CPSC 417: Midterm Exam Answer Sheet

Robin Cockett

November 3, 2005

PART I: Haskell

1. The fold and map functions for Rose trees are:

```
data Rose a = Rs [(a,Rose a)]
  deriving (Show, Eq)

foldRose:: ([(a,b)] -> b) -> (Rose a) -> b
foldRose f (Rs xs) = f (map (\(a,r) -> (a,foldRose f r)) xs)

mapRose::(a -> b) -> (Rose a) -> (Rose b)
mapRose f = foldRose (\k -> Rs (map (\(a,x) -> (f a,x)) k))
```

2. The Rose tree of tails of a list:

```
tailRose::[a] -> (Rose a)
tailRose = foldr
           (\a (Rs branches) -> Rs ((a,Rs branches):branches))
           (Rs [])
```

3. The height of a Rose tree:

```
maxInList:: [Int] -> Int
maxInList = foldr max 0

maxRoseHgt:: (Rose a) -> Int
maxRoseHgt = foldRose (maxInList.(map (\(_,x) -> x+1)))
```

4. The challenge: the substitution monad for Rose trees together with a filter function:

```
instance Monad Rose where
      return x = Rs [(x,Rs [])]
      (Rs xs) >>= f = Rs [(b,follow t branches)
                         | (a,Rs branches) <- xs
                         , (b,t) <- strip (f a)] where
```

```
          strip (Rs bs) = bs
          follow (Rs []) branches = Rs branches >>= f
          follow (Rs xs) branches
                 = Rs (map (\(b,bs) -> (b,follow bs branches)) xs)


filterRose::(a -> Bool) -> (Rose a) -> (Rose a)
filterRose p t =
    do a <- t
       if p a then return a
              else (Rs [])
```

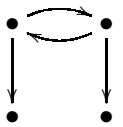
PART II: Rewriting

1. They are all false!

   (a) "If a rewriting system is locally confluent but it is not terminating then it cannot be (globally) confluent."
   Consider the rewriting system of β-reduction on the λ-calculus: we know it is confluent but it certainly is not terminating.

   (b) "If a rewriting system has elements which do not have a normal form it cannot be confluent."
   Again consider the rewriting system of β-reduction on the λ-calculus: we know it is confluent but Ω, for example, has no normal form.

   (c) "In a rewriting system which is locally confluent if an element has a normal form it must be unique."
   Consider the rewriting system (from the notes):

   this is locally confluent but does not have unique normal forms.

   (d) "A rewriting system in which every object has a normal form must be terminating."
   Consider the β-reduction system for the term $(\lambda xy.x)\mathsf{True}\Omega$ it does have a normal form, namely True but it is not terminating.

   (e) "Rewriting systems which are modulo equations are always confluent."
   You would be so lucky! Try this one:

   This is the same as the above but identifies the two-way rewriting: it still is not confluent!

   (f) "If a rewriting system is confluent one can decide the equality of elements."
   Well ... the confluence of β-reduction in the λ-calculus failed to have this beneficial effect! The problem is that simply being able to rewrite things into a common form does not tell you when one can stop the attempt to find a common form (unless you reach a normal form).

2. The term rewriting system is

$$\begin{aligned} x + 0 &\longrightarrow x && (1)\\ 0 + x &\longrightarrow x && (2)\\ (x + y) + z &\longrightarrow x + (y + z) && (3) \end{aligned}$$

   We start by showing it is terminating. Define a weight function as follows:

$$\begin{aligned} w(0) &= 1\\ w(x) &= 1\\ w(t + t') &= 2w(t) + w(t') \end{aligned}$$

We have to show that this weight function always strictly reduces across a term and so acts as a bound on the number of reductions which can be applied to the term. Note that $w(t) \geq 1$ always.

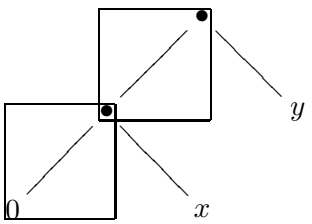
$$\begin{aligned} w(t+0) &= 2w(t)+1 > w(t) \\ w(0+t) &= 2+w(t) > w(t) \\ w((t_1+t_2)+t_3) &= 2(2(w(t_1)+w(t_2))+w(t_3) = 4w(t_1)+2w(t_2)+w(t_3) \\ &> 2w(t_1)+w(t_2)+w(t_3) = w(t_1+(t_2+t_3)) \end{aligned}$$

Clearly this is a very generous bound. An interesting question is: what is the minimum bound? Can you prove that the following function provides the minimum bound for this rewriting system

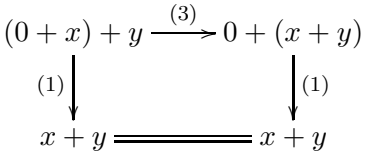
$$\begin{aligned} w(t_1+t_2) &= w(t_1)+w(t_2)+1 \quad \text{when } t_1 = t_{11}+t_{12} \\ w(t_1+t_2) &= w(t_1)+w(t_2) \qquad \text{otherwise} \\ w(0) &= 1 \\ w(x) &= 0 \qquad x \text{ is a variable} \end{aligned}$$

At this stage we know the rewriting system is strongly terminating so it remains to look at the *critical pairs* that is minimal terms in which there are overlapping redexes. It is useful here to visualize the terms as trees so that one can see what is going on. There are five critical pairs:

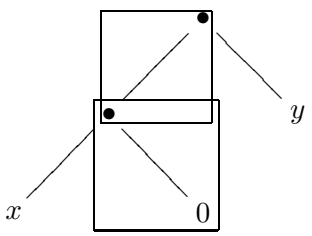
(a) Rules (1) and (3) have a critical pair arising from the following overlapping redexes:



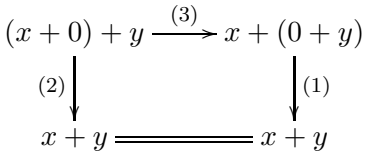
This critical pair is resolved as follows:



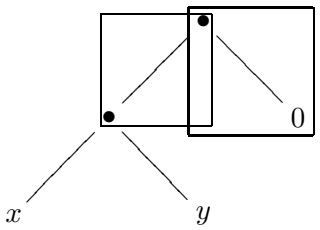
(b) Rules (2) and (3) have a first critical pair arising from the following overlapping redexes:



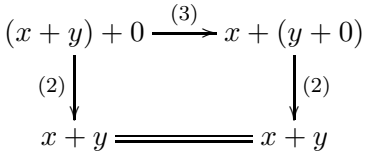
This critical pair is resolved as follows:



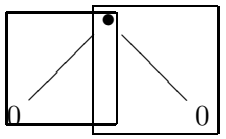
(c) Rules (3) and (2) have a critical pair arising from the following overlapping redexes:



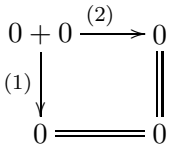
This critical pair is resolved as follows:



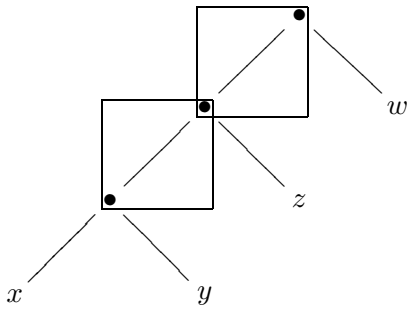
(d) Rules (2) and (1) have a critical pair arising from the following overlapping redexes:



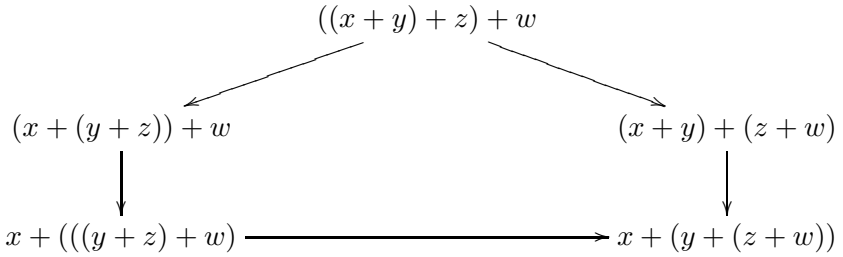
This critical pair is (trivially) resolved as follows:



(e) Rules (3) and (3) have a critical pair arising from the following overlapping redexes:



This critical pair is resolved as follows (this is actually an instance of something very famous sometimes called "the MacLane pentagon"):



PART III: λ-Calculus

1. To determine whether $\lambda xy.((\lambda f.y(fyx))(\lambda xy.x))$ and $\lambda x.(\lambda z.(\lambda x.((\lambda xy.xy)xx)z))$ are equal the plan is to try and reduce them to normal form. Unfortunately you have to read these terms correctly (although I was forgiving if you did not). Here are the reductions:

$$\begin{array}{rcl} \lambda xy.(\underline{(\lambda f.y(fyx))(\lambda xy.x)}) & \longrightarrow & \lambda xy.y(\underline{(\lambda xy.x)yx}) \\ & \longrightarrow & \lambda xy.yy \end{array}$$

and (I have added a parenthesis to emphasize the correct reading of this term!)

$$\begin{array}{rcl} \lambda x.(\lambda z.(\lambda x.(((\lambda xy.xy)xx)z))) & =_\alpha & \lambda x.(\lambda z.(\lambda w.(\underline{(\lambda xy.xy)ww}z))) \\ & \longrightarrow & \lambda x.(\lambda z.(\lambda w.wwz)) \end{array}$$

as the results of these reductions are in normal form, and we know normal forms are unique in the λ-calculus – and these are not α-equivalent – the terms cannot be equal.

2. Representing data in the λ-calculus is actually a translation process. Unfortunately, this does not mean it is particularly easy BUT once you have tried it a few times you will probably be able to appreciate the general process. Here is an explicit description of the process:

   (a) Start with the datatype definition:

```
data NewType a1 .. an = Con1 OldType1 a1 ... an (NewType a1 ... an)
                      | ...
                      | Conm Oldtypem a1 ... an (NewType a1 ... an)
```

   First notice that each constructor has a type: this is rather hidden in the Haskell syntax so lets make it explicit.

```
Coni :: (OldType1 a1 ... an (NewType a1 ... an)) -> (NewType a1 ... an)
```

   Now when we come to construct the `fold` function each constructor turns into a function so we need to think about the type of these functions. It is obtained by replacing the type of the new datatype with an arbitrary type:

```
Coni :: (OldTypei a1 ... an b) -> b
```

   This makes the type of the `fold` look like

```
fold :: ((OldType1 a1 ... an b) -> b) ->
             .....
        ((OldTypem a1 ... an b) -> b) ->
        (NewType a1 ... an)) ->
        b
```

   This just sorts out the typings involved but you must be clear on this to start with!

   (b) Now the next step is that you wish to represent this datatype in the λ-calculus and the basic idea is that you will represent it as the abstracted fold. So in the end you know that the actual fold will become:

```
fold c1 ... cm v = v c1 ... cm
```

which you then have to write as an abstraction.

Of course you are done if you can write down the constructors as then you can inductively construct representing elements of this datatype (and so say inductively what it means to be a element of this datatype).

(c) So what do the constructors look like?

```
Ci y c1 ... cm = c_i (mapi id ... id (\z -> z c1 ... cm) y)
```

where `mapi` is the map function for the i^{th} old type!

Ok so this is really dense and needs alot of unwrapping!! Further Haskell tends to use the curried types for constructors (so the `cons` for a list is not expressed as `Product(a,List a) -> List a` but as `a -> (List a) -> (List a)` so all this must be translated out. This means rather than having a product type you have to deal with `a -> b -> b` for the type of the replacing function in the fold and alter the notion of mapping accordingly. However, if you just follow this through carefully you will get:

```
Cons a l cons nil = cons a (l cons nil)
```

So with this pattern in mind you can just write down the representations:

(a)

$$\begin{aligned}
\mathbf{BZ} &= \lambda znf.z \\
\mathbf{Bon} &= \lambda xznf.n(xznf) \\
\mathbf{Boff} &= \lambda xznf.f(xznf) \\
\mathbf{fold} &= \lambda znfv.vznf
\end{aligned}$$

(b)

$$\begin{aligned}
\mathbf{Nil} &= \lambda cn.n \\
\mathbf{Cons} &= \lambda alcn.ca(lcn) \\
\mathbf{fold} &= \lambda cnl.lcn \\
\mathbf{map} &= \lambda fl.\mathbf{fold}(\lambda al.\mathbf{Cons}(fa)l)\mathbf{Nil}l \\
\mathbf{case} &= \lambda lfg.\mathbf{fst}(\mathbf{fold}(\lambda a(x,y).(gay,\mathbf{Cons}ay))(f,\mathbf{Nil})l
\end{aligned}$$

To get the **case** you needed to go to the dentist and have inspiration!

(c)

$$\begin{aligned}
\mathbf{Rs} &= \lambda fal.fa(\mathbf{map}(\lambda x.xf)l) \\
\mathbf{foldR} &= \lambda fx.xf
\end{aligned}$$

3. A fixed point combinator $\mathbf{Y}$ has the property that for every f we have $f(\mathbf{Y}f) = \mathbf{Y}f$. To show that $\mathbf{Y}'$ is a fixed point combinator we have:

$$\begin{aligned}
\mathbf{Y}'f &:= \underline{(\lambda xy.y(xxy))(\lambda xy.y(xxy))}f \\
&\rightarrow (\lambda y.y(\mathbf{Y}'y)f \\
&\rightarrow f(\mathbf{Y}'f)
\end{aligned}$$

To implement the recursive (non-terminating) function `f n = n:(f (n+1))` we set `f` $= \mathbf{Y}(\lambda fn.n : (f(n+1)))$ reducing gives the desired recursive behaviour:

$$\begin{array}{rcl} \texttt{f}n & := & (\mathbf{Y}(\lambda fn.n : (f(n+1))))n \\ & \longrightarrow & (\lambda fn.n : (f(n+1)))(\mathbf{Y}(\lambda fn.n : (f(n+1))))n \\ & \longrightarrow & n : ((\mathbf{Y}(\lambda fn.n : (f(n+1))))n + 1) \\ & =: & n : (\texttt{f}n + 1) \end{array}$$

4. They are all false! Or are they ...

   - "Every λ-term is α, β-equivalent to a term in normal form."
     Well, of course not: just consider Ω!
   - "There is only one fixed point combinator up to α, β-equivalence."
     Well one should be suspicious of this one as you have seen at least two ... but could they be equivalent?

     $$\begin{array}{rcll} \mathbf{Y} & = & \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) & \text{due to Curry} \\ \mathbf{Y}' & = & (\lambda xy.y(xxy))(\lambda xy.y(xxy)) & \text{due to Turing} \end{array}$$

     To see that they are not the same look at the β-reductions of $\mathbf{Y}$ setting $\mathbf{X} = \lambda x.f(xx)$

     $$\mathbf{Y} \longrightarrow \lambda f.f(\mathbf{XX}) \longrightarrow \lambda f.f(f(\mathbf{XX})) \longrightarrow .. \longrightarrow \lambda f.f^n(\mathbf{XX}) \longrightarrow ...$$

     Now consider Turing's fixed point combinator set, $\mathbf{X}' = \lambda xf.f(xxf)$, then

     $$\mathbf{Y}' = \mathbf{X}'\mathbf{X}' \longrightarrow \lambda f.f(\mathbf{X}'\mathbf{X}'f) \longrightarrow ... \longrightarrow \lambda f.f^n(\mathbf{X}'\mathbf{X}'f) \longrightarrow ...$$

     so they never have a common β-reduction and so they cannot be equal. Of course, there are infinitely many different fixed point combinators ... can you prove this?
   - "When reducing a λ-term to normal form using β-reduction one must be careful (normal order reduction) so that one never reaches a term from which infinitely many β-reduction are possible."
     The point is that the purpose of normal order reduction is not to avoid terms from which infinitely many β-reductions are possible it is to avoid doing infinitely many reductions when finitely many could get you to a normal form! Essentially normal order reduction ensures that you do only the *necessary* reductions.
     So, while, of course, you do need to be careful about which order you do the reductions, the reason given is not correct!
   - "If a term has a normal form then, because the λ-calculus is confluent no matter what order you apply β-reductions you will reach the normal form."
     Well given the discussion above this has to be false. For example, $\mathbf{K}\mathsf{True}\Omega$ has a normal form (namely True) but has infinitely many reductions leaving it if you choose to rewrite Ω (which, of course, is not *necessary*).

- "If a λ-calculus term has a normal form there is a fixed equation in the size of the term which bounds the number of β-reductions necessary to reduce it to that normal form."

  This would mean that you could solve the halting problem: apply this formula to the size of the term now do no more than that number of reductions. If you reach a normal form you know it reduces to normal form so halts ... but more importantly if it does not you know it cannot have a normal form.

- "Two λ-terms are α, β-equivalent if and only if they both have β-normal forms which are α-equivalent."

  If $t \xrightarrow{\beta} t'$ then $t =_{\alpha,\beta} t'$ but there is no guarantee that there will be a normal form to which this reduction leads (for example set $t = \mathbf{Y}$).

- "There are some functions which can be computed by the λ-calculus which cannot be computed by a Turing Machine."

  Everyone knows the Church-Turing thesis. It says that the only computable functions *on the natural numbers* are those computable by a Turing machine. But what about the infinite list calculation `f n = n:(f (n+1))`? The interesting thing is that this it quite naturally represented in λ-calculus (using head normal form repeatedly to generate what is called the Bohm tree). However, a Turing machine has a really hard time with representing this non-terminating but information providing computation.

  So it depends what you mean by function!