

HFusion

A Fusion Tool for Haskell programs

ALBERTO PARDO

Instituto de Computación
Universidad de la República
Montevideo - Uruguay

Modularity in FP

- In functional programming one often uses a compositional style of programming.
- Programs are constructed as the composition of simple and easy to write functions.
- Programs so defined are more modular and easier to understand.
- General purpose operators (like fold, map, filter, zip, etc.) play an important role in this design.

Example: *trail*

Function *trail* returns the last n lines of a text.

$$\textit{trail } n = \textit{unlines} \circ \textit{reverse} \circ \textit{take } n \circ \textit{reverse} \circ \textit{lines}$$

Example: *count*

count :: *Word* → *Text* → *Integer*
count *w* = *length* ∘ *filter* (== *w*) ∘ *words*

words :: *Text* → [*Words*]
words *t* = **case** *dropWhile isSpace t* **of**
 "" → []
 t' → **let** (*w*, *t''*) = *break isSpace t'*
 in *w* : *words t''*

filter :: (*a* → *Bool*) → [*a*] → [*a*]
filter *p* [] = []
filter *p* (*a* : *as*) = **if** *p a* **then** *a* : *filter p as*
 else *filter p as*

Drawbacks of modularity

- Modular functions are not necessarily efficient.
- Each functional composition implies information passing through an intermediate data structure.

$$A \xrightarrow{f} T \xrightarrow{g} B$$

- Nodes of the intermediate data structure are generated/allocated by f and subsequently consumed/deallocated by g .
- This may lead to repeated invocations to the garbage collector.

Deforestation

- **Deforestation** is a program transformation technique.
- Provided certain conditions hold, deforestation permits the derivation of equivalent functions that do not build intermediate data structures.

$$A \xrightarrow{f} T \xrightarrow{g} B \quad \rightsquigarrow \quad A \xrightarrow{h} B$$

- Our approach to deforestation is based on recursion program schemes.
- Associated with the recursion schemes there are algebraic laws –called *fusion laws*– which represent a form of deforestation.

Program Fusion

count w = length \circ *filter* (*== w*) \circ *words*



count w t = case dropWhile isSpace t of
 "" \rightarrow 0
 t' \rightarrow **let** (*w'*, *t''*) = *break isSpace t'*
 in if *w' == w*
 then 1 + *count w t''*
 else *count w t''*

How fusion proceeds

$lenfil\ p = length \circ filter\ p$

$length\ [] = 0$

$length\ (x : xs) = h\ x\ (length\ xs)$

where

$h\ x\ n = 1 + n$

$filter\ p\ [] = []$

$filter\ p\ (a : as) = \mathbf{if}\ p\ a\ \mathbf{then}\ a : filter\ p\ as$
 $\mathbf{else}\ filter\ p\ as$

How fusion proceeds (cont.)

In the body of the first function,

- replace every occurrence of the constructors used to build the intermediate data structure by the corresponding operations in the second function used to calculate the final result.
- replace recursive calls by calls to the new function

How fusion proceeds (cont.)

$length [] = 0$
 $length (x : xs) = h\ x\ (length\ xs)$
where
 $h\ x\ n = 1 + n$

$filter\ p\ [] = []$
 $filter\ p\ (a : as) = \mathbf{if}\ p\ a\ \mathbf{then}\ a : filter\ p\ as$
else $filter\ p\ as$

The result:

$lenfil\ p\ [] = 0$
 $lenfil\ p\ (a : as) = \mathbf{if}\ p\ a\ \mathbf{then}\ h\ a\ (lenfil\ p\ as)$
else $lenfil\ p\ as$
where
 $h\ x\ n = 1 + n$

Recursion schemes on data types

- They capture general patterns of computation commonly used in practice.
- The schemes and their fusion laws can be defined *generically* for a family of data types.

Standard program schemes

- Fold (structural recursion)
- Unfold (structural co-recursion)
- Hylomorphism (general recursion)

Capturing the structure of functions

$fact :: Int \rightarrow Int$
 $fact\ n \mid n < 1 = 1$
 $\mid otherwise = n * fact\ (n - 1)$

Capturing the structure of functions (2)

data $a + b = \text{Left } a \mid \text{Right } b$

$\psi :: \text{Int} \rightarrow () + \text{Int} \times \text{Int}$

$\psi \ n \mid n < 1 = \text{Left } ()$

$\mid \text{otherwise} = \text{Right } (n, n - 1)$

$\text{fmap } f (\text{Left } ()) = \text{Left } ()$

$\text{fmap } f (\text{Right } (m, n)) = \text{Right } (m, f \ n)$

$\varphi :: () + \text{Int} \times \text{Int} \rightarrow \text{Int}$

$\varphi (\text{Left } ()) = 1$

$\varphi (\text{Right } (m, n)) = m * n$

Capturing the structure of functions (3)

$$fact = \varphi \circ fmap\ fact \circ \psi$$

$$\begin{array}{ccc} Int & \xrightarrow{fact} & Int \\ \psi \downarrow & & \uparrow \varphi \\ () + Int \times Int & \xrightarrow{fmap\ fact} & () + Int \times Int \end{array}$$

Capturing the structure of functions (4)

Let us define,

$$F\ a = () + Int \times a$$

Therefore,

$$\begin{array}{ccc} Int & \xrightarrow{fact} & Int \\ \psi \downarrow & & \uparrow \varphi \\ F\ Int & \xrightarrow{fmap\ fact} & F\ Int \end{array}$$

Functor

A **functor** $(F, fmap)$ consists of two components:

- a type constructor F , and
- a mapping function $fmap :: (a \rightarrow b) \rightarrow (F\ a \rightarrow F\ b)$, which preserves identities and compositions:

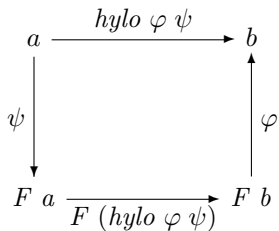
$$fmap\ id = id$$

$$fmap\ (f \circ g) = fmap\ f \circ fmap\ g$$

\leadsto it is usual to denote both components by F .

Hylomorphism

$$\begin{aligned} \text{hyla} &:: (F\ b \rightarrow b) \rightarrow (a \rightarrow F\ a) \rightarrow a \rightarrow b \\ \text{hyla } \varphi\ \psi &= \varphi \circ F\ (\text{hyla } \varphi\ \psi) \circ \psi \end{aligned}$$



A commutative square diagram illustrating the hylomorphism. The top-left node is a , the top-right node is b , the bottom-left node is $F\ a$, and the bottom-right node is $F\ b$. The top horizontal arrow is labeled $\text{hyla } \varphi\ \psi$. The bottom horizontal arrow is labeled $F\ (\text{hyla } \varphi\ \psi)$. The left vertical arrow is labeled ψ and points downwards. The right vertical arrow is labeled φ and points upwards.

$\rightsquigarrow \varphi$ is called an *algebra*

$\rightsquigarrow \psi$ is called a *coalgebra*.

Data types

Functors describe the top level structure of data types.

For each data type declaration

$$\mathbf{data} \tau = C_1 \tau_{1,1} \cdots \tau_{1,k_1} \mid \cdots \mid C_n \tau_{n,1} \cdots \tau_{n,k_n}$$

a functor F can be derived:

- constructor domains are packed in tuples;
- constant constructors are represented by the empty tuple $()$;
- alternatives are regarded as sums (replace $|$ by $+$);
- occurrences of τ are replaced by a type variable x in every $\tau_{i,j}$.

Examples: Lists

$List\ a = Nil \mid Cons\ a\ (List\ a)$



$L_a\ x = () + a \times x$

$L_a :: (x \rightarrow y) \rightarrow (L_a\ x \rightarrow L_a\ y)$

$L_a\ f\ (Left\ ()) = Left\ ()$

$L_a\ f\ (Right\ (a, x)) = Right\ (a, f\ x)$

Example: Leaf-labelled binary trees

data *Btree* a = *Leaf* a | *Join* (*Btree* a) (*Btree* a)



$$B_a x = a + b \times x$$

$$B_a :: (x \rightarrow y) \rightarrow (B_a x \rightarrow B_a y)$$

$$B_a f (\text{Left } a) = \text{Left } a$$

$$B_a f (\text{Right } (x, x')) = \text{Right } (f x, f x')$$

Example: Internally-labelled binary trees

data *Tree a = Empty | Node (Tree a) a (Tree a)*



$$T_a x = () + x \times a \times x$$

$$T_a :: (x \rightarrow y) \rightarrow (T_a x \rightarrow T_a y)$$

$$T_a f (\text{Left } ()) = \text{Left } ()$$

$$T_a f (\text{Right } (x, a, x')) = \text{Right } (f x, a, f x')$$

Constructors / Destructors

For every data type τ with functor F , there exists an isomorphism

$$F \mu F \begin{array}{c} \xrightarrow{\text{in}_F} \\ \xleftarrow{\text{out}_F} \end{array} \mu F$$

where

- μF denotes the data type
- in_F packs the constructors
- out_F packs the destructors

Example: Leaf-labelled binary trees

data *Btree* a = *Leaf* a | *Join* (*Btree* a) (*Btree* a)

$B_a x = a + x \times x$

$out_{B_a} :: B_a (Btree\ a) \rightarrow Btree\ a$

$out_{B_a} (Leaf\ a) = Leaf\ a$

$out_{B_a} (Right\ (t, t')) = Join\ t\ t'$

$outBa :: Btree\ a \rightarrow B_a (Btree\ a)$

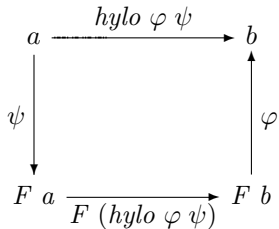
$outBa (Leaf\ a) = Left\ a$

$outBa (Join\ t\ t') = Right\ (t, t')$

Hylomorphism

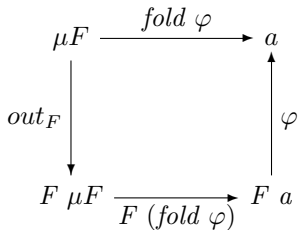
$hylo :: (F\ b \rightarrow b) \rightarrow (a \rightarrow F\ a) \rightarrow a \rightarrow b$

$hylo\ \varphi\ \psi = \varphi \circ F\ (hylo\ \varphi\ \psi) \circ \psi$



Fold

$fold :: (F\ a \rightarrow a) \rightarrow \mu F \rightarrow a$
 $fold\ \varphi = \varphi \circ F\ (fold\ \varphi) \circ out_F$



Fold: Lists

$$\text{fold}_L :: (b, a \rightarrow b \rightarrow b) \rightarrow \text{List } a \rightarrow b$$
$$\text{fold}_L (b, h) \text{ Nil} = b$$
$$\text{fold}_L (b, h) (\text{Cons } a \text{ as}) = h a (\text{fold}_L (b, h) \text{ as})$$

Example:

$$\text{prod} :: \text{List } \text{Int} \rightarrow \text{Int}$$
$$\text{prod Nil} = 1$$
$$\text{prod} (\text{Cons } n \text{ ns}) = n * \text{prod ns}$$

As a fold,

$$\text{prod} = \text{fold}_L (1, (*))$$

Unfold

$$\begin{aligned} \text{unfold} &:: (a \rightarrow F\ a) \rightarrow a \rightarrow \mu F \\ \text{unfold } \psi &= \text{in}_F \circ F\ (\text{unfold } \psi) \circ \psi \end{aligned}$$

$$\begin{array}{ccc} a & \xrightarrow{\text{unfold } \psi} & \mu F \\ \psi \downarrow & & \uparrow \text{in}_F \\ F\ a & \xrightarrow{F\ (\text{unfold } \psi)} & F\ \mu F \end{array}$$

Unfold: Lists

$$\begin{aligned} \text{unfold}_L &:: (b \rightarrow L_a b) \rightarrow b \rightarrow \text{List } a \\ \text{unfold}_L \psi b &= \mathbf{case} (\psi b) \mathbf{of} \\ &\quad \text{Left } () \rightarrow \text{Nil} \\ &\quad \text{Right } (a, b') \rightarrow \text{Cons } a (\text{unfold}_L \psi b') \end{aligned}$$

Example:

$$\begin{aligned} \text{upto} &:: \text{Int} \rightarrow \text{Int} \\ \text{upto } n \mid n < 1 &= \text{Nil} \\ &\mid \text{otherwise} = \text{Cons } n (\text{upto } (n - 1)) \end{aligned}$$

As an unfold,

$$\begin{aligned} \text{upto} &= \text{unfold}_L \psi \\ &\mathbf{where} \\ &\quad \psi n \mid n < 1 = \text{Left } () \\ &\quad \mid \text{otherwise} = \text{Right } (n, n - 1) \end{aligned}$$

Factorisation

$$\mathit{hylo} \varphi \psi = \mathit{fold} \varphi \circ \mathit{unfold} \psi$$

Factorisation: factorial

$$fact = prod \circ upto$$

$$prod :: List Int \rightarrow Int$$

$$prod Nil = 1$$

$$prod (Cons n ns) = n * prod ns$$

$$upto :: Int \rightarrow Int$$

$$upto n \mid n < 1 = Nil$$

$$\mid otherwise = Cons n (upto (n - 1))$$

Applying factorisation,

$$fact :: Int \rightarrow Int$$

$$fact n \mid n < 1 = 1$$

$$\mid otherwise = n * fact (n - 1)$$

Factorisation: quicksort

$qsort :: Ord\ a \Rightarrow [a] \rightarrow [a]$
 $qsort = inorder \circ mkTree$

$inorder :: Tree\ a \rightarrow List\ a$
 $inorder\ Empty = Nil$
 $inorder\ (Node\ t\ a\ t') = inorder\ t ++ [a] ++ inorder\ t'$

$mkTree :: Ord\ a \Rightarrow [a] \rightarrow Tree\ a$
 $mkTree\ [] = Empty$
 $mkTree\ (a : as) = Node\ (mkTree\ [x \mid x \leftarrow as; x \leq a])$
 $\quad\quad\quad a$
 $\quad\quad\quad (mkTree\ [x \mid x \leftarrow as; x > a])$

Quicksort

$qsort :: Ord\ a \Rightarrow [a] \rightarrow [a]$

$qsort [] = []$

$qsort (a : as) = qsort [x \mid x \leftarrow as; x \leq a]$

++ [a] ++

$qsort [x \mid x \leftarrow as; x > a]$

Fusion laws

Factorisation

$$\mathit{hylo} \ \varphi \ \psi = \mathit{hylo} \ \varphi \ \mathit{out}_F \circ \mathit{hylo} \ \mathit{in}_F \ \psi$$

Hylo-Fold Fusion

$$\tau :: \forall a . (F \ a \rightarrow a) \rightarrow (G \ a \rightarrow a)$$

\Rightarrow

$$\mathit{fold} \ \varphi \circ \mathit{hylo} \ (\tau \ \mathit{in}_F) \ \psi = \mathit{hylo} \ (\tau \ \varphi) \ \psi$$

Unfold-Hylo Fusion

$$\sigma :: (a \rightarrow F \ a) \rightarrow (a \rightarrow G \ a)$$

\Rightarrow

$$\mathit{hylo} \ \varphi \ (\sigma \ \mathit{out}_F) \circ \mathit{unfold} \ \psi = \mathit{hylo} \ \varphi \ (\sigma \ \psi)$$

Hylo-Fold Fusion

data *Maybe* a = *Nothing* | *Just* a

mapcoll :: (a → b) → List (Maybe a) → List b
mapcoll = *map f* ∘ *collect*

map f Nil = *Nil*
map f (Cons a as) = *Cons (f a) (map f as)*

collect :: List (Maybe Int) → List Int
collect Nil = *Nil*
collect (Cons m ms) = **case** m **of**
 Nothing → *collect ms*
 Just a → *Cons a (collect ms)*

Hylo-Fold Fusion

$$\begin{aligned} \tau &:: (b, a \rightarrow b \rightarrow b) \rightarrow (b, \text{Maybe } a \rightarrow b \rightarrow b) \\ \tau (h_1, h_2) &= (h_1, \\ &\quad \lambda m b \rightarrow \mathbf{case\ } m \mathbf{ of} \\ &\quad \quad \text{Nothing} \rightarrow b \\ &\quad \quad \text{Just } a \rightarrow h_2\ a\ b) \end{aligned}$$

Applying hylo-fold fusion,

$$\begin{aligned} \text{mapcoll} &:: (a \rightarrow b) \rightarrow \text{List } (\text{Maybe } a) \rightarrow \text{List } b \\ \text{mapcoll } f \text{ Nil} &= \text{Nil} \\ \text{mapcoll } f (\text{Cons } m \text{ ms}) &= \mathbf{case\ } m \mathbf{ of} \\ &\quad \text{Nothing} \rightarrow \text{mapcoll } f \text{ ms} \\ &\quad \text{Just } a \rightarrow \text{Cons } (f\ a) (\text{mapcoll } f \text{ ms}) \end{aligned}$$

HFusion

- HFusion is an extension of the HYLO system implemented at the University of Tokyo (97-98) and at MIT (2000) in the context of pH (parallel Haskell)
- The tool is implemented in Haskell.
- It can be used in three different modalities:
 - Command line
 - Web interface
 - Inside HaRe (Haskell Refactorer)