

Haskell intro

Assignment1

Kai Arne S. Myklebust, Silvan Adrian

Handed in: September 19, 2018



Contents

1	Design/Implementation	1
2	Code Assessment	2
A	Code Listing	2

1 Design/Implementation

Overall in this assignment our goal was to not use helper functions, where not specifically needed. This to make readability easier and making the code less complex since many of our helper functions from the previous assignment were unnecessary. We started by making a helper function for each arithmetic operation from "initial context". This was to get the first and second element from the list. After a while we got really annoyed doing that and found out that you can just get the first and second element by changing to [] list-brackets. We used head and tail in equality, but onlineTA says that we should not use them. We check for empty lists and a list of different length, so we check for possible errors which we found while testing from head and tail. In evalExpr we used the do notation to make it more readable when we have multiple actions in the same statement.

We worked our way through the monads, by doing many tests and figuring out what return values where needed. In addition we read a lot of monad introductions and articles and used more than 20 hours just to partly understand this assignment. Also we got help in another TA class. In the monad return we knew that we needed the right value of SubsM and worked our way from there. In fail we knew that we needed the left side value (Error). For the bind we used the slides from the lecture

and the haskell wiki to work our way forward. Underway we did testing to see that we got the right return values.

2 Code Assessment

According to our own tests and onlineTa, everything except array compression works. Array compression was the hardest part and is only partly working. ACFor only works for arrays with numbers. If you have a String it sees the string as only one element and not multiple characters. ACFor does not work with nested for's. We weren't able to make nested for's working. We use putVar and know that it works. So the ACFor can see the variable, but one problem is that only the body should see the variable but now the whole ACFor sees the new variable. In ACIf we have the problem when the if clause evaluates to false it has to return a Value, but the assignment says it should return nothing. IF ACIf is inside a ACFor our solution does not work, but if there's a single ACIf then it works.

We ran our own tests to show these failures. These can be run by stack test (the last 6 out of 84 tests fail, which we also described in the assessment above).

One place where our test cases were able to help us find errors was in equality and having arrays of different lengths. We fixed it by checking for empty arrays and for different array lengths then we return a FalseVal.

A Code Listing

```
1  module SubsInterpreter
2      (
3          Value(..)
4      , runExpr
5      , equality
6      , smallerThen
7      , add
8      , mul
9      , sub
10     , modulo
11     , mkArray
12     -- You may include additional exports here, if you want to
13     -- write unit tests for them.
14     )
15     where
16
17  import SubsAst
```

```

18
19  -- You might need the following imports
20  import Control.Monad
21  import qualified Data.Map as Map
22  import Data.Map (Map)
23
24
25  -- | A value is either an integer, the special constant
    ↳ undefined,
26  --   true, false, a string, or an array of values.
27  -- Expressions are evaluated to values.
28  data Value = IntVal Int
29             | UndefinedVal
30             | TrueVal | FalseVal
31             | StringVal String
32             | ArrayVal [Value]
33             deriving (Eq, Show)
34
35
36  type Error = String
37  type Env = Map Ident Value
38  type Primitive = [Value] -> Either Error Value
39  type PEnv = Map FunName Primitive
40  type Context = (Env, PEnv)
41
42  initialContext :: Context
43  initialContext = (Map.empty, initialPEnv)
44  where initialPEnv =
45          Map.fromList [ ("===", equality)
46                      , ("<", smallerThen)
47                      , ("+", add)
48                      , ("*", mul)
49                      , ("-", sub)
50                      , ("%", modulo)
51                      , ("Array", mkArray)
52                      ]
53
54  newtype SubsM a = SubsM {runSubsM :: Context -> Either Error (a,
    ↳ Env) }
55
56  instance Monad SubsM where
57    return x = SubsM $ \ (e, _) -> Right (x,e)
58    m >>= f = SubsM $ \c@(_, p) -> runSubsM m c >>= \ (x, e') ->
    ↳ runSubsM (f x) (e', p)

```

```

59     fail s = SubsM $ \_ -> Left s
60
61     -- You may modify these if you want, but it shouldn't be
62     --    necessary
63     instance Functor SubsM where
64         fmap = liftM
65     instance Applicative SubsM where
66         pure = return
67         (<*>) = ap
68
69     equality :: Primitive
70     equality [IntVal a, IntVal b] = if (a == b) then Right TrueVal
71         --> else Right FalseVal
72     equality [UndefinedVal, UndefinedVal] = Right TrueVal
73     equality [StringVal a, StringVal b] = if a == b then Right
74         --> TrueVal else Right FalseVal
75     equality [TrueVal, TrueVal] = Right TrueVal
76     equality [FalseVal, FalseVal] = Right TrueVal
77     equality [ArrayVal [], ArrayVal []] = Right TrueVal
78     equality [ArrayVal [], ArrayVal _] = Right FalseVal
79     equality [ArrayVal _, ArrayVal []] = Right FalseVal
80     equality [ArrayVal a, ArrayVal b] = if head a == head b
81         --> then equality [ArrayVal (tail a), ArrayVal (tail b)]
82         --> else Right FalseVal
83     equality [_ , _] = Right FalseVal
84     equality _ = Left "Wrong number of arguments"
85
86     smallerThen :: Primitive
87     smallerThen [IntVal a, IntVal b] = if a < b then Right TrueVal
88         --> else Right FalseVal
89     smallerThen [StringVal a, StringVal b] = if a < b then Right
90         --> TrueVal else Right FalseVal
91     smallerThen [_ , _] = Right FalseVal
92     smallerThen _ = Left "Wrong number of arguments"
93
94     add :: Primitive
95     add [IntVal a, IntVal b] = Right (IntVal(a + b))
96     add [StringVal a, StringVal b] = Right (StringVal(a ++ b))
97     add [IntVal a, StringVal b] = Right (StringVal(show a ++ b))
98     add [StringVal a, IntVal b] = Right (StringVal(a ++ show b))
99     add [_ , _] = Left "No Int or String"
100    add _ = Left "Wrong number of arguments"
101
102    mul :: Primitive

```

```

98 mul [IntVal a, IntVal b] = Right (IntVal(a*b))
99 mul [_, _] = Left "No Integer"
100 mul _ = Left "Wrong number of arguments"
101
102 sub :: Primitive
103 sub [IntVal a, IntVal b] = Right (IntVal(a-b))
104 sub [_, _] = Left "No Integer"
105 sub _ = Left "Wrong number of arguments"
106
107 modulo :: Primitive
108 modulo [IntVal a, IntVal b] = if b == 0 then Left "Division by
    ↳ Zero" else Right (IntVal(mod a b))
109 modulo [_, _] = Left "No Integer"
110 modulo _ = Left "Wrong number of arguments"
111
112 mkArray :: Primitive
113 mkArray [IntVal n] | n >= 0 = return $ ArrayVal (replicate n
    ↳ UndefinedVal)
114 mkArray _ = Left "Array() called with wrong number or type of
    ↳ arguments"
115
116 modifyEnv :: (Env -> Env) -> SubsM ()
117 modifyEnv f = SubsM $ \ (e, _) -> Right ((), f e)
118
119 putVar :: Ident -> Value -> SubsM ()
120 putVar name val = modifyEnv $ \e -> Map.insert name val e
121
122 getVar :: Ident -> SubsM Value
123 getVar name = SubsM $ \ (e, _) -> case Map.lookup name e of
124     Just v -> Right (v, e)
125     Nothing -> Left "No value
    ↳ found in map"
126
127 getFunction :: FunName -> SubsM Primitive
128 getFunction name = SubsM $ \ (e, p) -> case Map.lookup name p of
129     Just v -> Right (v, e)
130     Nothing -> Left "No value
    ↳ found in map"
131
132 evalExpr :: Expr -> SubsM Value
133 evalExpr Undefined = return UndefinedVal
134 evalExpr TrueConst = return TrueVal
135 evalExpr FalseConst = return FalseVal
136 evalExpr (Number a) = return $ IntVal a

```

```

137 evalExpr (String a) = return $ StringVal a
138 evalExpr (Var a) = getVar a
139 evalExpr (Array []) = return (ArrayVal [])
140 evalExpr (Array (a:ax)) = do
141   a <- evalExpr a
142   ArrayVal ax <- evalExpr(Array ax)
143   return (ArrayVal (a:ax))
144 evalExpr (Compr (ACBody e)) = evalExpr e
145 evalExpr (Compr (ACFor i e c)) = do
146   a <- evalExpr e
147   case a of
148     ArrayVal xa -> do
149       val <- mapM (\x -> do
150         putVar i x
151         evalExpr(Compr c)) xa
152       return (ArrayVal val)
153     StringVal xs -> do
154       (StringVal s) <- (\_ -> evalExpr(Compr c)) xs
155       return (StringVal s)
156     _ -> fail "FOR needs an array or string"
157
158 evalExpr (Compr (ACIf e c)) = do
159   a <- evalExpr e
160   case a of
161     TrueVal -> evalExpr (Compr c)
162     FalseVal -> return (ArrayVal [])
163     _ -> fail "IF needs a boolean"
164
165 evalExpr (Call a b) = do
166   f <- getFunction a
167   ArrayVal bv <- evalExpr (Array b)
168   case f bv of
169     Right r -> return r
170     Left l -> fail l
171
172 evalExpr (Assign a b) = do
173   v <- evalExpr b
174   putVar a v
175   return v
176
177 evalExpr (Comma a b) = do
178   _ <- evalExpr a
179   evalExpr b
180

```

```
181 runExpr :: Expr -> Either Error Value
182 runExpr expr = case runSubsM (evalExpr expr) initialContext of
183     Right r -> Right (fst r)
184     Left l -> Left l
```
