Design and Selection of Programming Languages

11 October 2006

Exercise 5.1 — Haskell Evaluation (36% of Midterm 1, 2004)

Assume the following Haskell definitions to be given:

\[
succ \; n = n + 1
\]

\[
take :: \text{Int} \rightarrow \text{[a]} \rightarrow \text{[a]}
\]

\[
take \; 0 \_ \_ = \[]
\]

\[
take \_ \_ \_ = \[]
\]

\[
take \; n \; (x:xs) = x : take \; (n-1) \; xs
\]

\[
feed \; h \; q \; y = q : feed \; h \; (q + y) \; (h \; y)
\]

Simulate Haskell evaluation for the following expression (write down the sequence of intermediate expressions):

\[
\text{take} \; 3 \; (\text{feed} \; \text{succ} \; 0 \; 1)
\]

Note: You may introduce abbreviations for repeated subexpressions, or use repetition marks for material that is unchanged from the previous line. In particular, write “s” instead of “succ”!

Solution Hints

13 steps, 1 contractible arith

\[
take \; 3 \; (\text{feed} \; \text{succ} \; 0 \; 1)
\]

\[
= \text{take} \; 3 \; (0 : \text{feed} \; \text{succ} \; (0 + 1) \; (\text{succ} \; 1))
\]

\[
= 0 : \text{take} \; (3-1) \; (\text{feed} \; \text{succ} \; (0 + 1) \; (\text{succ} \; 1))
\]

\[
= 0 : \text{take} \; 2 \; (\text{feed} \; \text{succ} \; (0 + 1) \; (\text{succ} \; 1))
\]

\[
= 0 : \text{take} \; 2 \; ((0 + 1) : \text{feed} \; \text{succ} \; ((0 + 1) + \text{succ} \; 1) \; (\text{succ} \; (\text{succ} \; 1)))
\]

\[
= 0 : (0 + 1) : \text{take} \; (2-1) \; (\text{feed} \; \text{succ} \; ((0 + 1) + \text{succ} \; 1) \; (\text{succ} \; (\text{succ} \; 1)))
\]

\[
= 0 : 1 : \text{take} \; 1 \; ((1 + \text{succ} \; 1) : \text{feed} \; (+) \; \text{succ} \; ((1 + \text{succ} \; 1) + \text{succ} \; (\text{succ} \; 1)) \; (\text{succ} \; (\text{succ} \; (\text{succ} \; 1))))
\]

\[
= 0 : 1 : (1 + \text{succ} \; 1) : \text{take} \; (1-1) \; (\text{feed} \; \text{succ} \; ((1 + \text{succ} \; 1) + \text{succ} \; (\text{succ} \; 1)) \; (\text{succ} \; (\text{succ} \; (\text{succ} \; 1))))
\]

\[
= 0 : 1 : (1 + 2) : \text{take} \; (1-1) \; (\text{feed} \; \text{succ} \; ((1 + 2) + \text{succ} \; (\text{succ} \; 1)) \; (\text{succ} \; (\text{succ} \; 2)))
\]

\[
= 0 : 1 : 3 : \text{take} \; (1-1) \; (\text{feed} \; \text{succ} \; (3 + \text{succ} \; (\text{succ} \; 1)) \; (\text{succ} \; (\text{succ} \; 2)))
\]

\[
= 0 : 1 : 3 : []
\]
3% per necessary step: • 1% for reducing the right redex
  • 2% for performing the reduction correctly
  • -1% for not writing down

Exercise 5.2 — Finite-State Machines (25% of Midterm 1, 2004)

Let the following type synonyms be given, as in the presentation in the first lecture:

```haskell
type State = Int
type Symbol = Char
type TransRel = [(State, Symbol, State)]
type FSM = (State, TransRel, [State])
```

(a) Define \( \text{fsm1} :: FSM \) such that it represents the finite-state machine drawn above (with start state circled and end states in boxes):

(b) Define the Haskell function \( \text{isDet} :: FSM \rightarrow \text{Bool} \) such that \( \text{isDet} \ \text{fsm} \) evaluates to the Boolean value indicating whether the finite-state machine \( \text{fsm} \) is deterministic or not.

For example, \( \text{isDet} \ \text{fsm1} = \text{False} \) since there are two \( b \)-edges from state 1 to different nodes.

**Hint:** Define auxiliary functions! For example:
- Calculate all start nodes of transitions in a \( \text{TransRel} \).
- Given a state, calculate all edges leaving that state in a \( \text{TransRel} \).
- Given a \( \text{Symbol} \) and a \( \text{TransRel} \), find all target nodes of edges with that symbol.
- Given a \( \text{State} \) and a \( \text{TransRel} \), find out whether any edges from that state violate determinacy.

Other functions may be useful, too. **Document your functions!**

**Solution Hints**

```haskell
type State = Int
type Symbol = Char
type TransRel = [(State, Symbol, State)]
type FSM = (State, TransRel, [State])
```

```haskell
fsm1 :: FSM          -- 6%
fsm1 = (0, tr1, [1])
where
  tr1 =
    [ (0, 'a', 1)
    , (1, 'b', 2)
    , (1, 'b', 3)
    , (2, 'a', 1)
    , (2, 'c', 0)
    , (3, 'a', 2)
    ]
```

```haskell
edgeStarts tr = [ s | (s, c, t) <- tr ]                -- 3%
```
outEdges tr s = \((c, t) \mid (s', c, t) \leftarrow tr, s' \equiv s\) — 3%

isUnique es (c, t) = all \((t' \mid (c', t') \leftarrow es, c' \equiv c)\) — 5%

isDetState tr s = all \((isUnique es)\) es — 4%

where es = outEdges tr s

isDet (s0, tr, fin) = all \((isDetState tr)\) (edgeStarts tr) — 4%

---

**Exercise 5.3 — Haskell Typing** *(19% of Midterm 1, 2004)*

Provide **detailed derivations** of the Haskell types of the following functions:

\[
\text{swibble} \ x \ y = \{ (x, y), (x +\ ''', y + 1) \} \\
\text{swoon} \ g \ h = \{ g \ ((1 +) \ . \ h) \}
\]

**Solution Hints**

Type classes have not been taught yet, only mentioned: Numeric types can be defaulted to *Integer* or *Int*.

\[
\text{swibble} : (\text{Num } n) \Rightarrow \text{String} \to n \to \{(\text{String}, n)\}
\]

Assuming \(1 : \text{Integer}\), we must have \(y : \text{Integer}\) because of \(y + 1\).

Since \(''\ : \text{String}\), we also have \(x : \text{String}\) because of \(x + ''\ : \text{String}\).

Then \((x, y) : (\text{String}, \text{Integer})\), and the type of *swibble* follows easily.

\[
\text{swoon} : (\text{Num } n) \Rightarrow ((a \to n) \to b) \to (a \to n) \to [b]
\]

Assuming \(1 : \text{Integer}\), we have \((1 +) : \text{Integer} \to \text{Integer}\), and because of the composition, we must have

\(h : a \to \text{Integer}\) for some type \(a\).

Therefore, we have \((1 +) \circ h : a \to \text{Integer}\), and may assume \(g : (a \to \text{Integer}) \to b\) for some type \(b\).

Then we have \([g \ ((1 +) \circ h)] : b\), and therefore

\[
\text{swoon} \ g : (a \to \text{Integer}) \to b
\]

and

\[
\text{swoon} : ((a \to \text{Integer}) \to b) \to (a \to \text{Integer}) \to b.
\]

---

**Exercise 5.4** *(Skeleton file is on the course page)*

We define a type of transition functions that define state transitions triggered by *inputs* and also producing *outputs*:

\[
\text{type Transition state input output} = (\text{state, input}) \to (\text{state, output})
\]

(a) Define a Haskell function
process :: Transition state input output → state → [ input ] → [ output ]

that calculates the list of outputs produced by a transition function given a starting state and a list of inputs.

Solution Hints

process tr s [] = []
process tr s ( input : inputs ) = let
    ( s' , output ) = tr ( s , input )
    in output : process tr s' inputs

Using process from (b) and prelude functions, the definition

runprocess :: Transition state String String → state → IO ()
runprocess tr s = do
    hSetBuffering stdout LineBuffering -- requires: “import System.IO” at beginning of module
    interact ( unlines ° process tr s ° lines )

allows runprocess to turn a transition with String inputs and outputs into a runnable program.

Try: runprocess id 0

(b) Define a transition function

countEcho :: Transition Integer String String

that keeps a counter as its state and otherwise just reproduces the input prefixed with line numbers as output.

Try: runprocess countEcho 0

Solution Hints

countEcho ( count , input ) = ( count' , shows count' ( ' ' : input ) )
    where count' = succ count

(c) Define a transition function

trAdd :: Transition Integer String String

that uses the prelude functions read and show to add the Integer reading of the input to the accumulating state, and outputs that state as a string.

Try: runprocess trAdd 0

Solution Hints

trAdd ( s , input ) = ( s' , show s' )
    where
        n = read input
        s' = s + n

(d) Define a transition function

polish :: Transition [ Integer ] String String

that implements a reverse Polish notation calculator by pushing number inputs on the stack, always outputing the top of the stack (if present), and interpreting +, −, ∗, / as taking their arguments
from the stack and pushing the result back onto the stack.

Try: \textit{runprocess polish [ ]}

\textbf{Solution Hints}

\begin{align*}
\textit{polish} (n : m : ks, \text{"\textbf{+}\text{"}) } & = (k : ks, \text{ show } k) \quad \textbf{where } k = m + n \\
\textit{polish} (n : m : ks, \text{"\textbf{-}\text{"}) } & = (k : ks, \text{ show } k) \quad \textbf{where } k = m - n \\
\textit{polish} (n : m : ks, \text{"\textbf{\text{*}\text{"}) } & = (k : ks, \text{ show } k) \quad \textbf{where } k = m \cdot n \\
\textit{polish} (n : m : ks, \text{"\textbf{/}\text{"}) } & = (k : ks, \text{ show } k) \quad \textbf{where } k = m \text{ \textquoteleft \text{div} \textquoteright } n \\
\textit{polish} (ks \quad , \text{ input}) \quad & = (k : ks, \text{ show } k) \quad \textbf{where } k = \text{ read input}
\end{align*}