A Shortcut Fusion Rule for Circular Program Calculation

João Fernandes[1] Alberto Pardo[2] João Saraiva[1]

[1]Departmento de Informática
Universidade do Minho
Portugal

[2]Instituto de Computación
Universidad de la República
Uruguay

Circular programs

- Circular programs were proposed by R. Bird as a technique to eliminate multiple traversals of data structures.
- Circular definitions are of the form:

$$(\ldots, x, \ldots) = f\ (\ldots, x, \ldots)$$

- Circular programs have been used to express pretty-printers or type systems.
- They are the natural representation of attribute grammars in a lazy setting.

Motivation:

- In this work we show the derivation of circular programs from non-circular ones.

Why don't we write circular programs directly?

1. most programmers find it very difficult;
2. it is easy to write a circular program that does not terminate, i.e. a program with a real circularity.

Motivation:

- In this work we show the derivation of circular programs from non-circular ones.

Why don't we write circular programs directly?

1. most programmers find it very difficult;
2. it is easy to write a circular program that does not terminate, i.e. a program with a <u>real</u> circularity.

Motivation:

- We may see circular programs as an intermediate stage for further transformations.

Post-processing of the derived circular programs:

1. *Tools and Libraries to Model and Manipulate Circular Programs*, (Fernandes & Saraiva, PEPM 07);
2. very efficient, completely data-structure free, programs are obtained.

Motivation:

- We may see circular programs as an intermediate stage for further transformations.

Post-processing of the derived circular programs:

1. *Tools and Libraries to Model and Manipulate Circular Programs*, (Fernandes & Saraiva, PEPM 07);
2. very efficient, completely data-structure free, programs are obtained.

Bird's *repmin*

```
data Tree = Leaf Int | Fork Tree Tree

transform     :: Tree -> Tree
transform t    = replace t (tmin t)

replace :: Tree -> Int -> Tree
replace (Leaf n) m = Leaf m
replace (Fork l r) m = Fork (replace l m)
                            (replace r m)

tmint :: Tree -> Int
tmint (Leaf n) = n
tmint (Fork l r) = min (tmin l) (tmin r)
```

Bird's method

```
repmin t m = (replace t m, tmin t)
```

⇓

```
repmin (Leaf n) m = (Leaf m,n)
repmin (Fork l r) m = (Fork l' r', min ml mr)
            where (l',ml) = repmin l m
                  (r',mr) = repmin r m
```

⇓

```
transform t = t'
     where (t',m) = repmin t m
```

Bird's method

```
repmin t m = (replace t m, tmin t)
```

$$\Downarrow$$

```
repmin (Leaf n) m = (Leaf m,n)
repmin (Fork l r) m = (Fork l' r', min ml mr)
             where (l',ml) = repmin l m
                   (r',mr) = repmin r m
```

$$\Downarrow$$

```
transform t = t'
     where (t',m) = repmin t m
```

Bird's method

```
repmin t m = (replace t m, tmin t)
```

$$\Downarrow$$

```
repmin (Leaf n) m = (Leaf m,n)
repmin (Fork l r) m = (Fork l' r', min ml mr)
               where (l',ml) = repmin l m
                     (r',mr) = repmin r m
```

$$\Downarrow$$

```
transform t = t'
     where (t',m) = repmin t m
```

Our method

- We present a calculational rule for circular program derivation
- It calculates circular programs from compositions of the form:

$$a \xrightarrow{prod} (t, z) \xrightarrow{cons} b$$

- Our calculational rule is:
  - generic
  - correct (preserves termination properties)

Our method

- The rule we present is a variant of shortcut fusion (fold/build).

We achieve:

1. intermediate structure deforestation;
2. multiple traversal elimination;
3. correctness guarantees.

Our method

- The rule we present is a variant of shortcut fusion (fold/build).

We achieve:

1. intermediate structure deforestation;
2. multiple traversal elimination;
3. correctness guarantees.

Increase Average Merge Sort

- increase the elements of a list by the list's average:

$$[8, 4, 6] \xrightarrow{(+6)} [14, 10, 12]$$

- sort the output list:

$$[14, 10, 12] \xrightarrow{\textit{mergesort}} [10, 12, 14]$$

Initial solution:

1. compute the input list's sum and length;
2. implement merge-sort using a leaf tree that contains the numbers in the input list;
3. increase all elements by the list's average while sorting the increased values.

Initial program

```
incavgMS :: [Int] -> [Float]
incavgMS [] = []
incavgMS xs = incsort (ltreesumlen xs)


ltreesumlen :: [Int] -> (Tree, (Int, Int))
ltreesumlen [x] = (Leaf x, (x, 1))
ltreesumlen xs  =  let (xs1, xs2)       = splitl xs
                       (t1, (s1, l1)) = ltreesumlen xs1
                       (t2, (s2, l2)) = ltreesumlen xs2
                   in (Fork t1 t2, (s1 + s2, l1 + l2))


incsort  :: (Tree, (Int, Int)) -> [Float]
incsort (Leaf n, (s,l)) = [n + s / l]
incsort (Fork t1 t2, p) = merge (incsort (t1, p))
                                (incsort (t2, p))
```

Calculating the circular program

```
incavgMS xs
     = incsort (ltreesumlen  xs)

     = incsort (fst (ltreesumlen xs), snd (ltreesumlen xs))

     = incsort′ ∘ fst ∘ ltreesumlen $ xs
         where
           incsort′ t = incsort (t, (s,l))
           (s,l)      = snd (ltreesumlen xs)

     = fst ∘ (incsort′ × id) ∘ ltreesumlen $ xs
         where
           incsort′ t = incsort (t, (s,l))
           (s,l)      = snd (ltreesumlen xs)
```

Calculating the circular program (2)

```
incavgMS xs
   = ys
     where
       (ys, _)      = incavgMS' xs
       incavgMS'    = (incsort' × id) ∘ ltreesumlen $ xs
       incsort' t   = incsort (t, (s, l))
       (s,l)        = snd (ltreesumlen xs)
```

Calculating the circular program (3)

We can synthesize a recursive definition for `incavgMS'`:

```
incavgMS xs
   = ys
     where
       (ys, _)          = incavgMS' xs
       (s,l)            = snd (ltreesumlen xs)
       incavgMS' [x] = ([x + s/l], (x,1))
       incavgMS' xs  = let (xs1, xs2)        = splitl xs
                           (ys1, (s1,l1)) = incavgMS' xs1
                           (ys2, (s2,l2)) = incavgMS' xs2
                       in (merge ys1 ys2, (s1+s2, l1+l2))
```

Calculating the circular program (4)

Multiple traversal elimination:

```
snd ∘ ltreesumlen = snd ∘ incavgMS'
```

$$\Downarrow$$

```
incavgMS xs = ys
    where
      (ys, (s,l))    = incavgMS' xs
      incavgMS' [x] = ([x + s/l], (x,1))
      incavgMS' xs  = let (xs1, xs2)       = splitl xs
                          (ys1, (s1,l1)) = incavgMS' xs1
                          (ys2, (s2,l2)) = incavgMS' xs2
                      in (merge ys1 ys2, (s1+s2, l1+l2))
```

The method

```
incsort = pfold (hleaf,hfork)
   where
     hleaf n (s,l) = [n + s/l]
     hfork ys zs _ = merge ys zs
```

```
pfold :: (Int -> z -> a,a -> a -> z -> a) -> (Tree,z) -> a
pfold (h1,h2) = p
   where
     p (Leaf n,z)     = h1 n z
     p (Fork l r, z) = h2 (p l z) (p r z) z
```

The method (2)

```
ltreesumlen = g (Leaf,Fork)


g :: ∀ a. (Int -> a,a -> a -> a) -> [Int] -> (a,(Int,Int))
g (leaf,fork) [x]
            = (leaf x, (x, 1))
g (leaf,fork) xs
            =  let (xs1, xs2)       = splitl xs
                    (t1, (s1, l1)) = g (leaf,fork) xs1
                    (t2, (s2, l2)) = g (leaf,fork) xs2
               in (fork t1 t2, (s1+s2, l1+l2))
```

The method (3)

```
incavgMS xs
     = incsort (ltreesumlen  xs)

     = pfold (hleaf,hfork) ∘ g (Leaf,Fork) $ xs
       where hleaf n (s,l) = [n + s/l]
             hfork ys zs _ = merge ys zs

     = ys
       where (ys,(s,l))  = g (kleaf,kfork) xs
             kleaf n     = hleaf n (s,l)
             kfork ys zs = hfork ys zs (s,l)

     = ys
       where (ys,(s,l))  = g (kleaf,kfork) xs
             kleaf n     = [n + s/l]
             kfork ys zs = merge ys zs
```

The method (3)

```
incavgMS xs
     = incsort (ltreesumlen  xs)

     = pfold (hleaf,hfork) ∘ g (Leaf,Fork) $ xs
       where hleaf n (s,l) = [n + s/l]
             hfork ys zs _ = merge ys zs

     = ys
       where (ys,(s,l))  = g (kleaf,kfork) xs
             kleaf n     = hleaf n (s,l)
             kfork ys zs = hfork ys zs (s,l)

     = ys
       where (ys,(s,l))  = g (kleaf,kfork) xs
             kleaf n     = [n + s/l]
             kfork ys zs = merge ys zs
```

The method (3)

```
incavgMS xs
     = incsort (ltreesumlen  xs)

     = pfold (hleaf,hfork) o g (Leaf,Fork) $ xs
       where hleaf n (s,l) = [n + s/l]
             hfork ys zs _ = merge ys zs

     = ys
       where (ys,(s,l))  = g (kleaf,kfork) xs
             kleaf n     = hleaf n (s,l)
             kfork ys zs = hfork ys zs (s,l)

     = ys
       where (ys,(s,l))  = g (kleaf,kfork) xs
             kleaf n     = [n + s/l]
             kfork ys zs = merge ys zs
```

The method (3)

```
incavgMS xs
     = incsort (ltreesumlen  xs)

     = pfold (hleaf,hfork) o g (Leaf,Fork) $ xs
       where hleaf n (s,l) = [n + s/l]
             hfork ys zs _ = merge ys zs

     = ys
       where (ys,(s,l))   = g (kleaf,kfork) xs
             kleaf n      = hleaf n (s,l)
             kfork ys zs = hfork ys zs (s,l)

     = ys
       where (ys,(s,l))   = g (kleaf,kfork) xs
             kleaf n      = [n + s/l]
             kfork ys zs = merge ys zs
```

Shortcut fusion: pfold/buildp rule

```
pfold (hleaf,hfork) ∘ buildp g $ c
    = v
      where
        (v,z)       = g (kleaf,kfork)
        kleaf n     = hleaf n z
        kfork l r = hleaf l r z
```

```
buildp :: (∀ a. (Int -> a,a -> a -> a) -> c -> (a,z))
          -> c -> (Tree,z)
buildp g = g (Leaf,Fork)
```

fold/buildp

$$(\texttt{fold (kleaf,kfork)} \times \texttt{id}) \circ \texttt{buildp g} = \texttt{g (kleaf,kfork)}$$

```
fold :: (Int -> a,a -> a -> a) -> Tree -> a
fold (k1,k2) = f
  where
    f (Leaf n)     = k1 n
    f (Fork l r) = k2 (f l) (f r)
```

Relationship pold-fold

```
pfold (hleaf,hfork) (t,z) = fold (kleaf,kfork) t
   where
     kleaf n     = hleaf n z
     kfork l r = hfork l r z
```

```
pfold (h1,h2) = p
   where
     p (Leaf n,z)      = h1 n z
     p (Fork l r, z) = h2 (p l z) (p r z) z
```

```
fold (k1,k2) = f
   where
     f (Leaf n)     = k1 n
     f (Fork l r) = k2 (f l) (f r)
```

Essential law

```
snd ∘ g (Leaf,Fork) = snd ∘ g (hleaf,hfork)
```

```
g :: ∀ a. (Int -> a,a -> a -> a) -> c -> (a,z)
```

The proof

```
pfold (hleaf,hfork) ∘ buildp g $ c

    = pfold (hleaf,hfork) ∘ g (Leaf,Fork) $ c

    = pfold (hleaf,hfork) (fst ∘ g (Leaf,Fork) $ c,
                           snd ∘ g (Leaf,Fork) $ c)

    = fold (kleaf,kfork) ∘ fst ∘ g (Leaf,Fork) $ c
      where
          z          = snd ∘ g (Leaf,Fork) $ c
          kleaf n    = hleaf n z
          kfork l r  = hfork l r z

    = fst ∘ (fold (kleaf,kfork) × id) ∘ g (Leaf,Fork) $ c
```

The proof

```
pfold (hleaf,hfork) ∘ buildp g $ c

    = pfold (hleaf,hfork) ∘ g (Leaf,Fork) $ c

    = pfold (hleaf,hfork) (fst ∘ g (Leaf,Fork) $ c,
                           snd ∘ g (Leaf,Fork) $ c)

    = fold (kleaf,kfork) ∘ fst ∘ g (Leaf,Fork) $ c
      where
          z          = snd ∘ g (Leaf,Fork) $ c
          kleaf n    = hleaf n z
          kfork l r  = hfork l r z

    = fst ∘ (fold (kleaf,kfork) × id) ∘ g (Leaf,Fork) $ c
```

The proof

```
pfold (hleaf,hfork) ∘ buildp g $ c

    = pfold (hleaf,hfork) ∘ g (Leaf,Fork) $ c

    = pfold (hleaf,hfork) (fst ∘ g (Leaf,Fork) $ c,
                            snd ∘ g (Leaf,Fork) $ c)

    = fold (kleaf,kfork) ∘ fst ∘ g (Leaf,Fork) $ c
        where
            z          = snd ∘ g (Leaf,Fork) $ c
            kleaf n    = hleaf n z
            kfork l r  = hfork l r z

    = fst ∘ (fold (kleaf,kfork) × id) ∘ g (Leaf,Fork) $ c
```

The proof

```
pfold (hleaf,hfork) ∘ buildp g $ c

    = pfold (hleaf,hfork) ∘ g (Leaf,Fork) $ c

    = pfold (hleaf,hfork) (fst ∘ g (Leaf,Fork) $ c,
                           snd ∘ g (Leaf,Fork) $ c)

    = fold (kleaf,kfork) ∘ fst ∘ g (Leaf,Fork) $ c
      where
         z          = snd ∘ g (Leaf,Fork) $ c
         kleaf n    = hleaf n z
         kfork l r  = hfork l r z

    = fst ∘ (fold (kleaf,kfork) × id) ∘ g (Leaf,Fork) $ c
```

The proof (2)

```
= fst ∘ g (kleaf,kfork) $ c
    where
      z            = snd ∘ g (Leaf,Fork) $ c
      kleaf n      = hleaf n z
      kfork l r    = hfork l r z

= fst ∘ g (kleaf,kfork) $ c
    where
      z            = snd ∘ g (kleaf,kfork) $ c
      kleaf n      = hleaf n z
      kfork l r    = hfork l r z

= v
    where
      (v,z) = g (kleaf,kfork) c
      kleaf n    = hleaf n z
      kfork l r  = hfork l r z
```

The proof (2)

```
= fst ∘ g (kleaf,kfork) $ c
    where
      z            = snd ∘ g (Leaf,Fork) $ c
      kleaf n      = hleaf n z
      kfork l r    = hfork l r z

= fst ∘ g (kleaf,kfork) $ c
    where
      z            = snd ∘ g (kleaf,kfork) $ c
      kleaf n      = hleaf n z
      kfork l r    = hfork l r z

= v
    where
     (v,z) = g (kleaf,kfork) c
     kleaf n     = hleaf n z
     kfork l r = hfork l r z
```

The proof (2)

```
= fst ∘ g (kleaf,kfork) $ c
    where
       z          = snd ∘ g (Leaf,Fork) $ c
       kleaf n    = hleaf n z
       kfork l r = hfork l r z

= fst ∘ g (kleaf,kfork) $ c
    where
       z          = snd ∘ g (kleaf,kfork) $ c
       kleaf n    = hleaf n z
       kfork l r = hfork l r z

= v
    where
      (v,z) = g (kleaf,kfork) c
      kleaf n    = hleaf n z
      kfork l r = hfork l r z
```

Conclusions

- Calculational approach to circular programming
  1. Intermediate Structure Deforestation
  2. Multiple Traversal Elimination
- Our Calculational Rule is in the style of shortcut fusion
  1. Easy to apply
  2. Effective
  3. Proved correct

Conclusions

- Calculational approach to circular programming
  1. Intermediate Structure Deforestation
  2. Multiple Traversal Elimination
- Our Calculational Rule is in the style of shortcut fusion
  1. Easy to apply
  2. Effective
  3. Proved correct

Conclusions

- Like the usual fold/build rule, our rule can also be implemented in GHC using the RULES pragma (rewrite rules).
- Bad news: In Haskell our rule we presented is morally correct only.
  - surjective pairing is not valid in Haskell due to the presence of lifted products: $\bot \neq (\bot, \bot)$
  - and this property is an essential step in the proof of the rule

Conclusions

- Like the usual fold/build rule, our rule can also be implemented in GHC using the RULES pragma (rewrite rules).
- Bad news: In Haskell our rule we presented is morally correct only.
  - surjective pairing is not valid in Haskell due to the presence of lifted products: $\perp \neq (\perp, \perp)$
  - and this property is an essential step in the proof of the rule