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*

$$
\begin{aligned}
&\mathit{count} :: \mathit{Word} \to \mathit{Text} \to \mathit{Integer}\\
&\mathit{count}\ w = \mathit{length} \circ \mathit{filter}\ (== w) \circ \mathit{words}
\end{aligned}
$$

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

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

Drawbacks of modularity

- Modular functions are not necessarily efficient.

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

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

- 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.

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

- 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

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

$$\Downarrow$$

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

How fusion proceeds

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

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

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

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.)

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

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

The result:

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

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

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

Capturing the structure of functions (2)

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

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

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

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

Capturing the structure of functions (3)

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

$$\begin{tikzcd}
Int \arrow[r, "fact"] \arrow[d, "\psi"'] & Int \\
() + Int \times Int \arrow[r, "fmap\ fact"'] & () + Int \times Int \arrow[u, "\varphi"']
\end{tikzcd}$$

Capturing the structure of functions (4)

Let us define,

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

Therefore,

$$\begin{tikzcd} Int \arrow[r, "fact"] \arrow[d, "\psi"'] & Int \\ F\ Int \arrow[r, "fmap\ fact"'] & F\ Int \arrow[u, "\varphi"'] \end{tikzcd}$$

Functor

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

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

$$\begin{aligned} \mathit{fmap}\ \mathit{id} &= \mathit{id} \\ \mathit{fmap}\ (f \circ g) &= \mathit{fmap}\ f \circ \mathit{fmap}\ g \end{aligned}$$

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

Hylomorphism

$$
\begin{aligned}
&hylo :: (F\ b \to b) \to (a \to F\ a) \to a \to b\\
&hylo\ \varphi\ \psi = \varphi \circ F\ (hylo\ \varphi\ \psi) \circ \psi
\end{aligned}
$$

$$
\begin{tikzcd}
a \arrow[r, "hylo\ \varphi\ \psi"] \arrow[d, "\psi"'] & b \\
F\ a \arrow[r, "F\ (hylo\ \varphi\ \psi)"'] & F\ b \arrow[u, "\varphi"']
\end{tikzcd}
$$

$\rightsquigarrow$ 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

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

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

Examples: Lists

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

$$\Downarrow$$

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

$$\begin{aligned} &L_a :: (x \to y) \to (L_a\ x \to L_a\ y) \\ &L_a\ f\ (Left\ ()) = Left\ () \\ &L_a\ f\ (Right\ (a, x)) = Right\ (a, f\ x) \end{aligned}$$

Example: Leaf-labelled binary trees

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

$$\Downarrow$$

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

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

Example: Internally-labelled binary trees

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

$$\Downarrow$$

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

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

Constructors / Destructors

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

$$\begin{tikzcd} F\mu F \arrow[r, shift left, "in_F"] & \mu F \arrow[l, shift left, "out_F"] \end{tikzcd}$$

where

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

Example: Leaf-labelled binary trees

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

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

$$\begin{array}{l} in_{B_a} :: B_a\ (\mathit{Btree}\ a) \to \mathit{Btree}\ a \\ in_{B_a}\ (\mathit{Left}\ a) = \mathit{Leaf}\ a \\ in_{B_a}\ (\mathit{Right}\ (t, t')) = \mathit{Join}\ t\ t' \end{array}$$

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

Hylomorphism

$$
\begin{aligned}
&hylo :: (F\ b \to b) \to (a \to F\ a) \to a \to b \\
&hylo\ \varphi\ \psi = \varphi \circ F\ (hylo\ \varphi\ \psi) \circ \psi
\end{aligned}
$$

$$
\begin{tikzcd}
a \arrow[r, "hylo\ \varphi\ \psi"] \arrow[d, "\psi"'] & b \\
F\ a \arrow[r, "F\ (hylo\ \varphi\ \psi)"'] & F\ b \arrow[u, "\varphi"']
\end{tikzcd}
$$

Fold

$$
\begin{aligned}
&\mathit{fold} :: (F\ a \to a) \to \mu F \to a \\
&\mathit{fold}\ \varphi = \varphi \circ F\ (\mathit{fold}\ \varphi) \circ \mathit{out}_F
\end{aligned}
$$

$$
\begin{tikzcd}
\mu F \arrow[r, "\mathit{fold}\ \varphi"] \arrow[d, "\mathit{out}_F"'] & a \\
F\ \mu F \arrow[r, "F\ (\mathit{fold}\ \varphi)"'] & F\ a \arrow[u, "\varphi"']
\end{tikzcd}
$$

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

$$unfold :: (a \to F\ a) \to a \to \mu F$$
$$unfold\ \psi = in_F \circ F\ (unfold\ \psi) \circ \psi$$

$$\begin{tikzcd}
a \arrow[r, "unfold\ \psi"] \arrow[d, "\psi"'] & \mu F \\
F\ a \arrow[r, "F\ (unfold\ \psi)"'] & F\ \mu F \arrow[u, "in_F"']
\end{tikzcd}$$

Unfold: Lists

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

Example:

$$
\begin{array}{l}
\mathit{upto} :: \mathit{Int} \rightarrow \mathit{Int} \\
\mathit{upto}\ n \mid n < 1 = \mathit{Nil} \\
\qquad\quad \mid \mathit{otherwise} = \mathit{Cons}\ n\ (\mathit{upto}\ (n-1))
\end{array}
$$

As an unfold,

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

Factorisation

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

Factorisation: factorial

$$fact = prod \circ upto$$

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

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

Applying factorisation,

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

Factorisation: quicksort

$$
\begin{aligned}
&\mathit{qsort} :: \mathit{Ord}\ a \Rightarrow [\,a\,] \rightarrow [\,a\,] \\
&\mathit{qsort} = \mathit{inorder} \circ \mathit{mkTree}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{inorder} :: \mathit{Tree}\ a \rightarrow \mathit{List}\ a \\
&\mathit{inorder}\ \mathit{Empty} = \mathit{Nil} \\
&\mathit{inorder}\ (\mathit{Node}\ t\ a\ t') = \mathit{inorder}\ t\ +\!\!+\ [\,a\,]\ +\!\!+\ \mathit{inorder}\ t'
\end{aligned}
$$

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

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

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

$$\begin{aligned} &mapcoll :: (a \to b) \to List\ (Maybe\ a) \to List\ b \\ &mapcoll = map\ f \circ collect \end{aligned}$$

$$\begin{aligned} &map\ f\ Nil = Nil \\ &map\ f\ (Cons\ a\ as) = Cons\ (f\ a)\ (map\ f\ as) \end{aligned}$$

$$\begin{aligned} &collect :: List\ (Maybe\ Int) \to List\ Int \\ &collect\ Nil = Nil \\ &collect\ (Cons\ m\ ms) = \mathbf{case}\ m\ \mathbf{of} \\ &\qquad\qquad\qquad\qquad Nothing \to collect\ ms \\ &\qquad\qquad\qquad\qquad Just\ a \to Cons\ a\ (collect\ ms) \end{aligned}$$

Hylo-Fold Fusion

$$
\begin{array}{l}
\tau :: (b, a \to b \to b) \to (b, \mathit{Maybe}\ a \to b \to b) \\
\tau\ (h_1, h_2) = (h_1, \\
\qquad\qquad \lambda m\ b \to \mathbf{case}\ m\ \mathbf{of} \\
\qquad\qquad\qquad \mathit{Nothing} \to b \\
\qquad\qquad\qquad \mathit{Just}\ a \to h_2\ a\ b)
\end{array}
$$

Applying hylo-fold fusion,

$$
\begin{array}{l}
\mathit{mapcoll} :: (a \to b) \to \mathit{List}\ (\mathit{Maybe}\ a) \to \mathit{List}\ b \\
\mathit{mapcoll}\ f\ \mathit{Nil} = \mathit{Nil} \\
\mathit{mapcoll}\ f\ (\mathit{Cons}\ m\ \mathit{ms}) = \mathbf{case}\ m\ \mathbf{of} \\
\qquad\qquad \mathit{Nothing} \to \mathit{mapcoll}\ f\ \mathit{ms} \\
\qquad\qquad \mathit{Just}\ a \to \mathit{Cons}\ (f\ a)\ (\mathit{mapcoll}\ f\ \mathit{ms})
\end{array}
$$

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/`