Haskell intro

Assignement0

Kai Arne S. Myklebust, Silvan Adrian

Handed in: September 17, 2018

Contents

1 Design/Implementation

2 Code Assessment

A Code Listing

```haskell
module SubsInterpreter
        (
          Value(..)
        , runExpr
        -- You may include additional exports here, if you want to
        -- write unit tests for them.
        )
        where

import SubsAst

-- You might need the following imports
import Control.Monad
```

```haskell
import qualified Data.Map as Map
import Data.Map(Map)


-- | A value is either an integer, the special constant undefined,
--   true, false, a string, or an array of values.
-- Expressions are evaluated to values.
data Value = IntVal Int
           | UndefinedVal
           | TrueVal | FalseVal
           | StringVal String
           | ArrayVal [Value]
           deriving (Eq, Show)


type Error = String
type Env = Map Ident Value
type Primitive = [Value] -> Either Error Value
type PEnv = Map FunName Primitive
type Context = (Env, PEnv)

initialContext :: Context
initialContext = (Map.empty, initialPEnv)
  where initialPEnv =
          Map.fromList [ ("===", equality)
                       , ("<", smallerThen)
                       , ("+", add)
                       , ("*", mul)
                       , ("-", sub)
                       , ("%", modulo)
                       , ("Array", mkArray)
                       ]

newtype SubsM a = SubsM {runSubsM :: Context -> Either Error (a, Env)}

instance Monad SubsM where
  return x = SubsM $ \(e, _) -> Right (x,e)
  m >>= f = SubsM $ \c@(e, p) -> runSubsM m c >>= \(x, e') -> runSubsM (f x) (e', p)
  fail s = SubsM $ \_ -> Left s

-- You may modify these if you want, but it shouldn't be necessary
```

```haskell
instance Functor SubsM where
  fmap = liftM
instance Applicative SubsM where
  pure = return
  (<*>) = ap

equality :: Primitive
equality a = if length a > 2 then equality2 (head a) (head (tail
  ↪  a)) else Left "List is smaller or bigger then 2"

equality2 :: Value -> Value -> Either Error Value
equality2 (IntVal a) (IntVal b) = if (a == b) then Right TrueVal
  ↪  else Right FalseVal
equality2 UndefinedVal UndefinedVal = Right TrueVal
equality2 (StringVal a) (StringVal b) = if (a == b) then Right
  ↪  TrueVal else Right FalseVal
equality2 TrueVal TrueVal = Right TrueVal
equality2 FalseVal FalseVal = Right TrueVal
equality2 (ArrayVal []) (ArrayVal []) = Right TrueVal
equality2 (ArrayVal a) (ArrayVal b) = if head a == head b then
  ↪  equality2 (ArrayVal (tail a)) (ArrayVal (tail b)) else Right
  ↪  FalseVal
equality2 _ _ = Right FalseVal

smallerThen :: Primitive
smallerThen a = if length a > 2 then smallerThen2 (head a) (head
  ↪  (tail a)) else Left "List is smaller or bigger then 2"

smallerThen2 :: Value -> Value -> Either Error Value
smallerThen2 (IntVal a) (IntVal b) = if (a < b) then Right
  ↪  TrueVal else Right FalseVal
smallerThen2 (StringVal a) (StringVal b) = if (a < b) then Right
  ↪  TrueVal else Right FalseVal
smallerThen2 _ _ = Right FalseVal

add :: Primitive
add a = if length a > 2 then add2 (head a) (head (tail a)) else
  ↪  Left "List is smaller or bigger then 2"

add2 :: Value -> Value -> Either Error Value
add2 (IntVal a) (IntVal b) = Right (IntVal(a + b))
add2 (StringVal a) (StringVal b) = Right (StringVal(a ++ b))
add2 (IntVal a) (StringVal b) = Right(StringVal(show a ++ b))
add2 (StringVal a) (IntVal b) = Right(StringVal(a ++ show b))
```

```haskell
add2 _ _ = Left "No Int or String"

mul :: Primitive
mul a = if length a > 2 then mul2 (head a) (head (tail a)) else
  ↪ Left "List is smaller or bigger then 2"

mul2 :: Value -> Value -> Either Error Value
mul2 (IntVal a) (IntVal b) = Right (IntVal(a*b))
mul2 _ _ = Left "No Integer"

sub :: Primitive
sub a = if length a > 2 then sub2 (head a) (head (tail a)) else
  ↪ Left "List is smaller or bigger then 2"

sub2 :: Value -> Value -> Either Error Value
sub2 (IntVal a) (IntVal b) = Right (IntVal(a-b))
sub2 _ _ = Left "No Integer"

modulo :: Primitive
modulo a = if length a > 2 then mod2 (head a) (head (tail a))
  ↪ else Left "List is smaller or bigger then 2"

mod2 :: Value -> Value -> Either Error Value
mod2 (IntVal a) (IntVal b) = if b == 0 then Left "Division by
  ↪ Zero" else Right (IntVal(mod a b))
mod2 _ _ = Left "Not integer"

mkArray :: Primitive
mkArray [IntVal n] | n >= 0 = return $ ArrayVal (replicate n
  ↪ UndefinedVal)
mkArray _ = Left "Array() called with wrong number or type of
  ↪ arguments"

modifyEnv :: (Env -> Env) -> SubsM ()
modifyEnv f = SubsM $ \(e, _) -> Right ((),(f e))

putVar :: Ident -> Value -> SubsM ()
putVar name val = modifyEnv $ \e -> Map.insert name val e

getVar :: Ident -> SubsM Value
getVar name = SubsM $ \(e, _) -> case Map.lookup name e of
                                   Just v -> Right (v, e)
                                   Nothing -> Left "No value
  ↪ found in map"
```

```haskell

getFunction :: FunName -> SubsM Primitive
getFunction name = SubsM $ \(e, p) -> case Map.lookup name p of
                                        Just v -> Right (v, e)
                                        Nothing -> Left "No value
  ↪  found in map"

evalExpr :: Expr -> SubsM Value
evalExpr Undefined = return UndefinedVal
evalExpr TrueConst = return TrueVal
evalExpr FalseConst = return FalseVal
evalExpr (Number a) = return $ IntVal a
evalExpr (String a) = return $ StringVal a
evalExpr (Array []) = return (ArrayVal [])
evalExpr (Var a) = getVar a

runExpr :: Expr -> Either Error Value
runExpr expr = undefined
```