CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 18: Comments on Project

- IO Monad and Creating a Repl
- Simple Type Checking

Project Schedule

- Milestone 1: **4/23/19**
  - Join a Piazza group as organized by Prof Snyder
  - Create a group repo, and summarize your plans in the readme:
    - Which Additional Features you plan to do (tentative, can be changed)
    - Who has primary responsibility for what (should be approximately equal, and, again, can be changed)
  - Completion is worth 5pts/100.
- Milestone 2: 4/29/19
  - Details TBA
  - Completion is worth 5pts/100.
- Final Deadline: 5/3/19
  - Completion is worth 90pts/100.

Required Features

- Additional operators and functions as explained below, including:
  - Add support for the following types of data: floats, characters, strings, and lists.
  - All operators and functions should report a sensible error message when applied incorrectly during execution (this is called "dynamic type checking"). For example `[0,1] !! 10` should return an error like "Can't get element 10 from a 2 element list" and `7 !! 10` should return "7 is not a list". No input should cause your program to crash, instead, you need to check for all possible errors at execution time.
  - Additional syntax for lists. For example `[1,2,3]`
  - Single and multiline comments. For example `x+ 7 -- this is a comment` and `x+ {- this is a multi-line comment-} 7`

Let's look at the list of operators and functions...

https://github.com/BU-CS320/Spring-2019/blob/master/project/INSTRUCTIONS.md

For parsing, look at the new version of the basic parsing code posted with Lecture 14....

http://www.cs.bu.edu/fac/snyder/cs320/Homeworks%20and%20Labs/ParserExample.hs

Changing the EnvUnsafe monad

- Appropriate changes to "EnvUnsafe" monad, including support for logging:
  - a `print` keyword
  - a sequencing infix operator `;`

You will have to combine the `EnvUnsafe` monad and the `Writer` monad to create a new monad, called, perhaps, `EnvUnsafeLog`.

Changing the EnvUnsafe monad

You will have to combine the `EnvUnsafe` monad and the `Writer` monad to create a new monad, called, perhaps, `EnvUnsafeLog`.

`EnvUnsafe = Reader + Ok`

```
data Unsafe a = Error String | Ok a deriving (Show, Eq)

data EnvUnsafe env a = EnvUnsafe (env -> Unsafe a )

instance Functor (EnvUnsafe e) where
  -- fmap :: (a -> b) -> EnvUnsafe env a -> EnvUnsafe env b
  fmap f (EnvUnsafe g) = EnvUnsafe $ \e -> case g e of
                                             Error str -> Error str
                                             Ok x -> Ok (f x)

instance Monad (EnvUnsafe env) where
  --return :: a -> EnvUnsafe a
  return a = EnvUnsafe (\_ -> Ok a)

  --(>>=) :: EnvUnsafe a -> (a -> EnvUnsafe b) -> EnvUnsafe b
  (EnvUnsafe g) >>= f = EnvUnsafe $ \e -> case g e of
                                            Error str -> Error str
                                            Ok x -> let (EnvUnsafe h) = f x
                                                    in h e
```

`Writer`

```
data Writer a = Writer a [b]

-- Make it into a functor

instance Functor Writer where
  -- fmap :: (a -> b) -> Writer a -> Writer b
  fmap f (Writer x log) = Writer (f x) log


-- boilerplate, don't touch!
instance Applicative Writer where
  pure = return
  (<*>) = ap        -- imported from Control.Monad


instance Monad Writer where

  -- return :: a -> Writer a
  return x = Writer x []

  -- (>>=) :: Writer a -> (a -> Writer b) -> Writer b
  (Writer x log) >>= f = let (Writer y log2) = f x
                         in (Writer y (log ++ log'))
```

`EnvUnsafeLog = EnvUnsafe + Writer`

```
data Unsafe a = Error String | Ok a deriving (Show, Eq)

data EnvUnsafeLog envType resType logType = EnvUnsafeLog (envType -> (Unsafe resType,[logType]))
```

Changing the EnvUnsafe monad

These changes will also be reflected in the Value returned by eval, since a function may now write to the log AND return a value:

```
data Unsafe a = Error String | Ok a deriving (Show, Eq)

data EnvUnsafeLog env a = EnvUnsafeLog (env -> (Unsafe a,[String]))
```

```
data Val = I Integer | F Double | B Bool | C Char
         | Ls [Val]
         | Fun (Val -> (Unsafe Val,[String]))
               ---------------------------
```

Static Checking

Static error checking takes place after parsing but before evaluation:

- Implement a static check that takes in an `Ast` and warns when a variable is used when not declared. For instance `\ x -> y + 10` should warn something like "y is not in scope". This will not be part of your parser or interperter(eval), but should be implemented in a separate `check` function which is normally executed between the parser and the evaluator.

This is not TOO different than your check for whether a lambda expression was closed (had no free variables) in Week 9 except it goes the opposite direction (outside to the innermost terms):

```
freeVars :: Term -> Set String
freeVars (Lam v bod) = Set.delete v $ freeVars bod -- TODO: undefined
freeVars (Var v) = Set.singleton v
freeVars (App f a) = freeVars f `Set.union` freeVars a
```

```
isClosed :: Term -> Bool
isClosed t = Set.null (freeVars t)
```

Creating a REPL: Interactive Programming with the IO Monad

- 5pt A Read-Eval-Print loop, so that users can work interactively with your language, including preloading a Prelude-like initialization file. You would need to learn about the IO monad (start with Chapter 10 in Hutton).

The **do** notation (with a new monad, the IO Monad) provides a clean way to deal with computations whose context is input and output with the user, say the Keeper of the Bridge of Death:

```
bridgeOfDeath =
    do putStr "What is your  name? "
       name <- getLine
       putStr "What is your quest? "
       quest <- getLine
       putStr "What is your favorite color? "
       color <- getLine

Main> bridgeOfDeath
What is your  name? Sir Launcelot of Camlot
What is your quest? To seek the Holy Grail
What is your favorite color? Blue
Right. Off you go, Sir Launcelot of Camlot!
```

Interactive Programming with the IO Monad

The IO Monad is polymorphic and three basic IO functions which operate in the context of the screen and keyboard to read and write characters:

```
-- get a character from the keyboard via repl
getChar :: IO Char
getChar =  ... built in, you don't want to know....

-- write a character to the repl
putChar :: Char -> IO ()      -- returns null value
putChar c =  ... built in, you don't want to know....


-- just return a value in Haskell code, no input/output
return :: a -> IO a
return v =  ... built in, you don't want to know....
```

Interactive Programming with the IO Monad

```
act :: IO (Char, Char)
act = do x <- getChar
         getChar
         y <- getChar
         return (x,y)
```

```
Main> act
4
6('4','6')

Main> act
a
b('a','b')

Main>
```

Interactive Programming with the IO Monad

How to loop through user input and respond?

```
main = do putStr "Input> "
          line <- getLine
          putStrLn line
          main
```

```
Repl> main
Input> hi there!
hi there!
Input> ok
ok
Input> how
how
Input> do
do
Input> I end this>
I end this>
Input> ^CInterrupted.
Repl>
```

Interactive Programming with the IO Monad

Adding bells and whistles is not too hard....

```
miniH :: IO ()
miniH = do putStr "\nMiniHaskell> "
           line <- getLine
           case line of
              "" -> miniH
              ('q':_) -> putStrLn "Bye!\n"
              ('Q':_) -> putStrLn "Bye!\n"
              inp     -> do putStrLn $ process inp
                            miniH

process :: String -> String
process s = s
```

To create a REPL:

1. Add a global Map which is carried along during interaction (either explicit parameter or Reader monad);
2. Add a definition "let x = e" to the Ast as (Def String Ast), which inserts a definition into the global Map;
3. Modify eval so it checks this global evironment after checking local environment.

```
Main> miniH

MiniHaskell> 7
7

MiniHaskell> hi there!
hi there!

MiniHaskell>

MiniHaskell>

MiniHaskell> q
Bye!

Main>
```

Simple Static Type Checking

Static Checking

- 5pt Warn when a var is intruduced but never used
- 15pt Checking simple types, where every variable has a type annotation
- 20-30pt Advanced type checking: Bidirectional, Hindly-milner, or dependently typed **

Simple Static Type Checking

A recursive, bottom-up strategy works well for static type checking:

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

data Type = IntegerType | FloatType deriving (Eq, Show)

data Ast = ValInt Integer
         | ValFloat Double
         | Plus Ast Ast
         deriving Show

-- simple types

getType :: Ast -> Ok Type
getType (ValInt _ )   = return IntegerType
getType (ValFloat _ ) = return FloatType
getType (Plus x y)    =
   case (getType x, getType y) of
      (Ok IntegerType, Ok IntegerType) -> return IntegerType
      (Ok FloatType, Ok FloatType)     -> return FloatType
      (_ , _ ) -> Error $ "Type error for " ++ show (Plus x y)
```

```
Main> getType (ValInt 5)
Ok IntegerType
Main> getType (Plus (Plus (ValInt 5) (ValInt 2)) (ValInt 6))
Ok IntegerType
Main> getType (Plus (Plus (ValInt 5) (ValFloat 2.4)) (ValInt 6))
Error "Type error for Plus (Plus (ValInt 5) (ValFloat 2.4)) (ValInt 6)"
```

Simple Static Type Checking

This can be modified in various ways, for example, to allow coercion (integer -> float):

```
-- allow coercion

getType2 :: Ast -> Ok Type
getType2 (ValInt _ )   = return IntegerType
getType2 (ValFloat _ ) = return FloatType
getType2 (Plus x y)    =
   case (getType2 x, getType2 y) of
      (Ok IntegerType, Ok IntegerType) -> return IntegerType
      (Ok FloatType,   Ok IntegerType) -> return FloatType
      (Ok IntegerType, Ok FloatType  ) -> return FloatType
      (_ , _ ) ->  Error $ "Type error for " ++ show (Plus x y)
```

Simple Static Type Checking

For your project you would need to change the syntax so that whenever variables are declared, you must give the type:

```
(\x :: int -> x + 1)          (let x :: float = 3.4 in x * 2)
```

This information is stored in the Ast:

```
(Lambda String Type Ast Ast)    (Let String Type Ast Ast)
```

and then used to verify all types, using all the rules you would ordinarily use for dynamic checking (e.g., head must take a list as argument), plus:

1. No types are polymorphic;
2. All elements in lists must have same type; and
3. Functions follow the usual typing rules:

$$\frac{x :: a \qquad e :: b}{(\backslash x \rightarrow e) :: (a \rightarrow b)} \qquad\qquad \frac{f :: (a \rightarrow b) \qquad e :: a}{(f\ e) :: b}$$

The Problem with Recursion...

The basic project language does not allow recursion!

Why? Because of the scope of a let definition:

```
let fact = (\n -> if n < 2 then 1 else n * (fact(n - 1)) in fact 5
```

What is the scope of the definition of fact?

```
let fact = (\n -> if n < 2 then 1 else n * (fact(n - 1)) in fact 5
                                          ----------
```

unbound

How to solve this?

- Create a special `letrec` function which extends the scope of let definitions to include the defining expression; or
- Create a global memory whose scope is the entire program, in which variables can be looked up any time.