Two Tricks for Program Optimization

Difference lists and the Codensity monad

Mio Alter

January 13, 2016

Overview

Who am I?

Places, things, and things

1. `NY -> Paris -> NY -> Chicago -> NY -> Austin -> SF`
2. `Cinema Studies -> Math -> ML Data Scientist (@6sense)`
3. `Movies -> Cooking -> Dogs (standard poodles)`

Shape of Things to Come

Tricks for program optimzation

1. Replace mon(oid|ad) with mon(oid|ad) of functions
2. Replace recursive constructions with function composition/application
3. `rep :: m -> mm`
   `abs :: mm -> m`
4. `abs . rep = id`
5. `rep (x * y) = (rep x) @ (rep y)`

Reversing a list

Suppose we want to reverse a list

```
rev :: [a] -> [a]
rev [] = []
rev (x:xs) = rev xs ++ [x]
```

Reversing a list

Suppose we want to reverse a list

```
rev :: [a] -> [a]
rev [] = []
rev (x:xs) = rev xs ++ [x]
```

The problem is

```
(++) :: [a] -> [a] -> [a]
(++) [] ys = ys
(++) (x:xs) ys = x : xs ++ ys
```

is O(length of the first list)

Reversing a list: complexity

Telescoping work

```
rev [1..3]
= (rev [2,3]) ++ [1]
= ((rev [3]) ++ [2]) ++ [1]
= (((rev []) ++ [3]) ++ [2]) ++ [1]
= (([] ++ [3]) ++ [2]) ++ [1] -- 0 steps
= ([3] ++ [2]) ++ [1] -- 1 step
= ([3,2]) ++ [1] -- 2 steps
= [3,2,1]
```

so reversing a list of length n takes

$$0 + 1 + 2 + ... + n - 1 = \frac{(n-1)n}{2} \sim n^2$$

steps.

Definition of difference list

Hughes' idea: make a new datatype and functions to map there and back

```
type EList a = [a] -> [a]

rep :: [a] -> EList a
rep xs = (xs ++)

abs :: EList a -> [a]
abs f = f []
```

which satisfy

```
abs . rep = id
rep (xs ++ ys) = rep xs . rep ys
```

Fast reverse

Now, we can define

```
rev' :: [a] -> EList a
rev' [] = rep []
rev' (x:xs) = rev' xs . rep [x]
```

and

```
fastReverse :: [a] -> [a]
fastReverse = abs . rev'
```

so

```
fastReverse xs = rev' xs []
```

List vs. Difference List

We replaced appending

```
rev :: [a] -> [a]
rev [] = []
rev (x:xs) = rev xs ++ [x]
```

with function composition

```
rev' :: [a] -> EList a
rev' [] = rep []
rev' (x:xs) = rev' xs . rep [x]
```

which is $O(1)$.

Difference Lists: Why? How?

Why would anyone think to replace list appending with function composition?

Difference Lists: Why? How?

Why would anyone think to replace list appending with function composition? Both are monoids

```
instance Monoid [a] where
        mempty = []
        mappend = (++)

instance Monoid (EList a) where
        mempty = id
        mappend = (.)
```

Difference Lists: Why? How?

Why would anyone think to replace list appending with function composition? Both are monoids

```
instance Monoid [a] where
        mempty = []
        mappend = (++)

instance Monoid (EList a) where
        mempty = id
        mappend = (.)
```

and

```
rep (xs ++ ys) = rep xs . rep ys
```

is a monoid homomorphism.

Cayley's Theorem

Theorem (Cayley)

Every monoid is isomorphic to a submonoid of its monoid of endomorphisms.

1. (Endomorphisms are just functions from a thing to itself)
2. Meaning: given a monoid m, the functions $m \to m$ are also a monoid, and we can *monoid*-embed m into $m \to m$.
3. This is from 1854.

Difference lists review

Not a novel solution, not the best solution, but has good ideas

1. Replace monoid with monoid of functions
2. Replace appending with function composition
3. ```
   rep xs = mappend xs
   abs f = f mempty
   ```
4. `abs . rep = id`
5. `rep (mappend x y) = mappend (rep x) (rep y)`
   ```

A tree monad

Substitution makes a tree a monad

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

subst :: Tree a -> (a -> Tree b) -> Tree b
subst (Leaf x) k = k x
subst (Node l r) k = Node (subst l k) (subst r k)

instance Monad Tree where
        return = Leaf
        (>>=) = subst
```

Growing a tree

Helper functions to grow trees

```
sprout :: Int -> Int -> Tree Int
sprout n = \i -> Node (Leaf (n - 1 - i)) (Leaf (i+1))

fullTree :: Int -> Tree Int
fullTree 1 = Leaf 1
fullTree n = fullTree (n-1) >>= sprout n
```

A length-n computation

```
zigzag :: Tree Int -> Int
zigzag = zig
       where
               zig (Leaf n) = n
               zig (Node l r) = zag l
               zag (Leaf n) = n
               zag (Node l r) = zig r
```

```
zigzag (fullTree 3)
= zigzag (Leaf 1 >>= sprout 2 >>= sprout 3)
```

which is $O(n^2)$

Codensity monad

The codensity monad in general

```
type CodT m a = forall z . (a -> m z) -> m z

instance Monad (CodT m) where
        return x = \c -> c x
        f >>= g = \c -> f (\x -> g x c)
```

and for us

```
type Coden a = CodT Tree a
```

Codensity rep and abs

Getting there and back

```
rep :: Tree a -> Coden a
rep t = (t >>=)

abs :: Coden a -> Tree a
abs p = p return -- Remember: return = Leaf!
```

Clearly

```
abs . rep = id
```

Codensity "type constructors"

We can make functions like our Tree type constructors

```
leaf :: a -> Coden a
leaf = return

node :: Coden a -> Coden a -> Coden a
node p q = \h -> Node (p h) (q h)
```

Codensity homomorphism property

We can make functions like our Tree type constructors

```
leaf :: a -> Coden a
leaf = return

node :: Coden a -> Coden a -> Coden a
node p q = \h -> Node (p h) (q h)
```

and we get our homomorphism property

```
rep (Leaf i) = leaf i
rep (t >>= f) = rep t >>= (rep . f)
rep (Node l r) = node (rep l) (rep r)
```

Growing a Coden

Helper functions to grow Codens

```
sproden :: Int -> Int -> Coden Int
sproden n = \i -> node (leaf (n - 1 - i)) (leaf (i + 1))

fullCoden :: Int -> Coden Int
fullCoden 1 = leaf 1
fullCoden n = fullCoden (n-1) >>= sproden n
```

Codensity punchline

Now zigzagging is $O(n)$

```
rep (fullTree n) = fullCoden n
abs (fullCoden n) = fullTree n

fullCoden 3
= \h -> Node (Node (h 2) (h 1)) (Node (h 0) (h 3))

zigzag (fullTree 3)
= zigzag (abs (fullCoden 3))
= zigzag (abs (\ h -> Node (Node (_) (h 1)) (_) ))
= zigzag (Node ((_) (Leaf 1)) (_))
```

Similarities

Same trick

1. Replace mon(oid|ad) with mon(oid|ad) of functions
2. Replace recursive constructions with function composition/application
3. ```
   rep :: m -> mm -- (xs ++) vs. (t >>=)
   abs :: mm -> m -- f [] vs. f return
   ```
4. `abs . rep = id`
5. `rep (x * y) = (rep x) @ (rep y)`
   ```

Differences

But what about

1. `m -> m` vs. `forall z . (a -> m z) -> m z` ?
2. monoid vs. monad ?

Differences?

Wait a minute!

1. why not just

   `forall z . m z -> m z ?`
2. how do we make "functions from a functor to itself" a functor?
3. is "a monad *really* just a monoid in the category of endofunctors"?

So much more

We still have all this to do!

1. Categories, functors (polymorphic datatypes), and natural transformations (polymorphic functions)
2. Two categories: category of datatypes, category of functors
3. Of monoids and monads
4. Currying/uncurrying in the category of functors
5. Yoneda Lemma
6. Derive $\forall z.(a \to mz) \to mz$ from these

References

See here

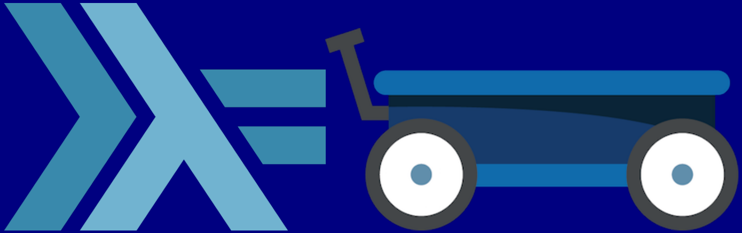
1. Rivas and Jaskelioff ▸ Notions of Computations as Monoids
2. Hughes ▸ A Novel Representation of Lists and its Application to the Function "Reverse"
3. Voigtlander ▸ Asymptotic Improvement of Computations over Free Monads
4. Hutton, Jaskelioff, and Gill ▸ Factorising Folds for Faster Functions
5. Hinze ▸ Kan Extensions for Program Optimisation Or: Art and Dan Explain an Old Trick

Still good tricks!

Nonetheless, good stuff!

1. Replace mon(oid|ad) with mon(oid|ad) of functions
2. Replace recursive constructions with function composition/application
3. `rep :: m -> mm`
   `abs :: mm -> m`
4. `abs . rep = id`
5. `rep (x * y) = (rep x) @ (rep y)`

Thanks!



Thanks!