Super combinators

Frank Wijmans

March 28, 2012

1 Rewriting Haskell basics in lambda-calculus

1.1 Functional representations of data structures

We can write data structures like functions, we gain a continuation programming style.

```
data Pair a b = Pair a b
pair a b = \\f. f a b -- more intuitive
pair a b f = f a b

fst(pair a b) = a
fst pab = pab (\ a b. a)
snd pab = pab (\ a b. b)
```

Normally we would ask a pair for a first or second element. `fst` calls the function `f` in `pair`. This is a continuation function, pair 'stores' values and returns them to the function `f`. Each example begins stating the normal data structure

```
data Bool = True | False

if c t e
true = \t e. t
false = \t e. e

if true  t e => t
if false t e => e
```

```
data Mayba a = Just a | Nothing

just  a j n = \f j a
nothing j n = \f n

--example:
case ma of
  Nothing = e1
  Just = e2

maybe ma e1 e2 = ma e1 e2
```

They are equal, but e2 actually is a function that takes an a, because the Just carries an a that is still in scope. This can also be done for lists.

```
data List a = Cons a (List a) | Nil

cons a as c n = c a as
nil       c n = n
```

The steps you take are, first, you create functions out of the constructors. Secondly you give the functions the arguments they need. Next you add arguments containing functions for every constructor. Then apply the functions with the arguments.

1.2 Remove (recursive) let/in structures

```
let x = e1 in e2 = (\xdef. e2) e1

Recursive let
x = e1 in e2 -- in e1 an x is bound.

let x = .. x ..

let x = rx
                where rx = x rx | rx = fix rx
                          x x = .. x ..
                                        ...(...x...)...

fix x = let rx = x rx
        in rx
```

In recursive let constructions you need to rewrite the x such that it is not recursive. Fix ix a recursive pattern, that will evaluate a part of it self plus itself. We can write fix as a lambda expression.

$(\lambda x.xx)(\lambda x.xx)$ rewrites to itself. Another expression will rewrite to itself: $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))F$ and gives back the F. Recursion over the F

```
fix f = \xdef. f(xx)(\xdef.f(xx))
```

Recursion can be written in many ways, it's normally named Y (capital).

In lambda calculus we can formalize all the data structures we use in Haskell.

2 Combinator Reduction SKI

In the following example the y+2 is already bound by the lambda.

```
(\x .. x x x) ()
(\x \y   x     x) (y+2)
   (\y   y+2   y+2)
```

This can be done by beta reduction.

But SKI combinators can build lambda expressions without having to compute all parts of an expression, but combine them together.

Now we will rewrite lambda expressions, such that it doesn't contain a lambda. Using a data structure, we can rewrite an example lambda.

I for Identity, K for constant and S is

```
data Lambda = App Lambda Lambda
                          | Var String
                          | Lambda String Lambda
```

```
\x. e1
\x. x -> I
\x. y -> K y
K y x = y
```

When $e1$ is rewritten e1 contains no variable or lambda, so it is a application $e1e2$. Next we need to substitute the x in $e1$.

```
e1 = (e1 e2)
\x. e1 = \x. ((\x. e1) x) ((\x. e2) x)
               ( re1   )      ( re2   )

(\f g x. (f x) (g x)) re1 re2
```

For a function that takes the n^{th} element from a list, we have the following.

```
el n = \l. if (=n 1) (hd l) (el (-n 1) (tl l))
       = S (\l if (=n i) (hd l)) (\l el (-n 1) (tl l))
           (S (\l if (=n 1)) (\l hd l)) (S (\l. el (-n 1)) (\l tl l))

       (\l. hd l) = (S (\l. hd) (\l l))
                                (S (K hd) I)
```

By rewriting all the way, you see that you do much work, and at the end you throw it away. The expression blows up if you do it this way. You will create a tree of S K I. S means send the argument in the function and the argument part. K means don't use it, it's not needed. I means, here you can use the lambda, apply it.

2.1 SKI ⇒ G

2.2 SKI-BC-S'B'C'

$Bfgx = f(gx)$ Remember that C stands not for compose, so B does. B has a function and a argument part, and the x is only needed in the argument part. And c is equal to flip, $Cfgx = fxg$. C takes an x, and applies it to the f, instead of a g. B and C are like each other.

NExt, the `app` is a smart constructor.

```
data Lambda = S | K | I | B | C | App L L

S K K x = (K x) (K x) = x
S K K = I

app S (K f) (
app      (App (K f)))(K g) = App K (App f g)

app (App (S (K f) g)) = App (App B f) g [??]
```

```
app (App (S f (K g))) = App f (App C g) [??]
```

In other words you can use R and L for B and C respectively. R for right, because the argument goes into the right, and with L the argument goes to the left.

Expressions grow exponentially because the transformations require that all arguments are passed to all parts of the expression, even though you don't need it. Introducing the S'

```
S' c f g x = c (f x) (g x)
B' c f g x = c f     (g x)
C' c f g x = c (f x)  g
```

The c argument is a constant part of the expression. A director string is the part of combinators that states the number of arguments times the number of application nodes in the graph. This string tells evaluation which path to take.

```
foldr op e []     = e
                (x:Xs) = x `op` foldr op e xs

sum = foldr (+) 0

sum [] = 0
sum (x:xs) = x + (sum xs)
```

foldr passes three parameters, the 'normal' way only passes one.

```
foldr op e = go
  where go [] = e
              go (x:xs) = x `op` go xs

--using combinators
foldr = \op e. Y go'

go' go = \l. if (nil l) e (op (hd l) (go (tl l)))
           = S (L (R if nil) e) (S (R op hd) (R go tl))

--take out go

go' = R' S (L (R if nil) e) (R' (S (R op hd))(L' (R I tl)))
```

Doing graph rewriting this way, you get self optimizing programs. In Haskell you want to program as abstractly as possible, but you don't want to pay for abstraction. Clean uses partial evaluation and graph rewriting to evaluate a code.

```
Y f = f (Y f)
```

2.3 Super Combinator

Define a special purpose set of combinators accustomed to the program at hand.

```
\x. x .. x .. x .. x
```

These x's are reachable by subtrees in the expression, and some trees have no x inside them. In the graph, the substitution is visible, you see parts where x's are not needed. The sub expressions without x named e_1, e_2, e_n. A function that will build a graph with x's given the parts without x.

```
f x = spnx e1 e2 en x
```

Not all combinators are interpreted, but the substitution is clear. By looking at free expressions, expressions that are not using an x, you can make a function that takes free expressions and substitutes x between them. The free expressions don't depend on a argument, so they don't have to be evaluated as if the were using it. The function is always the same, given any argument.

What about ordering e1 e2 en..?