Assignment 3 — Advanced Functional Programming, 2011/2012

Beerend Lauwers
Augusto Passalaqua
{B.Lauwers, A.PassalaquaMartins}@students.uu.nl

Utrecht University, The Netherlands

December 9, 2011

Exercise 1

This is troublesome in Haskell due to the fact that the 'let' expression is polymorphic and unsafe, so any IO operation on x can accept a different type.

ML's typing system splits types in two camps: strong and weak. Strong type variables can only be used in strong types. Only strong types can be stored in references, preventing polymorphic references.

Exercise 2

type *Square a* = *Square′ Nil a*
data *Square′ t a* = *Zero* (*t* (*t a*)) | *Succ* (*Square′* (*Cons t*) *a*)
data *Nil a* = *Nil*
data *Cons t a* = *Cons a* (*t a*)

Raw: sq2 = Succ *Succ* Zero (Cons (Cons 5 $Cons5Nil)(Cons(Cons5$ Cons 5 Nil) Nil)) Prettified:

cons x = *Cons x Nil*
sq2 = *Succ* \$ *Succ* \$ *Zero* \$ *Cons* (*Cons* 1 \$ *cons* 0) \$ *Cons* (*Cons* 0 \$ *cons* 1) *Nil*

Raw : sq3 = Succ *Succ* Succ *Zero* Cons (Cons 1 (Cons 2 (Cons 3 Nil))) $(Cons(Cons4(Cons5(Cons6Nil)))$ Cons (Cons 7 (Cons 8 (Cons 9 Nil))) Nil) Prettified:

sq3 = *Succ* \$ *Succ* \$ *Succ* \$ *Zero* \$ *Cons* (*Cons* 1 \$ *Cons* 2 \$ *cons* 3) \$
(*Cons* (*Cons* 4 \$ *Cons* 5 \$ *cons* 6) \$ *Cons* (*Cons* 7 \$ *Cons* 8 \$ *cons* 6) *Nil*)

Exercise 3

$$\mathit{forceBoolList} :: [\mathit{Bool}] \to r \to r$$
$$\mathit{forceBoolList}\ (\mathit{True} : \mathit{xs})\ r = \mathit{forceBoolList}\ \mathit{xs}\ r$$
$$\mathit{forceBoolList}\ (\mathit{False} : \mathit{xs})\ r = \mathit{forceBoolList}\ \mathit{xs}\ r$$
$$\mathit{forceBoolList}\ [\,]\qquad r = r$$

A function of type [Bool] -¿ [Bool] would not allow the construction of expressions that use the list of bools to depend on the forced evaluation of the list. Due to lazyness, the function forceBoolList will not be necessarily called before (or might not be called at all) before the evaluation of an expression that depends on the list of bools. In practice, this means no strictness at all.

A function defined as

$$\mathit{force} :: a \to a$$
$$\mathit{force}\ a = \mathit{seq}\ a\ a$$

will present the same problem mentioned above. There is no functional dependency between the "force a" expression and any other that might depend on its value a. In a lazy environment this call will be deferred until the value of "force a" is necessary.

Exercise 4

Exercise 4 is defined in its own module (imported by the lhs version of this document) as the following:

module *Trie* **where**

import *qualified Data.Map as M*

data *Trie a* = *Node* (*Maybe a*) (*M.Map Char* (*Trie a*)) **deriving** (*Show*, *Eq*)

$$\mathit{empty} :: \mathit{Trie}\ a$$
$$\mathit{empty} = \mathit{Node}\ \mathit{Nothing}\ M.\mathit{empty}$$

$$\mathit{null} :: (\mathit{Eq}\ a) \Rightarrow \mathit{Trie}\ a \to \mathit{Bool}$$
$$\mathit{null} = (\equiv)\ \mathit{empty}$$

valid :: *Trie a* → *Bool*
valid (*Node a m*) = **case** *a* **of**
  Nothing → **if** *M.null m* **then** *True* **else** *validLeafs′ m*
  otherwise → *validLeafs′ m*
  where *validLeafs′* = *M.fold* (λ*t a* → *validLeafs t* ∧ *a*) *True*

Checks if any leaf node is present without any value. The idea is that if true, that means we're at a Node with Nothing for (Maybe a) but one empty node at one point in the subtrie.

$$\mathit{validLeafs} :: \mathit{Trie}\ a \to \mathit{Bool}$$
$$\mathit{validLeafs}\ (\mathit{Node}\ \mathit{Nothing}\ m) \mid M.\mathit{null}\ m = \mathit{False}$$
$$\mathit{validLeafs}\ (\mathit{Node}\ \_\ m) = M.\mathit{fold}\ (\lambda t\ a \to \mathit{validLeafs}\ t \wedge a)\ \mathit{True}\ m$$

$$\mathit{insert} :: \mathit{String} \to a \to \mathit{Trie}\ a \to \mathit{Trie}\ a$$
$$\mathit{insert}\ (x : \mathit{xs})\ a\ (\mathit{Node}\ b\ m) = \mathit{Node}\ b\ \$\ M.\mathit{insertWithKey}\ f\ x\ (\mathit{insert}\ \mathit{xs}\ a\ \mathit{empty})\ m$$

  where *f k n o* = *insert xs a o*

```
insert [ ]  a (Node _ m) = Node (Just a) m
lookup :: String → Trie a → Maybe a
lookup (x : xs) (Node _ m) = do m′ ← M.lookup x m
  r ← Trie.lookup xs m′
  return r
lookup [ ] (Node a _) = a
delete :: (Eq a) ⇒ String → Trie a → Trie a
delete (x : xs) (Node a m) = Node a $ M.filter (¬ ∘ Trie.null)
   $ M.adjust (delete xs) x m
delete [ ]  (Node _ m) = Node Nothing m
```

Examples

Examples to test the functions above. Example 4 gives us exactly what is on the PDF.

```
example1 = insert "f" 0 Trie.empty
example2 = insert "foo" 1 example1
example3 = insert "bar" 2 example2
example4 = insert "baz" 3 example3
```

Exercise 5

Exercise 5 has also been defined in its own file to allow the normal execution of the code of the other tasks.

We'll use this function to generate exponential amounts of type variables:

```
func2 a b c d e f g = (a, b, c, d, e, f, g)
```

By applying func2 to itself, we generate $(7^2) - 1$ type variables. Applying the resulting function to func2 again results in $(7^3) - 1$ type variables. This way, we can easily generate an expression with an exponential amount of type variables.

sf generates $7^{n+1} - 1$ type variables.

```
sf1 = func2 func2 func2 func2 func2 func2 func2 func2   -- 48 type variables
sf2 = func2 sf1 sf1 sf1 sf1 sf1 sf1 sf1   -- 342 type variables
sf3 = func2 sf2 sf2 sf2 sf2 sf2 sf2 sf2   -- 2400 different type variables
sf4 = func2 sf3 sf3 sf3 sf3 sf3 sf3 sf3   -- 16806 different type variables
sf5 = func2 sf4 sf4 sf4 sf4 sf4 sf4 sf4   -- 117648 different type variables
sf6 = func2 sf5 sf5 sf5 sf5 sf5 sf5 sf5   -- 823542 different type variables
sf7 = func2 sf6 sf6 sf6 sf6 sf6 sf6 sf6   -- 5764800 different type variables
sf8 = func2 sf7 sf7 sf7 sf7 sf7 sf7 sf7   -- 40353606 different type variables
sf9 = func2 sf8 sf8 sf8 sf8 sf8 sf8 sf8   -- 282475248 different type variables,
    -- that should be enough.
```

Also, we have:

```
func x = (x, x)
```

By composing *func* with itself, the tuple (x, x) expands to $((x, x), (x, x))$. Composing it again results in four tuples that in total contain eight x's. This way, an exponentional type signature can be generated trivially, but there is only a single type variable in the signature: a. By applying this very long pointfree expression to one of the above functions, each a is expanded to a tuple that has a large amount of type variables. The type variable generation is also exponential, so it can quickly enough overwhelm the type checker on its own.

$$
\begin{aligned}
&\mathit{two} = \mathit{func} \circ \mathit{func} \\
&\mathit{four} = \mathit{two} \circ \mathit{two} \\
&\mathit{eight} = \mathit{four} \circ \mathit{four} \\
&\mathit{sixteen} = \mathit{eight} \circ \mathit{eight} \\
&\mathit{test} = \mathit{sixteen} \; \$ \; \mathit{sf4}
\end{aligned}
$$