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.

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

Example: *count*

$$count :: Word \to Text \to Integer$$
$$count\ w = length \circ filter\ (== w) \circ words$$

$$words :: Text \to [Words]$$

$$\begin{aligned} words\ t = \mathbf{case}\ & dropWhile\ isSpace\ t\ \mathbf{of} \\ & \texttt{""} \to [\,] \\ & t' \to \mathbf{let}\ (w, t'') = break\ isSpace\ t' \\ & \qquad \mathbf{in}\ w : words\ t'' \end{aligned}$$

$$filter :: (a \to Bool) \to [a] \to [a]$$
$$filter\ p\ [\,] = [\,]$$
$$\begin{aligned} filter\ p\ (a : as) = \mathbf{if}\ p\ a\ & \mathbf{then}\ a : filter\ p\ as \\ & \mathbf{else}\ filter\ p\ as \end{aligned}$$

Drawbacks of modularity

- Modular functions are not necessarily efficient.

- Each functional composition implies information passing through an intermediate data structure.

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

- 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.

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

- 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

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

$$\Downarrow$$

$$
\begin{aligned}
count\ w\ t = \mathbf{case}\ & dropWhile\ isSpace\ t\ \mathbf{of} \\
& \texttt{""} \rightarrow 0 \\
& t' \rightarrow \mathbf{let}\ (w', t'') = break\ isSpace\ t' \\
& \qquad \mathbf{in\ if}\ w' == w \\
& \qquad\qquad \mathbf{then}\ 1 + count\ w\ t'' \\
& \qquad\qquad \mathbf{else}\ count\ w\ t''
\end{aligned}
$$

How fusion proceeds

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

$$\begin{aligned}
&length\ [\,] = 0\\
&length\ (x : xs) = h\ x\ (length\ xs)\\
&\qquad\qquad\qquad\quad \mathbf{where}\\
&\qquad\qquad\qquad\qquad h\ x\ n = 1 + n
\end{aligned}$$

$$\begin{aligned}
&filter\ p\ [\,] = [\,]\\
&filter\ p\ (a : as) = \mathbf{if}\ p\ a\ \mathbf{then}\ a : filter\ p\ as\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ filter\ p\ as
\end{aligned}$$

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.)

$$\begin{array}{l} length\ [\,] = 0 \\ length\ (x : xs) = h\ x\ (length\ xs) \\ \qquad\qquad\qquad\quad \mathbf{where} \\ \qquad\qquad\qquad\qquad h\ x\ n = 1 + n \end{array}$$

$$\begin{array}{l} filter\ p\ [\,] = [\,] \\ filter\ p\ (a : as) = \mathbf{if}\ p\ a\ \mathbf{then}\ a : filter\ p\ as \\ \qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ filter\ p\ as \end{array}$$

The result:

$$\begin{array}{l} lenfil\ p\ [\,] = 0 \\ lenfil\ p\ (a : as) = \mathbf{if}\ p\ a\ \mathbf{then}\ h\ a\ (lenfil\ p\ as) \\ \qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ lenfil\ p\ as \\ \qquad\qquad\qquad\quad \mathbf{where} \\ \qquad\qquad\qquad\qquad h\ x\ n = 1 + n \end{array}$$

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

$$
\begin{aligned}
&fact :: Int \to Int\\
&fact\ n \mid n < 1 = 1\\
&\qquad\ \mid otherwise = n * fact\ (n - 1)
\end{aligned}
$$

Capturing the structure of functions (2)

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

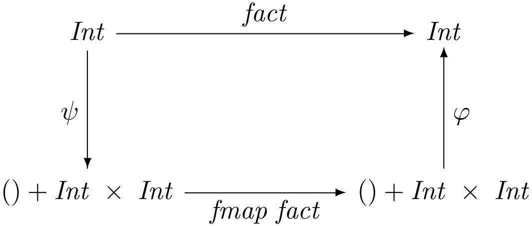
$$\begin{aligned}
&\psi :: \mathit{Int} \to () + \mathit{Int}\ \times\ \mathit{Int} \\
&\psi\ n \mid n < 1 = \mathit{Left}\ () \\
&\quad\ \ \mid \mathit{otherwise} = \mathit{Right}\ (n, n-1)
\end{aligned}$$

$$\begin{aligned}
&\mathit{fmap}\ f\ (\mathit{Left}\ ()) = \mathit{Left}\ () \\
&\mathit{fmap}\ f\ (\mathit{Right}\ (m, n)) = \mathit{Right}\ (m, f\ n)
\end{aligned}$$

$$\begin{aligned}
&\varphi :: () + \mathit{Int}\ \times\ \mathit{Int} \to \mathit{Int} \\
&\varphi\ (\mathit{Left}\ ()) = 1 \\
&\varphi\ (\mathit{Right}\ (m, n)) = m * n
\end{aligned}$$

Capturing the structure of functions (3)

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

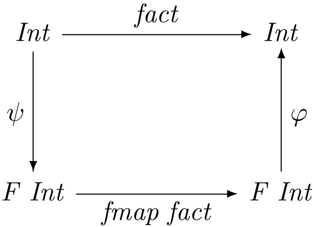


Capturing the structure of functions (4)

Let us define,

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

Therefore,



Functor

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

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

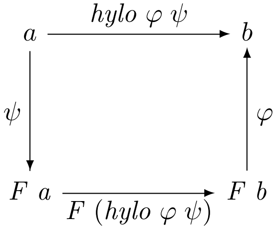
$$fmap\ id = id$$

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

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

Hylomorphism

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



- $\rightsquigarrow$ φ is called an *algebra*
- $\rightsquigarrow$ ψ is called a *coalgebra*.

Data types

Functors describe the top level structure of data types.

For each data type declaration

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

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

Examples: Lists

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

$$\Downarrow$$

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

$$L_a :: (x \to y) \to (L_a\ x \to L_a\ y)$$
$$L_a\ f\ (Left\ ()) = Left\ ()$$
$$L_a\ f\ (Right\ (a, x)) = Right\ (a, f\ x)$$

Example: Leaf-labelled binary trees

$$\mathbf{data}\ Btree\ a = Leaf\ \ a\ \mid\ Join\ (Btree\ a)\ (Btree\ a)$$

$$\Downarrow$$

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

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

Example: Internally-labelled binary trees

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Empty}\ \mid\ \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)$$

$$\Downarrow$$

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

$$\begin{array}{l} T_a :: (x \rightarrow y) \rightarrow (T_a\ x \rightarrow T_a\ y) \\ T_a\ f\ (\mathit{Left}\ ()) = \mathit{Left}\ () \\ T_a\ f\ (\mathit{Right}\ (x, a, x')) = \mathit{Right}\ (f\ x, a, f\ x') \end{array}$$

Constructors / Destructors

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

$$F\mu F \underset{out_F}{\overset{in_F}{\rightleftarrows}} \mu F$$

where

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

Example: Leaf-labelled binary trees

$$\mathbf{data}\ \mathit{Btree}\ a = \mathit{Leaf}\ a \mid \mathit{Join}\ (\mathit{Btree}\ a)\ (\mathit{Btree}\ a)$$

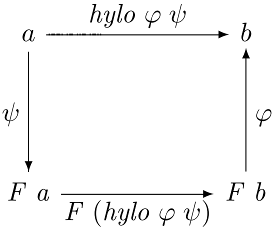
$$B_a\ x = a + x\ \times\ x$$

$$\begin{aligned}
&\mathit{out}_{B_a} :: B_a\ (\mathit{Btree}\ a) \rightarrow \mathit{Btree}\ a \\
&\mathit{out}_{B_a}\ (\mathit{Left}\ a) = \mathit{Leaf}\ a \\
&\mathit{out}_{B_a}\ (\mathit{Right}\ (t, t')) = \mathit{Join}\ t\ t'
\end{aligned}$$

$$\begin{aligned}
&\mathit{outBa} :: \mathit{Btree}\ a \rightarrow B_a\ (\mathit{Btree}\ a) \\
&\mathit{outBa}\ (\mathit{Leaf}\ a) = \mathit{Left}\ a \\
&\mathit{outBa}\ (\mathit{Join}\ t\ t') = \mathit{Right}\ (t, t')
\end{aligned}$$

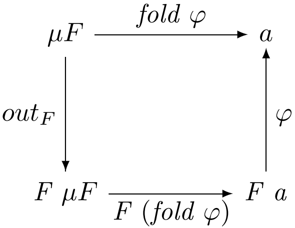
Hylomorphism

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



Fold

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



Fold: Lists

$$fold_L :: (b, a \to b \to b) \to List\ a \to b$$
$$fold_L\ (b, h)\ Nil = b$$
$$fold_L\ (b, h)\ (Cons\ a\ as) = h\ a\ (fold_L\ (b, h)\ as)$$

Example:

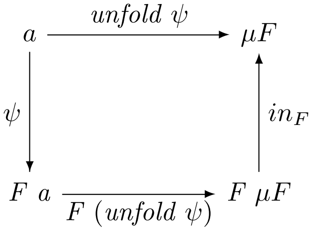
$$prod :: List\ Int \to Int$$
$$prod\ Nil = 1$$
$$prod\ (Cons\ n\ ns) = n * prod\ ns$$

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

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

Example:

$$
\begin{aligned}
&\mathit{upto} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{upto}\ n \mid n < 1 = \mathit{Nil} \\
&\qquad\quad \mid \mathit{otherwise} = \mathit{Cons}\ n\ (\mathit{upto}\ (n-1))
\end{aligned}
$$

As an unfold,

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

Factorisation

$$hylo\ \varphi\ \psi = fold\ \varphi \circ unfold\ \psi$$

Factorisation: factorial

$$fact = prod \circ upto$$

$$\begin{aligned}
&prod :: List\ Int \rightarrow Int\\
&prod\ Nil = 1\\
&prod\ (Cons\ n\ ns) = n * prod\ ns
\end{aligned}$$

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

Applying factorisation,

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

Factorisation: quicksort

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

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

$$mkTree :: Ord\ a \Rightarrow [a] \rightarrow Tree\ a$$
$$mkTree\ [\,] = Empty$$
$$\begin{aligned} mkTree\ (a : as) = Node\ & (mkTree\ [x \mid x \leftarrow as; x \leqslant a]) \\ & a \\ & (mkTree\ [x \mid x \leftarrow as; x > a]) \end{aligned}$$

Quicksort

$$
\begin{aligned}
&qsort :: Ord\ a \Rightarrow [a] \rightarrow [a] \\
&qsort\ [\,] = [\,] \\
&qsort\ (a : as) = qsort\ [x \mid x \leftarrow as; x \leqslant a] \\
&\qquad\qquad\qquad ++\ [a]\ ++ \\
&\qquad\qquad\qquad qsort\ [x \mid x \leftarrow as; x > a]
\end{aligned}
$$

Fusion laws

Factorisation

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

Hylo-Fold Fusion

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

$$\Rightarrow$$

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

Unfold-Hylo Fusion

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

$$\Rightarrow$$

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

Hylo-Fold Fusion

$$\textbf{data}\ Maybe\ a = Nothing \mid Just\ a$$

$$mapcoll :: (a \to b) \to List\ (Maybe\ a) \to List\ b$$
$$mapcoll = map\ f \circ collect$$

$$map\ f\ Nil = Nil$$
$$map\ f\ (Cons\ a\ as) = Cons\ (f\ a)\ (map\ f\ as)$$

$$collect :: List\ (Maybe\ Int) \to List\ Int$$
$$collect\ Nil = Nil$$
$$collect\ (Cons\ m\ ms) = \textbf{case}\ m\ \textbf{of}$$
$$Nothing \to collect\ ms$$
$$Just\ a \to Cons\ a\ (collect\ ms)$$

Hylo-Fold Fusion

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

Applying hylo-fold fusion,

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

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)