An Extended Form of Shortcut Fusion with Multiple Applications

ALBERTO PARDO

Instituto de Computación
Universidad de la República
Montevideo - Uruguay

(joint work with João Fernandes, João Saraiva and Cecilia Manzino)

Tallinn, March 24, 2009

Modular programs

Separate parts are combined using intermediate data structures.

```
factorial   :: Int -> Int
factorial n = product (down n)

product         :: [Int] -> Int
product []       = 1
product (a:as)  = a * product as

down   :: Int -> [Int]
down 0 = []
down n = n : down (n-1)
```

Modular programs

Benefits

- Easier to understand
- Easier to maintain

Modular programs

Benefits

- Easier to understand
- Easier to maintain

Drawbacks

- Poor performance

Program fusion

```
factorial m = product (down m)

product []     = 1
product (a:as) = a * product as

down 0 = []
down m = m : down (m-1)
```

Program fusion

```
factorial m = product (down m)

product []     = 1
product (a:as) = a * product as

down 0 = []
down m = m : down (m-1)
```

$$\Downarrow$$

```
factorial 0 = 1
factorial m = m * factorial (m-1)
```

Shortcut fusion for lists

Consumer

```
fold :: (b,a -> b -> b) -> [a] -> b
fold (n,c) []       = n
fold (n,c) (a:as) = c a (fold (n,c) as)
```

Shortcut fusion for lists

Consumer

```
fold :: (b,a -> b -> b) -> [a] -> b
fold (n,c) []     = n
fold (n,c) (a:as) = c a (fold (n,c) as)
```

Producer

```
build :: (forall b. (b,a -> b -> b) -> c -> b)
         -> c -> [a]
build g  = g ([],(:))
```

Shortcut fusion for lists

Consumer

```
fold :: (b,a -> b -> b) -> [a] -> b
fold (n,c) []       = n
fold (n,c) (a:as) = c a (fold (n,c) as)
```

Producer

```
build :: (forall b. (b,a -> b -> b) -> c -> b)
          -> c -> [a]
build g  = g ([],(:))
```

fold/build

```
    fold (n,c) . build g = g (n,c)
```

Consumer: `product`

```
product []     = 1
product (a:as) = a * product as
```

$$\Downarrow$$

```
product = fold (1,(*))
```

Producer: down

```
down 0 = []
down m = m : down (m-1)
```

$\Downarrow$

```
down = build gdown
  where
    gdown (n,c) 0 = n
    gdown (n,c) m = c m (gdown (n,c) (m-1))
```

Fusion: factorial

```
product = fold (1,(*))

down = build gdown
  where
    gdown (n,c) 0 = n
    gdown (n,c) m = c m (gdown (n,c) (m-1))
```

Fusion: factorial

```
product = fold (1,(*))

down = build gdown
  where
    gdown (n,c) 0 = n
    gdown (n,c) m = c m (gdown (n,c) (m-1))



factorial
    = product . down
```

Fusion: `factorial`

```
product = fold (1,(*))

down = build gdown
  where
    gdown (n,c) 0 = n
    gdown (n,c) m = c m (gdown (n,c) (m-1))



factorial
    = product . down
    = fold (1,(*)) . build gdown
```

Fusion: factorial

```
product = fold (1,(*))

down = build gdown
  where
    gdown (n,c) 0 = n
    gdown (n,c) m = c m (gdown (n,c) (m-1))


factorial
    = product . down
    = fold (1,(*)) . build gdown
    = gdown (1,(*))
```

Extended shortcut fusion

Let `N` be a type constructor with an associated map function

```
mapN :: (a -> b) -> (N a -> N b)
```

Extended shortcut fusion

Let `N` be a type constructor with an associated map function

```
mapN :: (a -> b) -> (N a -> N b)
```

Producer

```
buildN :: (forall b. (b,a -> b -> b) -> c -> N b)
              -> c -> N [a]
buildN g  = g ([],(:))
```

Extended shortcut fusion

Let N be a type constructor with an associated map function

```
mapN :: (a -> b) -> (N a -> N b)
```

Producer

```
buildN :: (forall b. (b,a -> b -> b) -> c -> N b)
          -> c -> N [a]
buildN g  = g ([],(:))
```

extended fold/build

```
    mapN (fold (n,c)) . buildN g = g (n,c)
```

Monadic shortcut fusion [Manzino & Pardo, SBLP'08]

```
type N a = m a

mmap :: Monad m => (a -> b) -> (m a -> m b)
mmap f m = do {a <- m; return (f a)}
```

Monadic shortcut fusion [Manzino & Pardo, SBLP'08]

```
type N a = m a

mmap :: Monad m => (a -> b) -> (m a -> m b)
mmap f m = do {a <- m; return (f a)}
```

Producer

```
mbuild :: Monad m
        => (forall b. (b,a -> b -> b) -> c -> m b)
        -> c -> m [a]
mbuild g  = g ([],(:))
```

Monadic shortcut fusion [Manzino & Pardo, SBLP'08]

```
type N a = m a

mmap :: Monad m => (a -> b) -> (m a -> m b)
mmap f m = do {a <- m; return (f a)}
```

Producer

```
mbuild :: Monad m
        => (forall b. (b,a -> b -> b) -> c -> m b)
        -> c -> m [a]
mbuild g  = g ([],(:))
```

fold/mbuild

```
  do {as <- mbuild g x; return (fold (n,c) as)}
= g (n,c) x
```

Example: `lenLine`

```
lenLine = do {cs <- getLine; return(length cs)}

length         :: [a] -> Int
length []       = 0
length (x:xs)   = 1 + length xs

getLine :: IO String
getLine = do c <- getChar
             if c == eol
                then return []
                else do cs <- getLine
                        return (c : cs)
```

Example: `lenLine` (2)

```
lenLine = do {cs <- getLine; return(length xs)}

length         :: [a] -> Int
length []       = 0
length (x:xs)  = 1 + length xs

getLine :: IO String
getLine = do c <- getChar
             if c == eol
                then return []
                else do cs <- getLine
                        return (c : cs)
```

Example: lenLine (3)

```
length = fold (0,h) where h x y = 1 + y

getLine = mbuild ggL
  where
    ggL (n,c) = do c' <- getChar
                   if c' == eol
                      then return n
                      else do b <- ggL (n,c)
                              return (c c' b)
```

Example: `lenLine` (4)

```
lenLine
  = do {cs <- getLine; return(length cs)}
  = do {cs <- mbuild ggL; return(fold (0,h) cs)}
  = ggL (0,h)
```

Example: `lenLine` (4)

```
lenLine
  = do {cs <- getLine; return(length cs)}
  = do {cs <- mbuild ggL; return(fold (0,h) cs)}
  = ggL (0,h)
```

```
lenLine = do c <- getChar
             if c == eol
                then return 0
                else do n <- lenLine
                        return (1 + n)
```

Fusion of effectful functions [Ghani & Johann 08], [Chitil 00]

effectful fold/mbuild

```
  do {as <- mbuild g x; fold (n,c) as}
=
  do {m <- g (n,c) x; m}
```

where

```
n :: m b
c :: a -> m b -> m b
fold (n,c) :: [a] -> m b
```

Circular program derivation [Fernandes & Pardo & Saraiva, HW'07]

```
type N a = (a,z)

mapN :: (a -> b) -> ((a,z) -> (b,z))
mapN f (a,z) = (f a,z)
```

Circular program derivation [Fernandes & Pardo & Saraiva, HW'07]

```
type N a = (a,z)

mapN :: (a -> b) -> ((a,z) -> (b,z))
mapN f (a,z) = (f a,z)
```

Producer

```
buildp :: (forall b. (b,a -> b -> b) -> c -> (b,z))
            -> c -> ([a],z)
buildp g = g ([],(:))
```

fold/buildp

```
(fold (n,c) × id) . buildp g = g (n,c)
```

Circular program derivation [Fernandes & Pardo & Saraiva, HW'07]

Consumer

```
pfold :: (z -> b, a -> b -> z -> b)
      -> ([a], z) -> b
pfold (hn,hc) = pL
  where  pL ([],z)     = hn z
         pL (a:as,z) = hc a (pL (as,z)) z
```

pfold/buildp

```
pfold (hn,hc) . buildp g $ i
    = let (v, z) = g (n,c) i
          n = hn z
          c x y = hc x y z
      in v
```

Example: `repmax`

```
repmax = replace . copymax

replace :: ([a],a) -> [a]
replace ([], a) = []
replace (x:xs, a) = a : replace (xs, a)

{- lists with nonnegative elements -}
copymax :: Ord a => [a] -> ([a],a)
copymax [] = ([], 0)
copymax (x:xs) = let (ys,m) = copymax xs
                 in (x : ys, max x m)
```

Example: `repmax` (2)

```
repmax = replace . copymax

replace :: ([a],a) -> [a]
replace ([], a) = []
replace (x:xs, a) = a : replace (xs, a)

{- lists with nonnegative elements -}
copymax :: Ord a => [a] -> ([a],a)
copymax [] = ([], 0)
copymax (x:xs) = let (ys,m) = copymax xs
                 in (x : ys, max x m)
```

Example: repmax (3)

```
repmax = replace . copymax

replace :: ([a],a) -> [a]
replace = pfold (hn,hc)
    where hn _ = []
          hc _ l m = m:l

copymax :: Ord a => [a] -> ([a],a)
copymax = buildp g
    where g (n, c) [] = (n, 0)
          g (n, c) (x:xs)
             = let (ys, m) = g (n, c) xs
               in (c x ys, max x m)
```

Example: `repmax` (4)

```
repmax xs = zs
  where
    (zs, m ) = repm xs
    repm [] = ([],0)
    repm (x:xs) = let (ys,m') = repm xs
                  in ( m : ys, max x m')
```

Monadic circular program derivation [Pardo & Fernandes & Saraiva, PEPM'09]

```
type N a = m (a,z)

mapN f = mmap (f × id)

mmap f m = do {a <- m; return (f a)}
```

Producer

```
mbuildp :: Monad m =>
    (forall b. (b,a -> b -> b) -> m (b,z))
     -> m ([a],z)
mbuildp g = g ([],(:))
```

Monadic circular program derivation

fold/mbuildp

```
do {(xs,z) <- mbuildp g;return (fold (n,c) xs,z)}
= g (n,c)
```

pfold/mbuildp Let *m* be a recursive monad.

```
do  {(xs,z) <- mbuildp g;
       return (pfold (hn, hc) (xs,z))}
=
mdo {(v,z) <- let n = hn z
                  c x y  = hc x y z
              in g (n, c);
       return v}
```

Example: Parsing

```
newtype Parser a = P (String -> [(a,String)])

instance Monad Parser where
  return a = P (\cs -> [(a,cs)])
  p »= f   = ...

pzero :: Parser a
pzero = P (\cs -> [])

(<|>) :: Parser a -> Parser a -> Parser a
(P p) <|> (P q)
      = P (\cs ->  case p cs ++ q cs of
                         []       -> []
                         (x:xs)   -> [x])
```

Example: Parsing (2)

```
transform = do (bs, s) <- bitstring
               return (applyXor (bs, s))

applyXor :: ([Bit], Bit) -> [Bit]
applyXor ([], _) = []
applyXor (b:bs, s) = xor s b : applyXor (bs, s)

bitstring :: Parser ([Bit], Bit)
bitstring =  do b          <- bit
                (bs, s)  <- bitstring
                return (b:bs, xor s b)
          <|> return ([], 0)
```

Example: Parsing (3)

```
transform = do (bs, s) <- bitstring
               return (applyXor (bs, s))

applyXor = pfold (hn,hc)
  where hn _        = []
        hc b r s    = xor b s : r

bitstring = mbuildp g
  where g (n, c)
             =  do b       <- bit
                   (bs, s) <- g (n, c)
                   return (c b bs,xor b s)
             <|>  return (n,0)
```

Example: Parsing (4)

```
transform
  = mdo (bs, s )
          <- let gbits
                 = do b         <- bit
                      (bs', s') <- gbits
                      return (xor s  b : bs',
                              xor s' b)
                 <|>  return ([],0)
             in gbits
        return bs
```

pfold as higher-order fold

```
pfold (hn, hc) :: ([a], z) -> b
```

pfold as higher-order fold

```
pfold (hn, hc) :: ([a], z) -> b

fold (fn, fc) :: [a] -> (z -> b)
```

pfold as higher-order fold

```
pfold (hn, hc) :: ([a], z) -> b

fold (fn, fc) :: [a] -> (z -> b)

pfold (hn, hc) = apply . ((fold (fn, fc)) × id)
```

Monadic H.O. program derivation [PEPM'09]

```
type N a = m (a,z)

mapN f = mmap (f × id)

mmap f m = do {a <- m; return (f a)}
```

Producer

```
mbuildp :: Monad m =>
    (forall b. (b,a -> b -> b) -> c -> m (b,z))
    -> c -> m ([a],z)
mbuildp g = g ([],(:))
```

Monadic H.O. program derivation

pfold as higher-order fold

```
pfold (hn, hc) = apply . ((fold (fn, fc)) × id)
```

higher-order pfold/mbuildp

```
do {(t,z)  <- mbuildp g;
     return (pfold (hn, hc) (t,z))}
=
do {(f,z) <-  g (fn, fc));
     return (f z)}
```

Example: Parsing

```
applyXor = pfold (hn,hc)
  where hn _        = []
        hc b r s   = (xor b s) : r
```

$$\Downarrow$$

```
applyXor = fold (fn,fc)
  where fn       = \_ -> []
        fc b r  = \s -> (xor b s): r s
```

Example: Parsing (2)

```
transform = do (bs, s) <- bitstring
               return (applyXor (bs, s))
```

$$\Downarrow$$

```
transform = do (f,s) <- gbits
                  return (f s)
  where
   gbits = do b <- bit
                 (f,s)  <- gbits
                 return (\s' -> (xor b s'): f s',
                         xor b s)
           <|>  return (\_ -> [],0)
```

Conclusions

- We presented shortcut fusion laws for the derivation of circular and higher-order (monadic) programs.
- The laws are simple and easy to apply in practice.
- The laws developed are generic, in the sense that they can be defined for a wide class of datatyes and monads.
- Like standard shortcut fusion (fold/build), our laws can also be implemented in GHC using the RULES pragma (rewrite rules).

Summary of results

$$
\begin{tikzcd}
& cons \circ prod \arrow[dl, "\alpha"'] \arrow[dr, "\alpha"] & \\
circular \arrow[ur, dashed, "\beta"'] \arrow[rr, dashed, "\beta"'] & & higher\ order
\end{tikzcd}
$$

Future Work

- Multiple intermediate data structure elimination;

```
prog = fn . ... . f2 . f1
```

- Relation with Attribute Grammars.