Design and Selection of Programming Languages
18 October 2006

Exercise 6.1 — Haskell Evaluation  (25% of 90 minutes Midterm 2, 2005)
Let the following Haskell definition be given:

from k = k : from (k+1)
prune True xs = []
prune False xs = xs
eat p [] = from (7 * 8)
eat p (x : xs) = x : prune (p x) (eat (not . p) xs)

Simulate Haskell evaluation for the following expression, i.e., write down the complete sequence of intermediate expressions:

\[
\text{eat (< 5) (from 5)}
\]

Note: You may introduce abbreviations for repeated subexpressions, or use repetition marks for material that is unchanged from the previous line.

Solution Hints
\[
eat (< 5) (from 5) \\
\rightarrow \text{eat (< 5) (5 : from (5 + 1))} \\
\rightarrow 5 : \text{prune ((< 5) 5) (eat (not . (< 5)) (from (5 + 1)))} -- * \\
\rightarrow 5 : \text{prune (5 < 5) (eat (not . (< 5)) (from (5 + 1)))} \\
\rightarrow 5 : \text{prune False (eat (not . (< 5)) (from (5 + 1)))} \\
\rightarrow 5 : \text{eat (not . (< 5)) (from (5 + 1))} \\
\rightarrow 5 : \text{eat (not . (< 5)) ((5 + 1) : from ((5 + 1) + 1))} \\
\rightarrow 5 : (5 + 1) : \text{prune ((not . (< 5)) (5 + 1)) (eat (not . (not . (< 5))) (from ((5 + 1) + 1)))} \\
\rightarrow 5 : 6 : \text{prune ((not . (< 5)) 6) (eat (not . (not . (< 5))) (from (6 + 1))))} \\
\rightarrow 5 : 6 : \text{prune (not ((< 5) 6)) (eat (not . (not . (< 5))) (from (6 + 1))))} \\
\rightarrow 5 : 6 : \text{prune (not (6 < 5)) (eat (not . (not . (< 5))) (from (6 + 1)))} -- * \\
\rightarrow 5 : 6 : \text{prune (not False) (eat (not . (not . (< 5))) (from (6 + 1))))} \\
\rightarrow 5 : 6 : \text{prune True (eat (not . (not . (< 5))) (from (6 + 1))))} \\
\rightarrow 5 : 6 : [ ] \\
\rightarrow 5 , 6 \\
\]

Exercise 6.2 — Haskell Typing  (22% of Midterm 2, 2005)
Provide detailed derivations of the most general Haskell types of the following functions:
maybe \( x \mapsto f \) Nothing = \( x \)
maybe \( x \mapsto f \) (Just \( y \)) = \( f \ y \)
\[
\text{keepof2 } k \ h \ (x,y) = k \ (\text{curry} \ h \ x) \ y
\]

Remember: \( \text{curry} :: ( (a,b) \to c) \to a \to b \to c \)

**Solution Hints**
The prelude definition

\[
\text{data} \ Maybe \ a = \text{Nothing} \mid \text{Just} \ a
\]
implies the following types for the constructors of this datatype:

\[
\text{Nothing} :: \text{Maybe} \ a
\]
\[
\text{Just} :: a \to \text{Maybe} \ a
\]

Starting from the second equation and assuming \( x :: q \) and \( f :: a \to b \), we see that \( y :: a \) and obtain:

\[
\text{maybe} :: q \to (a \to b) \to (\text{Maybe} \ a) \to b
\]

With the first equation, we see from the right-hand side that \( x :: b \), too, so we have:

\[
\text{maybe} :: b \to (a \to b) \to (\text{Maybe} \ a) \to b
\]

Using \( \text{curry} :: ( (a,b) \to c) \to a \to b \to c \), we obtain \( x :: a \) and \( h :: (a,b) \to c \).

So \( (\text{curry} \ h \ x) :: b \to c \). Now let us assume that \( y :: d \); then we have

\[
k :: ((b \to c) \to d \to e)
\]
for some \( e \), and therefore:

\[
\text{keepof2} :: ((b \to c) \to d \to e) \to ((a,b) \to c) \to (a,d) \to e
\]

**Exercise 6.3 — Defining Haskell Functions  \((19\% \text{ of Midterm 2, 2005})\)**

Define the following Haskell functions (the solutions are independent of each other, but each can use functions specified in previous items):

(a) \( \text{inits} :: [a] \to [[[a]]] \)

such that \( \text{inits} \) \( xs \) evaluates to a list consisting of exactly all prefixes of \( xs \) (in which order is irrelevant).

E.g., \( \text{inits} [1,2,3] = [[[],[1],[1,2],[1,2,3]]] \)

(This is a function exported by the standard library module \texttt{List}.)

**Solution Hints**

\( \text{inits} :: [a] \to [[[a]]] \) \(--= \text{List} \text{.inits}\)

\( \text{inits} [ ] = [ [ ] ] \)

\( \text{inits} \ (x:xs) = [ ] : \text{map} \ (x:) \ (\text{inits} \ xs) \)

Or:
\( \text{inits'} :: [a] \rightarrow [[a]] \)
\( \text{init'} :: [a] \rightarrow [a] \)  
\( \text{--- = Prelude.init} \)
\( \text{inits'} [] = [[]] \)
\( \text{init'} [x] = [x] \)
\( \text{inits'} xs = xs \) : \( \text{inits'} (\text{init'} xs) \)
\( \text{init'} (x:xs) = x : \text{init'} xs \)

(b) \( \text{fromThen} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{[Integer]} \)

such that \( \text{fromThen} x1 x2 = [x1, x2 ..] \).

Solution Hints
\( \text{fromThen} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{[Integer]} \)
\( \text{fromThen} x1 x2 = x1 : \text{fromThen} x2 (x2 + x2 - x1) \)
\( \text{fromThen'} x1 x2 = ft x1 \)
where
\( ft x1 = x1 : ft (x1 + d) \)
\( d = x2 - x1 \)

(c) \( \text{fromThenTo} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{[Integer]} \)

such that \( \text{fromThenTo} x1 x2 x3 = [x1, x2 .. x3] \), e.g.:

\( \text{fromThenTo} \ 5 \ 7 \ 9 = [5,7,9] \)
\( \text{fromThenTo} \ 5 \ 7 \ 10 = [5,7,9] \)
\( \text{fromThenTo} \ 7 \ 5 \ 10 = [] \)
\( \text{fromThenTo} \ 7 \ 5 \ 1 = [7,5,3,1] \)

Solution Hints
\( \text{fromThenTo} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{[Integer]} \)
\( \text{fromThenTo} x1 x2 x3 = \text{takeWhile} p \$ \text{fromThen} x1 x2 \)
where
\( p = \text{if } x2 \geq x1 \text{ then } (\leq x3) \text{ else } (\geq x3) \)

Or:

\( \text{fromThenTo'} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{[Integer]} \)
\( \text{fromThenTo'} x1 x2 x3 = \text{ftt} x1 \)
where
\( \text{ftt} x1 \mid p x1 = x1 : \text{ftt} (x1 + d) \)
\( | \text{otherwise} = [] \)
\( d = x2 - x1 \)
\( p = \text{if } d \geq 0 \text{ then } (\leq x3) \text{ else } (\geq x3) \)

Note: \( \text{fromThen} \) and \( \text{fromThenTo} \) are the functions underlying the syntactic sugar \([1,3 ..]\) and \([1,3 .. 10]\) — you should not use this syntax to define these functions.

Exercise 6.4 — Simple Graphs  (34% of Midterm 2, 2005)
A simple graph can be (naïvely) represented in Haskell as a list of pairs, where an edge from node \( x \) to node \( y \) is represented by the pair \((x, y)\), and the sequencing of pairs in the list does not matter.
For example, one representation of the graph drawn to the left is
\[
\text{gr} = [(1, 2), (2, 3), (2, 5), (3, 4), (4, 1)]
\]

Let the following type synonym be given:

\[
\text{type} \hspace{1em} \text{Graph} \hspace{1em} a = [(a, a)]
\]

(a) \(\approx 6\%\) Define \(\text{successors} :: \text{Eq} \ a \Rightarrow \text{Graph} \ a \rightarrow a \rightarrow [a]\) such that \(\text{successors} \ g \ n\) returns a list containing exactly the endnodes of those edges of the graph \(g\) that start at node \(n\).

E.g., \(\text{successors} \ gr \ 2 = [3, 5]\) and \(\text{successors} \ gr \ 5 = []\)

Solution Hints

\[
\text{successors} \ g \ n = \left[ \ y \mid (x, y) \leftarrow g, x \equiv n \right] \quad \quad \text{\(\sim\)} = \text{map snd} \ (\text{filter} ((n ==) \ . \ \text{fst}) \ g)
\]

\[
\text{successors'} \ [\ ] \ n = [\ ]
\]

\[
\text{successors'} \ ((x, y) : ps) \ n = \text{if} \ x \equiv n \ \text{then} \ y : \text{successors'} \ ps \ n \ \text{else} \ \text{successors'} \ ps \ n
\]

(b) \(\approx 10\%\) \(\text{pathGraph} :: [a] \rightarrow \text{Graph} \ a\)

such that \(\text{pathGraph} [x_1, ..., x_n]\) evaluates to the list \([(x_1, x_2), ..., (x_{n-1}, x_n)]\) containing the pairs of immediately consecutive elements in \(xs\), e.g.,

\(\text{pathGraph} [2, 3, 4, 1, 2, 5] = [(2, 3), (3, 4), (4, 1), (1, 2), (2, 5)]\), which is just another representation of the graph drawn above.

Solution Hints

\[
\text{pathGraph} :: [a] \rightarrow [(a, a)]
\]

\[
\text{pathGraph} \ (x : xs \equiv (y : ys)) = (x, y) : \text{pathGraph} \ xs
\]

\[
\text{pathGraph} _ = [\ ]
\]

(c) \(\approx 8\%\) A \textit{path} in a simple graph can be represented as a list of nodes, as above in (b). Define the Haskell function \(\text{hasCycle} :: \text{Eq} \ a \Rightarrow [a] \rightarrow \text{Bool}\) such that \(\text{hasCycle} \ p\) is true if path \(p\) contains a cycle, i.e., if there is a node that occurs at least twice in \(p\). For example, the path \([2, 3, 4, 1, 2, 5]\) has a cycle around node 2.

Solution Hints

\[
\text{hasCycle} :: \text{Eq} \ a \Rightarrow [a] \rightarrow \text{Bool}
\]

\[
\text{hasCycle} \ [\ ] = \text{True}
\]

\[
\text{hasCycle} \ (x : xs) = x \ '\text{elem}' \ xs \parallel \text{hasCycle} \ xs
\]

(d) \(\approx 10\%\) Define \(\text{edgeGraph} :: \text{Eq} \ a \Rightarrow \text{Graph} \ a \rightarrow \text{Graph} \ (a, a)\) such that \(\text{edgeGraph} \ g\) returns the \textit{edge graph} of \(g\). This edge graph has edges of \(g\) as nodes, and has an edge from \(e1\) to \(e2\) iff the end node of \(e1\) is equal to the start node of \(e2\) (as edges in \(g\)).

Solution Hints
Define `paths :: Eq a => Graph a -> [[a]]` to calculate all non-empty cycle-free paths of a graph.

**Solution Hints**

We use induction over the number of edges: Adding an edge to a graph may combine two previously existing paths, or extend one previously existing path either at the beginning or at the end.

```haskell
paths :: Eq a => Graph a -> [[a]]
paths [] = []
paths (e : es) = let ps = paths es in
    ps ++
    [x : zs | zs ≡ (z : zs') ← ps, y ≡ z, x `notElem` zs ]
  ++
    [zs ++ [y] | zs ← ps, x ≡ last zs, x `notElem` zs ]
  ++
    [zs ++ zs' | zs ← ps, x ≡ last zs, zs' ← ps, y ≡ head zs',
              all (`notElem` zs) zs']
  ++
    if x ≡ y then [] else [[x, y]]
```