Point-free Programming with Hylomorphisms

Alcino Cunha

Departamento de Informática, Universidade do Minho
4710-057 Braga, Portugal
alcino@di.uminho.pt

WDGP'04, June 3rd

Introduction

- Pointless - a Haskell library for point-free programming with recursion patterns:
  - Categorical combinators + Hylomorphisms;
  - Types as fixed points of functors;
  - Mimics the theoretical notation;
  - Mainly built on top of PolyP's ideas [NJ03].
- Examples of how to implement some of the standard recursion patterns using hylomorphisms.
- Some related tools: visualization of intermediate data structures, derivation of hylomorphisms and point-free definitions.

Categorical Combinators in Haskell

$$A \times B = \{(x, y) \mid x \in A, y \in B\}$$

$$\mathsf{fst}\ (x, y) = x$$

$$\mathsf{snd}\ (x, y) = y$$

$$(f \vartriangle g)\ x = (f\ x, g\ x)$$

$$f \times g = (f \circ \mathsf{fst}) \vartriangle (g \circ \mathsf{snd})$$

Categorical Combinators in Haskell

- Products are implemented by Haskell pairs, even if these aren't true products: $(\perp, \perp) \neq \perp$.

```
-- fst :: (a, b) -> a
-- fst (x, y) = x

-- snd :: (a, b) -> b
-- snd (x, y) = y

(/\) :: (a -> b) -> (a -> c) -> a -> (b,c)
(/\) f g x = (f x, g x)

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

Categorical Combinators in Haskell

$$A + B = \{0\} \times A \cup \{1\} \times B \cup \{\perp\}$$

$$\mathsf{inl}\ x = (0, x)$$

$$\mathsf{inr}\ x = (1, x)$$

$$(f \triangledown g)\ x = \begin{cases} f\ (\mathsf{snd}\ x) & \text{if } \mathsf{fst}\ x = 0 \\ g\ (\mathsf{snd}\ x) & \text{if } \mathsf{fst}\ x = 1 \\ \perp & \text{otherwise} \end{cases}$$

$$f + g = (\mathsf{inl} \circ f) \triangledown (\mathsf{inr} \circ g)$$

Categorical Combinators in Haskell

- Sums are implemented by the `Either` data type.

```
inl :: a -> Either a b
inl = Left

inr :: b -> Either a b
inr = Right

(\/) :: (b -> a) -> (c -> a) -> Either b c -> a
(\/) f g (Left x)  = f x
(\/) f g (Right x) = g x

(-|-) :: (a -> b) -> (c -> d) -> Either a c -> Either b d
f -|- g = (inl . f) \/ (inr . g)
```

Categorical Combinators in Haskell

$$B^A = \{f \mid f : A \to B\}$$

$$\mathsf{ap}\ (f, x) = f\ x$$

$$\overline{f}\ x\ y = f\ (x, y)$$

$$f^\bullet = \overline{f \circ \mathsf{ap}}$$

Categorical Combinators in Haskell

- Exponentials are implemented by Haskell functions. Notice that these aren't also true functions: $\lambda x.\bot \neq \bot$

- Function exponentiation is implemented by the left sectioning of the composition combinator.

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

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

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

Categorical Combinators in Haskell

- Postfix operators for constants and guards are simulated by using left-sectionings of binary operators.

```
(!) :: a -> b -> a
(!) = const

(?) :: (a -> Bool) -> a -> Either a a
p ? x = if p x then inl x else inr x
```

- Since the bottom element will be used frequently, we also define the following alias.

```
_L :: a
_L = undefined
```

Recursive Data Types

- A recursive data type is modeled by the least fixed point of a base functor that captures the recursive structure of its constructors.
- For a regular functor F, the data type μF, together with two strict functions $\mathsf{in}_F : F\ (\mu F) \to \mu F$ and $\mathsf{out}_F : \mu F \to F\ (\mu F)$, are guaranteed to exist [Rey77, FM91].

$$\mathsf{Nat} = \mu(1! + \mathsf{Id})$$
$$\mathsf{in} = \mathsf{zero}! \triangledown \mathsf{succ}$$
$$\mathsf{out} = (1! + \mathsf{pred}) \circ \mathsf{iszero}?$$

$$\mathsf{List}\ A = \mu(1! + A! \times \mathsf{Id})$$
$$\mathsf{in} = \mathsf{nil}! \triangledown \mathsf{cons}$$
$$\mathsf{out} = (1! + (\mathsf{head} \vartriangle \mathsf{tail})) \circ \mathsf{null}?$$

Hylomorphisms

- Given a functor F, a function $g : F\ B \to B$, and a function $h : A \to F\ A$, a hylomorphism [MFP91] is defined by the following recursive function of type $A \to B$.

$$[\![g, h]\!]_F = \mu(\lambda f . g \circ F f \circ h)$$

- The recursion pattern of the hylomorphism is characterized by the functor F. Function h handles all computations prior to recursion, and g combines the results of the recursive calls in order to determine the output.

$$\begin{array}{lcl} \mathsf{fact} & : & \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{fact} & = & [\![\mathsf{succ} \circ \mathsf{zero!} \triangledown \mathsf{mult}, (\mathsf{id} + \mathsf{succ} \vartriangle \mathsf{id}) \circ \mathsf{out}]\!]_{\mathsf{1!}+\mathsf{Nat!}\times\mathsf{Id}} \end{array}$$

Recursion Patterns as Hylomorphisms

- Folds and unfolds are simple restrictions of hylomorphisms.

$$(\!|g|\!)_F = [\![g, \mathsf{out}_F]\!]_F \quad [\!(h)\!]_F = [\![\mathsf{in}_F, h]\!]_F$$

- An alternative way to understand hylomorphisms results from the following law.

$$[\![g, h]\!]_F = (\!|g|\!)_F \circ [\!(h)\!]_F$$

- Paramorphisms [Mee92]. Given $g : F\ (A \times \mu F) \rightarrow A$ we get $\langle\!|g|\!\rangle_F : \mu F \rightarrow A$ as follows.

$$\langle\!|g|\!\rangle_F = [\![g, F\ (\mathsf{id} \vartriangle \mathsf{id}) \circ \mathsf{out}_F]\!]_{F \circ (\mathsf{Id} \times \mu F!)}$$

- The virtual intermediate data structure stores at each node a copy of the recursive values, that are passed intact to the fold.

Generic Programming with Explicit Functors

- This method is known at least since [MH95]. First, define the explicit fixpoint operator and the in and out isomorphisms.

```
newtype (Functor f) => Mu f = Mu {unMu :: f (Mu f)}
```

- Types are declared as expected.

```
newtype FNat x = FNat {unFNat :: Either () x}

instance Functor FNat
    where fmap f = FNat . (id -|- f) . unFNat

type Nat = Mu FNat
```

Generic Programming with Explicit Functors

- We can have polytypic definitions of the fundamental recursion patterns.

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

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

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

- The definition of the factorial is not so straightforward.

```
fact = hylo g h
    where h = FList . (id -|- succ /\ id) . unFNat . unMu
          g = (succ . (zero!) \/ mult) . unFList
```

Generic Programming with Explicit Functors

- If a definition involves a functor change we need to explicitly define higher-order functors.

```
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 (id /\ id) . unMu)
```

- This approach has some disadvantages:
  - Extensive use of coercing constructors and destructors;
  - Recursion operators can not be used with the standard Haskell types;
  - The `Functor` instances must be defined explicitly.

The PolyP Approach

- Relating types to functors using a multi-parameter type class [JJM97] with a functional dependency [Jon00] (we use a simplified version).

```
class (Functor f) => FunctorOf f d | d -> f
    where inn' :: f d -> d
          out' :: d -> f d
```

- We can now use the standard Haskell types.

```
instance FunctorOf FNat Int
    where inn' = ((0!) \/ succ) . unFNat
          out' = FNat . ((()!) -|- pred) . (iszero?)
```

The PolyP Approach

- We can still use data types declared explicitly as fixed points of functors.

```
instance (Functor f) => FunctorOf f (Mu f)
    where inn' = Mu
          out' = unMu
```

- Polytypism is achieved through a typed representation for functors.

```
newtype Id x          = Id {unId :: x}
newtype Const t x     = Const {unConst :: t}
data (g :+: h) x      = Inl (g x) | Inr (h x)
data (g :*: h) x      = g x :*: h x
newtype (g :@: h) x = Comp {unComp :: g (h x)}
```

The PolyP Approach

```
instance Functor Id
    where fmap f (Id x) = Id (f x)

instance Functor (Const t)
    where fmap f (Const x) = Const x

instance (Functor g, Functor h) => Functor (g :+: h)
    where fmap f (Inl x) = Inl (fmap f x)
          fmap f (Inr x) = Inr (fmap f x)

instance (Functor g, Functor h) => Functor (g :*: h)
    where fmap f (x :*: y) = (fmap f x) :*: (fmap f y)

instance (Functor g, Functor h) => Functor (g :@: h)
    where fmap f (Comp x) = Comp (fmap (fmap f) x)
```

The PolyP Approach

- Now we must use this functor representation in the `FunctorOf` instances.

```
instance FunctorOf (Const () :+: (Const a :*: Id)) [a]
    where inn' (Inl (Const ()))          = []
          inn' (Inr (Const x :*: Id xs)) = x:xs
          out' []     = Inl (Const ())
          out' (x:xs) = Inr (Const x :*: Id xs)
```

- Unfortunately, the price to pay for polytypism is an enormous growth in the use of coercing constructors, rendering point-free programming almost impossible.

- Our solution is to mix PolyP with a limited form of implicit coercion.

Implicit Coercions

- Again, we use a multi-parameter type class to relate elements defined using the functor combinators and the standard (sums of products) Haskell types.

```
class Rep a b | a -> b
    where to   :: a -> b
          from :: b -> a
```

- Similar to Hinze's embedding-projection pairs [CH02].

- The functional dependency restricts the implementation to at most one standard type, but makes the type-checking feasible.

Implicit Coercions

- Since in the context of the instance declaration, we do not have simple type variables, `-fallow-undecidable-instances` is necessary.

```
instance Rep (Id x) x
    where to (Id x) = x
          from x = Id x

instance (Rep (g x) y,Rep (h x) z) => Rep ((g :+: h) x) (Either y z)
    where to (Inl a) = Left (to a)
          to (Inr a) = Right (to a)
          from (Left a) = Inl (from a)
          from (Right a) = Inr (from a)

instance (Rep (g x) y,Rep (h x) z) => Rep ((g :*: h) x) (y,z)
    where to (a :*: b) = (to a, to b)
          from (a, b) = from a :*: from b
...
```

Implicit Coercions

- A possible interaction with a Haskell interpreter could now be

```
> to (Id 'a' :*: Const 'b')
('a','b')
> from ('a','b') :: (Id :*: Const Char) Char
Id 'a' :*: Const 'b'
> from ('a','b') :: (Id :*: Id) Char
Id 'a' :*: Id 'b'
```

- To enable type-checking, polytypic functions are annotated with the functor (indirectly through the corresponding type) to which they should be instantiated (thus, mimicking the theoretical notation).

- This is done through the use of a "dummy" variable and scoped type variables [JS02].

Implicit Coercions

```
pmap :: (FunctorOf f d, Rep (f a) fa, Rep (f b) fb) =>
        d -> (a -> b) -> (fa -> fb)
pmap (_::d) f =
   to . (fmap :: (FunctorOf f d) => (a -> b) -> (f a -> f b)) f . from

hylo :: (FunctorOf f d, Rep (f b) fb, Rep (f a) fa) =>
        d -> (fb -> b) -> (a -> fa) -> a -> b
hylo (_::d) g h = g . pmap (_L::d) (hylo (_L::d) g h) . h

out :: (FunctorOf f d, Rep (f d) fd) => d -> fd
out = to . out'

inn :: (FunctorOf f d, Rep (f d) fd) => fd -> d
inn = inn' . from
```

Using the Library

- Now we can program in a true point-free style.

```
fact :: Int -> Int
fact = hylo (_L :: [Int]) f g
    where g = (id -|- succ /\ id) . out
          f = (1!) \/ mult
```

- Since we can still use explicit fixed points to model data types, we don't have to define neither the type of intermediate data structure, nor the respective instance of the `FunctorOf` class.

```
fib :: Int -> Int
fib = hylo (_L :: Mu (Const () :+: (Id :*: Id))) f g
    where g = ((()!) -|- pred /\ (pred . pred)) . (iszeroorone?)
          f = (1!) \/ plus
          iszeroorone = (<=1)
```

Using the Library

- Folds (and unfolds) are defined as expected.

```
cata :: (FunctorOf f d, Rep (f a) fa, Rep (f d) fd) =>
        d -> (fa -> a) -> d -> a
cata (_::d) g = hylo (_L::d) g out
```

- And for paramorphisms we no longer need to define the higher-order functors, due to the ability to explicitly define the intermediate data type as a fixed point of a functor.

```
para (_::d) g =
    hylo (_L :: FunctorOf f d => Mu (f :@: (Id :*: Const d)))
         g (pmap (_L::d) (id /\ id) . out)
```

Using the Library

```
preorder :: Rose a -> [a]
preorder = cata (_L::Rose a) (cons . (id >< aux))
    where aux = cata (_L::[[a]]) (([]!) \/ cat)

fact :: Int -> Int
fact = para (_L::Int) g
    where g = (1!) \/ (mult . (id >< succ))

plus :: (Int,Int) -> Int
plus = accum (_L::Int) g t
    where t = (fst -|- id >< succ) . distl
          g = (snd \/ fst) . distl
```

Visualization of Intermediate Data Structures

- The visualization uses GHood [Rei01], a graphical animation tool built on top of Hood (*Haskell Object Observation Debugger*) [Gil00].

- Hood introduces a combinator that returns the second argument, and as a side-effect stores it into some persistent structure for later rendering.

```
observe :: (Observable a) => String -> a -> a
```

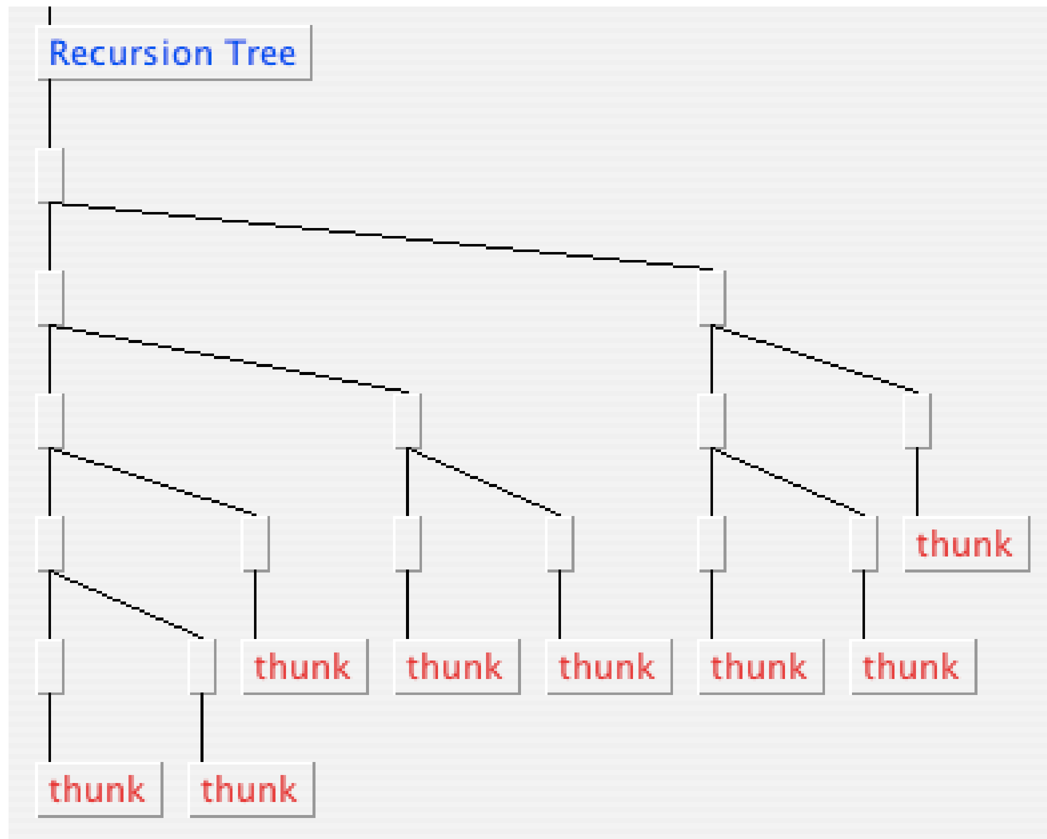
- Likewise to `Functor`, we defined a generic instance for the `Observable` class.

- The hylo-split law allows us to expose the intermediate virtual data structure of the hylomorphisms.

```
hyloO (f::d) g h = cata (f::d) g . (observe "Tree") . ana (f::d) h
```

Visualization of Intermediate Data Structures

- The intermediate binary shape tree of `fib 5` is



Derivation of Hylomorphisms

- DrHylo - a tool that derives (pointwise) hylomorphisms from explicit recursive definitions. Essentially, it implements the algorithm of [HIT96].

```
fact 0 = 1
fact n = n * fact (n-1)
```

```
fact = hylo (_L :: Mu (Const () :+: (Const v2 :*: Id))) g h
  where g (Left (())) = 1
        g (Right (((n), v1))) = n * v1
        h 0 = Left (())
        h n = Right (((n), (n - 1)))
```

- Together with the previous tool it can be used to visualize the recursion tree of a recursive definition.

Derivation of Point-free Definitions

- We also have a prototype of a tool that derives point-free definitions from pointwise ones. It still has many limitations:
  - It only handles user defined data-types;
  - The pattern matching clauses must be exhaustive and disjunct;
  - ...
- It will be incorporated in DrHylo in order to derive point-free hylomorphisms.
- It's being implemented in a rather heuristic way, but we are also studying the similar transformation of the Categorical Abstract Machine [Cur93] in order to achieve a more formally justified implementation.

Conclusions

- Pointless is a Haskell library that can be used to program with recursion patterns in a point-free style. It is also supported by a growing set of useful tools.
- Built on top of existing techniques ideas from the generic programming community.
- It uses some extensions to the Haskell type system.
- The type annotations introduced in order to make inference possible were precisely the same that were already used in the theoretical notation, namely the functors that characterize recursion.
- Unfortunately, the error messages obtained when programming with the library are of limited help for the programmer.

Future Work

- We are starting to implement an equational reasoning tool for point-free expressions.

- Based on rewrite systems for lambda calculus with categorical sums and pairs [dCK93, Dou93].

- Together with DrHylo it will be used as the basis for a program transformation tool for Haskell programs (similar to MAG [dMS03]).

- For more information visit the PURe project website.

`http://www.di.uminho.pt/pure`

References

[CH02] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the ACM SIGPLAN 2002 Haskell Workshop*, pages 90–104, 2002.

[Cur93] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Birkhäuser, 2nd edition, 1993.

[dCK93] Roberto di Cosmo and Delia Kesner. A confluent reduction for the extensional typed lambda-calculus with pairs, sums, recursion and terminal object. In Andrzej Lingas, editor, *Proceedings of the International Conference on Automata, Languages and Programming*, volume 700 of *LNCS*, pages 645–656. Springer-Verlag, 1993.

[dMS03] Oege de Moor and Ganesh Sittampalam. Mechanising fusion. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, chapter 5, pages 79–104. Palgrave Macmillan, 2003.

[Dou93] Daniel Dougherty. Some lambda calculi with categorical sums and products. In Claude Kirchner, editor, *Proceedings of the 5th International Conference on Rewriting Techniques and Applications*, volume 690 of *LNCS*, pages 137–151. Springer-Verlag, 1993.

[FM91] Maarten Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, January 1991.

[Gil00] Andy Gill. Debugging Haskell by observing intermediate data structures. In G. Hutton, editor, *Proceedings of the 4th ACM SIGPLAN Haskell Workshop*, 2000.

[HIT96] 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.

[JJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: exploring the design space. In *Proceedings of the Haskell Workshop*, 1997.

[Jon00] Mark Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

[JS02] Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. To be submitted to The Journal of Functional Programming, March 2002.

[Mee92] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[MFP91] 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.

[MH95] 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.

[NJ03] Ulf Norell and Patrik Jansson. Polytypic programming in haskell. In *Draft proceedings of the 15th International Workshop on the Implementation of Functional Languages (IFL'03)*, 2003.

[Rei01] Claus Reinke. GHood - graphical visualisation and animation of Haskell object observations. In Ralf Hinze, editor, *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, volume 59 of *ENTCS*. Elsevier, 2001.

[Rey77] J.C. Reynolds. Semantics of the domain of flow diagrams. *Journal of the ACM*, 24(3):484–503, July 1977.