Deforestation in the presence of effects

Alberto Pardo

Instituto de Computación
Universidad de la República
Montevideo - Uruguay

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

Program construction in FP

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

Example: *count*

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

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

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

A drawback

- Functions may not have a well performance when defined using a compositional style.

- Information is passed from one function to another through an intermediate data structure.

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

- It often happens that the nodes of the intermediate data structure are generated/allocated by f, and immediately consumed/deallocated by g.

- The allocation/deallocation loop leads to repeated invocations of the garbage collector.

Deforestation

Deforestation is a program transformation technique for the elimination of intermediate data structures.

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

Deforestation of *count*

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

$$\Downarrow$$

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

How deforestation proceeds

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

Example

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

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

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

The result:

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

Programs with effects

The compositional style of programming is also useful in the presence of effects.

For example,

$$
\begin{aligned}
& \mathit{lenline} :: \mathit{IO}\ \mathit{Int} \\
& \mathit{lenLine} = \mathbf{do}\ \mathit{xs} \leftarrow \mathit{getLine} \\
& \qquad\qquad\quad\ \mathit{return}\ (\mathit{length}\ \mathit{xs})
\end{aligned}
$$

where

$$
\begin{aligned}
& \mathit{getLine} :: \mathit{IO}\ \mathit{String} \\
& \mathit{getLine} = \mathbf{do}\ c \leftarrow \mathit{getChar} \\
& \qquad\qquad\quad\ \mathbf{if}\ c == \texttt{'\textbackslash n'}\ \mathbf{then}\ \mathit{return}\ [\,] \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathbf{do}\ \mathit{cs} \leftarrow \mathit{getLine} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \mathit{return}\ (c : \mathit{cs})
\end{aligned}
$$

Deforestation with effects

The same considerations about the intermediate data structure apply in this case.

$$lenline = \mathbf{do}\; xs \leftarrow getLine; return\; (length\; xs)$$

$$\Downarrow$$

$$\begin{array}{lll} lenLine = & \mathbf{do} & c \leftarrow getChar \\ & & \mathbf{if}\; c == \texttt{'\textbackslash n'} \\ & & \mathbf{then}\; return\; 0 \\ & & \mathbf{else}\; \mathbf{do}\; n \leftarrow lenLine; return\; (1 + n) \end{array}$$

Deforestation with effects

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

$$
\begin{aligned}
&\mathit{getLine} :: \mathit{IO}\ \mathit{String}\\
&\mathit{getLine} = \textbf{do}\ c \leftarrow \mathit{getChar}\\
&\qquad\qquad\quad \textbf{if}\ c == \texttt{'\textbackslash n'}\ \textbf{then}\ \mathit{return}\ [\,]\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else do}\ cs \leftarrow \mathit{getLine}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (c : cs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{lenLine} = \textbf{do}\ c \leftarrow \mathit{getChar}\\
&\qquad\qquad\quad \textbf{if}\ c == \texttt{'\textbackslash n'}\ \textbf{then}\ \mathit{return}\ 0\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else do}\ n \leftarrow \mathit{lenLine}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{return}\ (h\ c\ n)
\end{aligned}
$$

Our approach to deforestation

- We adopt an approach based on recursion program schemes (like fold, map).

- They capture general patterns of computation commonly used in practice.

- Each recursion scheme has associated a set of algebraic laws.

- Some of these laws –called *fusion laws*– correspond to deforestation.

Capturing the structure of functions

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

Capturing the structure of functions (2)

Let us define,

$$
\begin{aligned}
\psi\ n \mid n < 1 &= \mathit{Left}\ () \\
\mid \mathit{otherwise} &= \mathit{Right}\ (n, n-1)
\end{aligned}
$$

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

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

Then,

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

Capturing the structure of functions (3)

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

Therefore,

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

Capturing the structure of functions (4)

Let us define,

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

Therefore,

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

Hylomorphism

$$
\begin{aligned}
&\mathit{hylo} :: (F\ b \to b) \to (a \to F\ a) \to a \to b \\
&\mathit{hylo}\ h\ g = h \circ \mathit{fmap}\ (\mathit{hylo}\ h\ g) \circ g
\end{aligned}
$$

$$
\begin{tikzcd}
a \arrow[r, "\mathit{hylo}\ h\ g"] \arrow[d, "g"'] & b \\
F\ a \arrow[r, "\mathit{fmap}\ (\mathit{hylo}\ h\ g)"'] & F\ b \arrow[u, "h"']
\end{tikzcd}
$$

h is called an *algebra* and g a *coalgebra*.

Data types

Functors describe the top level structure of data types.

Given a data type declaration

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

the derivation of the corresp. functor F proceeds as follows:

- pack the arguments to constructors in tuples;
- for constant constructors place the empty tuple ();
- regard alternatives as sums (replace $\mid$ by $+$);
- substitute the occurrences of τ by a type variable a in every $\tau_{i,j}$.

Example: Lists

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

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

$$\begin{aligned}
&fmap :: (b \to c) \to (L_a\ b \to L_a\ c)\\
&fmap\ f\ (Left\ ()) = Left\ ()\\
&fmap\ f\ (Right\ (a, b)) = Right\ (a, f\ b)
\end{aligned}$$

Example: Leaf-labelled binary trees

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

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

$$\begin{array}{l} \mathit{fmap} :: (b \to c) \to (B_a\ b \to B_a\ c) \\ \mathit{fmap}\ f\ (\mathit{Left}\ a) = (\mathit{Left}\ a) \\ \mathit{fmap}\ f\ (\mathit{Right}\ (b1, b2)) = \mathit{Right}\ (f\ b1, f\ b2) \end{array}$$

Data type constructors / destructors

Given a functor $(F, fmap)$ 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

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

Fold / Unfold

Fold

$$fold :: (F\ a \to a) \to \mu F \to a$$
$$fold\ h = hylo\ h\ out_F$$

Unfold

$$unfold :: (a \to F\ a) \to a \to \mu F$$
$$unfold\ g = hylo\ in_F\ g$$

Factorisation

$$hylo\ h\ g = fold\ h \circ unfold\ g$$

Examples of unfold

Lists

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

Leaf-labelled binary trees

$$
\begin{aligned}
&\mathit{unfold}_B :: (b \to B_a\ b) \to b \to \mathit{Btree}\ a\\
&\mathit{unfold}_B\ g\ b = \textbf{case}\ (g\ b)\ \textbf{of}\\
&\qquad\qquad \mathit{Left}\ a \to \mathit{Leaf}\ a\\
&\qquad\qquad \mathit{Right}\ (b1, b2) \to \mathit{Join}\ (\mathit{unfold}_B\ g\ b1)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{unfold}_B\ g\ b2)
\end{aligned}
$$

Fusion laws

Factorisation

$$hylo\ h\ g = hylo\ h\ out_F \circ hylo\ in_F\ g$$

Hylo-Fold Fusion

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

$$\Rightarrow$$

$$fold\ h \circ hylo\ (\tau\ in_F)\ g = hylo\ (\tau\ h)\ g$$

Unfold-Hylo Fusion

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

$$\Rightarrow$$

$$hylo\ h\ (\sigma\ out_F) \circ unfold\ g = hylo\ h\ (\sigma\ g)$$

Factorisation

$$fact = prod \circ upto$$

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

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

Hylo-Fold Fusion

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

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

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

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

Hylo-Fold Fusion

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

Monads

A *monad* is a triple $(m, return, \gg\!=)$, where

- m is a type constructor,

- $return :: a \to m\ a$

  is a polymorphic function

- $(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b$

  is a polymorphic operator, often pronounced *bind*.

plus some monad laws.

do notation

Translation rules:

$$\mathbf{do}\,\{\,x \leftarrow m; m'\,\} \;=\; m \gg\!= \lambda x \rightarrow \mathbf{do}\,\{\,m'\,\}$$

$$\mathbf{do}\,\{\,m\,\} \;=\; m$$

Recursion with effects

$$
\begin{tikzcd}
a \arrow[rr, "hylo\ h\ g"] \arrow[d, "g"'] & & b \\
F\ a \arrow[rr, "fmap\ (hylo\ h\ g)"'] & & F\ b \arrow[u, "h"']
\end{tikzcd}
$$

Monadic hylomorphism

$$mhylo\ h\ g = h \bullet fmapM\ (mhylo\ h\ g) \bullet g$$

$$\begin{tikzcd}
a \arrow[r, "mhylo\ h\ g"] \arrow[d, "g"'] & m\ b \\
m\ (F\ a) \arrow[r, "(fmapM\ (mhylo\ h\ g))^{\star}"'] & m\ (F\ b) \arrow[u, "h^{\star}"']
\end{tikzcd}$$

$$fmapM\ f\ =\ F\ a \xrightarrow{F\ f} F\ (m\ b) \xrightarrow{dist_F} m\ (F\ b)$$

Lists

$$
\begin{array}{l}
mhylo_L\ (h_1, h_2)\ g = mh_L \\
\quad \textbf{where} \\
\qquad mh_L\ b = \textbf{do}\ x \leftarrow g\ b \\
\qquad\qquad\qquad\quad \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Left}\ () \rightarrow h_1 \\
\qquad\qquad\qquad\qquad \mathit{Right}\ (a, b') \rightarrow \textbf{do}\ c \leftarrow mh_L\ b' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h_2\ a\ c
\end{array}
$$

Example

$$
\begin{array}{l}
msum_L :: \mathit{Monad}\ m \Rightarrow \mathit{List}\ (m\ \mathit{Int}) \rightarrow m\ \mathit{Int} \\
msum_L\ \mathit{Nil} = \mathit{return}\ 0 \\
msum_L\ (\mathit{Cons}\ m\ ms) = \textbf{do}\ y \leftarrow msum_L\ ms \\
\qquad\qquad\qquad\qquad\qquad\quad x \leftarrow m \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (x + y)
\end{array}
$$

A more practical approach

$$mhylo\ h\ g = h \bullet mmap\ (fmap\ (mhylo\ h\ g)) \circ g$$

$$\begin{tikzcd}
a \arrow[r, "mhylo\ h\ g"] \arrow[d, "g"'] & m\ b \\
m\ (F\ a) \arrow[r, "mmap\ (fmap\ (mhylo\ h\ g))"'] & m\ (F\ (m\ b)) \arrow[u, "h^{\star}"']
\end{tikzcd}$$

Now $h :: F\ (m\ b) \to m\ b$ is an algebra with monadic carrier.

$mmap :: (a \to b) \to (m\ a \to m\ b)$

Examples

$$
\begin{aligned}
&\mathit{sequence} :: \mathit{Monad}\ m \Rightarrow \mathit{List}\ (m\ a) \rightarrow m\ (\mathit{List}\ a)\\
&\mathit{sequence}\ \mathit{Nil} = \mathit{return}\ \mathit{Nil}\\
&\mathit{sequence}\ (\mathit{Cons}\ m\ \mathit{ms}) = \mathbf{do}\ \begin{array}[t]{l} a \leftarrow m\\ \mathit{as} \leftarrow \mathit{sequence}\ \mathit{ms}\\ \mathit{return}\ (\mathit{Cons}\ a\ \mathit{as}) \end{array}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{msum}_L :: \mathit{Monad}\ m \Rightarrow \mathit{List}\ (m\ \mathit{Int}) \rightarrow m\ \mathit{Int}\\
&\mathit{msum}_L\ \mathit{Nil} = \mathit{return}\ 0\\
&\mathit{msum}_L\ (\mathit{Cons}\ m\ \mathit{ms}) = \mathbf{do}\ \begin{array}[t]{l} x \leftarrow m\\ y \leftarrow \mathit{msum}_L\ \mathit{ms}\\ \mathit{return}\ (x + y) \end{array}
\end{aligned}
$$

Properties

MHylo-Fold Fusion

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

$$\Rightarrow$$

$$mmap\ (fold\ h) \circ mhylo\ (\tau\ in_F)\ g = mhylo\ (\tau\ h)\ g$$

Unfold-MHylo Fusion

$$\sigma :: \forall\ a\ .\ (a \to F\ a) \to (a \to m\ (G\ a))$$

$$\Rightarrow$$

$$mhylo\ h\ (\sigma\ out_F) \circ unfold\ g = mhylo\ h\ (\sigma\ g)$$

MHylo-Fold Fusion

$$msum_L :: List\ (m\ Int) \rightarrow m\ Int$$
$$msum_L = mmap\ sum_L \circ sequence$$

$$sum_L :: List\ Int \rightarrow Int$$
$$sum_L\ Nil = 0$$
$$sum_L\ (Cons\ a\ as) = a + sum_L\ as$$

$$sequence :: Monad\ m \Rightarrow List\ (m\ a) \rightarrow m\ (List\ a)$$
$$sequence\ Nil = return\ [\,]$$
$$\begin{aligned} sequence\ (Cons\ m\ ms) = \mathbf{do}\ & a \leftarrow m \\ & as \leftarrow sequence\ ms \\ & return\ (Cons\ a\ as) \end{aligned}$$

MHylo-Fold Fusion

$$
\begin{aligned}
&\tau :: (b, Int \to b \to b) \to (m\ b, m\ Int \to m\ b \to m\ b)\\
&\tau\ (h_1, h_2) = (return\ h_1,\\
&\qquad\qquad\quad \lambda m\ mb \to \mathbf{do}\ a \leftarrow m\\
&\qquad\qquad\qquad\qquad\qquad\quad b \leftarrow mb\\
&\qquad\qquad\qquad\qquad\qquad\quad return\ (h_2\ a\ b))
\end{aligned}
$$

$$
\begin{aligned}
&msum_L :: Monad\ m \Rightarrow List\ (m\ Int) \to m\ Int\\
&msum_L\ Nil = return\ 0\\
&msum_L\ (Cons\ m\ ms) = \mathbf{do}\ x \leftarrow m\\
&\qquad\qquad\qquad\qquad\qquad\quad y \leftarrow msum_L\ ms\\
&\qquad\qquad\qquad\qquad\qquad\quad return\ (x + y)
\end{aligned}
$$