Recursion Patterns as Hylomorphisms

Manuel Alcino Cunha
alcino@di.uminho.pt

Techn. Report DI-PURe-03.11.01
2003, November

PURe
Program Understanding and Re-engineering: Calculi and Applications
(Project POSI/ICHS/44304/2002)

Departamento de Informática da Universidade do Minho
Campus de Gualtar — Braga — Portugal

DI-PURe-03.11.01
Recursion Patterns as Hylomorphisms by Manuel Alcino Cunha

Abstract

In this paper we show how some of the recursion patterns typically used in algebraic programming can be defined using hylomorphisms. Most of these definitions were previously known. However, unlike previous approaches that use fixpoint induction, we show how to derive the standard laws of each recursion pattern by using just the basic laws of hylomorphisms. We also define the accumulation recursion pattern introduced by Pardo using a hylomorphism, and use this definition to derive the strictness conditions that characterize this operator in the presence of partiality. All definitions are implemented and exemplified in Haskell.

1 Introduction

The exponential growth in the use of computers in the past decades raised important questions about the correctness of software, specially in safety critical systems. Compared to other areas of engineering, software engineers are still very unfamiliar with the mathematical foundations of computation, and tend to approach programming more as (black) art and less as a science [5]. Ideally, final implementations should be calculated from their specifications, using simple laws that relate programs, in the same way we calculate with mathematical expressions in algebra.

The calculational approach is particular appealing in the area of functional programming, since referential transparency ensures that expressions in functional programs behave as ordinary expressions in mathematics. However, in order to make it more effective, we must avoid the use of arbitrarily recursively defined functions, using only a limited set of high order functions that encode typical recursion patterns with well known properties, such as folds and unfolds. This philosophy is the same that fostered the use of structured control primitives instead of arbitrary gotos. This move also made more feasible to formally reason about imperative programs.

As will be seen, the recursion patterns arise naturally from an algebraic view of recursive data types. Although initially they were only defined for finite lists, it became clear that they could be generalized for any data type [10]. Most of the research done in this area is being carried in the context of total functions and total elements. Technically, this means that the underlying category is $\mathcal{S}et$ (of sets and total functions). Unfortunately, this category is not a good semantic model of most functional languages, since it hardens the treatment of arbitrary recursive or partial definitions. Another problem is that finite and infinite data types are different things that cannot be combined, thus excluding, for example, functions defined by induction that work for both. These problems are overcome in the category $\mathcal{CPO}$ (of complete partial orders and continuous functions), by imposing some additional structure on objects and morphisms. The dawn side of this move is that some of laws became polluted with strictness side-conditions.

Meijer, Fokkinga, and Paterson pioneered the study of recursion patterns in $\mathcal{CPO}$ [12, 4]. Besides studying the properties of folds, unfolds, and paramorphisms in this category, they introduced a new operator, called hylomorphism, whose expressive power was equivalent to the composition of a fold after an unfold of some intermediate data structure. Although noticing that this operator was expressive enough to define the others, these definitions were made directly by fixpoint, and the fundamental laws that characterize them proved using fixpoint induction. In this paper we take a slightly different approach. Given the definition of hylomorphisms by fixpoint, and the basic set of laws that characterize them, we define all the other recursion patterns as hylomorphisms, and show how their properties can be derived (without induction) from that definition. We also show how this approach can be used to investigate the properties in $\mathcal{CPO}$ of other recently introduced recursion patterns, like the accumulation operator defined by Pardo [16].

The next section presents some basic concepts about category theory particularized to the category $\mathcal{C}PO$. Most of these concepts appear in any standard book about category theory, such as [17], and the readers acquainted with the area can skip this section. In Section 3 we show how to model recursive data types in this category. Section 4 introduces hylomorphisms, and in the following sections we show how to define folds, unfolds, paramorphisms, and accumulations using this recursion pattern. Section 9 concludes with some brief remarks. All definitions and recursion patterns are implemented and exemplified in Haskell [8].

2 Preliminary Concepts

The category $\mathcal{C}PO$ has pointed complete partial orders as objects (sets equipped with a partial order, with a least element, and closed under limits of ascending chains) and continuous functions as arrows (monotonic functions that preserve limits). Sometimes we will also refer the category $\mathcal{C}PO_\perp$, that is the subcategory of $\mathcal{C}PO$ with the same objects but only the strict (continuous) functions as arrows. A strict functions is defined as follows.

$$f \text{ strict} \quad \Leftrightarrow \quad f \circ \perp = \perp \qquad \text{strict-DEF}$$

The *product* of two data types is defined as follows.

$$\begin{aligned} A \times B &= \{(x, y) | x \in A, y \in B\} \\ (f \times g)\ (x, y) &= (f\ x, g\ y) \end{aligned}$$

Related to these we have the projections and split combinators.

$$\begin{aligned} \pi_1\ (x, y) &= x \\ \pi_2\ (x, y) &= y \\ \langle f, g\rangle\ x &= (f\ x, g\ x) \end{aligned}$$

This is a categorical product on $\mathcal{C}PO$ (and $\mathcal{C}PO_\perp$) since that it satisfies the following uniqueness property.

$$f = \langle g, h\rangle \quad \Leftrightarrow \quad \pi_1 \circ f = g \wedge \pi_2 \circ f = h \qquad \times\text{-UNIQ}$$

One can also define the product of functions using split and the projections.

$$f \times g = \langle f \circ \pi_1, g \circ \pi_2\rangle \qquad \times\text{-DEF}$$

A consequence of the product definition is

$$\langle f, g\rangle \text{ strict} \quad \Leftrightarrow \quad f \text{ strict} \wedge g \text{ strict} \qquad \times\text{-STRICT}$$

The following properties can all be derived from these.

$$\langle \pi_1, \pi_2 \rangle = \mathtt{id} \qquad \times\text{-Reflex}$$
$$\pi_1 \circ \langle f, g \rangle = f \ \wedge \ \pi_2 \circ \langle f, g \rangle = g \qquad \times\text{-Cancel}$$
$$\langle f, g \rangle \circ h = \langle f \circ h, g \circ h \rangle \qquad \times\text{-Fusion}$$
$$(f \times g) \circ \langle h, i \rangle = \langle f \circ h, g \circ i \rangle \qquad \times\text{-Absor}$$
$$(f \times g) \circ (h \times i) = f \circ h \times g \circ i \qquad \times\text{-Functor}$$
$$\langle f, g \rangle = \langle h, i \rangle \quad \Leftrightarrow \quad f = h \ \wedge \ g = i \qquad \times\text{-Equal}$$

The product and split combinators can be defined in Haskell as follows. The projections are the predefined `fst` and `snd` functions.

```
split :: (a -> b) -> (a -> c) -> a -> (b,c)
split f g x = (f x, g x)

(><) :: (a -> b) -> (c -> d) -> (a,c) -> (b,d)
f >< g = split (f . fst) (g . snd)
```

Sometimes we will also refer the notion of *left strictness* of binary function.

$$f \text{ left strict} \quad \Leftrightarrow \quad f \circ (\bot \times \mathtt{id}) = \bot \qquad \text{lstrict-Def}$$

Obviously, left strictness implies strictness.

$$f \text{ strict} \quad \Leftarrow \quad f \text{ left strict} \qquad \text{lstrict-Strict}$$

It is know that $\mathcal{C}PO$ does not have coproducts. In $\mathcal{C}PO_\bot$ it can be defined as the *coalesced sum* of A and B, by identifying $\bot_A$ and $\bot_B$ as $\bot_{A+B}$. However, this is not the coproduct tipically implemented in lazy functional languages. Instead, we have the *separated sum* that adds a new bottom element to the data type. It is defined as follows.

$$A + B = \{0\} \times A \cup \{1\} \times B \cup \{\bot\}$$
$$(f + g) \ \bot = \bot$$
$$(f + g) \ (0, x) = (0, f \ x)$$
$$(f + g) \ (1, x) = (1, g \ x)$$

Related to these we have the injections and either combinators.

$$i_1 \ x = (0, x)$$
$$i_2 \ x = (1, x)$$
$$[f, g] \ \bot = \bot$$
$$[f, g] \ (0, x) = f \ x$$
$$[f, g] \ (1, x) = g \ x$$

Although we do not have a proper coproduct we still have an uniqueness law.

$$f = [g, h] \quad \Leftrightarrow \quad f \circ i_1 = g \ \wedge \ f \circ i_2 = h \ \wedge \ f \text{ strict} \qquad +\text{-Uniq}$$

Similarly to products, we can define the sum using either and the injections.

$$f + g = [i_1 \circ f, i_2 \circ g] \qquad +\text{-Def}$$

A consequence of the sum definition is

$$\forall f, g \cdot [f, g] \text{ strict} \qquad +\text{-Strict}$$

Again, we can derive the following laws from these.

$$\begin{aligned}
&[i_1, i_2] = \texttt{id} && +\text{-Reflex}\\
&[f, g] \circ i_1 = f \ \wedge\ [f, g] \circ i_2 = g && +\text{-Cancel}\\
&f \circ [g, h] = [f \circ g, f \circ h] \quad \Leftarrow \quad f \text{ strict} && +\text{-Fusion}\\
&[f, g] \circ (h + i) = [f \circ h, g \circ i] && +\text{-Absor}\\
&(f + g) \circ (h + i) = f \circ h + g \circ i && +\text{-Functor}\\
&[f, g] = [h, i] \quad \Leftrightarrow \quad f = h \ \wedge\ g = i && +\text{-Equal}
\end{aligned}$$

The sum and either (predefined) combinators can be defined in Haskell as follows. The injections are the `Left` and `Right` constructors of the `Either` data type.

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either l r (Left x)  = l x
either l r (Right y) = r y

(-|-) :: (a -> b) -> (c -> d) -> Either a c -> Either b d
f -|- g = either (Left . f) (Right . g)
```

The *exponentiation* to a data type is defined as follows.

$$\begin{aligned}
B^A &= \{f : A \to B\}\\
f^A\ g &= f \circ g
\end{aligned}$$

Related to this we have the curry and apply operators.

$$\begin{aligned}
\overline{f}\ x\ y &= f\ (x, y)\\
\texttt{ap}\ (f, x) &= f\ x
\end{aligned}$$

Like products, this is a categorical exponentiation in $\mathcal{C}PO$, and satisfies the following uniqueness law.

$$f = \overline{g} \quad \Leftrightarrow \quad g = \texttt{ap} \circ (f \times \texttt{id}) \qquad \wedge\text{-Uniq}$$

Again, we can define exponentiation in terms of curry and application.

$$f^A = \overline{f \circ \texttt{ap}} \qquad \wedge\text{-Def}$$

A consequence of exponentiation definition is

$$\overline{f} \text{ strict} \quad \Leftrightarrow \quad f \text{ left strict} \qquad \wedge\text{-Strict}$$

From the definitions and uniqueness we can define the following laws.

$$\overline{\mathtt{ap}} = \mathtt{id} \qquad \wedge\text{-Reflex}$$
$$f = \mathtt{ap} \circ (\overline{f} \times \mathtt{id}) \qquad \wedge\text{-Cancel}$$
$$\overline{f \circ (g \times \mathtt{id})} = \overline{f} \circ g \qquad \wedge\text{-Fusion}$$
$$f^A \circ \overline{g} = \overline{f \circ g} \qquad \wedge\text{-Absor}$$
$$(f \circ g)^A = f^A \circ g^A \qquad \wedge\text{-Functor}$$
$$\overline{f} = \overline{g} \quad \Leftrightarrow \quad f = g \qquad \wedge\text{-Equal}$$

In Haskell we can implement curry (predefined), application and exponentiation as follows.

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x, y)

app :: (a -> b,a) -> b
app (f, x) = f x

expn :: (a -> b) -> (c -> a) -> (c -> b)
expn f = curry (f . app)
```

A *functor* F is a mapping between categories (maps objects to objects, and morphisms to morphisms) such that

$$F\ f : F\ A \to F\ B \quad \Leftarrow \quad f : A \to B \qquad \text{functor-Type}$$
$$F(f \circ g) = F\ f \circ F\ g \qquad \text{functor-Compose}$$
$$F\ \mathtt{id}_A = \mathtt{id}_{F\ A} \qquad \text{functor-Id}$$

In Haskell, a functor `f` is a type constructor that is a member of the following class.

```
class Functor f
    where fmap :: (a -> b) -> (f a -> f b)
```

The burden that the map function obeys the functor laws (the last two equations above) is left to the programmer.

A *polynomial functor* is either: the identity functor `Id`; a constant functor $\underline{A}$ for a given type A; the composition of other polynomial functors; or the pointwise product or coproduct of other polynomial functors, defined by

$$(F + G)\ h = F\ h + G\ h$$
$$(F \times G)\ h = F\ h \times G\ h$$

Given an endofunctor F, an F-*algebra* is a strict function of type $F\ A \to A$, and an F-*coalgebra* is a, not necessarily strict, function of type $A \to F\ A$. The set A is called the *carrier* of the algebra. An F-*homomorphism* is a function

$h : A \to B$ from an F-algebra $\varphi : F\ A \to A$ to an F-algebra $\psi : F\ B \to B$ that makes the following diagram commute.

$$\begin{array}{ccc} A & \xleftarrow{\varphi} & F\ A \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle F\ h} \\ B & \xleftarrow[\psi]{} & F\ B \end{array}$$

We also have the dual notion of *F-cohomomorphism*.

A *natural transformation* η between functors F and G, denoted by $\eta : F \dot{\to} G$, is a function that assigns for each type A a function $\eta_A : F\ A \to G\ A$ such that for any function $f : A \to B$ the following diagram commutes.

$$\begin{array}{ccc} F\ A & \xrightarrow{\eta_A} & G\ A \\ {\scriptstyle F\ f}\downarrow & & \downarrow{\scriptstyle G\ f} \\ F\ B & \xrightarrow{\eta_B} & G\ B \end{array}$$

3 Recursive Data Types

In order to generalize the results to any recursive data type, we must present a general and simple framework to describe them in the category $\mathcal{CPO}$. Typically this is done using fixed points of functors. Given a locally continuous and strictness preserving functor F, there exists a data type μF and two functions $\mathtt{in}_F : F(\mu F) \to \mu F$ and $\mathtt{out}_F : \mu F \to F(\mu F)$ such that

$$\mathtt{in}_F \text{ strict} \wedge \mathtt{out}_F \text{ strict} \qquad \text{in-out-STRICT}$$

$$\mathtt{in}_F \circ \mathtt{out}_F = \mathtt{id}_{\mu F} \wedge \mathtt{out}_F \circ \mathtt{in}_F = \mathtt{id}_{F(\mu F)} \qquad \text{in-out-INV}$$

$$\mathtt{id}_{\mu F} = \mu(\lambda f.\mathtt{in}_F \circ F f \circ \mathtt{out}_F) \qquad \text{fix-REFLEX}$$

The data type μF is the least fixed point of F, the functor that captures the signature of its constructors. The functions $\mathtt{in}_F$ and $\mathtt{out}_F$ are used, respectively, to construct and destruct values of the data type μF. This result is acknowledged to [18]. Note that all functors that we will use through the paper are polynomial, and thus locally continuous and strictness preserving.

We can implement these concepts directly in Haskell [13]. As an example, we will show how to implement the following recursive Haskell data type that models natural numbers.

```
data Nat = Zero | Succ Nat

one = Succ Zero
two = Succ one
```

First, we define an explicit fixpoint operator using the keyword `newtype` to enforce the isomorphism.

```
newtype Mu f = In {out :: f (Mu f)}
```

Then, for each data type, it is necessary to define the functor that captures its signature (i.e, its constructors) and apply the fixpoint operator in order to obtain the data type itself. The idea is to separate the recursion from the signature definition. The functor captures the signature and defines a *one-layer* map, and the *recursive knot* is tied separately by the fixpoint operator. This technique is also the main idea of programming with recursion patterns and is kind of folklore in the functional programming community [9]. The isomorphic encoding of the `Nat` data type as a fixed point is

```
data FNat x = Zero | Succ x

instance Functor FNat
    where fmap f Zero     = Zero
          fmap f (Succ x) = Succ (f x)

type Nat = Mu FNat

one = In (Succ (In Zero))
two = In (Succ one)
```

The `Mu` operator is always applied to a monofunctor. In order to declare a polymorphic data type, we must apply sectioning to a bifunctor by treating the parameter type as a constant type. For example, the usual list data type can be obtained as follows.

```
data FList a x = Nil | Cons a x

instance Functor (FList a)
    where fmap f Nil        = Nil
          fmap f (Cons a x) = Cons a (f x)

type List a = Mu (FList a)
```

The list `[1,2,3]` can be defined as follows.

```
list :: List Int
list = In (Cons 1 (In (Cons 2 (In (Cons 3 (In Nil))))))
```

As a final example, lets see how to encode polymorphic binary trees.

```
data FTree a x = Empty | Node a x x

instance Functor (FTree a)
    where fmap f Empty        = Empty
          fmap f (Node a x y) = Node a (f x) (f y)

type Tree a = Mu (FTree a)
```

4 Hylomorphisms

The recursion pattern *hylomorphism* was first defined in [12, 4], and denotes the following recursive function defined by a fixed point:

$$[\![g, h]\!]_F = \mu(\lambda f.g \circ Ff \circ h) \qquad \text{hylo-DEF}$$

The typing information in this equation is summarized in the following diagram, where it is clear that the h function is a F-coalgebra and the g function a F-algebra.

$$\begin{array}{ccc} A & \xrightarrow{h} & FA \\ {\scriptstyle [\![g,h]\!]_F}\downarrow & & \downarrow{\scriptstyle F[\![g,h]\!]_F} \\ B & \xleftarrow[g]{} & FB \end{array}$$

How does a hylomorphism computes its result? The first thing to be noticed is that the recursion is characterized by the functor F. The map function for a particular functor determines where the recursion will occur. Suppose, for example, that the functor was `FNat`. Then, the resulting function is linear recursive. On the other way, to define a birecursive function we must use a functor like `FTree`. Technically, μF is the data type that models the recursion tree of a hylomorphism. The coalgebra is responsible for all the work that must be done before recursion, namely, compute the values passed to the recursive calls from the input parameters. The algebra is applied after the recursion, and is responsible to determine the final result, essentially by combining the results of the recursive call(s). Notice, however, that the result can be computed using other information beside the results of the recursive calls. If we use a functor like `FList`, the recursion will only be applied to the second argument of the constructor `Cons`. The other argument can be used to pass some information unchanged from the coalgebra to the algebra. This recursion pattern can be expressed in Haskell by an high-order function:

```
hylo :: Functor f => (f b -> b) -> (a -> f a) -> a -> b
hylo g h = g . fmap (hylo g h) . h
```

Likewise to the `Mu` operator in the approach to define recursive data types presented in the previous section, the hylomorphism will be responsible to tie the *recursive knot* when defining a function. The programmer should select the appropriate functor and define the non-recursive code. As an example, we will implement the following typical factorial definition using a hylomorphism.

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

This is a linear recursive function, but to compute the result we need not only the recursive result of applying `fact` to `n-1`, but also the input parameter `n` itself. This means that the recursion tree must be a list, where it is possible to store information in the intermediate nodes of computation. As such, the factorial function can be defined as a hylomorphism using as functor `FList`.

```
fact :: Int -> Int
fact = hylo g h
    where h 0          = Nil
          h n          = Cons n (n-1)
          g Nil        = 1
          g (Cons n r) = n * r
```

The stop case is triggered by selecting the `Nil` constructor. The coalgebra computes the parameter of the recursive call `n-1` and stores it in the second argument of `Cons`. In the first argument it stores the input `n` in order to be passed unchanged to the algebra, that just multiplies it with the result of the recursive call.

This process of converting an explicit recursive definition of a function into a hylomorphism can be systematized. Hu, Iwasaki and Takeichi implemented an algorithm that automatically derives the functors, algebras and coalgebras from most recursive definitions [7]. The main restrictions to this algorithm is that no mutual recursion is allowed, and the recursive function calls should not be nested. This algorithm was mainly used in the context of program optimization through deforestation of intermediate data structures [14]. In the past we used it in the context of program understanding, at the core of a tool to automatically visualize recursion trees of Haskell functions [3, 2].

The main advantage of expressing recursive functions as hylomorphisms is that they have several interesting laws that enable program calculation and transformation. The following fundamental laws about hylomorphisms follow directly from the definition or can be proved by fixpoint induction [12, 4]:

$$[\![g, h]\!]_F = g \circ F[\![g, h]\!]_F \circ h \qquad \text{hylo-CANCEL}$$

$$[\![g, h]\!]_F \text{ strict} \quad \Leftarrow \quad g, h \text{ strict} \qquad \text{hylo-STRICT}$$

$$[\![\mathtt{in}_F, \mathtt{out}_F]\!]_F = \mathtt{id}_{\mu F} \qquad \text{hylo-REFLEX}$$

$$[\![g \circ \eta, h]\!]_F = [\![g, \eta \circ h]\!]_G \quad \Leftarrow \quad \eta : F \dot{\rightarrow} G \qquad \text{hylo-SHIFT}$$

$$[\![g, \phi]\!]_F \circ [\![\psi, h]\!]_F = [\![g, h]\!]_F \quad \Leftarrow \quad \phi \circ \psi = \mathtt{id} \qquad \text{hylo-COMPOSE}$$

$$\begin{array}{c} g \circ [\![\phi, \psi]\!]_F \circ h = [\![\beta, \gamma]\!]_F \\ \Leftarrow \\ g \text{ strict} \wedge\ g \circ \phi = \beta \circ F\ g \wedge\ \psi \circ h = F\ h \circ \gamma \end{array} \qquad \text{hylo-FUSION}$$

Since the fixpoint operator can itself be defined as a hylomorphism, this recursion pattern gives us the full power of recursion [13]. This means that it should be possible to implement other recursion patterns using hylomorphisms instead of explicit recursion. In the remaining of the paper we will show how this implementation can be done for most of the typical recursion patterns, and also show how the laws that characterize them can be derived from the basic set of laws presented above, thus avoiding the use of fixpoint induction.

5 Catamorphisms

Each inductive data type is characterized by a standard way of recursively consuming its values by iteration according to its shape. This standard recursion

pattern is usually known as fold or catamorphism. In the case of Haskell lists it is encoded in the standard function `foldr`.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

This function can be generalized to any inductive data type. As an example we will define the catamorphism for the binary trees declared in section 3. Similarly to `foldr`, if we want to produce a value of type `b` from a value of type `Tree a`, we will need as arguments a function to combine the value in each node of type `a` with two recursive results of type `b`, and a value of type `b` to return when an empty tree is reached. Both of this arguments can be combined in a single function of type `FTree a b -> b`, yielding the following definition.

```
cataTree :: (FTree a b -> b) -> Tree a -> b
cataTree f (In Empty)        = f Empty
cataTree f (In (Node x l r)) = f (Node x (cataTree f l) (cataTree f r))
```

Using this recursion pattern, we can, for example, define the function that determines the depth of a tree as follows.

```
depth :: BTree a -> Int
depth = cataTree f
    where f Empty        = 0
          f (Node _ l r) = 1 + max l r
```

`cataTree` can itself be defined as a hylomorphism as follows.

```
cataTree :: (FTree a b -> b) -> Tree a -> b
cataTree f = hylo g h
    where h (In Empty)        = Empty
          h (In (Node x l r)) = Node x l r
          g Empty = f Empty
          g (Node x l r) = f (Node x l r)
```

In this definition is trivial to see that `h` can be implemented by `out`, and `g` equals the argument function `f`. Thus the catamorphism for binary trees can be expressed just as

```
cataTree :: (FTree a b -> b) -> Tree a -> b
cataTree f = hylo f out
```

In fact, for any recursive data type, the catamorphism can be defined using a hylomorphism in the same way. Given a F-algebra $f : FA \to A$ we define $(\!|f|\!) : \mu F \to A$ as follows.

$$(\!|f|\!)_F = [\![f, \mathtt{out}_F]\!]_F \qquad \text{cata-DEF}$$

If we declare our recursive types in Haskell explicitly as fixed points of functors we can have a single generic definition of this recursion pattern.

```
cata :: Functor f => (f a -> a) -> Mu f -> a
cata g = hylo g out
```

From cata-DEF and the laws of hylomorphisms we immediately get a set of useful laws to reason about catamorphisms:

$$(\!|\mathtt{in}_F|\!)_F = \mathtt{id}_{\mu F} \qquad \text{cata-REFLEX}$$

$$(\!|f|\!)_F \text{ strict} \quad \Leftarrow \quad f \text{ strict} \qquad \text{cata-STRICT}$$

$$(\!|f|\!)_F \circ \mathtt{in}_F = f \circ F(\!|f|\!)_F \qquad \text{cata-CANCEL}$$

$$f \circ (\!|g|\!)_F = (\!|h|\!)_F \quad \Leftarrow \quad f \text{ strict} \wedge f \circ g = h \circ Ff \qquad \text{cata-FUSION}$$

Catamorphisms also have an uniqueness law under some strictness conditions:

$$f = (\!|g|\!)_F \wedge g \text{ strict} \quad \Leftrightarrow \quad f \circ \mathtt{in}_F = g \circ Ff \wedge f \text{ strict} \qquad \text{cata-UNIQ}$$

Proof. The $\Rightarrow$ implication is trivial from cata-CANCEL and cata-STRICT. The $\Leftarrow$ is also trivial from cata-FUSION with $g = \mathtt{in}_F$ and cata-REFLEX. For the strictness of g we could argue as follows:

$$\begin{array}{ll}
\lceil & \dagger \;\; f \circ \mathtt{in}_F = g \circ Ff \\
& \ddagger \;\; f \text{ strict} \\
| & \\
| & \quad \mathtt{true} \\
\Leftrightarrow & \quad \{\, \dagger \,\} \\
| & \quad f \circ \mathtt{in}_F = g \circ Ff \\
\Rightarrow & \quad \{\, \text{Extensionality} \,\} \\
| & \quad f \circ \mathtt{in}_F \circ \bot = g \circ Ff \circ \bot \\
\Leftrightarrow & \quad \{\, \text{in-out-STRICT}, \ddagger \,\} \\
| & \quad \bot = g \circ \bot \\
\Leftrightarrow & \quad \{\, \text{strict-DEF} \,\} \\
\lfloor & \quad g \text{ strict}
\end{array}$$

This uniqueness law says that $\mathtt{in}_F$ is an initial F-algebra in the category of F-algebras whose arrows are F-homomorphisms in $\mathcal{CPO}_\bot$ (this initiality result does not extend to $\mathcal{CPO}$). This means that for any strict F-algebra $g : FA \to A$, there exists a unique strict function $f : \mu F \to A$ such that $f \circ \mathtt{in}_F = g \circ Ff$. This homomorphism is the catamorphism $(\!|g|\!)_F$. Schematically, $(\!|f|\!)_F$ is the only function that makes the following diagram commute in $\mathcal{CPO}_\bot$:

$$\begin{array}{ccc}
\mu F & \xleftarrow{\mathtt{in}_F} & F(\mu F) \\
{\scriptstyle (\!|g|\!)_F}\downarrow & & \downarrow{\scriptstyle F(\!|g|\!)_F} \\
A & \xleftarrow[g]{} & FA
\end{array}$$

To exemplify the utility of these laws, lets see how can we prove, without using the traditional induction, the following property about map and length.

```
length . map f = length
```

First, we present the explicit recursive definition of these functions using the data types `List` and `Nat` declared in section 3.

```
length :: List a -> Nat
length (In Nil)         = In Zero
length (In (Cons x xs)) = In (Succ (length xs))

map :: (a -> b) -> List a -> List b
map f (In Nil)         = In Nil
map f (In (Cons x xs)) = In (Cons (f x) (map f xs))
```

Then, we translate them into catamorphisms.

```
length :: List a -> Nat
length = cata h
    where h Nil        = In Zero
          h (Cons x r) = In (Succ r)

map :: (a -> b) -> List a -> List b
map f = cata g
    where g Nil        = In Nil
          g (Cons x r) = In (Cons (f x) r)
```

According to the law cata-FUSION, and knowing that `length` is a strict function, we have to prove that

```
length . g = h . fmap length
```

Going pointwise, and using the definition of `fmap` for `FList`, this is equivalent to prove the following facts.

```
          length (g Nil) = h Nil
length (g (Cons x r)) = h (Cons x (length r))
```

Both follow from trivial rewritings using the definitions. For the former we have

```
          length (g Nil) = h Nil
        length (In Nil) = In Zero
           In Zero = In Zero
```

And for the later we have

```
          length (g (Cons x r)) = h (Cons x (length r))
        length (In (Cons (f x) r)) = In (Succ (length r))
           In (Succ (length r)) = In (Succ (length r))
```

6 Anamorphisms

The dual of fold is the unfold or anamorphism. Although already known for a long time it is still not very used by programmers [6]. This recursion pattern encodes a standard way of producing a value of a given data type, and for lists it is defined as `unfoldr` in one of the standard libraries of Haskell.

```
unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
unfoldr f b = case f b of Nothing    -> []
                          Just (a,b) -> a : unfoldr f b
```

The argument function `f` dictates the generation of the list. If it returns `Nothing` the generation stops yielding the empty list. Otherwise it should return both the value to put at the head of the list being constructed, and a seed to carry on with the generation.

Similarly to catamorphisms, the anamorphism can be generalized to any recursive data type, and can be easily defined as a hylomorphism. Given a F-coalgebra $f : A \to FA$, we define $(\!|f|\!) : A \to \mu F$ as follows.

$$(\!|f|\!)_F = [\![\mathtt{in}_F, f]\!]_F \qquad \text{ana-DEF}$$

Following our approach, in Haskell we can have the following generic definition of this recursion pattern.

```
ana :: Functor f => (a -> f a) -> a -> Mu f
ana h = hylo In h
```

A function that, given an integer `n`, produces a list with all integers from `n` downto (but not including) zero can be defined with a hylomorphism as follows.

```
downto :: Int -> List Int
downto = ana h
    where h 0 = Nil
          h n = Cons n (n-1)
```

From ana-DEF and the hylomorphism laws we can derive the following laws about anamorphisms:

$$(\!|\mathtt{out}_F|\!)_F = \mathtt{id}_{\mu F} \qquad \text{ana-REFLEX}$$

$$(\!|f|\!)_F \text{ strict} \quad \Leftarrow \quad f \text{ strict} \qquad \text{ana-STRICT}$$

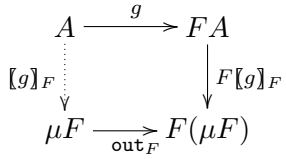
$$\mathtt{out}_F \circ (\!|f|\!)_F = F(\!|f|\!)_F \circ f \qquad \text{ana-CANCEL}$$

$$(\!|f|\!)_F \circ g = (\!|h|\!)_F \quad \Leftarrow \quad f \circ g = Fg \circ h \qquad \text{ana-FUSION}$$

And, likewise to catamorphisms, from these we can derive the following uniqueness law:

$$f = (\!|g|\!)_F \quad \Leftrightarrow \quad \mathtt{out}_F \circ f = Ff \circ g \qquad \text{ana-UNIQ}$$

This means that $\mathtt{out}_F$ is a final F-coalgebra in the category of F-algebras whose arrows are F-homomorphisms in both $\mathcal{C}PO$ and $\mathcal{C}PO_\perp$. Dually to catamorphisms, this means that for any F-coalgebra $g : A \to FA$, there exists a unique F-cohomomorphism $f : A \to \mu F$ such that $\mathtt{out}_F \circ f = Ff \circ g$. This cohomomorphism is the anamorphism $(\!|g|\!)_F$. Schematically, $(\!|g|\!)_F$ is the only function that makes the following diagram commute:



After defining these recursion patterns, we can present one of the most essential law about hylomorphisms, that is the one that states their decomposability into catamorphisms and anamorphisms, and that follows directly from the definitions and hylo-Compose.

$$[\![f, g]\!]_F = (\!|f|\!)_F \circ [\![g]\!]_F \qquad \text{hylo-Split}$$

This law can be used to expose the call tree of a recursive definition as an intermediate data structure, and to present the hylomorphism as the composition of two functions: one that builds this data structure from the input (the anamorphism), and another that traverses it in order to produce the result (the catamorphism). It also as some nice implications in program understanding. For example, the factorial definition as a hylomorphism puts in evidence the fact that this function is just the product of all integers from a given `n` down to `1`: the catamorphism is a function that multiplies all numbers in a list and the anamorphism is the function `downto` defined above.

Another nice example is the quick-sort. Given its explicit recursive definition

```
qsort :: (Ord a) => [a] -> [a]
qsort []     = []
qsort (x:xs) = (qsort (filter (<=x) xs)) ++ [x] ++
               (qsort (filter (>x)  xs))
```

we can translate it into a hylomorphism using binary trees as intermediate data structure [1].

```
qsort :: (Ord a) => [a] -> [a]
qsort = hylo g h
    where h []           = Empty
          h (x:xs)       = Node x (filter (<=x) xs) (filter (>x) xs)
          g Empty        = []
          g (Node x l r) = l ++ [x] ++ r
```

From this definition it is clear that the quick-sort is the composition of an inorder traversal after a function that builds a binary search tree from an unordered list.

7 Paramorphisms

While catamorphisms can express functions that can be defined by iteration, paramorphisms can express all functions that can be defined by primitive recursion [11]. In practice, this means that, for example, when defining a function over a list, the result may depend not only on the recursive result of applying the function to the tail of the list, but also on the tail itself, as can be seen in the following definition of this recursion pattern to Haskell lists in the style of `foldr`.

```
paralist :: (a -> [a] -> b -> b) -> b -> [a] -> b
paralist f z []     = z
paralist f z (x:xs) = f x xs (paralist f z xs)
```

An example of a function that can not be defined directly with a catamorphism, but that can be declared with a paramorphism is the insertion of an element in an ordered list.

```
insert :: (Ord a) => a -> [a] -> [a]
insert x = paralist f [x]
    where f y ys r | x<=y = x:y:ys
                   | x>y  = y:r
```

The definition of a paramorphism as a hylomorphism is known at least since [12]. The idea is that the anamorphism should make a copy of the parameter of a recursive invocation to be passed intact into the catamorphism. This means that, unlike the definition of the catamorphism where the intermediate data type was equal to the data type being consumed, in this case we will have a different data type. Given an input of type μF, the functor that generates the intermediate data structure will be $F \circ (\mathtt{Id} \times \underline{\mu F})$, that is, every recursive occurrence of the original type is replaced by a new recursive occurrence and a copy of the older one that will be left intact when recursing. For example, the intermediate data type for paramorphisms over naturals will be a list of naturals (notice that `FNat (x, Nat)` is isomorphic to `FList Nat x`). Instead of an F-algebra, the paramorphism $\langle\!|f|\!\rangle : \mu F \to A$ is parameterized by a function $f : F(A \times \underline{\mu F})$, and is defined by the following hylomorphism.

$$\langle\!|f|\!\rangle_F = [\![f, F\langle \mathtt{id}, \mathtt{id}\rangle \circ \mathtt{out}_F]\!]_{F\circ(\mathtt{Id}\times\underline{\mu F})} \qquad \text{para-DEF}$$

Schematically we have the following diagram. Notice the use of the doubling combinator $\langle \mathtt{id}, \mathtt{id}\rangle$ to replicate the recursive occurrences of the input data type.

$$\begin{array}{ccccc} \mu F & \xrightarrow{\mathtt{out}_F} & F(\mu F) & \xrightarrow{F\langle \mathtt{id},\mathtt{id}\rangle} & F(\mu F \times \mu F) \\ {\scriptstyle \langle\!|f|\!\rangle_F}\downarrow & & & & \downarrow{\scriptstyle F(\langle\!|f|\!\rangle_F \times \mathtt{id}_{\mu F}} \\ A & \xleftarrow[f]{} & & & F(A \times \mu F) \end{array}$$

To define a generic paramorphism in Haskell we first define a functor transformer from F to $F\circ(\mathtt{Id}\times\underline{\mu F})$, and then encode directly the previous definition.

```
newtype FPara f x = FPara {unFPara :: f (x, Mu f)}

instance Functor f => Functor (FPara f)
    where fmap f = FPara . fmap (f >< id) . unFPara

para :: Functor f => (f (a, Mu f) -> a) -> Mu f -> a
para g = hylo (g . unFPara) (FPara . fmap (split id id) . out)
```

Although we already presented it as a hylomorphism, the factorial function is one of the most typical examples of a paramorphism over naturals. Assuming that `mult` is a function that multiplies a pair of naturals, it can be redefined for the `Nat` data type as follows.

```
fact = para g
    where g Zero          = one
          g (Succ (r, i)) = mult (r, suck i)
```

Some laws about paramorphisms that can be easily derived from its definition and the laws of hylomorphisms.

$$\langle\!|\mathtt{in}_F \circ F\pi_1|\!\rangle_F = \mathtt{id}_{\mu F} \qquad \text{para-REFLEX}$$

$$\langle\!|f|\!\rangle_F \text{ strict} \quad \Leftarrow \quad f \text{ strict} \qquad \text{para-STRICT}$$

$$\langle\!|f|\!\rangle_F \circ \mathtt{in}_F = f \circ F\langle\langle\!|f|\!\rangle_F, \mathtt{id}_{\mu F}\rangle \qquad \text{para-CANCEL}$$

$$f \circ \langle\!|g|\!\rangle_F = \langle\!|h|\!\rangle_F \quad \Leftarrow \quad f \text{ strict} \wedge f \circ g = h \circ F(f \times \mathtt{id}_{\mu F}) \qquad \text{para-FUSION}$$

However, to derive its uniqueness law it is necessary to prove first the equivalence between the definition using hylomorphisms and the original definition by Meertens using pairs and catamorphisms [11]. Due to its size, the proof of the following equation is in the appendix A.

$$[\![f, F\langle \mathtt{id}, \mathtt{id}\rangle \circ \mathtt{out}_F]\!]_{F\circ(\mathtt{Id}\times\underline{\mu F})} = \pi_1 \circ (\!|\langle f, \mathtt{in}_F \circ F\pi_2\rangle|\!)_F$$
$$\Leftarrow$$
$$f \text{ strict}$$

Given the previous equivalence, the proof of the uniqueness law can be easily adapted from similar proofs in the $\mathcal{S}et$ category, such as the one presented in [19].

$$f = \langle\!|g|\!\rangle_F \wedge g \text{ strict} \quad \Leftrightarrow \quad f \circ \mathtt{in}_F = g \circ F\langle f, \mathtt{id}_{\mu F}\rangle \wedge f \text{ strict} \qquad \text{para-UNIQ}$$

8 Accumulations

Accumulations are binary recursive functions that use the second parameter as an accumulator of intermediate results. The accumulation technique is typically used in function programming in order to implement efficient versions of many recursive functions. For example, the reverse function for `List` can be implemented in linear time as follows.

```
reverse :: List a -> List a
reverse l = aux (l, In Nil)

aux :: (List a, List a) -> List a
aux (In Nil, l) = l
aux (In (Cons x xs), l) = aux (xs, In (Cons x l))
```

Alberto Pardo defined for the $\mathcal{S}et$ category a generic version of an accumulation recursion pattern, called `afold`, that can applied to any inductive data type [16]. This recursion pattern can be expressed as a hylomorphism using as intermediate data type a labeled variant of the input type. Lets suppose that the type of the accumulator is X. Given an input of type μF, the functor that generates the labeled variant is $F \times \underline{X}$. Given $f : FA \times X \to A$ and τ proper for

accumulation (defined later) we define the accumulation $\{\!|\tau, f|\!\} : \mu F \times X \to A$ in $\mathcal{C}PO$ as the following hylomorphism.

$$\{\!|\tau, f|\!\}_F = [\![f, \langle \tau, \pi_2 \rangle \circ (\mathtt{out}_F \times \mathtt{id})]\!]_{F \times \underline{X}} \qquad \text{afold-DEF}$$

Schematically we have

$$\begin{array}{ccccc} \mu F \times X & \xrightarrow{\mathtt{out}_F \times \mathtt{id}} & F(\mu F) \times X & \xrightarrow{\langle \tau, \pi_2 \rangle} & F(\mu F \times X) \times X \\ {\scriptstyle \{\!|\tau,f|\!\}}\downarrow & & & & \downarrow{\scriptstyle F\{\!|\tau,f|\!\} \times \mathtt{id}} \\ A & \xleftarrow[f]{} & & & FA \times X \end{array}$$

As can be seen in this definition, τ is the function responsible for propagating the accumulator to the recursive calls. This function should be *proper for accumulation*, that is, is should be a natural transformation of type

$$\tau : F \times \underline{X} \mathrel{\dot{\to}} F \circ (\mathtt{Id} \times \underline{X}) \qquad \tau\text{-NAT}$$

which guarantees that the propagated accumulation does not depend on the recursive values of the data type, and it cannot modify the shape of the data type neither the data contained in it.

$$F\pi_1 \circ \tau = \pi_1 \qquad \tau\text{-CANCEL}$$

To define this generic accumulations in Haskell we first define the appropriate functor transformer, and then encode directly the definition using rank-2 polymorphism to ensure the naturality of τ. Likewise to the functor implementation the burden to ensure τ-CANCEL is left to the programmer.

```
newtype FAfold f a x = FAfold {unFAfold :: (f x, a)}

instance Functor f => Functor (FAfold f a)
    where fmap f = FAfold . (fmap f >< id) . unFAfold

afold :: Functor f => (forall a . (f a, x) -> f (a, x)) ->
                      ((f a, x) -> a) -> (Mu f, x) -> a
afold t h = hylo (h . unFAfold) (FAfold . (split t snd) . (out >< id))
```

Using this recursion pattern we can redefine the auxiliary function of reverse as follows.

```
aux :: (List a, List a) -> List a
aux = afold t h
    where t (Nil, l)       = Nil
          t (Cons x xs, l) = Cons x (xs, In (Cons x l))
          h (Nil, l)       = l
          h (Cons x xs, l) = xs
```

Most of the laws about accumulations presented in [16] can be derived directly from the their definition as hylomorphisms. Some of them rely on the notion of X-*actions*, functions of type $A \times X \to B$, for fix X and arbitrary A and B. Given a function $f : A \to B$, it is lifted to a X-action as follows.

$$\widetilde{f} = f \circ \pi_1 \qquad \text{xaction-LIFT}$$

To exemplify the proof technique we will prove the following law.

$$\widetilde{(\!|h|\!)_F} = \{\!|\widetilde{\tau, h}|\!\}_F \qquad \text{afold-LIFT}$$

Proof.

$$\left[\begin{array}{ll} & \widetilde{(\!|h|\!)_F} = \{\!|\widetilde{\tau, h}|\!\}_F \\ \Leftrightarrow & \quad \{ \text{ xaction-LIFT, cata-DEF, afold-DEF } \} \\ & [\![h, \mathtt{out}_F]\!]_F \circ \pi_1 = [\![h \circ \pi_1, \langle \tau, \pi_2 \rangle \circ (\mathtt{out}_F \times \mathtt{id})]\!]_{F \times \underline{X}} \\ \Leftrightarrow & \quad \{ \pi_1 : F \times \underline{X} \dot{\to} F \} \\ & [\![h, \mathtt{out}_F]\!]_F \circ \pi_1 = [\![h, \pi_1 \circ \langle \tau, \pi_2 \rangle \circ (\mathtt{out}_F \times \mathtt{id})]\!]_F \\ \Leftrightarrow & \quad \{ \times\text{-CANCEL} \} \\ & [\![h, \mathtt{out}_F]\!]_F \circ \pi_1 = [\![h, \tau \circ (\mathtt{out}_F \times \mathtt{id})]\!]_F \\ \Leftarrow & \quad \{ \text{hylo-FUSION} \} \\ & \mathtt{out}_F \circ \pi_1 = F\pi_1 \circ \tau \circ (\mathtt{out}_F \times \mathtt{id}) \\ \Leftrightarrow & \quad \{ \tau\text{-CANCEL} \} \\ & \mathtt{out}_F \circ \pi_1 = \pi_1 \circ (\mathtt{out}_F \times \mathtt{id}) \\ \Leftrightarrow & \quad \{ \times\text{-DEF}, \times\text{-CANCEL} \} \\ & \mathtt{true} \end{array}\right.$$

A similar approach can be applied to the following. Notice that some laws have additional strictness conditions due to the change of the base category.

$$\{\!|\tau, \widetilde{in_F}|\!\}_F = \pi_1 \qquad \text{afold-CANCEL}$$

$$\begin{array}{c} f \circ \{\!|\tau, g|\!\}_F = \{\!|\tau, h|\!\}_F \\ \Leftarrow \\ f \text{ strict } \wedge\ f \circ g = h \circ (Ff \times \mathtt{id}) \end{array} \qquad \text{afold-FUSION}$$

$$\begin{array}{c} \{\!|\tau', h|\!\}_F \circ ((\!|\mathtt{in}_F \circ \kappa|\!)_G \times \mathtt{id}) = \{\!|\tau, h \circ (\kappa \times \mathtt{id})|\!\}_G \\ \Leftarrow \\ \kappa : G \dot{\to} F\ \wedge\ \kappa \circ \tau = \tau' \circ (\kappa \times \mathtt{id}) \end{array} \qquad \text{afold-cata-FUSION}$$

$$\begin{array}{c} \{\!|\tau', h|\!\}_F \circ (\mathtt{id} \times f) = \{\!|\tau, h \circ (\mathtt{id} \times f)|\!\}_F \\ \Leftarrow \\ \tau' \circ (\mathtt{id} \times f) = F(\mathtt{id} \times f) \circ \tau \end{array} \qquad \text{afold-morph-FUSION}$$

Two prove uniqueness we will again resort to an alternative definition of (curried) accumulations using catamorphisms, presented in [15]. The proof of the following equivalence is on appendix B due to its size.

$$\begin{array}{c} [\![h, \langle \tau, \pi_2 \rangle \circ (\mathtt{out}_F \times \mathtt{id})]\!]_{F \times \underline{X}} = \mathtt{ap} \circ (\overline{(\!|h \circ \langle F\mathtt{ap} \circ \tau, \pi_2 \rangle|\!)_F} \times \mathtt{id}) \\ \Leftarrow \\ h \text{ left strict } \wedge\ \tau \text{ left strict} \end{array}$$

With this equivalence, we can prove the following uniqueness law given a left strict τ.

$$\begin{array}{c} f = \{\!|\tau, g|\!\}_F \ \wedge\ g \text{ left strict} \\ \Leftrightarrow \\ f \circ (\mathtt{in}_F \times \mathtt{id}) = g \circ \langle Ff \circ \tau, \pi_2 \rangle \ \wedge\ f \text{ left strict} \end{array} \qquad \text{afold-UNIQ}$$

9 Conclusions

In this paper we have shown how to define some of the usual recursion patterns of algebraic programming in $\mathcal{C}PO$ using hylomorphisms. Traditionally this definition was made by fixpoint. Although it is already known that most recursion patterns can be defined using hylomorphisms, we have shown how to derive their properties directly from that definition and the basic laws of hylomorphisms, without using fixpoint induction such as in [12, 4]. As acknowledged in [16], the definition of accumulations in terms of hylomorphisms was introduced by us and, to our knowledge, this is the first time that the strictness conditions that characterize this recursion pattern in $\mathcal{C}PO$ are clarified. We have also shown how to implement directly the definitions in Haskell, by defining data types explicitly as fixed points of functors, and by using functor transformers.

The main advantages of this approach is to avoid proofs by fixpoint induction, and to increase the understanding of the recursion patterns by factoring them into the composition of a producer and a consumer of a (virtual) intermediate data structure. For example, for accumulations, the intermediate data structure is a labeled variant of the input data type, that explicitly stores the propagated accumulators in each node. We have already successfully applied the same approach to other recursion patterns, like apomorphisms [21, **?**], the dual to paramorphisms, and also to generalized catamorphisms [20], a recursion pattern parameterized with a comonad that abstracts the way in which the argument is used in each recursive step.

References

1. Lex Augusteijn. Sorting morphisms. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 1–27. Springer-Verlag, 1999.
2. Alcino Cunha. Automatic visualization of recursion trees: a case study on generic programming. *Electronic Notes in Theoretical Computer Science*, 86(3), 2003. Selected papers of the 12th International Workshop on Functional and (Constraint) Logic Programming.
3. Alcino Cunha, José Barros, and João Saraiva. Deriving animations from recursive definitions. In *Draft Proceedings of the 14th International Workshop on the Implementation of Functional Languages (IFL'02)*, 2002.
4. Maarten Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, January 1991.
5. Jeremy Gibbons. Calculating functional programs. In R. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, chapter 5, pages 148–203. Springer-Verlag, 2002.
6. Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 273–279. ACM Press, 1998.

7. Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 73–82. ACM Press, 1996.
8. Simon Peyton Jones and John Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language*. February 1999.
9. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 26–37. ACM Press, 2003.
10. Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, October 1990.
11. Lambert Meertens. Paramorphisms. Technical Report RUU-CS-90-4, Utrecht University, Department of Computer Science, January 1990.
12. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS*. Springer-Verlag, 1991.
13. Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*. ACM Press, 1995.
14. Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system HYLO. In *Proceedings of the IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106. Chapman & Hall, 1997.
15. Alberto Pardo. Towards merging recursion and comonads. In Johan Jeuring, editor, *Proceedings of the 2nd Workshop on Generic Programming*, pages 50–68, Ponte de Lima, Portugal, 2000. Department of Computer Science, Utrecht University. Technical Report UU-CS-2000-19.
16. Alberto Pardo. Generic accumulations. In J. Gibbons and J. Jeuring, editors, *Proceedings of the 2002 IFIP TC2 Working Conference on Generic Programming*, pages 49–78, Schloss Dagstuhl, Germany, 2003. Kluwer Academic Publishers.
17. Benjamin Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing Series. MIT Press, 1991.
18. J.C. Reynolds. Semantics of the domain of flow diagrams. *Journal of the ACM*, 24(3):484–503, July 1977.
19. Tarmo Uustalu and Varmo Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *INFORMATICA*, 10(1):5–26, 1999.
20. Tarmo Uustalu, Varmo Vene, and Alberto Pardo. Recursion schemes from comonads. *Nordic Journal of Computing*, 8(3):366–390, 2001.
21. Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, 47(3):147–161, 1998.

A Correctness of the Paramorphism Definition

$$[\![f, F\langle \mathtt{id}, \mathtt{id}\rangle \circ \mathtt{out}_F]\!]_{F\circ(\mathtt{Id}\times\underline{\mu F})} = \pi_1 \circ (\!|\langle f, \mathtt{in}_F \circ F\pi_2\rangle|\!)_F$$
$$\Leftarrow$$
$$f \text{ strict}$$

Proof.

$$
\begin{array}{l}
\lceil\ \dagger\ f \text{ strict} \\
\ \ \beta\ [\![f, F\langle \mathtt{id}, \mathtt{id}\rangle \circ \mathtt{out}_F]\!]_{F\circ(\mathtt{Id}\times\underline{\mu F})} \\
\ \ \\
\ \ \ \ \beta \\
\ \ = \quad \{\ \times\text{-Cancel}\ \} \\
\ \ \ \ \pi_1 \circ \langle \beta, \mathtt{id}\rangle \\
\ \ = \quad \{\ \text{cata-Uniq}\ \} \\
\ \ \ \ \lceil\ \ \langle \beta, \mathtt{id}\rangle \circ \mathtt{in}_F \\
\ \ \ \ \ = \quad \{\ \text{hylo-Cancel}\ \} \\
\ \ \ \ \ \ \langle f \circ F(\beta \times \mathtt{id}) \circ F\langle \mathtt{id}, \mathtt{id}\rangle \circ \mathtt{out}_F, \mathtt{id}\rangle \circ \mathtt{in}_F \\
\ \ \ \ \ = \quad \{\ \times\text{-Absor}\ \} \\
\ \ \ \ \ \ \langle f \circ F\langle \beta, \mathtt{id}\rangle \circ \mathtt{out}_F, \mathtt{id}\rangle \circ \mathtt{in}_F \\
\ \ \ \ \ = \quad \{\ \text{in-out-Inv}\ \} \\
\ \ \ \ \ \ \langle f \circ F\langle \beta, \mathtt{id}\rangle \circ \mathtt{out}_F, \mathtt{in}_F \circ \mathtt{out}_F\rangle \circ \mathtt{in}_F \\
\ \ \ \ \ = \quad \{\ \text{functor-Id}\ \} \\
\ \ \ \ \ \ \langle f \circ F\langle \beta, \mathtt{id}\rangle \circ \mathtt{out}_F, \mathtt{in}_F \circ F\mathtt{id} \circ \mathtt{out}_F\rangle \circ \mathtt{in}_F \\
\ \ \ \ \ = \quad \{\ \times\text{-Cancel}\ \} \\
\ \ \ \ \ \ \langle f \circ F\langle \beta, \mathtt{id}\rangle \circ \mathtt{out}_F, \mathtt{in}_F \circ F(\pi_2 \circ \langle \beta, \mathtt{id}\rangle) \circ \mathtt{out}_F\rangle \circ \mathtt{in}_F \\
\ \ \ \ \ = \quad \{\ \text{functor-Compose}\ \} \\
\ \ \ \ \ \ \langle f \circ F\langle \beta, \mathtt{id}\rangle \circ \mathtt{out}_F, \mathtt{in}_F \circ F\pi_2 \circ F\langle \beta, \mathtt{id}\rangle \circ \mathtt{out}_F\rangle \circ \mathtt{in}_F \\
\ \ \ \ \ = \quad \{\ \times\text{-Fusion}\ \} \\
\ \ \ \ \ \ \langle f, \mathtt{in}_F \circ F\pi_2\rangle \circ F\langle \beta, \mathtt{id}\rangle \circ \mathtt{out}_F \circ \mathtt{in}_F \\
\ \ \ \ \ = \quad \{\ \text{in-out-Inv}\ \} \\
\ \ \ \ \ \ \langle f, \mathtt{in}_F \circ F\pi_2\rangle \circ F\langle \beta, \mathtt{id}\rangle \\
\ \ \ \ \\
\ \ \ \ \lceil\ \ \langle \beta, \mathtt{id}\rangle \text{ strict} \\
\ \ \ \ \Leftrightarrow \quad \{\ \times\text{-Strict}\ \} \\
\ \ \ \ \ \ \beta \text{ strict} \\
\ \ \ \ \Leftarrow \quad \{\ \text{hylo-Strict}\ \} \\
\ \ \ \ \ \ f \text{ strict} \wedge F\langle \mathtt{id}, \mathtt{id}\rangle \circ \mathtt{out}_F \text{ strict} \\
\ \ \ \ \Leftrightarrow \quad \{\ \dagger,\ \times\text{-Strict, in-out-Strict, } \mathtt{id} \text{ strict}\ \} \\
\ \ \ \ \ \ \mathtt{true} \\
\ \ \ \ \pi_1 \circ (\!|\langle f, \mathtt{in}_F \circ F\pi_2\rangle|\!)_F
\end{array}
$$

B Correctness of the Accumulation Definition

$$
\begin{array}{c}
[\![h, \langle \tau, \pi_2\rangle \circ (\mathtt{out}_F \times \mathtt{id})]\!]_{F\times\underline{X}} = \mathtt{ap} \circ ((\!|\overline{h \circ \langle F\mathtt{ap} \circ \tau, \pi_2\rangle}|\!)_F \times \mathtt{id}) \\
\Leftarrow \\
h \text{ left strict} \wedge \tau \text{ left strict}
\end{array}
$$

Proof.

$$
\begin{array}{l}
\dagger\ h \text{ left strict} \\
\ddagger\ \tau \text{ left strict} \\
\beta\ [\![h, \langle \tau, \pi_2 \rangle \circ (\mathtt{out}_F \times \mathtt{id})]\!]_{F \times \underline{X}} \\
\\
\quad \beta \\
= \quad \{\, \wedge\text{-Cancel} \,\} \\
\quad \mathtt{ap} \circ (\overline{\beta} \times \mathtt{id}) \\
= \quad \{\, \text{cata-Uniq} \,\} \\
\qquad \left[\begin{array}{l}
\quad \overline{\beta} \circ \mathtt{in}_F \\
= \quad \{\, \text{hylo-Cancel} \,\} \\
\quad \overline{h \circ (F\beta \times \mathtt{id}) \circ \langle \tau, \pi_2 \rangle \circ (\mathtt{out}_F \times \mathtt{id})} \circ \mathtt{in}_F \\
= \quad \{\, \wedge\text{-Fusion, in-out-Inv} \,\} \\
\quad \overline{h \circ (F\beta \times \mathtt{id}) \circ \langle \tau, \pi_2 \rangle} \\
= \quad \{\, \times\text{-Absor} \,\} \\
\quad \overline{h \circ \langle F\beta \circ \tau, \pi_2 \rangle} \\
= \quad \{\, \wedge\text{-Cancel} \,\} \\
\quad \overline{h \circ \langle F(\mathtt{ap} \circ (\overline{\beta} \times \mathtt{id})) \circ \tau, \pi_2 \rangle} \\
= \quad \{\, \text{functor-Compose} \,\} \\
\quad \overline{h \circ \langle F\mathtt{ap} \circ F(\overline{\beta} \times \mathtt{id}) \circ \tau, \pi_2 \rangle} \\
= \quad \{\, \tau\text{-Nat} \,\} \\
\quad \overline{h \circ \langle F\mathtt{ap} \circ \tau \circ (F\overline{\beta} \times \mathtt{id}), \pi_2 \rangle} \\
= \quad \{\, \times\text{-Cancel}, \times\text{-Def} \,\} \\
\quad \overline{h \circ \langle F\mathtt{ap} \circ \tau \circ (F\overline{\beta} \times \mathtt{id}), \pi_2 \circ (F\overline{\beta} \times \mathtt{id}) \rangle} \\
= \quad \{\, \times\text{-Fusion} \,\} \\
\quad \overline{h \circ \langle F\mathtt{ap} \circ \tau, \pi_2 \rangle} \circ (F\overline{\beta} \times \mathtt{id}) \\
= \quad \{\, \wedge\text{-Fusion} \,\} \\
\quad \overline{h \circ \langle F\mathtt{ap} \circ \tau, \pi_2 \rangle} \circ F\overline{\beta}
\end{array}\right. \\
\\
\qquad \left[\begin{array}{l}
\quad \overline{\beta} \text{ strict} \\
\Leftrightarrow \quad \{\, \wedge\text{-Strict,lstrict-Def,hylo-Cancel} \,\} \\
\quad h \circ (F\beta \times \mathtt{id}) \circ \langle \tau, \pi_2 \rangle \circ (\mathtt{out}_F \times \mathtt{id}) \circ (\bot \times \mathtt{id}) = \bot) \\
\Leftrightarrow \quad \{\, \times\text{-Functor, in-out-Strict, } \times\text{-Fusion, } \ddagger, \times\text{-Def, } \times\text{-Cancel} \,\} \\
\quad h \circ (F\beta \times \mathtt{id}) \circ \langle \bot, \pi_2 \rangle = \bot) \\
\Leftrightarrow \quad \{\, \times\text{-Absor, } \times\text{-Functor} \,\} \\
\quad h \circ (F\beta \circ \bot \times \mathtt{id}) \circ \langle \mathtt{id}, \pi_2 \rangle = \bot) \\
\Leftrightarrow \quad \{\, \beta \text{ strict} \,\} \\
\qquad \left[\begin{array}{l}
\quad \beta \text{ strict} \\
\Leftarrow \quad \{\, \text{hylo-Strict} \,\} \\
\quad h \text{ strict} \langle \tau, \pi_2 \rangle \circ (\mathtt{out}_F \times \mathtt{id}) \text{ strict} \\
\Leftrightarrow \quad \{\, \text{in-out-Strict,} \times\text{-Strict,} \pi_2 \text{ strict} \,\} \\
\quad h \text{ strict} \wedge \tau \text{ strict} \\
\Leftarrow \quad \{\, \text{lstrict-Strict, } \dagger, \ddagger \,\} \\
\quad \mathtt{true}
\end{array}\right. \\
\quad h \circ (\bot \times \mathtt{id}) \circ \langle \mathtt{id}, \pi_2 \rangle = \bot \\
\Leftrightarrow \quad \{\, \dagger, \bot\text{-Def} \,\} \\
\quad \mathtt{true}
\end{array}\right. \\
\quad \mathtt{ap} \circ ((\overline{h \circ \langle F\mathtt{ap} \circ \tau, \pi_2 \rangle})_F \times \mathtt{id})
\end{array}
$$