mu\tau g(\mathbf{In}_\mu x) \longrightarrow\!\!\!\!\!\rightarrow g(\Phi(<\mathrm{Rec}_\mu\tau g,\mathrm{id}>)x)$ easily follows. To define F_{corec} in F_{ret} take

$$\begin{array}{rcl}
\nu\alpha.\Phi(\alpha) & := & \rho\alpha.\exists\gamma.(\gamma\to\Phi(\gamma+\alpha))\to\gamma), \\
\mathrm{Corec}_\nu & := & \lambda\gamma.\lambda f{:}\gamma\to\Phi(\gamma+\nu).\lambda x{:}\gamma.\mathbf{i}(\lambda\beta\lambda k.k\gamma<f,x>), \\
\mathbf{Out}_\nu & := & \lambda x{:}\nu.\mathbf{o}x(\Phi(\nu))(\lambda\gamma.\lambda f{:}\gamma\to\Phi(\gamma+\nu)\times\nu.\Phi([\mathrm{Corec}_\nu\gamma(\mathrm{fst}f),\mathrm{id}])(\mathrm{fst}f(\mathrm{snd}f))),
\end{array}$$

and $\mathbf{Out}_\nu(\mathrm{Corec}_\nu\tau g) \longrightarrow\!\!\!\!\!\rightarrow \Phi([\mathrm{Corec}_\nu\tau g,\mathrm{id}])(gx)$ easily follows.

To define F_{ret} in terms of F_{rec} or F_{corec}, take respectively

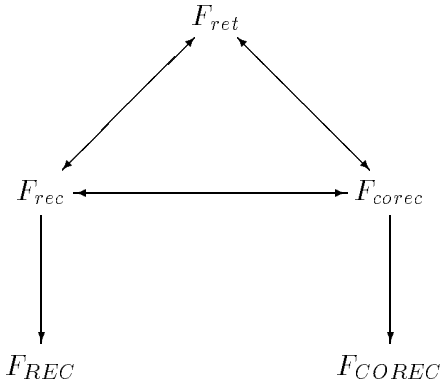
$$\begin{array}{rcl}
\rho\alpha.\Phi(\alpha) & := & \mu\alpha.\Phi(\alpha), \\
\mathbf{i} & := & \mathbf{In}_\mu, \\
\mathbf{o} & := & \mathrm{Rec}_\mu\Phi(\mu)(\Phi(\mathrm{snd}))
\end{array}$$

(so $\mathbf{o}(\mathbf{i}x) \longrightarrow\!\!\!\!\!\rightarrow x$) and

$$\begin{array}{rcl}
\rho\alpha.\Phi(\alpha) & := & \nu\alpha.\Phi(\alpha), \\
\mathbf{o} & := & \mathbf{Out}_\nu, \\
\mathbf{i} & := & \mathrm{Corec}_\mu\Phi(\nu)(\Phi(\mathrm{inr}))
\end{array}$$

(so again, $\mathbf{o}(\mathbf{i}x) \longrightarrow\!\!\!\!\!\rightarrow x$.)

We can collect the results from Theorems 5.2 and 6.2 and Corollary 5.6 in a picture as follows. (An arrow from A to B means that the system A can be translated in the system B.)



If we translate in F_{rec} the type $\mu\alpha.\Phi(\alpha)$ in terms of the ρ-type, which is defined in terms of the μ-type, we obtain the type $\mu\alpha.\forall\gamma.(\Phi(\gamma \times \alpha)\to\gamma)\to\gamma$. A similar situation occurs if we translate in F_{ret} a ρ-type in terms of a μ-type, which is defined in terms of the ρ-type: $\rho\alpha.\Phi(\alpha)$ becomes $\rho\alpha.\forall\gamma.(\Phi(\gamma\times\alpha)\to\gamma)\to\gamma$. Using these double translations, we can deduce the following facts about the systems F_{rec}, F_{corec} and F_{ret} themselves.

Fact 6.3 *1. For ρ a retract type of $\Psi(\alpha) \equiv \forall\gamma.(\Phi(\gamma\times\alpha)\to\gamma)\to\gamma$ or of $\Psi(\alpha) \equiv \exists\gamma.(\gamma\to\Phi(\gamma + \alpha)) \times \gamma$, ρ is also a retract type of Φ.*

2. For μ a recursive type of $\Psi(\alpha) \equiv \forall\gamma.(\Phi(\gamma \times \alpha)\to\gamma)\to\gamma$, μ is also a recursive type of Φ.

3. For ν a corecursive type of $\Psi(\alpha) \equiv \exists\gamma.(\gamma\to\Phi(\gamma + \alpha)) \times \gamma$, ν is also a corecursive type of Φ.

We can also compose the translations to obtain new interpretations of μ-types in ν-types and vice versa:

Fact 6.4 *1. For $\Phi(\alpha)$ a positive type scheme, we can interpret ν-types in F_{rec} by taking $\nu\alpha.\Phi(\alpha)$ and* Corec_ν *as in 5.2 and*

$$\mathbf{Out}_\nu :\equiv \lambda x.\mathrm{Rec}_\nu(\Theta(\nu))(\Theta(\mathrm{snd}))x(\Phi(\nu))(\lambda\gamma f.\Phi([\mathrm{Corec}\gamma(\mathrm{fst}f), \mathrm{id}])(\mathrm{fst}f(\mathrm{snd}f))).$$

2. For $\Phi(\alpha)$ a positive type scheme, we can interpret μ-types in F_{corec} by taking $\mu\alpha.\Phi(\alpha)$ and Rec_μ *as in 5.2 and*

$$\mathbf{In}_\mu :\equiv \lambda x.\mathrm{Corec}_\mu(\Theta(\mu))(\Theta(\mathrm{inr}))(\lambda\gamma f.f(\Phi(<\mathrm{Rec}\gamma f, \mathrm{id}>)x)).$$

References

[Böhm and Berarducci 1985] C. Böhm and A. Berarducci, Automatic synthesis of typed Λ-programs on term algebras *Theor. Comput. Science*, 39, pp 135-154.

[Coquand and Huet 1988] Th. Coquand and G. Huet, The calculus of constructions, *Information and Computation*, 76, pp 95-120.

[Coquand and Mohring 1990] Inductively defined types, In P. Martin-Löf and G. Mints editors. *COLOG-88 : International conference on computer logic, LNCS 417.*

[Dowek e.a. 1991] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin-Mohring, B. Werner, The Coq proof assistant version 5.6, user's guide. INRIA Rocquencourt - CNRS ENS Lyon.

[Girard et al. 1989] J.Y. Girard, Y. Lafont and P. Taylor, *Proofs and types*, Camb. Tracts in Theoretical Computer Science 7, Cambridge University Press.

[Hagino 1987a] T. Hagino, A categorical programming language, Ph. D. thesis, University of Edinburgh.

[Hagino 1987b] 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, LNCS 283* pp 140-157.

[Hayashi 1985] S. Hayashi, Adjunction of semifunctors: categorical structures in nonextensional lambda calculus. *Theor. Comp. Sc. 41*, pp 95-104.

[Kleene 1936] S.C. Kleene, λ-definability and recursiveness. *Duke Math. J.* 2, pp 340-353.

[Lambek 1968] J. Lambek, A fixed point theorem for complete categories. *Mathematisches Zeitschrift 103* pp 151-161.

[Leivant 1989] D. Leivant, Contracting proofs to programs. In P. Odifreddi, editor. *Logic in Computer Science*, Academic Press, pp 279-327.

[Mendler 1987] N.P. Mendler, Inductive types and type constraints in second-order lambda calculus. *Proceedings of the Second Symposium of Logic in Computer Science.* Ithaca, N.Y., IEEE, pp 30-36.

[Mendler 1991] N.P. Mendler, Predicative type universes and primitive recursion. *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science.* Amsterdam, The Netherlands, IEEE, pp 173-184

[Newman 1942] M.H.A. Newman, On theories with a combinatorial definition of "equivalence". *Ann. of Math.* (2) 43, pp 223-243.

[Paulin 1992] Ch. Paulin-Mohring, private communication.

[Parigot 1988] M. Parigot, Programming with proofs: a second order type theory. *ESOP '88, LNCS 300*, pp 145-159.

[Parigot 1992] M. Parigot, Recursive programming with proofs. *Theor. Comp. Science* 94, pp 335-356.

[Reynolds and Plotkin 1990] J.C. Reynolds and G.D. Plotkin, On functors expressible in the polymorphic lambda calculus. In G. Huet, editor. *Logical Foundations of Functional Programming*, In 'The UT Year of Programming Series', Austin, Texas, pp 127-152.

[Wraith 1989] G.C. Wraith, A note on categorical datatypes In D.H. Pitt, A. Poigné and D.E. Rydeheard, editors. *Category Theory and Computer Science, LNCS 389* pp 118-127.