# 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*

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

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

$$\begin{array}{l} \textit{filter} :: (a \to Bool) \to [a] \to [a] \\ \textit{filter} \ p \ [] = [] \\ \textit{filter} \ p \ (a : as) = \mathbf{if} \ p \ a \ \mathbf{then} \ a : \textit{filter} \ p \ as \\ \mathbf{else} \ \textit{filter} \ p \ as \end{array}$$

# 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.

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

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

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

### **Deforestation of** count

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

```
count w \ t = \mathbf{case} \ drop \ While \ is Space \ t \ \mathbf{of}

"" \rightarrow 0

t' \rightarrow \mathbf{let} \ (w', t'') = break \ is Space \ t'

in if w' == w

then 1 + count \ w \ t''

else count w \ t''
```

# 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

$$lenfil \ p = length \circ filter \ p$$
$$length \ [] = 0$$
$$length \ (x : xs) = h \ x \ (length \ xs)$$
$$where \ h \ x \ n = 1 + n$$
$$filter \ p \ [] = []$$
$$filter \ p \ (a : as) = if \ p \ a \ then \ a : filter \ p \ as$$
$$else \ filter \ p \ as$$

The result:

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

# **Programs with effects**

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

For example,

where

$$\begin{array}{l} getLine :: IO \ String\\ getLine = \mathbf{do} \ c \leftarrow getChar\\ \mathbf{if} \ c == `\n' \ \mathbf{then} \ return \ []\\ \mathbf{else} \ \mathbf{do} \ cs \leftarrow getLine\\ return \ (c:cs) \end{array}$$

### **Deforestation with effects**

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

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

$$u = \mathbf{do} \ c \leftarrow getChar$$

$$\mathbf{if} \ c == \cdot \mathbf{n},$$

$$\mathbf{then} \ return \ 0$$

$$\mathbf{else} \ \mathbf{do} \ n \leftarrow lenLine; return \ (1+n)$$

### **Deforestation with effects**

$$length [] = 0$$

$$length (x : xs) = h x (length xs)$$
where
$$h x n = 1 + n$$

$$getLine :: IO String$$

$$getLine = do c \leftarrow getChar$$
if  $c == '\n'$  then return []
else do  $cs \leftarrow getLine$ 

$$return (c : cs)$$

$$lenLine = do \ c \leftarrow getChar \\ if \ c == ``n' then \ return \ 0 \\ else \ do \ n \leftarrow lenLine \\ return \ (h \ c \ n) \end{cases}$$

# 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

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

# Capturing the structure of functions (2)

Let us define,

$$\psi \ n \mid n < 1 = Left \ ()$$
$$\mid otherwise = Right \ (n, n-1)$$

$$\begin{array}{l} fmap \ f \ (Left \ ()) = Left \ () \\ fmap \ f \ (Right \ (m,n)) = Right \ (m,f \ n) \end{array}$$

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

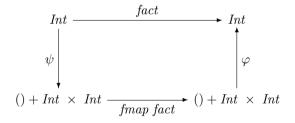
Then,

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

### Capturing the structure of functions (3)

$$\begin{array}{l} fmap \ f \ (Left \ ()) = Left \ () \\ fmap \ f \ (Right \ (m,n)) = Right \ (m,f \ n) \end{array}$$

Therefore,

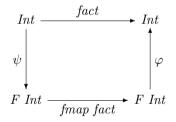


# Capturing the structure of functions (4)

Let us define,

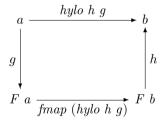
$$F \ a = () + Int \ \times \ a$$

Therefore,



### Hylomorphism

$$\begin{array}{l} hylo :: (F \ b \to b) \to (a \to F \ a) \to a \to b \\ hylo \ h \ g = h \circ fmap \ (hylo \ h \ g) \circ g \end{array}$$



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

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

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 | by +);
- substitute the occurrences of  $\tau$  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**

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

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

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

# Data type constructors / destructors

Given a functor (F, fmap) there exists an isomorphism

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

where

- $in_F$  packs the constructors of the data type  $\mu F$
- $out_F$  packs the destructors of  $\mu F$

# Fold / Unfold

#### Fold

$$\begin{array}{l} \textit{fold} :: (F \ a \to a) \to \mu F \to a \\ \textit{fold} \ h = hylo \ h \ out_F \end{array}$$

#### 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

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

#### Leaf-labelled binary trees

$$\begin{array}{l} unfold_B :: (b \to B_a \ b) \to b \to Btree \ a \\ unfold_B \ g \ b = \mathbf{case} \ (g \ b) \ \mathbf{of} \\ Left \ a \to Leaf \ a \\ Right \ (b1, b2) \to Join \ (unfold_B \ g \ b1) \\ (unfold_B \ g \ b2) \end{array}$$

### **Fusion laws**

#### Factorisation

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

#### Hylo-Fold Fusion

$$\begin{aligned} \tau :: \forall \ a \ . \ (F \ a \to a) \to (G \ a \to a) \\ \Rightarrow \\ fold \ h \circ hylo \ (\tau \ in_F) \ g = hylo \ (\tau \ h) \ g \end{aligned}$$

#### **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**

 $\mathit{fact} = \mathit{prod} \circ \mathit{upto}$ 

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

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

### **Hylo-Fold Fusion**

**data** Maybe  $a = Nothing \mid Just a$ 

 $\begin{array}{l} mapcoll :: (a \rightarrow b) \rightarrow List \ (Maybe \ a) \rightarrow List \ b \\ mapcoll = map \ f \circ collect \end{array}$ 

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

 $collect :: List (Maybe Int) \rightarrow List Int$  collect Nil = Nil collect (Cons m ms) = case m of  $Nothing \rightarrow collect ms$  $Just a \rightarrow Cons a (collect ms)$ 

### **Hylo-Fold Fusion**

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

### 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.

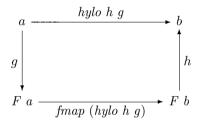
plus some monad laws.

### do notation

Translation rules:

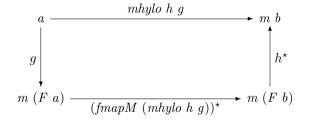
$$\mathbf{do} \{ x \leftarrow m; m' \} = m \gg \lambda x \to \mathbf{do} \{ m' \}$$
$$\mathbf{do} \{ m \} = m$$

### **Recursion with effects**



### Monadic hylomorphism

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



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

### Lists

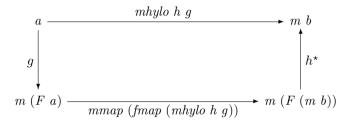
$$\begin{array}{l} \textit{mhylo}_L \ (h_1, h_2) \ g = \textit{mh}_L \\ \textbf{where} \\ \textit{mh}_L \ b = \textbf{do} \ x \leftarrow g \ b \\ \textbf{case} \ x \ \textbf{of} \\ \textit{Left} \ () \rightarrow h_1 \\ \textit{Right} \ (a, b') \rightarrow \textbf{do} \ c \leftarrow \textit{mh}_L \ b' \\ h_2 \ a \ c \end{array}$$

#### Example

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

### A more practical approach

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



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**

sequence :: Monad  $m \Rightarrow List (m \ a) \rightarrow m$  (List a) sequence Nil = return Nil sequence (Cons m ms) = do  $a \leftarrow m$  $as \leftarrow sequence ms$ return (Cons a as)

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

 $\Rightarrow$ 

#### MHylo-Fold Fusion

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

### mmap (fold h) $\circ$ mhylo ( $\tau$ in<sub>F</sub>) 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**

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

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

sequence :: Monad  $m \Rightarrow List (m \ a) \rightarrow m \ (List \ a)$ sequence Nil = return [] sequence (Cons m ms) = do  $a \leftarrow m$  $as \leftarrow sequence \ ms$ return (Cons a as)

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

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