

Stream Fusion

Wouter Swierstra

30/11/2010

Exercise:

Write a function that sums the square of all the odd integers between two arguments m and n .

Haskell solution

```
f : (Int,Int) -> Int
```

```
f = sum . map square . filter isOdd . between
```

where

```
square x = x * x
```

```
isOdd x = x % 2 == 1
```

```
between (m,n) = [m .. n]
```

Haskell solution

```
f : (Int,Int) -> Int
```

```
f = sum . map square . filter isOdd . between
```

where

```
square x = x * x
```

```
isOdd x = x % 2 == 0
```

```
between (m,n) = [m .. n]
```



**Each intermediate computation
creates an additional list**



Pseudocode analogue

```
int a[]; int b[]; int c[]; int d[];
for (int i = 0; i < N; i++)
    a[i] = i
for (int j = 0; j < N; j++)
    if (a[j] % 2 == 0)
        then b[j] = a[j]
        else b[j] = 0
for (int k = 0; k < N; k++)
    c[k] = b[k] * b[k]
```

...

A “better” solution

```
f : (Int,Int) -> Int
```

```
f (m,n) = recurse m
```

where

```
recurse m =
```

```
  if m > n then 0
```

```
  else let rest = recurse (m + 1)
```

```
    if isOdd m then square m + rest
```

```
    else rest
```

Composability



Efficiency

Yes – it is a problem

- Slightly changing the specification needs a complete rewrite/copy-paste of the original solution;
- Despite existing optimization techniques, the solution I will present gives significant performance improvements (up to 50% speedup on certain benchmarks).

Problem:

How can we write
programs that are both
efficient and *modular*?

Solution

- Represent constituent traversals as explicit *folds* and *unfolds*;
- Exploit their universal properties to *fuse* multiple traversals into one;
- (Teach the Haskell compiler – GHC – to do this for you.)

Algebras

- Given a functor $F : \mathcal{C} \rightarrow \mathcal{C}$, an F -algebra is a pair (a, A) where
 - A an object in \mathcal{C} and
 - $a : FA \rightarrow A$ is a morphism in \mathcal{C} .

$$\begin{array}{c} FA \\ \downarrow a \\ A \end{array}$$

Fixed points

- You can form a category of F -algebras;
- If it has an initial object, call it the *least-fixed point* of F , usually written $(in, \mu F)$

$$\begin{array}{ccc} F(\mu F) & \dashrightarrow & FA \\ \downarrow in & & \downarrow a \\ \mu F & \dashrightarrow & A \end{array}$$

Folds

- As the μF is initial, any algebra gives rise to a unique morphism from the least fixed-point to that algebra.

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{F(\text{fold } a)} & F A \\ \downarrow \text{in} & & \downarrow a \\ \mu F & \xrightarrow{\text{fold } a} & A \end{array}$$

Example: sum a list

We can represent integer lists as the least-fixed point of the functor:

$$L a = \text{Int} \times a + \text{I}$$

Or in Haskell:

```
data L a = Cons Int a | Nil
```

Example: sum a list

Taking the fixed point of the functor gives the “usual” lists:

```
data L a = Cons Int a | Nil
```

```
data Fix f = In (f (Fix f))
```

```
type List = Fix L
```

Example: sum a list

An example *L*-algebra that sums a list:

```
data L a = Cons Int a | Nil
```

```
sumAlg : L Int -> Int
```

```
sumAlg Nil = 0
```

```
sumAlg (Cons x sum) = x + sum
```

Example: sum a list

And finally, use this algebra to fold over the list, computing the sum:

```
data L a = Cons Int a | Nil
fold : (L a -> a) -> List -> a
fold f (In t) = f (mapL (fold f) t)
sum : List -> Int
sum = fold sumAlg
```

...and all this dualizes to
coalgebras, greatest-
fixed points, and unfolds.

Example: unfolding

Use an unfold to generate a list of all numbers between two integers n and m :

```
data L a = Cons Int a | Nil
```

```
unfold : (a -> L a) -> a -> List
```

```
betweenAlg : (Int,Int) -> L (Int,Int)
```

```
betweenAlg (m,n) = if n > m then Nil  
                else Cons m (m + 1,n)
```

A technical point

- Most programming languages do not distinguish between least and greatest fixed points.
- Haskell’s “ambient category” – CPO_{\perp} – is algebraically compact and identifies the two, which some might consider a bit of a theoretical “hack”.

Recursive coalgebras

A co-algebra (C,c) is said to be *recursive* if for every algebra (A,a) the equation:

$$h = a . F h . c$$

has a unique solution for h , written *hylo* (a,c) .

Such h (sometimes called a *hylomorphisms*) capture common divide-and-conquer algorithms.

Hylo fusion

From the universal property of hylomorphisms, we can derive for all algebras a and recursive coalgebras c :

$$\mathit{fold} \ a \ . \ \mathit{unfold} \ c \ = \ \mathit{hylo} \ (a,c)$$

Applying this rule right-to-left is called *fusion*, as two traversals are fused into one, getting rid of an intermediate data type.

Example: interval sum

Can we sum all the numbers between two arguments n and m ?

First attempt:

```
fold sumAlg . unfold betweenAlg
```

But this creates an unnecessary intermediate list...

Example: interval sum

But fold/unfold law we saw previously guarantees the existence of a hylomorphism, that computes the sum directly:

```
intervalSum : (Int,Int) -> Int
```

```
intervalSum (m,n) =
```

```
  if m > n then 0
```

```
  else m + intervalSum (m + 1,n)
```

Perspective

- So what? We already could have written that solution directly.
- Compilers tend to be very good at optimizing non-recursive functions.
- Writing functions as a composition of folds and unfolds, can generate more opportunities for optimization.

Stream fusion

- Instead of writing functions over (infinite) lists directly, write functions over their coalgebraic representation.
- (I should point out, it's not really about streams but lazy lists.)

Example: map

`map : (a -> b) -> List a -> List b`

`map f Nil = Nil`

`map f (Cons x xs) =`

`Cons (f x) (map f xs)`

Example: map

`map : (a -> b) -> List a -> List b`

`map f Nil = Nil`

`map f (Cons x xs) =`

`Cons (f x) (map f xs)`

How we teach our compiler that

`map f . map g = map (f . g) ?`

List Coalgebras

```
data LC a =  $\exists$  s. (Step s a)  $\times$  s
```

```
data Step s a =
```

```
    Done
```

```
  | Yield (a  $\times$  s)
```

To-and-fro

It's fairly straightforward to convert between the coalgebra representation of lists and the usual inductive definition:

```
toLC : List a -> LC a
```

```
fromLC : LC a -> List a
```

Non-recursive map

```
map : (a -> b) -> List a -> List b
```

```
map f = fromLC . mapLC f . toLC
```

```
mapLC f (next, s) = (next', s)
```

where

```
next' Done = Done
```

```
next' (Yield (x, s')) =
```

```
  Yield (f x, s')
```

Fusing traversals

A simple calculation teaches us:

`map f . map g ≡`

`fromLC . mapLC f . toLC . fromLC . mapLC g . toLC`

Fusing traversals

A simple calculation teaches us:

```
map f . map g ≡  
fromLC . mapLC f . toLC . fromLC . mapLC g . toLC
```

Optimization stuck behind recursive functions

Cunning plan

- If only we could teach the compiler that:

$$\forall c . \text{toLC} (\text{fromLC } c) = c$$

we might be able to trigger more optimization.

- Using GHC's rewrite rules, we can achieve just this!

Fusing traversals

Calculating again:

```
map f . map g ≡  
fromLC . mapLC f . toLC . fromLC . mapLC g . toLC ≡  
fromLC . mapLC f . mapLC g . toLC ≡ (compiler magic)  
fromLC . mapLC (f . g) . toLC ≡  
map (f . g)
```

Result!

- At the cost of converting to-and-fro between (infinite) lists and their coalgebraic representation, we can expose more optimization opportunities to the compiler.
- But does this always work?

Filter

```
filter :  
  (a -> Bool) -> List a -> List a  
filter p Nil = Nil  
filter p (Cons x xs) =  
  if p x then Cons x (filter p xs)  
  else filter p xs
```

Filter

```
filter :  
  (a -> Bool) -> List a -> List a
```

```
filter p  
  = fromLC . filterLC p . toLC
```

```
filterLC p (next,s) = (next',s)
```

where

```
next' Done = Done
```

```
next' (Yield (x,s')) = ...
```

Filter

```
filterLC p (next,s) = (next',s)
```

```
  where
```

```
    next' (Yield (x, s')) =
```

```
      if p x then Yield (x, s')
```

```
      else filterLC s'
```

Filter

```
filterLC p (next,s) = (next',s)
```

where

```
next' (Yield (x, s')) =
```

```
  if p x then Yield (x, s')
```

```
  else filterLC s'
```

This function is recursive!

Stuttering

```
data LC a =  $\exists$  s. (Step s a)  $\times$  s
```

```
data Step s a =
```

```
    Done
```

```
  | Yield a  $\times$  s
```

```
  | Skip s
```

Filter – revisited

```
next' (Skip s') = Skip s'  
next' (Yield (x, s')) =  
  if p x then Yield (x, s')  
  else Skip s'
```

Filter – revisited

```
next' (Skip s') = Skip s'  
next' (Yield (x, s')) =  
    if p x then Yield (x, s')  
    else Skip s'
```

This function is no longer recursive!

Taking stock

- These ideas have made their way into an alternative implementation of Haskell's list library.
- Performance is 'usually better' than lists.
- Runs on co-algebraic technology.
- Many similar optimization techniques have a solid theoretical justification (foldr/build fusion, deforestation, ...)

References

- *Theory and Practice of Fusion*; Ralf Hinze, Thomas Harper, and Daniel James.
- *Stream Fusion: from Lists to Streams to Nothing at All*; Duncan Coutts, Roman Leshchinskiy, Don Stewart.