HFusion A Fusion Tool for Haskell programs

Alberto Pardo

Instituto de Computación Universidad de la República Montevideo - Uruguay

http://www.fing.edu.uy/~pardo

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.

 $trail\ n = unlines \circ reverse \circ take\ n \circ reverse \circ lines$

Example: count

```
count :: Word \rightarrow Text \rightarrow Integer
count \ w = length \circ filter \ (== w) \circ words
words :: Text \rightarrow [Words]
words t = \mathbf{case} \ drop While \ isSpace \ t \ \mathbf{of}
                    "" → []
                    t' \rightarrow \mathbf{let} \ (w, t'') = break \ isSpace \ t'
                           in w: words t''
filter :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [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.

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

- ullet Nodes of the intermediate data structure are generated/allocated by f and subsequently consumed 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

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



How fusion proceeds

How fusion proceeds (cont.)

In the body of the first function,

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

How fusion proceeds (cont.)

```
length \ [\ ] = 0 length \ (x:xs) = h \ x \ (length \ xs) \mathbf{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
```

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

- 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

```
 \begin{aligned} & \textit{fact} :: \textit{Int} \rightarrow \textit{Int} \\ & \textit{fact} \; n \mid n < 1 = 1 \\ & \mid \textit{otherwise} = n * \textit{fact} \; (n-1) \end{aligned}
```

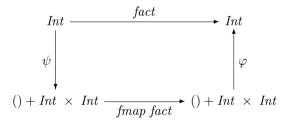
Capturing the structure of functions (2)

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

```
\psi :: Int \rightarrow () + Int \times Int
\psi n \mid n < 1 = Left ()
       | otherwise = Right (n, n-1)
fmap \ f \ (Left \ ()) = Left \ ()
fmap \ f \ (Right \ (m, n)) = Right \ (m, f \ n)
\varphi :: () + Int \times Int \rightarrow Int
\varphi \left( Left \left( \right) \right) = 1
\varphi\left(Right\left(m,n\right)\right) = m*n
```

Capturing the structure of functions (3)

$$\mathit{fact} = \varphi \circ \mathit{fmap} \; \mathit{fact} \circ \psi$$

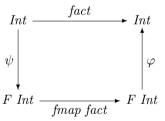


Capturing the structure of functions (4)

Let us define.

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

Therefore,



Functor

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

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

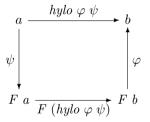
$$fmap\ id\ =\ id$$

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

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

Hylomorphism

$$hylo :: (F \ b \to b) \to (a \to F \ a) \to a \to b$$
$$hylo \ \varphi \ \psi = \varphi \circ F \ (hylo \ \varphi \ \psi) \circ \psi$$



 $\rightarrow \varphi$ is called an *algebra* $\rightarrow \psi$ is called a *coalgebra*.

Data types

Functors describe the top level structure of data types.

For each data type declaration

data
$$T = C_1 \ \tau_{1,1} \cdots \tau_{1,k_1} \ | \cdots | \ C_n \ \tau_{n,1} \cdots \tau_{n,k_n}$$

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 T are replaced by a type variable x in every $\tau_{i,j}$.

Examples: Lists

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



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

$$\begin{array}{l} L_a::(x\to y)\to (L_a\;x\to L_a\;y)\\ L_a\;f\;(Left\;())=Left\;()\\ L_a\;f\;(Right\;(a,x))=Right\;(a,f\;x) \end{array}$$

Example: Leaf-labelled binary trees

data $Btree\ a = Leaf\ a\ |\ Join\ (Btree\ a)\ (Btree\ a)$



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

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

 $B_a f (Left \ a) = Left \ a$
 $B_a f (Right \ (x, x')) = Right \ (f \ x, f \ x')$

Example: Internally-labelled binary trees

data $Tree\ a = Empty \mid Node\ (Tree\ a)\ a\ (Tree\ a)$



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

$$\begin{split} T_a &:: (x \to y) \to (T_a \ x \to T_a \ y) \\ T_a \ f \ (Left \ ()) &= Left \ () \\ T_a \ f \ (Right \ (x,a,x')) &= Right \ (f \ x,a,f \ x') \end{split}$$

Constructors / Destructors

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

$$F\mu F \xrightarrow{in_F} \mu F$$

where

- ullet μF denotes the data type
- \bullet in_F packs the constructors
- ullet out_F packs the destructors

Example: Leaf-labelled binary trees

data $Btree\ a = Leaf\ a \mid Join\ (Btree\ a)\ (Btree\ a)$

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

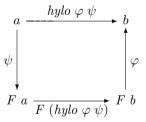
$$in_{B_a} :: B_a \ (Btree \ a) \to Btree \ a$$

 $in_{B_a} \ (Left \ a) = Leaf \ a$
 $in_{B_a} \ (Right \ (t,t')) = Join \ t \ t'$

 $out_{B_a} :: Btree \ a \to B_a \ (Btree \ a)$ $out_{B_a} \ (Leaf \ a) = Left \ a$ $out_{B_a} \ (Join \ t \ t') = Right \ (t, t')$

Hylomorphism

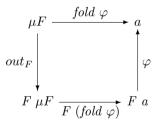
$$\begin{array}{l} \textit{hylo} :: (F \ b \rightarrow b) \rightarrow (a \rightarrow F \ a) \rightarrow a \rightarrow b \\ \textit{hylo} \ \varphi \ \psi = \varphi \circ F \ (\textit{hylo} \ \varphi \ \psi) \circ \psi \end{array}$$



Fold

$$fold :: (F \ a \to a) \to \mu F \to a$$

 $fold \ \varphi = \varphi \circ F \ (fold \ \varphi) \circ out_F$



Fold: Lists

```
 \begin{array}{l} fold_L :: (b,a \rightarrow b \rightarrow b) \rightarrow List \ a \rightarrow b \\ fold_L \ (b,h) \ Nil = b \\ fold_L \ (b,h) \ (Cons \ a \ as) = h \ a \ (fold_L \ (b,h) \ as) \end{array}
```

Example:

```
prod :: List \ Int \rightarrow Int

prod \ Nil = 1

prod \ (Cons \ n \ ns) = n * prod \ ns
```

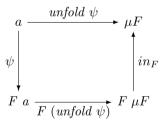
As a fold,

$$prod = fold_L(1, (*))$$

Unfold

$$unfold :: (a \to F \ a) \to a \to \mu F$$

 $unfold \ \psi = in_F \circ F \ (unfold \ \psi) \circ \psi$



Unfold: Lists

```
\begin{array}{c} unfold_L :: (b \to L_a \ b) \to b \to List \ a \\ unfold_L \ \psi \ b = \mathbf{case} \ (\psi \ b) \ \mathbf{of} \\ Left \ () \to Nil \\ Right \ (a,b') \to Cons \ a \ (unfold_L \ \psi \ b') \end{array}
```

Example:

```
 \begin{array}{l} \textit{upto} :: \textit{Int} \rightarrow \textit{Int} \\ \textit{upto} \; n \mid n < 1 = \textit{Nil} \\ \mid \textit{otherwise} = \textit{Cons} \; n \; (\textit{upto} \; (n-1)) \end{array}
```

As an unfold,

```
\begin{split} upto &= unfold_L \; \psi \\ \mathbf{where} \\ &\quad \psi \; n \mid n < 1 = Left \; () \\ &\quad \mid otherwise = Right \; (n,n-1) \end{split}
```

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 \ | \ n < 1 = Nil

| \ otherwise = Cons \ n \ (upto \ (n-1))
```

Applying factorisation,

```
 \begin{aligned} & \textit{fact} :: \textit{Int} \rightarrow \textit{Int} \\ & \textit{fact} \; n \mid n < 1 = 1 \\ & \mid \textit{otherwise} = n * \textit{fact} \; (n-1) \end{aligned}
```

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 \leqslant a])
                                (mkTree [x \mid x \leftarrow as; x > a])
```

Quicksort

```
\begin{array}{l} qsort :: Ord \ a \Rightarrow [\, a\,] \rightarrow [\, a\,] \\ qsort \ [\,] = [\,] \\ qsort \ (a : as) = qsort \ [\, x \mid x \leftarrow as; x \leqslant a\,] \\ ++ [\, a\,] \ ++ \\ qsort \ [\, x \mid x \leftarrow as; x > a\,] \end{array}
```

Fusion laws

Factorisation

$$hylo \varphi \psi = hylo \varphi out_F \circ hylo in_F \psi$$

Hylo-Fold Fusion

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

$$fold \ \varphi \circ hulo \ (\tau \ in_F) \ \psi = hulo \ (\tau \ \varphi) \ \psi$$

Unfold-Hylo Fusion

 \Rightarrow

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

$$hylo \ \varphi \ (\sigma \ out_F) \circ unfold \ \psi = hylo \ \varphi \ (\sigma \ \psi)$$

Hylo-Fold Fusion

```
data Maybe \ a = Nothing \mid Just \ a
mapcoll :: (a \rightarrow b) \rightarrow List \ (Maybe \ a) \rightarrow List \ b
mapcoll = map \ f \circ collect
map \ f \ Nil = Nil
map\ f\ (Cons\ a\ as) = Cons\ (f\ a)\ (map\ f\ as)
collect :: List (Maybe Int) \rightarrow List Int
collect\ Nil = Nil
collect (Cons \ m \ ms) = \mathbf{case} \ m \ \mathbf{of}
                                    Nothing \rightarrow collect \ ms
```

HFUSION 34

 $Just \ a \rightarrow Cons \ a \ (collect \ ms)$

Hylo-Fold Fusion

```
\tau :: (b, a \to b \to b) \to (b, Maybe \ a \to b \to b)
\tau \ (h_1, h_2) = (h_1, \\ \lambda m \ b \to \mathbf{case} \ m \ \mathbf{of}
Nothing \to b
Just \ a \to h_2 \ a \ b)
```

Applying hylo-fold fusion,

```
mapcoll :: (a \rightarrow b) \rightarrow List \ (Maybe \ a) \rightarrow List \ b
mapcoll \ f \ Nil = Nil
mapcoll \ f \ (Cons \ m \ ms) = \mathbf{case} \ m \ \mathbf{of}
Nothing \rightarrow mapcoll \ f \ ms
Just \ a \rightarrow Cons \ (f \ a) \ (mapcoll \ f \ ms)
```

HFusion

- HFusion is an extension of the HYLO system:
 - University of Tokyo, 1997-98
 - MIT, 2000, in the context of pH (parallel Haskell)
- HFusion is implemented in Haskell.
- It can be used in three different modalities:
 - Command line
 - Web interface
 - Inside HaRe (Haskell Refactorer)

Web access:

http://www.fing.edu.uy/inco/proyectos/fusion/tool/