Functional Programming 1b

Thomas Forster

November 26, 2006

Some suitable tripos questions.

1991:5:9, 1993:12:9, 1994:5:10, 1994:10:12, 1995:5:10, 1995:12:12, 1996:5:9, 1996:12:11, 1997:5:11, 1998:13:10

0.0.1 A Question from John Harrison's notes

```
Moscow ML version 1.40 (1 July 1996)
Enter `quit();' to quit.
- fun itlist f [] b = b
    | itlist f (h::t) b = f h (itlist f t b);
> val itlist = fn : ('a -> ('b -> 'b)) -> 'a list -> 'b -> 'b
- fun map f [] = []
    | map f (h::t) = (f h)::(map f t);
> val map = fn : ('a -> 'b) -> 'a list -> 'b list
- fun C f x y = f y x;
> val C = fn : ('a -> ('b -> 'c)) -> 'b -> 'a -> 'c
- local fun mem x [] = false
          | mem x (h::t) = x = h orelse mem x t;
        fun insert x l = if mem x l then l else x::l
     in fun union l1 l2 = itlist insert l1 l2
    end;
> val union = fn : ''a list -> ''a list -> ''a list
- fn f => fn l1 => fn l2 => itlist (union o C map l2 o f) l1 [];
> val it = fn : ('a -> ('b -> ''c)) -> ('a list -> ('b list -> ''c list))
- it (fn a => fn b => (a,b)) [1,2,3] [4,5,6];
> val it =
    [(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)]
    : (int * int) list
```

1 Answers to Tripos questions

1991:5:9

We declare some λ-terms as follows.
$\underline{\text{true}} = \lambda xy.x$; $\underline{\text{false}} = \lambda xy.y$; $\underline{0} = \lambda x.x$; $\underline{\text{succ}} = \lambda zf.f\ \underline{\text{false}}\ z$

- a We want $\underline{\text{succ}}\ (\underline{\text{succ}}\ \underline{0})$
  First we compute $\underline{\text{succ}}\ \underline{0} = \lambda zf.f\ \underline{\text{false}}\ z\ \ \lambda x.x$
  $= \lambda f.f\ \underline{\text{false}}\ (\lambda x.x)$. Do $\underline{\text{succ}}$ again
  $\underline{\text{succ}}\ (\lambda f.f\ \underline{\text{false}}\ (\lambda x.x))$
  $= \lambda yg.g\ \underline{\text{false}}\ y\ (\lambda f.f\ \underline{\text{false}}\ (\lambda x.x))$
  $= \lambda g.g\ \underline{\text{false}}\ (\lambda f.f\ \underline{\text{false}}\ (\lambda x.x))$
- b Evaluate at $\underline{\text{false}}$
- c Evaluate at $\underline{\text{true}}$
- d Distinct normal forms

1993:12:9

```
- fun N f x = x;;
val N = fn : 'a -> 'b -> 'b

-  fun P a k f x = f a (k f x);;
val P = fn
  : 'a -> (('a -> 'b -> 'c) -> 'd -> 'b) -> ('a -> 'b -> 'c) -> 'd -> 'c

- fun Q k l f x = k f (l f x);;
val Q = fn : ('a -> 'b -> 'c) -> ('a -> 'd -> 'b) -> 'a -> 'd -> 'c


-  fun W a k = Q k (P a k);;
val W = fn
  : 'a -> (('a -> 'b -> 'c) -> 'c -> 'b) -> ('a -> 'b -> 'c) -> 'c -> 'b

-  fun R k = k W N;;
val R = fn
  : ((('a -> (('a -> 'b -> 'c) -> 'c -> 'b) -> ('a -> 'b -> 'c) -> 'c -> 'b)
     -> ('d -> 'e -> 'e) -> 'f)
    -> 'f
```

Suppose further that `K` and `L` have `ML` definitions of the form:

```
val K = P a1 (P a2 (P a3 N ...))

val K = P b1 (P b2 (P b3 N ...))
```

1. *State the* `ML` *types of* `N` *and* `P`.

   `N` has type $\alpha \to (\beta \to \beta)$. `P` seems to have the type:
   $\alpha \to (((\alpha \to (\beta \to \gamma)) \to (\delta \to \beta)) \to ((\alpha \to (\beta \to \gamma)) \to (\delta \to \gamma)))$. God knows why.

2. *What does the expression* `K f x` *evaluate to?*

   Let us start by noting that `K` of the form `P a1 T` for some term `T`. Then

   `K f x` becomes

   `P a1 T f x` by expanding `K`. Then expand `P` to get

   `f a1 (T f x)`

   But now we have a task just like the one we started with, since `T` is an expression of the same form as `K`. We repeat this step until we have `T` that

is in fact `N`. But `N f x` is just `x`, so the result is going to be `K` with all the `P`s replaced by `f`s and the final `N` replaced by `x`.

3. *What does the expression* `Q K L f x` *evaluate to?*

   `Q K L f x` First expand `Q` to get

   `K f (L f x)` Next use the fact that `K` is of the form `P a1 T` as before.

   `P a1 T f (L f x)` then expand `P` getting

   `f a1 (T f (L f x))`

   But by now the embedded expression `T f (L f x)` is clearly of the same form as the term `K f (L f x)` we started with, so we can keep repeating this until we get down to a `T` that will in fact be `N`. Then `f a1 (N f (L f x))` will clearly β-reduce to `f a1 (L f x)` which means that the result is going to be a concatenation of the two lists!

4. *What does the expression* `R K f x` *evaluate to?* Given that the earlier ISWIM code in this question relates to an ISWIM implementation of lists, it's an obvious guess that `R` means `R`everse. Let's see.

   `R k f x` expand `R k`

   `k W N f x` expand `K` into `P a1 k'`

   `P a1 k' W N f x` expand `P a1 k' W N` using definition of `P`

   `W a1 (k' W N) f x` expand `W`

   `Q (k' W N) (P a1 N) f x` `W` now has 4 arguments, so expand it

   `k' W N f (P a1 N f x)`

   So by unpeeling the top level of the structure of `k`, into `P a1 k'`, we have turned

   `k W N f x` into

   `k' W N f (P a1 N f x)`

   These two look suspiciously alike. We match `k` to `k'`, `W` and `N` to themselves, and `P a1 N f x` to `x`. This tells us what will happen if we repeat what we have just done, this time on `k'` instead of on `k`. `R` is indeed `R`everse!

1994:5:10

'=' between λ-terms is not (syntactic) equality, but *convertibility* and it is defined by the following recursion

- $(\lambda x.M)N \ = \ M[x := N]$ (β-conversion)
- $M = M$
- $M = N \rightarrow N = M$
- $M = N \wedge N = L \rightarrow M = L$
- $M = N \rightarrow MZ = NZ$
- $M = N \rightarrow ZM = ZN$
- $M = N \rightarrow \lambda x.M = \lambda x.N$

Set **true** $= \lambda xy.x$ and **false** $= \lambda xy.y$. Then we can define the desired terms as follows.

cons Define **cons** $a\ l = \lambda fx.fa(lfx)$

null I am assuming that the null list is $\lambda fx.x$, which is **false** (i can't think what else the given definition of list could intend it to be). So **null** l must be $\lambda l.(lt)$**true** for some term t. Consideration of the null case doesn't tell us what t is to be.

Check this:

$((\lambda l.lt)$**true**$)$ **false**

β-reduces to (**false** t)**true**

which β-reduces to **true** as desired.

On the other hand a non-null list is $\lambda fx.faM$ for some term M containing an occurrence of 'x'. This will tell us what t has to be. So let's apply

$\lambda l.l\ t$ **true** to $\lambda fx.faM$.

$\lambda l.l\ t$ **true** $\lambda fx.faM$.

$(\lambda fx.faM)\ t$ **true**.

$(\lambda x.taM)$ **true**.

This has got to β-reduce to **false**, and we can achieve this only by tweaking t. Let's take t to be $K(K$**false**$)$. Then

$(\lambda x.taM)$ **true** is

$(\lambda x.K(K$**false**$)aM)$ **true**.

$(\lambda x.$**false**$)$ **true**.

false.

So **null** must be $\lambda l.l\ (K(K$**false**$))$ **true**.

append Define **append** $l_1\ l_2 = \lambda fx.(l_1 f)(l_2 f x)$

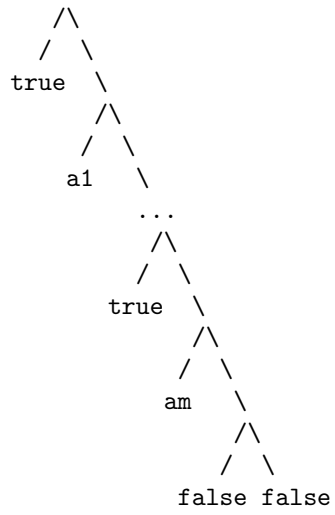
hd can be $\lambda l.l$ **true** t for any λ-term t.

tl (Matt Grimwade says:)

The clue is in the question: "assume λ-encodings of ordered pairs". So first use:

$$\texttt{mk\_pairs} = \lambda l.l(\lambda ax.\texttt{pair}\ \mathit{true}\ (\texttt{pair}\ ax))(\texttt{pair}\ \textbf{false}\ \textbf{false})$$

to turn the list into a lot of internesting pairs, flagging the end so we'll know when we get there:



get rid of the unwanted head using `snd`; and convert back with:

$$\texttt{mk\_list} = Y(\lambda gpfx.\texttt{if}(\texttt{fst}\ p)(f(\texttt{fst}(\texttt{snd}\ p))(g(\texttt{snd}(\texttt{snd}\ p))fx)x).$$

In total we have:

$$\texttt{tl} = \lambda lfx.\texttt{mk\_list}(\texttt{snd}(\texttt{snd}(\texttt{mk\_pairs}\ l)))fx.$$

1994:10:12

Recall that `f o g` is the function that sends x to $f(g(x))$. Consider the `ML` definitions: `fun I` $x = x$; `fun pair` $(f, g)(x, y) = ((fx), (gy))$; `fun pup` $(f, g)z = ((fz), (gz))$; `fun fst`$(x, y) = x$; `fun snd`$(x, y) = y$;

Describe the effect of the following functions:

- `pair`$(\texttt{I}, \texttt{I})$ is the identity relation of type $\alpha \times \beta \to \alpha \times \beta$.
- `pair`$((f1 \circ f2), (g1 \circ g2))$ sends (x, y) to $(f1(f2(x)), g1(g2(y)))$
- `pup`(`fst`, `snd`) is the identity relation of type $\alpha \times \beta \to \alpha \times \beta$.
- `pup(f o fst, g o snd)` sends x to $((f(\texttt{fst}(x)), g(\texttt{snd}(x)))$. That is, it behaves like `pair` (f, g) x.

Infinite lists can be represented in a functional language by triples. The triple (a, h, t) represents the infinite list whose nth element is $h(t^n(a))$ (for $n \geq 0$).

(a) Give a representation for the infinite list $n, n+1, n+2, \ldots$

This is $(n, \texttt{I}, succ)$

(b) Code in `ML` a map functional for this representation; given a function f and the infinite list $x_0, x_1, \ldots x_n \ldots$, it should yield a representation of $f(x_0), f(x_1), \ldots f(x_n) \ldots$,

`map` f $(a, h, t) =: (a, (f \circ h), t)$;

(c) Code in `ML` a *zip* function

`zip` (a, h, t) $(a', h', t') =: ((a, a'),$ `pair`$(h, h'),$ `pair`$(t, t'))$;

(d) Code in `ML` an *interleave* function

[*HOLE !!*]

- (e) Discuss

1995:5:10

Let

$A = \lambda xy.y(xxy)$

$\Theta = AA$

$\texttt{suc} = \lambda nfx.f(nfx)$

$\texttt{true} = \lambda xy.x$

$\texttt{false} = \lambda xy.y$

The first thing to do is to demonstrate that Θ is a fix-point combinator. Let us β-reduce $\Theta\ f$. This is $A\ A\ f$, or
$((\lambda xy.y(xxy))\ A)\ f$
The first occurrence of A (the one i have written out in full) has two variables at the front, 'x' and 'y'. We replace 'x' by A and 'y' by 'f', and lop off the '$\lambda xy.$' getting $f(\Theta f)$.
This will save a lot of trouble later on.

- Θ `suc` will β-reduce to $\texttt{suc}(\Theta\ \texttt{suc})$, which is $\lambda fx.f((\Theta\ \texttt{suc})\ f\ x)$ which is in head normal form, tho' clearly not in normal form.
- $\Theta\ (\lambda x.xx)$ β-reduces to $(\Theta(\lambda x.xx))(\Theta(\lambda x.xx))$
  This is going to go on getting bigger and will have no normal form.
- $\Theta(\texttt{suc}\ n)$ β-reduces (once one has relettered 'n' for 'x') to $(\texttt{suc}\ n)(\Theta\ (\texttt{suc}\ n))$. Notice that `suc` of anything has a head normal form. But this thing is not going to have a normal form.
- Θ `true` β-reduces to
  $\texttt{true}(\Theta\ \texttt{true})$. But `true` x is always $\lambda y.x$. So this is $\lambda y.(\Theta\ \texttt{true})$ which has our original formula as a subformula, so this will go on for ever. It doesn't even have a head normal form, beco's it never returns anything except itself.
- Θ `false` β-reduces to
  $\texttt{false}(\Theta\ \texttt{false})$. `false` of anything is I.
- $\Theta(\lambda x.fxx)$ β-reduces to $(\lambda x.fxx)(\Theta(\lambda x.fxx))$ and one more β-reduction will give
  $((f(\Theta(\lambda x.fxx))))(\Theta(\lambda x.fxx))$
  which is in head normal form.

If M has no hnf then $M[N/x]$ has no hnf, for any N. Use this fact to prove the following:

If M has no hnf then MN has no hnf for any N.

1995:12:12

The type of `curry` is $((\alpha \times \beta) \to \gamma)) \to (\alpha \to (\beta \to \gamma))$ and the type of `uncurry` is $(\alpha \to (\beta \to \gamma)) \to ((\alpha \times \beta) \to \gamma))$.

`curry(fn(x,y) => x)` is $\lambda xy.x$.

`uncurry o curry` is the identity function of type $((\alpha \times \beta) \to \gamma)) \to ((\alpha \times \beta) \to \gamma))$

`curry I` is $\lambda xy.\texttt{pair}(x, y)$. To deduce this we have to specialise `I` to a thing of type $(\alpha \times \beta) \to (\alpha \times \beta)$, and `curry` of a thing of this type must be of type $\alpha \to (\beta \to (\alpha \times \beta))$.

To determine what `uncurry I` evaluates to we must note that `uncurry` expects an argument of type $\alpha \to (\beta \to \gamma)$. So we must specialise `I` to identity of type $(\beta \to \gamma) \to (\beta \to \gamma)$. Applying `uncurry` to this will give a result of type $((\beta \to \gamma) \times \beta) \to \gamma$ and this must be

$\lambda x.((\texttt{fst}(x))\texttt{snd}(x))$.

(a) `fun` n `=>` $n * 2$`;`

(b) If `g` codes the list in question we want `f o g`.

(c) `drop` $f\ i\ n = f\ (n + i)$`;`

(d) `interleave` $f\ g\ n =$ `if even` n `then` $f\ (n$ `div` $2)$ `else` $g\ (n$ `div` $2)$`;`

(e)
```
fun ifilter(f,p,x,0,0) = if p(f(x)) then f(x) else
                            ifilter(f,p,x+1,0,0)
  | ifilter(f,p,x,n,l) = if n=l then f(x-1) else
                         if p(f(x)) then ifilter(f,p,x+1,n+1,l)
                                    else ifilter(f,p,x+1,n,l);;
fun filter f p x = ifilter(f,p,0,0,x);;
```

(Thanks to David Bradshaw)

1996:5:9

Consider binary trees that are either empty (written '`leaf`') or have the form `Br` x t_1 t_2 where t_1 and t_2 are themselves binary trees. Give an encoding of binary trees in the λ-calculus, including functions `isleaf`, `label`, `left` and **right** satisfying

`isleaf leaf` $\rightarrow$ **false**

`isleaf`(`Br` x t_1 t_2) $\rightarrow$ **true**

`label`(`Br` x t_1 t_2) $\rightarrow x$

`left`(`Br` x t_1 t_2) $\rightarrow t_1$

right(`Br` x t_1 t_2) $\rightarrow t_2$

If you use encodings of other data structures, state the properties assumed.

[6 marks]

Consider the `ML` functions `f` and `g` defined to satisfy

```
f([],ys) = ys

f(x::xs,ys) = f(xs,x::ys)

g([],ys) = ys

g(x::xs,ys) = x :: g(xs,ys)
```

Using list induction, prove `f(f(xs,[]), []) = xs`. [14 marks]

1.0.2 A model answer from Larry Paulson

Here are the definitions:

`leaf` $= \lambda z.z$

`Br` $= \lambda x t_1 t_2$.`pair` **false** (`pair` x (`pair` t_1 t_2))

`isleaf` $=$ `fst`

`label` $= \lambda t$.`fst` (`snd` t)

`left` $= \lambda t$.`fst`(`snd` (`snd` t))

right $= \lambda t$.`snd`(`snd` (`snd` t))

This assumes `fst`(`pair` x y) $\rightarrow x$ and `snd`(`pair` x y) $\rightarrow y$; the definition of `leaf` given above actually relies on `fst` $= \lambda p.p$ **true**. Thus `isleaf leaf` β-reduces successively to $(\lambda z.z)$**true** then to **true**. The other laws hold trivially.

Now for the second part.

Obviously, g is the append function. The given formula requires generalization before induction. Perform induction on xs in

$\forall ys. f(f(xs, ys), zs) = f(ys, g(xs, zs))$.

The result will then follow by the definition of f and by $g(xs, []) = xs$, itself provable by a trivial induction.

The base case of the induction holds by definition of f and g:
$f(f([], ys), zs) = f(ys, zs) = f(ys, g([], zs))$.
For the inductive step we have (mostly by definition)
$f(f(x :: xs, ys), zs) = f(f(xs, x :: ys), zs)$
$= f(x :: ys, g(xs, zs))$ (induction hypothesis)
$= f(ys, x :: g(xs, zs))$
$= f(ys, g(x :: xs, zs))$
The main difficulty lies in choosing the right generalization.

1996:12:11

Any recursive function f :`int -> int` declared like

$$fn =: \texttt{if } n = 0 \texttt{ then } a \texttt{ else } g(n, f(n-1))$$

is a fixed point for the function F:`(int -> int) -> (int -> int)` declared as

$$F\ f\ n =: \texttt{if } n = 0 \texttt{ then } a \texttt{ else } g(n, f(n-1))$$

If $\mathcal{F}$ is a λ-term for F then $Y\mathcal{F}$ is a λ-term for f.

The displayed attempt to declare Y in `ML` won't work because it won't type-check. Instead we should declare

```
fun Y f x = f (Y f) x;

val Y = fn : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
- -

val fac = Y ( fn g => fn x => if x=0 then 1 else x*(g (x-1)));
val fac = fn : int -> int
```

Notice that `fun Y f x => f (Y f) x`

doesn't cause non-termination because `ML` evaluates functions only when they have been given all their arguments (I think; I don't believe it does any partial evaluation). So if you evaluate `fac`, it goes something like this:

```
val fac => Y ( fn g => fn x => if x = 0 then 1 else x*(g (x-1)));

fac 1 => Y ( fn g => fn x => if x = 0 then 1 else x*(g (x-1))) 1
```

(now Y has enough arguments, so gets evaluated)

```
=> ( fn g => fn x => if x = 0 then 1 else x*(g (x-1)))
    (Y ( fn g => fn x => if x = 0 then 1 else x*(g (x-1)))) 1

=> if 1= 0 then 1
   else 1*(Y ( fn g => fn x => if x = 0 then 1 else x*(g (x-1)))) (1-1))

=> 1*((Y ( fn g => fn x => if x = 0 then 1 else x*(g (x-1)))) 0))
```

```
=> 1*( fn g => fn x => if x = 0 then 1 else x*(g (x-1)))
    (Y ( fn g => fn x => if x = 0 then 1 else x*(g (x-1)))) 0

=> 1*(if 0 = 0 then 1
      else x*((Y ( fn g => fn x => if x = 0 then 1 else x*(g (x-1)))) (0-1))))

=> 1*1

=> 1
```

Of course, we're just pretending that it uses substitution rather than keeping variables in store locations, but that would make what's already rather messy too much worse.

Anyway, the crucial things are that

- it does look at the definition of `Y`
- it works because the applicative order evaluation strategy doesn't apply to function arguments.

(Thanks to Jon Fairbairn)

Grant Warrell sez the answer that was wanted was one not using recursion, to wit:

```
- val Y = (fn t=>t(T t));
val Y = fn : ('a t -> 'a) -> 'a
```

[*HOLE ML detects failure of typechecking in cases like 'xx' by doing an occurs check. What happens if we write it in PROLOG which famously doesn't do an occurs check? What is the class of things that suddenly become OK?*]

1997:5:11

(a) [bookwork]

(b) (i) Y_M is $\lambda f.WWM$. This is

$$\lambda f.[\lambda x \lambda z.f(xxz)\ W\ M]$$

Now do two β-reductions inside the square bracket (W/x and M/z) getting

$$\lambda f.f(WWM)$$

which is in head normal form.

(ii) $Y_M(KI)$ is $(\lambda f.WWM)\ (KI)$
Using (i) this becomes $\lambda f.f(WWM))\ (KI)$ (then do a β-reduction) $(KI)(WWM)$ which of course is just I.

(iii) We follow the same track as (ii) to get $K(WWM)$. If that has a HNF i'll eat my hat.

(c) If a λ-term M is in HNF, then the body is of the form fN for some vbl f and some term N. If we now ensure that f gets instantiated to KI, we will find that $(KI)N$ β-reduces to I as desired. As far as i can see it doesn't matter what the other variables get instantiated to.

The fact that $Y_M(KI)$ β-reduces to I shows that Y_M is solvable. II is also I so $Y_M(KI)$ is solvable too.

The third one is not solvable. Y_M is a fixpoint combinator so $Y_M K$ is a fixpoint for K. But no fixpoint for K can possibly be solvable: whatever you apply it to you just get back what you started with and that can't be I—beco's I is not a fixed point for K!

1998:13:10

```
fun update(s, x, i) = if x = y then i else s(y)
   fun interpret  Assign(a, Expr(e)) s1 = fun update(s1, x, e(s1)))
    — interpret  Sequence(c1, c2) s1 = interpret(c2,interpret(c2, s1))
    — interpret  while_do(Expr(e), c) s1 = if not e(s1)<> 0 then interpret(c, s1)
elses1
```