Generalizing Hylo-shift

Jorge Sousa Pinto
jsp@di.uminho.pt

Techn. Report DI-PURe-04:10:01
October 2004

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-04:10:01
Generalizing Hylo-shift by Jorge Sousa Pinto

Abstract

This note proposes a generalization of the Hylo-shift law for functional program calculation. The generalization allows to handle transformations involving recursive types generated by polynomials where other recursive types occur (for instance, lists of binary trees).

1 Introduction

In the following we assume a general acquaintance with algebraic programming and program calculation concepts [1], including the notion of a *hylomorphism* [3].

Section 2 reviews the Hylo-shift law with many examples, and shows a situation where it would be desirable (but is not possible) to apply the law to perform a specific calculation. Section 3 then presents a generalization of the law that solves this problem.

2 The Hylomorphism Shifting Law

Let $\mu\mathsf{L}, \mu\mathsf{M}$ be the types generated by base functors L, M respectively, and $\varepsilon : \mathsf{L} \longrightarrow \mathsf{M}$ a natural transformation.

Then for any $\psi : A \to \mathsf{L}A, \varphi : \mathsf{M}B \to B$ the following shifting law applies, concerning hylomorphisms of type $A \to B$:

$$[\![\varphi \circ \varepsilon, \psi]\!]_{\mathsf{L}} = [\![\varphi, \varepsilon \circ \psi]\!]_{\mathsf{M}} \tag{1}$$

The proof is straightforward using the definition of hylomorphism:

$$\begin{array}{ll} & [\![\varphi \circ \varepsilon, \psi]\!]_{\mathsf{L}} \\ = & \quad \{\text{Definition of hylomorphism}\} \\ & \mu(\lambda f.(\varphi \circ \varepsilon \circ \mathsf{L}f \circ \psi)) \\ = & \quad \{\varepsilon : \mathsf{L} \longrightarrow \mathsf{M}\} \\ & \mu(\lambda f.(\varphi \circ \mathsf{M}f \circ \varepsilon \circ \psi)) \\ = & \quad \{\text{Definition of hylomorphism}\} \\ & [\![\varphi, \varepsilon \circ \psi]\!]_{\mathsf{M}} \end{array}$$

This shows how a recursive function can be converted from a hylomorphism of intermediate type $\mu\mathsf{L}$ to a hylomorphism of some other type $\mu\mathsf{M}$. The intermediate data-structure, on the other hand, can be converted by unfolding. We call ι this conversion function:

$$\iota = [\![\varepsilon \circ \mathsf{out}_{\mathsf{L}}]\!]_{\mathsf{M}} \tag{2}$$

This conversion can also be carried out by folding over the initial structure:

$$\iota = (\!|\mathsf{in}_{\mathsf{M}} \circ \varepsilon|\!)_{\mathsf{L}} \tag{3}$$

as can easily be shown:

$$
\begin{array}{ll}
 & [\![\varepsilon \circ \mathsf{out_L}]\!]_\mathsf{M} \\
= & \quad \{\text{Cata-refl}\} \\
 & (\!|\mathsf{in_M}|\!)_\mathsf{M} \circ [\![\varepsilon \circ \mathsf{out_L}]\!]_\mathsf{M} \\
= & \quad \{\text{Hylo-split}\} \\
 & [\![\mathsf{in_M}, \varepsilon \circ \mathsf{out_L}]\!]_\mathsf{M} \\
= & \quad \{\text{Hylo-shift}\} \\
 & [\![\mathsf{in_M} \circ \varepsilon, \mathsf{out_L}]\!]_\mathsf{L} \\
= & \quad \{\text{Hylo-split}\} \\
 & (\!|\mathsf{in_M} \circ \varepsilon|\!)_\mathsf{L} \circ [\![\mathsf{out_L}]\!]_\mathsf{L} \\
= & \quad \{\text{Ana-refl}\} \\
 & (\!|\mathsf{in_M} \circ \varepsilon|\!)_\mathsf{L}
\end{array}
$$

The following derived laws make precise what is meant by ι being a data-structure conversion function:

$$(\!|\varphi \circ \varepsilon|\!)_\mathsf{L} = (\!|\varphi|\!)_\mathsf{M} \circ \iota \tag{4}$$

$$[\![\varepsilon \circ \psi]\!]_\mathsf{M} = \iota \circ [\![\psi]\!]_\mathsf{L} \tag{5}$$

We prove the former:

$$
\begin{array}{ll}
 & (\!|\varphi \circ \varepsilon|\!)_\mathsf{L} \\
= & \quad \{\text{Ana-refl}\} \\
 & (\!|\varphi \circ \varepsilon|\!)_\mathsf{L} \circ [\![\mathsf{out_L}]\!]_\mathsf{L} \\
= & \quad \{\text{Hylo-split}\} \\
 & [\![\varphi \circ \varepsilon, \mathsf{out_L}]\!]_\mathsf{L} \\
= & \quad \{\text{Hylo-shift}\} \\
 & [\![\varphi, \varepsilon \circ \mathsf{out_L}]\!]_\mathsf{M} \\
= & \quad \{\text{Hylo-split}\} \\
 & (\!|\varphi|\!)_\mathsf{M} \circ [\![\varepsilon \circ \mathsf{out_L}]\!]_\mathsf{M} \\
= & \quad \{\text{def. } \iota\} \\
 & (\!|\varphi|\!)_\mathsf{M} \circ \iota
\end{array}
$$

From these it is easy to see that (1) can alternatively be written as the composition of an anamorphism of type $\mu\mathsf{L}$, the conversion function ι, and a catamorphism of type $\mu\mathsf{M}$.

$$[\![\varphi \circ \varepsilon, \psi]\!]_{\mathsf{L}} = (\!|\varphi|\!)_{\mathsf{M}} \circ \iota \circ [\!(\psi)\!]_{\mathsf{L}} = [\![\varphi, \varepsilon \circ \psi]\!]_{\mathsf{M}} \tag{6}$$

2.1 Examples

We give here some examples of using the Hylo-shift law. The first two correspond to situations in which the natural transformation is an isomorphism; as such, the law corresponds to a change in the representation of the intermediate data-structure, without loss of information.

Isomorphic Types Consider the factorial function:

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

Roughly, a recursive function can be seen as a hylomorphism where the intermediate structure is a tree shaped as the recursion tree of the function, and each node of the tree contains the extra information that is needed, apart from the results of recursive calls, to construct the result at each recursion level.

So factorial can be seen as a hylomorphism where the intermediate structure is a list containing all the numbers from n down to 1; to obtain the final result it simply remains to multiply all these numbers. The base functor defining the type of lists of integers is given on types and functions by

$$\mathsf{L}\ X = 1 + \mathsf{Int} \times X \qquad\qquad \mathsf{L}\ f = \mathsf{id} + \mathsf{id} \times f$$

The corresponding data-type is defined as

$$\mathsf{List} = \mu\mathsf{L}$$

and in Haskell:

```
data List = Nil | Cons (Int, List)
```

Then one can write

$$\mathsf{fact} = [\![g, f]\!]_{\mathsf{L}}$$

where $[\![f]\!]_{\mathsf{L}}$ constructs the list of all numbers from n down to 1 and $(\!|g|\!)_{\mathsf{L}}$ multiplies the numbers in this list. In Haskell:

```
f :: Int -> Either () (Int,Int)
f 0 = inl ()
f n = inr (n,n-1)

g :: Either () (Int,Int) -> Int
g (inl ()) = 1
g (inr (n,r)) = n * r
```

It is obvious that the same function can be seen as a hylomorphism of the following isomorphic type to `List`:

```
data SList = Lin | Snoc (List, Int)
```

This is called a *Snoc List* since the constructor takes its arguments in reversed order with respect to *Cons* above. The base functor corresponding to *Snoc Lists* is defined by

$$\mathsf{M}\ X = 1 + X \times \mathsf{Int} \qquad\qquad \mathsf{M}\ f = \mathsf{id} + f \times \mathsf{id}$$

This equivalence can be formally stated by presenting the following natural isomorphism $\varepsilon : L \longrightarrow M$:

$$\varepsilon = \mathsf{id} + \mathsf{swap}$$

where swap is an isomorphism that swaps the components of a pair. We note that $g = g' \circ \varepsilon$ with g' defined as

```
g' :: Either () (Int,Int) -> Int
g' (inl ()) = 1
g' (inr (r,n)) = n * r
```

Then Hylo-shift yields

$$\mathsf{fact} = [\![g' \circ \varepsilon, f]\!]_{\mathsf{L}} = [\![g', \varepsilon \circ f]\!]_{\mathsf{M}} = [\![g', f']\!]_{\mathsf{M}}$$

where

```
f' :: Int -> Either () (Int,Int)
f' 0 = inl ()
f' n = inr (n-1,n)
```

Finally, observe that the function allowing to convert intermediate data-structures from type $\mu\mathsf{L}$ to type $\mu\mathsf{M}$ is as expected both a fold of the former type and an unfold of the latter:

```
iota :: List -> Slist
iota Nil = Lin
iota Cons(x,xs) = Snoc(iota xs,x)
```

Polymorphic Data-types and Bifunctors Let us now turn to the data-type of polymorphic lists:

```
data List a = Nil | Cons (a, List a)
```

The previous example can of course also be considered as a hylomorphism with this intermediate type. The base functors of polymorphic data-types are in fact *bifunctors*. In this particular case we have for polymorphic lists:

$$\mathsf{B}\ (Z, X) = 1 + Z \times X \qquad\qquad \mathsf{B}\ (i, f) = \mathsf{id} + i \times f$$

An instantiation of the type variable corresponds to a *sectioning* of the base functor; for a given type A the (mono-)functor ${}^{A}\mathsf{B}$ is defined as follows

$${}^{A}\mathsf{B}X = \mathsf{B}\ (A, X) = 1 + A \times X \qquad\qquad {}^{A}\mathsf{B}f = \mathsf{B}\ (\mathsf{id}, f) = \mathsf{id} + \mathsf{id} \times f$$

So in fact we have

$$\mathsf{L} = {}^{\mathsf{Int}}\mathsf{B}$$

The polymorphic type of lists is then given as

$$\mathsf{PList}\ A = \mu({}^{A}\mathsf{B})$$

and this relates to the Integer list type as

$$\mathsf{List} \cong \mathsf{PList}\ \mathsf{Int}$$

Now if we define for $f : A \to B$

$$\mathsf{PList}\ f = \mathsf{B}\ (f, \mathsf{PList}\ f)$$

we have that $\mathsf{PList}\ f : \mathsf{PList}\ A \to \mathsf{PList}\ B$. This in fact defines a so-called *type functor*, which behaves as a *map* function for the type PList.

A convenient alternative representation for bifunctors uses infix symbols; for instance we could represent B by symbol $\oplus$:

$$Z \oplus X = 1 + Z \times X \qquad\qquad i \oplus f = \mathsf{id} + i \times f$$

And the sectioned functor is now called $A\oplus$:

$$(A\oplus)X = \mathsf{B}\ (A, X) = 1 + A \times X \qquad\qquad (A\oplus)f = \mathsf{B}\ (\mathsf{id}, f) = \mathsf{id} + \mathsf{id} \times f$$

Sectioning can also be done on the second argument of the functor. The *right-sectioning* of bifunctor $\oplus$ (as opposed to *left-sectioning*, defined before) is given by

$$(\oplus A)X = \mathsf{B}\ (X, A) = 1 + X \times A \qquad (\oplus A)f = \mathsf{B}\ (f, \mathsf{id}) = \mathsf{id} + f \times \mathsf{id}$$

All considerations concerning Hylo-shift and natural transformations apply as before to sectioned bifunctors. As an example of using Hylo-shift with polymorphism, let us express factorial as a hylomorphism of lists of a user-defined type of naturals, rather than using Haskell's integers.

We start by defining the data-type together with a conversion function from integers to naturals (valid for non-negative integers only).

```
data Nat = Zero | Succ Nat
i2n :: Int -> Nat
i2n 0 = Zero
i2n n = Succ (i2n (n-1))
```

Now we seek a natural transformation $\varepsilon : {}^{\mathsf{Int}}\mathsf{B} \longrightarrow^{\mathsf{Nat}} \mathsf{B}$. This is simply

$$\varepsilon = \mathsf{id} + \mathsf{i2n} \times \mathsf{id}$$

The next step is to rewrite fact as:

$$\mathsf{fact} = [\![g'' \circ \varepsilon, f]\!]_{\mathsf{Int}\mathsf{B}}$$

where

```
g'' :: Either () (Nat,Int) -> Int
g'' (inl ()) = 1
g'' (inr (n,r)) = (n2i n) * r
```

and n2i is the inverse of i2n (restricted to non-negative numbers). Then Hylo-shift yields

$$\mathsf{fact} = [\![g'', \varepsilon \circ f]\!]_{\mathsf{Nat}\mathsf{B}} = [\![g'', f'']\!]_{\mathsf{Nat}\mathsf{B}}$$

and

```
f'' :: Int -> Either () (Nat, Int)
f'' 0 = inl ()
f'' n = inr (i2n n,n-1)
```

Again, the function that converts intermediate data-structures is both a fold and an unfold of lists; in fact it is in this case a *map*:

```
iota :: List Int -> List Nat
iota Nil = Nil
iota Cons(x,xs) = Cons(i2n x,iota xs)
```

Non-isomorphic Types Our examples so far have been straightforward and corresponded to isomorphisms between intermediate types. The next example is a bit more interesting.

Consider the Haskell function

```
bubble :: [Int] -> [int]
bubble [] = []
bubble [x] = [x]
bubble (a:b:t) | a<=b = a : bubble (b:t)
               | otherwise = b : bubble (a:t)
```

This can be seen as a hylomorphism with a binary tree as intermediate type, with base functor

$$\mathsf{L}\ X = (1 + \mathsf{Int}) + (\mathsf{Int} \times \mathsf{Int}) \times (X \times X) \qquad \mathsf{L}\ f = \mathsf{id} + \mathsf{id} \times (f \times f)$$

The corresponding data-type is defined by

$$\mathsf{BTree} = \mu\mathsf{L}$$

and in Haskell:

```
data BTree = Leaf (Maybe Int) | Node ((Int, Int), (BTree, BTree))
```

Then one can write

$$\mathsf{bubble} = [\![g, f]\!]_{\mathsf{L}}$$

where f and g are

```
f :: [Int] -> Either (Maybe Int) ((Int, Int), ([Int], [Int]))
f [] = inl Nothing
f [x] = inl (Just x)
f (a:b:t) = inr ((a,b),(b:t,a:t))

g :: Either (Maybe Int) ((Int, Int), ([Int], [Int])) -> [Int]
g (inl Nothing) = []
g (inl (Just x)) = [x]
g (inr ((a,b),(r1,r2)) | a<=b = a:r1
                       | otherwise = b:r2
```

The anamorphism executed by itself would generate a tree containing all the possible traces of execution; in the context of the hylomorphism however (and Haskell being a lazy language), it will produce a linear tree: a single sub-tree will be constructed for each node. Positive (resp.

negative) tests of the guard `a<=b` correspond to left (resp. right) branches of the tree.

An alternative formulation of `bubble` as a hylomorphism uses a list of integers as intermediate type:

$$\mathsf{M}\ X = (1 + \mathsf{Int}) + \mathsf{Int} \times X \qquad\qquad \mathsf{M}\ f = \mathsf{id} + \mathsf{id} \times f$$

```
data Listb = End (Maybe Int) | Cons (Int, Listb)
```

This alternative formulation can be obtained using the Hylo-shift law with the following natural transformation $\varepsilon : \mathsf{L} \rightarrow \mathsf{M}$:

$$\varepsilon = \mathsf{id} + \mathsf{aux}$$

with aux defined as the polymorphic function

```
aux :: ((Int, Int), a, a) -> (Int, a)
aux ((u,v),(x,y)) | u<=v = (u,x)
                  | otherwise = (v,y)
```

The rest follows naturally: bubble is rewritten as

$$\mathsf{bubble} = [\![g' \circ \varepsilon, f]\!]_{\mathsf{L}}$$

where

```
g' :: Either (Maybe Int) (Int, [Int]) -> [Int]
g' (inl Nothing) = []
g' (inl (Just x)) = [x]
g' (inr (x,r)) = x:r
```

Applying Hylo-shift:

$$\mathsf{bubble} = [\![g', \varepsilon \circ f]\!]_{\mathsf{M}} = [\![g', f']\!]_{\mathsf{M}}$$

and

```
f' :: [Int] -> Either (Maybe Int) (Int, [Int])
f' [] = inl Nothing
f' [x] = inl (Just x)
f' (a:b:t) | a<=b = inr(a,b:t)
           | otherwise = inr(b,a:t)
```

We remark that we have in fact shifted the decision based on the boolean test from the fold component of the hylomorphism to the unfold. The function ι is not invertible (ε is not an isomorphism), which means that the intermediate data-structures can only be converted with loss of information. The first version of the hylomorphism is thus more informative.

Consider for instance an execution of `bubble` on the input `[1,3,4,2]`. The hylomorphism $[\![g', f']\!]_{\mathsf{M}}$ produces the intermediate structure

```
Cons(1,Cons(3,Cons(2,End (Just 4))))
```

whereas $[\![g, f]\!]_{\mathsf{L}}$ produces

```
Node((1,3),Node((3,4),Node((4,2),...,Leaf(Just 4)),...),...)
```

where parts of the structure have not been computed. We finish with the explicitly recursive definition of the intermediate structure conversion function:

```
iota : BTree -> Listb
iota (Leaf z) = End z
iota (Node((x,y),(l,r))) | x<=y = Cons (x,iota l)
                         | otherwise = Cons (y,iota r)
```

2.2 When Hylo-shift Fails to Apply

From the first two examples it seems that it should always be possible to transform any hylomorphism into an equivalent version that uses an isomorphic intermediate data-structure. We now present an example of a situation in which Hylo-shift does not allow to prove such an equivalence.

The example is based on the well-known isomorphism between the type of binary trees and the type of lists of binary trees. Let us focus for instance on node-labeled trees given by the base functor

$$\mathsf{F}\ X = 1 + \mathsf{Int} \times (X \times X) \qquad \mathsf{F}\ f = \mathsf{id} + \mathsf{id} \times (f \times f)$$

defining the type

$$\mathsf{BTree} = \mu\mathsf{F}$$

Each of the two *spines* [2] in the tree provides a way of viewing it as a list. In any case, the underlying isomorphism is between a binary tree and a list containing trees and base elements, as given by the functor

$$\mathsf{G}\ X = 1 + (\mathsf{Int} \times \mathsf{BTree}) \times X \qquad \mathsf{G}\ f = \mathsf{id} + \mathsf{id} \times f$$

which defines the type

$$\mathsf{TList} = \mu\mathsf{G}$$

Then it is easy to see that both types are solutions to the equation $X \cong GX$, and

$$\mathsf{BTree} \cong \mathsf{TList}$$

The problem here is that there is no natural isomorphism $\varepsilon : \mathsf{F} \longrightarrow \mathsf{G}$, due to the fixpoint type BTree occurring in G. A different formulation of the law is required to allow calculations to proceed.

3 The Generalized Hylo-shift Law

We need the following, straightforward to prove

Lemma 1. *Let $\oplus$ be a bifunctor and $f : A \to B$, $g : C \to D$. Then*

$$(B\oplus)g \circ (\oplus C)f = f \oplus g = (\oplus D)f \circ (A\oplus)g$$

Generalized Hylo-shift Law. Now let F, G be (mono)functors and $\oplus$ a bifunctor (represented using infix notation) such that $\mathsf{G} = (\mu\mathsf{F})\oplus$. Moreover, let α be a natural transformation $\alpha : \mathsf{F} \longrightarrow \mathsf{H}$, with H defined as

$$\begin{array}{ll} \mathsf{H}\ X = X \oplus X & \text{on types, and} \\ \mathsf{H}\ f = f \oplus f & \text{on functions} \end{array}$$

The following law holds, with $\psi : A \to \mathsf{F}A$ and $\varphi : \mathsf{G}B \to B$,

$$[\![\varphi \circ \alpha, \psi]\!]_{\mathsf{F}} = [\![\varphi \circ (\oplus B)\,(\!|\varphi \circ \alpha|\!)_{\mathsf{F}}\,, (\oplus A)[\!|\psi|\!]_{\mathsf{F}} \circ \alpha \circ \psi]\!]_{\mathsf{G}} \qquad (7)$$

This is proved as follows:

$$
\begin{array}{ll}
 & [\![\varphi \circ \alpha, \psi]\!]_{\mathsf{F}} \\
= & \quad \{\text{Def. Hylomorphism}\} \\
 & \mu(\lambda f.(\varphi \circ \alpha \circ \mathsf{F}f \circ \psi)) \\
= & \quad \{\alpha : \mathsf{F} \longrightarrow \mathsf{H}\} \\
 & \mu(\lambda f.(\varphi \circ \mathsf{H}f \circ \alpha \circ \psi)) \\
= & \quad \{\text{def. } \mathsf{H}\} \\
 & \mu(\lambda f.(\varphi \circ (f \oplus f) \circ \alpha \circ \psi)) \\
= & \quad \{\text{Lemma 1, } f : A \to B\} \\
 & \mu(\lambda f.(\varphi \circ (\oplus B)f \circ (A\oplus)f \circ \alpha \circ \psi)) \\
= & \quad \{\text{Reduction, fixpoint calculation}\} \\
 & \mu(\lambda f.(\varphi \circ (\oplus B)[\![\varphi \circ \alpha, \psi]\!]_{\mathsf{F}} \circ (A\oplus)f \circ \alpha \circ \psi)) \\
= & \quad \{\text{Hylo-split}\} \\
 & \mu(\lambda f.(\varphi \circ (\oplus B)\,((\!|\varphi \circ \alpha|\!)_{\mathsf{F}} \circ [\![\psi]\!]_{\mathsf{F}}) \circ (A\oplus)f \circ \alpha \circ \psi)) \\
= & \quad \{\text{Functors}\} \\
 & \mu(\lambda f.(\varphi \circ (\oplus B)\,(\!|\varphi \circ \alpha|\!)_{\mathsf{F}} \circ (\oplus B)[\![\psi]\!]_{\mathsf{F}} \circ (A\oplus)f \circ \alpha \circ \psi)) \\
= & \quad \{\text{Lemma 1, } f : A \to B,\ [\![\psi]\!]_{\mathsf{F}} : A \to \mu\mathsf{F}\} \\
 & \mu(\lambda f.(\varphi \circ (\oplus B)\,(\!|\varphi \circ \alpha|\!)_{\mathsf{F}} \circ ((\mu F)\oplus)f \circ (\oplus A)[\![\psi]\!]_{\mathsf{F}} \circ \alpha \circ \psi)) \\
= & \quad \{\text{Def. } \mathsf{G}\} \\
 & \mu(\lambda f.(\varphi \circ (\oplus B)\,(\!|\varphi \circ \alpha|\!)_{\mathsf{F}} \circ \mathsf{G}f \circ (\oplus A)[\![\psi]\!]_{\mathsf{F}} \circ \alpha \circ \psi)) \\
= & \quad \{\text{Def. Hylomorphism}\} \\
 & [\![\varphi \circ (\oplus B)\,(\!|\varphi \circ \alpha|\!)_{\mathsf{F}}\,, (\oplus A)[\![\psi]\!]_{\mathsf{F}} \circ \alpha \circ \psi]\!]_{\mathsf{G}}
\end{array}
$$

This new law may be written alternatively as

$$[\![\varphi \circ \alpha, \psi]\!]_{\mathsf{F}} = [\![\varphi \circ (\!|\varphi \circ \alpha|\!)_{\mathsf{F}} \oplus \mathsf{id}, [\![\psi]\!]_{\mathsf{F}} \oplus \mathsf{id} \circ \alpha \circ \psi]\!]_{\mathsf{G}} \tag{8}$$

Derived Laws If we define, as in (2)

$$\iota = [\![\alpha \circ \mathsf{out}_{\mathsf{F}}]\!]_{\mathsf{G}}$$

we can derive a similar law to (4):

$$(\!|\varphi \circ \alpha|\!)_{\mathsf{F}} = (\!|\varphi \circ (\!|\varphi \circ \alpha|\!)_{\mathsf{F}} \oplus \mathsf{id}|\!)_{\mathsf{G}} \circ \iota \tag{9}$$

and also

$$[\![\varphi \circ \alpha, \psi]\!]_{\mathsf{F}} = (\!|\varphi \circ (\!|\varphi \circ \alpha|\!)_{\mathsf{F}} \oplus \mathsf{id}|\!)_{\mathsf{G}} \circ \iota \circ [\!(\psi)\!]_{\mathsf{F}} \tag{10}$$

We remark however that $\iota = (\!|\mathsf{in}_{\mathsf{G}} \circ \alpha|\!)_{\mathsf{F}}$ does *not* hold in general. ι can only be written as a catamorphism when α is an isomorphism, in which case ι has an inverse.

To see that this is so we assume $\iota = (\!|\varphi \circ \alpha|\!)_{\mathsf{F}}$ for some φ; then we may calculate

$$\begin{array}{ll}
 & \iota \\
= & \quad \{\text{Ana-refl}\} \\
 & (\!|\varphi \circ \alpha|\!)_{\mathsf{F}} \circ [\!(\mathsf{out}_{\mathsf{F}})\!]_{\mathsf{F}} \\
= & \quad \{\text{Hylo-split}\} \\
 & [\![\varphi \circ \alpha, \mathsf{out}_{\mathsf{F}}]\!]_{\mathsf{F}} \\
= & \quad \{\text{law (8)}\} \\
 & [\![\varphi \circ (\!|\varphi \circ \alpha|\!)_{\mathsf{F}} \oplus \mathsf{id}, [\!(\mathsf{out}_{\mathsf{F}})\!]_{\mathsf{F}} \oplus \mathsf{id} \circ \alpha \circ \mathsf{out}_{\mathsf{F}}]\!]_{\mathsf{G}} \\
= & \quad \{\text{def. of } \iota, \text{ Ana-refl}\} \\
 & [\![\varphi \circ \iota \oplus \mathsf{id}, \mathsf{id} \oplus \mathsf{id} \circ \alpha \circ \mathsf{out}_{\mathsf{F}}]\!]_{\mathsf{G}} \\
= & \quad \{\text{Functors}\} \\
 & [\![\varphi \circ \iota \oplus \mathsf{id}, \alpha \circ \mathsf{out}_{\mathsf{F}}]\!]_{\mathsf{G}} \\
= & \quad \{\text{Hylo-split; def. } \iota\} \\
 & (\!|\varphi \circ \iota \oplus \mathsf{id}|\!)_{\mathsf{G}} \circ \iota
\end{array}$$

This is not in general true if $\varphi = \mathsf{in}_{\mathsf{G}}$. It is true if we let (Cata-refl)

$$\begin{array}{ll}
 & \varphi \circ \iota \oplus \mathsf{id} = \mathsf{in}_{\mathsf{G}} \\
\Leftrightarrow & \quad \{\iota \text{ invertible; Functors}\} \\
 & \varphi = \mathsf{in}_{\mathsf{G}} \circ \iota^{\circ} \oplus \mathsf{id}
\end{array}$$

this yields

$$\iota = (\!|\mathsf{in}_{\mathsf{G}} \circ \iota^{\circ} \oplus \mathsf{id} \circ \alpha|\!)_{\mathsf{F}} \tag{11}$$

Example. Take the "trees as lists" example. F and G are the following base functors:

$$\begin{array}{ll}
\mathsf{F}\ X = 1 + \mathsf{Int} \times (X \times X) & \mathsf{F}\ f = \mathsf{id} + \mathsf{id} \times (f \times f) \\
\mathsf{G}\ X = 1 + (\mathsf{Int} \times \mu\mathsf{F}) \times X & \mathsf{G}\ f = \mathsf{id} + \mathsf{id} \times f
\end{array}$$

We now define a bifunctor $\oplus$ by introducing a new parameter in G, abstracting from the type μF.

$$Z \oplus X = 1 + (\mathsf{Int} \times Z) \times X \qquad\qquad i \oplus f = \mathsf{id} + (\mathsf{id} \times i) \times f$$

We have $\mathsf{G} = (\mu \mathsf{F}) \oplus$, as required. This also gives us a definition for H:

$$\mathsf{H}\ X = 1 + (\mathsf{Int} \times X) \times X \qquad\qquad \mathsf{H}\ f = \mathsf{id} + (\mathsf{id} \times f) \times f$$

and $\alpha : \mathsf{F} \longrightarrow \mathsf{H}$ is easy to find:

$$\alpha = \mathsf{id} + \mathsf{assocl}$$

and assocl is the isomorphism $A \times (B \times C) \cong (A \times B) \times C$.

Notice also that

$$f \oplus \mathsf{id} = \mathsf{id} + (\mathsf{id} \times f) \times \mathsf{id}$$

and our law allows to write

$$[\![\varphi \circ (\mathsf{id} + \mathsf{assocl}), \psi]\!]_{\mathsf{F}} =$$
$$[\![\varphi \circ (\mathsf{id} + (\mathsf{id} \times (\!|\varphi \circ \mathsf{id} + \mathsf{assocl}|\!)_{\mathsf{F}}) \times \mathsf{id}), (\mathsf{id} + (\mathsf{id} \times [\!|\psi|\!]_{\mathsf{F}}) \times \mathsf{id}) \circ (\mathsf{id} + \mathsf{assocl}) \circ \psi]\!]_{\mathsf{G}}$$

To be more specific, let lists and binary trees be defined in Haskell as

```
data BTree = Empty | Node (Int, (BTree, BTree))
data List = Nil | Cons ((Int, BTree), List)
```

The conversion function, written with explicit recursion, is the anamorphism

```
iota Empty = Nil
iota (Node(x,(l,r))) = Cons((x,l),iota r)
```

A concrete example may provide further insight; we take the standard view of quicksort as a hylomorphism, using Haskell's predefined list type for the input and output.

$$\mathsf{qsort} = [\![g, f]\!]_{\mathsf{F}}$$

where

```
f :: [Int] -> Either () (Int, ([Int],[Int]))
f []    = inl()
f (h:t) = inr(h, split h t)

g :: Either () (Int, ([Int] ,[Int]) -> [Int]
g (inl())        = []
g (inr(x,(l,r)) = l++(x:r)
```

where `split :: Int -> [Int] -> ([Int],[Int])` separates elements smaller (or equal) and greater than a given value. From the above we obtain

$$\mathsf{qsort} = [\![g', f']\!]_{\mathsf{G}}$$

with

```
f' :: [Int] -> Either () ((Int, BTree), [Int])
f' []    = inl()
f' (h:t) = inr((h, unfoldBTree F a),b) where (a,b) = split h t

g' :: Either () ((Int, BTree) ,[Int]) -> [Int]
g' (inl())        = []
g' (inr((x,t),r)) = (foldBTree g t)++(x:r)
```

and `unfoldBTree`, `foldBTree` are the standard anamorphism and catamorphism recursion patterns for binary trees.

Example: Polymorphic Lists. The above result can in fact be written for polymorphic lists. Consider the bifunctors of binary trees and lists:

$$\mathsf{F}\ (Z, X) = 1 + Z \times (X \times X) \qquad \mathsf{F}\ (i, f) = \mathsf{id} + i \times (f \times f)$$
$$\mathsf{G}\ (Z, X) = 1 + Z \times X \qquad \mathsf{G}\ (i, f) = \mathsf{id} + i \times f$$

and the types

$$\mathsf{PBTree}\ A = \mu({}^{A}\mathsf{F})$$
$$\mathsf{PList}\ A = \mu({}^{A}\mathsf{G})$$

We now want to explore the isomorphism

$$\mathsf{PBTree}\ A \cong \mathsf{PList}\ (A \times \mathsf{PBTree}\ A)$$

It suffices to observe that the sectioned functors ${}^{A}F$ and ${}^{(A \times \mathsf{PBTree}\ A)}G$ are the same as in the monomorphic example (with A substituting Int).

4 Further Laws

If we consider again the "trees as lists" example, since there is an isomorphism between the two intermediate types $\mu\mathsf{F}$ and $\mu\mathsf{G}$, it should be possible to apply law (8) with the roles of the two functors reversed, using the natural transformation $\beta = \alpha^\circ$. It is easy to see however, that there is an asymmetry in the law that prevents this; the adequate law to use here is symmetric to (8) and can be derived in the same way.

Considering the same functors of the previous section and a natural transformation $\beta : \mathsf{H} \longrightarrow \mathsf{F}$, this law can be written as:

$$[\![\varphi, \beta \circ \psi]\!]_{\mathsf{F}} = [\![\varphi \circ \beta \circ (\!|\varphi|\!)_{\mathsf{F}} \oplus \mathsf{id}, [\!|\beta \circ \psi|\!]_{\mathsf{F}} \oplus \mathsf{id} \circ \psi]\!]_{\mathsf{G}}$$

The reader will not have difficulty in understanding that the conversion function will now be written as the catamorphism $\iota = (\!|\mathsf{in}_{\mathsf{M}} \circ \varepsilon|\!)_{\mathsf{L}}$, not as an anamorphism.

References

1. Richard Bird and Oege de Moor. *Algebra of Programming.* Prentice Hall, 1997.

2. Oege de Moor and Jeremy Gibbons. Pointwise relational programming. In *Proceedings of AMAST'00*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

3. 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.