Haskelterpreters

Definitional Interpreters in Haskell

COMS 6998: Types, Languages and Compilers

Xurxo Riesco (xr2154)

May 8th, 2023

1 Introduction

In his paper "Definitional Interpreters for Higher Order Programming Languages," John Reynold's explores various characteristics that can be used to categorize programming languages by defining interpreters for said languages in a pseudo-language more accessible to the reader. This paper aims to provide a similar exploration in the context of a yet better understood language, Haskell. The two key features what will govern the exploration are order-of-application dependence and use of higher-order functions.

Based on the order-of-application dependence the language defined by an interpreter can either be follow one of two different evaluation strategies: call by value or call by name. In call by value the arguments to a function are to be evaluated prior to the function call itself; namely, the evaluation of an applicative expression starts only after the operand and the operators have themselves been evaluated. In call by name, the evaluation begins as soon as the operator is evaluated, but the operands are evaluated at the time the application is dependent on the value of the operand, in other words, a function call receives un-evaluated arguments and only evaluates them at the time they are needed within the function body, which might never happen.

Similarly, based on the use of higher-order functions the defined language can either be in one of two categories: higher-order or first-order.Higher-order programming languages can operate on functions as data, meaning a function can take a function as an argument or return a function as its re-

turn value. A language in which functions can't serve as arguments or return values of other functions, is considered to be first-order.

Based on both of these categories of classification, the original paper presents four different interpreters, as shown in the figure below. The interpreter with order-of-application dependence and use of higher-order functions is considered to be Interpreter I, the rest are numbered clockwise from it.

Order-of-application dependence:	Use of higher-order functions: yes	no
yes	direct interpreter for GEDANKEN	McCarthy's definition of LISP
no	Morris-Wadsworth method	SECD machine, Vienna definition

Figure 1: Categorized interpreters.

2 Approach

While the original paper focused on defining an implementation of Interpreter I and derives the remaining three from it. The approach followed by this paper will deviate from it by first formalizing the characteristics of the defined language as opposed to the defining language, then providing an implementation of Interpreter I for the defined language written in Haskell and showcasing the necessary changes to alter the order-of-application dependence, providing a Haskell implementation of Interpreter IV. Alternatively, since the exploration of the paper is center around expressiveness, an implementation of Interpreter I in the defined language of Interpreter IV will be used in the exploration. An Implementation of Interpreter II and Interpreter III in Haskell won't be providing, leaving the exploration of higher-order vs first-order functions as future work.

3 Defined Language

3.1 Types

The support of two primitive types in the defined language are sufficient for exemplary purposes. The two primitive types are the boolean: `ValBool`, which will take on values `True` or `False`, and the integer type: `Valint`, which will take on numerical values. Since the function type is used as one of the exploration criteria, its definition will be present along with each particular interpreter, but the type, or an equivalent variation, is present all cases throughout the defined language.

3.2 Expressions

The defined language supports the following syntax in defining its expressions:

3.2.1 Constant

Any value conforming a primitive type can itself be considered a constant expression, a constant expression for the value `x` is written as: `x`.

3.2.2 Variable

The defined language supports binding; so variables (which follow the standard variable naming conventions of a sequence of alpha-numerical characters where the initial character is not a number) are to be considered expressions, a variable `x` is written as: `x`.

3.2.3 Application

Application of the operator `x` to the operand `y` is written as: `x y`.

3.2.4 Conditional

A basic control flow operation where the consequence `c` is evaluated if the result of the evaluation of the premise `p` is True, and where the alternative `a` is evaluated if the evaluation of the premise `p` results in False is written as: `if p then c else a`.

3.2.5 Lambdas

A lambda function which takes argument `x` and whose function is body is `b` is written as:
`x -> b`.

3.2.6 Recursive Let

A recursive binding of the `letrec` form specifies a function `f` which takes arguments `x` and whose body is `b` in an enviroment `e` is written as: `letrec f x = b in e`.

3.3 Lexer and Parser

The lexer and parser for the defined language add support for parenthesising and for all the aforementioned expressions and turns a set of characters into a list of tokens which then conform the following Abstract Syntax Tree, common to the defined language of all three interpreters and shown in the definining language, Haskell. Additionally, an environment type is provided to facilitate the evaluation function.

```
data Exp
  = Const Val
  | Var Var
  | App Exp Exp
  | Lambda Var Exp
  | Cond Exp Exp Exp
  | LetRec Var Var Exp Exp

type Var = String

data Val
  = ValBool Bool
  | ValInt Int

type Env = [(Var, Val)]
```

3.4 Interpreter I

3.4.1 Function Type

In the first interpreter, the function type is defined follows:

```
data Val =
    ...
    | ValFun (Val -> Val)
```

3.4.2 Evaluation

The defined language follows the following evaluation rules defined in Haskell and conforming Interpreter I:

```
eval :: Exp -> Env -> Val
eval (Const v) env = v
eval (Var var) env =
  case lookupVar var env of
    Just val -> val
    Nothing -> error $ "Unbound variable: " ++ var
eval (App e1 e2) env =
  let v1 = eval e1 env
  in case v1 of
      ValFun f -> let v2 = eval e2 env in f v2
      _ -> error $ "Non-function application: " ++ show v1
eval (Lambda var e) env = ValFun (\arg -> eval e ((var, arg) : env))
eval (Cond e1 e2 e3) env =
  let v1 = eval e1 env
  in case v1 of
      ValBool b -> if b then eval e2 env else eval e3 env
      _ -> error $ "Non-boolean in condition: " ++ show v1
eval (LetRec fun arg body e) env =
  let recEnv = (fun, ValFun
    (\argVal -> eval body ((arg, argVal) : recEnv))) : env
  in eval e recEnv
```

3.4.3 Exploration

The implementation provided by Reynold's does not clearly represent the order of application-dependence for the defined language, since the application case defers to the evaluation order of the defining language. Haskell is not considered to by call by name itself, but rather call by need. However, both call by name and call by need defer the evaluation of unneeded operands until strictly required by the operator, and only differ in that call by need optimizes call by name by storing the result of the evaluation of the operand and accessing the same evaluated value if it is further required by the operator. For the purposes of this exploration, this distinction is minor, and does not affect the results of comparing order of application dependence with its counterpart. Due to this feature in the defining language, and the lack of further processing in the application case, the defined language under Interpreter I, is understood to follow call by name semantics. The order of application dependence can be clearly shown on the following example program written in the defined language:

```
letrec loop x =
    loop x
in letrec test x =
    letrec innerTest y =
        x
    in innerTest
in test 42 (loop 0)
```

Under Interpreter I, the example program, will evaluate to `ValInt 42` as the operand `loop 0` is never evaluated. It will be shown later, that if the defined-language is to follow a call-by-name evaluation strategy, the evaluation of the same test program would result in an infinite loop. The implementation of Interpreter I which uses higher-order functions does not illustrate the concept in a very clear manner in the defined language, given that the defining language itself supports the use of higher-order functions and an implementation of higher-order functions in a first-order language would be much more illustrate. It is for that reason, that Interpreter II, which does not use higher-order functions, is to be explored now, in hopes that converting the higher-order defined language into first-order illustrates both the nature of higher-order and first-order.

3.5 Interpreter IV

3.5.1 Continuations

Prior to presenting the implementation of Interpreter IV, the concept of continuations is to be presented, as it is the governing concept behind the adaption of the defined language in Interpreter I to the defined language in Interpreter IV, and the allowing towards order-of-application independence. To provide order-of-application independence, actions are not to be performed after a function has returned, but to be embedded in a continuation to be passed to the function. That is, all functions are to be replace by functions which take an additional argument, the continuation, which it itself is a function that is to be applied to the old function and only then the result of the applying the continuation is returned.

3.5.2 Function Type

The introduction of continuations affects the function type as follows:

```
type Cont = Val -> Val
data Val =
    ...
  | ValFun (Val -> Cont -> Val)
```

3.5.3 Evaluation

The defined language follows the following evaluation rules defined in Haskell and conforming Interpreter IV:

```
eval :: Exp -> Env -> Cont -> Val
eval (Const v) env k = k v
eval (Var var) env k =
  case lookupVar var env of
    Just val -> k val
    Nothing -> error $ "Unbound variable: " ++ var
eval (App e1 e2) env k =
  eval e1 env $ \v1 ->
    case v1 of
      ValFun f -> eval e2 env $ \v2 -> f v2 k
      _ -> error $ "Non-function application: " ++ show v1
```

```
eval (Lambda var e) env k = k (ValFun (\arg -> eval e ((var, arg) : env)))
eval (Cond e1 e2 e3) env k =
  eval e1 env $ \v1 ->
    case v1 of
      ValBool b -> if b then eval e2 env k else eval e3 env k
      _ -> error $ "Non-boolean in condition: " ++ show v1
eval (LetRec fun arg body e) env k =
  let recEnv = (fun, ValFun
    (\argVal cont -> eval body ((arg, argVal) : recEnv) cont)) : env
   in eval e recEnv k
```

3.5.4 Exploration

The implementation very closely resembles that of Interpreter IV and it takes advantage of the usage of the higher order functions provided in the defining language while maintaining support for them in the defined-language. One caveat of continuation passing style, which is mentioned in Reynold's paper is that, in theory, we must take into account the difference between serious and trivial functions. Reynold's defines to be trivial any function who will always return, and serious if there is any possibility that the function might not return. In the defined language, there is no way to make this distinction prior to the evaluation of the function; therefore, order-of-application independence incurs a risk, as any serious function that does not return might lead to the program to hang. Taking into account that fact does however provide us with a simple way to verify that the defined language is indeed order-of-application dependence for the example program defined in the exploration of Interpreter I and containing a serious function which never returns, namely:

```
letrec loop x = loop x in loop 0
```

leads to the evaluation of the whole program never to be achieved. If we take the sequence of the application case as described in Reynold's paper. It is then clear that the hanging is indeed the correct behavior of the program for the defined language of Interpreter IV:

1. Evaluate the operator
2. Evaluate the operand
3. Apply the operator to the operand
4. Apply the continuation to the result.

3.6 Defined to Defining

By extending the features of the defined language to introduce a data container and support for strings as a primitive type. The defined language becomes powerful enough to be used as the defined language. It is for that reason that the implementation of Interpreter IV will be extended with support for two new primitive types, a tuple `ValPair` which holds two `a` and `b` elements irrespective of their type and can be constructed in the following manner `[a,b]`, and strings `ValString` which holds a sequence of characters `s` and can be constructed in the following manner `"s"`. For readiness purposes, the implementation will also be extended to support `let` bindings, which give a expression in their body b the name `n` within the context of environment `e`, thet can be constructed as: `let n = b in e`. Two basic functions, `fst` and `snd` are also introduced to support manipulation of the `ValPair` type, and are defined as follows in the initial enviroment:

```
"fst": ValFun (\ (ValPair (v1,v2)) k -> k v1))
"snd": ValFun (\ (ValPair (v1,v2)) k -> k v2))
```

The original defined language can now be defined with the constructs provided by Interpreter IV in the new defined language.

3.6.1 Big Step Evaluation Rules

Prior to considering the actual implementation, which might be harder to follow given that the defined language has now turned to be the defining language, and may be unfamiliar to the reader, the Big Step Evaluation Rules are presented below:

e ::=		*Expressions*
	x	Var
|	$e_1 e_2$	App
|	λx `->` e	Lambdas
|	`letrec` $x_1 x_2 = e_1$ `in` e_2	LetRec Bindings
|	`if` e_1 `then` e_2 `else` e_3	Conditionals
|	i	Int
|	b	Bool
|	s	String
|	$e_1 == e_2$	Equality
|	`[` e_1`,` e_2 `]`	Tuples
|	`fst` e	First
|	`snd` e	Second
|	`let` $x = e_1$ `in` e_2	Let Bindings

$$\frac{}{x \Downarrow x}$$

$$\frac{e_1 \Downarrow v_1 \qquad \dfrac{e_2 \Downarrow v_2 \qquad v_1 \Downarrow v_2}{v_1 e_2 \Downarrow v}}{e_1 e_2 \Downarrow v}$$

$$\frac{}{\lambda x -> e \Downarrow \lambda -> e}$$

$$\frac{Z(\lambda f x.e1) \Downarrow v_1 \qquad [f := v1]e2 \Downarrow v}{\texttt{letrec } x_1 x_2 = e_1 \texttt{ in } e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow v_1 \qquad true \ \texttt{eq} \ v_1}{\texttt{if} \ e1 \ \texttt{then} \ e_2 \ \texttt{else} \ e_3 \Downarrow \texttt{if true then} \ e_2 \ \texttt{else} \ e_3 \Downarrow v_2}$$

$$\frac{e_1 \Downarrow v_1 \qquad false \ \texttt{eq} \ v_1}{\texttt{if} \ e1 \ \texttt{then} \ e_2 \ \texttt{else} \ e_3 \Downarrow \texttt{if false then} \ e_2 \ \texttt{else} \ e_3}$$

$$\frac{e_2 \Downarrow v_3}{\texttt{if true then} \ e_2 \ \texttt{else} \ e_3 \Downarrow v_2}$$

$$\frac{e_3 \Downarrow v_3}{\texttt{if false then} \ e_2 \ \texttt{else} \ e_3 \Downarrow v_3}$$

$$\frac{i \in \mathbb{Z}}{i \Downarrow i}$$

$$\frac{b \in \texttt{True,False}}{b \Downarrow b}$$

$$\frac{}{s \Downarrow s}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e1 == e2 \Downarrow v1 == v2}$$

$$\frac{v1 \ \texttt{eq} \ v_2}{v1 == v2 \Downarrow true}$$

$$\frac{v1 \ \texttt{not eq} \ v_2}{v1 == v2 \Downarrow false}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{\texttt{[} \ e_1\texttt{,} \ e_2 \ \texttt{]} \Downarrow \texttt{[} \ v_1\texttt{,} \ v_2 \ \texttt{]}}$$

$$\frac{}{\texttt{fst [}v_1\texttt{,} \ v_2 \ \texttt{]} \Downarrow v_1}$$

$$\frac{}{\texttt{snd [}v_1\texttt{,} \ v_2 \ \texttt{]} \Downarrow v_2}$$

$$\frac{e_1 \Downarrow v_1 \qquad [x := v1]e2}{\texttt{let} \ x = e1 \ \texttt{in} \ e2 \Downarrow v}$$

3.6.2 Interpreter I revisited

The implementation of Interpreter I in the defined language of Interpreter IV is as follows:

```
letrec eval program =
  let exp = fst program in
    let env = snd program in
      if equalS (fst exp) "const"
        then snd exp
      else if equalS (fst exp) "var"
      then
          letrec lookupVar tup =
            let var = fst tup in
            let env = snd tup in
            if equalS (fst (fst env)) "END"
                then "Error: Unbound variable"
                else if equalS var (fst (fst env))
                    then snd (fst env)
                    else lookupVar [var, snd env]
          in lookupVar [snd exp, env]
      else if equalS (fst exp) "cond"
        then
          let e1 = fst (snd exp) in
          let e2 = fst (snd (snd exp)) in
          let e3 = snd (snd (snd exp)) in
          let cond = eval [e1, env] in
          if isValBool cond
            then if equalB (eval [e1,env]) True
              then eval [e2,env] else eval [e3,env]
          else "Error: Non boolean condition"
      else if equalS (fst exp) "lambda"
        then
          let var = fst (snd exp) in
          let e = snd (snd exp) in
          letrec f argExp = eval [e, [[var, argExp], env]] in
          f
      else if equalS (fst exp) "app"
        then
```

```
        let e1 = fst (snd exp) in
        let e2 = snd (snd exp) in
        let v1 = eval [e1, env] in
        if isValFun v1 then v1 (eval [e2, env])
        else "Error: Non function application"
    else if equalS (fst exp) "letrec"
      then
        let fun = (fst (fst (snd exp))) in
        let var = (snd (fst (snd exp))) in
        let body = (fst (snd (snd exp))) in
        let e = snd (snd (snd exp)) in
        letrec recEnv x =
            [[fun, letrec f arg =
                eval [body, [[var, arg], (recEnv x)]]
            in f], env]
        in eval [e, (recEnv 0)]
      else "Error"
```

3.6.3 Re-exploration

This implementation differs from the original implementation of Interpreter I in that it more closely resembles the application case to be what is described in Reynold's paper. Namely it removes the wrapping of `e2` in a lambda expression and therefore the evaluation order of the defined language is delegated to be the evaluation order of the defining language, which is order-of-application independence given that is built on Interpreter IV. Therefore, the adaption of the test program shown in the exploration of Interpreter I:

```
["letrec", [["loop","x"],
    [["app", [["var", "loop"],["var", "x"]]],
    ["letrec", [["test", "x"],
        [["letrec", [["innerTest", "y"], [["var", "x"],
    ["var", "innerTest"]]]],
["app", [["app", [["var", "test"], ["const", 42]]],
["app", [["var", "loop"], ["const", 0]]]]]]]]]]
```

will yield the results discussed on the exploration of Interpreter IV and not in Interpreter I. However, recuperating the order-of-application dependence

is a fairly straightforward process, as the interpreter just needs to modify the application case as shown:

```
else if equalS (fst exp) "app" then
  let e1 = fst (snd exp) in
  let e2 = snd (snd exp) in
  let v1 = eval [e1, env] in
  if isValFun v1 then v1 (eval [e2, env])
  else "Error: Non function application"
```

which can be verified by running the same test program and observing that the result now matches what was expected of Interpreter I.

Without said modification, the defined language of Interpreter I can be understood to be the defined language of Interpreter IV, and we have therefore successfully implemented a meta-circular interpreter of Interpreter IV as well as proven that the order-of-application dependence of an Interpreter can be fairly easily adapted to match or not the defining language.

3.7 Conclusions

As concluded by Reynolds, the call by value strategy could be understood as more predictable and more inline with the operational reality of computers, and the call by name strategy can be understood as more efficient when computations are not needed, but may be inefficient when computations are re-used. Languages like Haskell opted for an approach that more closely resembles call by name, while further optimizing its performance by avoiding re computations with the introduction of call by need. Showcasing how interpreters are finding clever ways to reduce the trade-offs between one implementation versus the other. However, the expressiveness power of both approaches has been shown to be nearly identical as the defined language with the call by value strategy can itself define a language that follows the call by name strategy while itself being defined in a language that uses a call by name strategy.

3.8 Further Work

One of the short comings of the paper, is that the support for higher-order functions is not clearly illustrated in Interpreter I, for the defining language used already has built-in support for higher-order functions. Haskell would

not provide any further illustrations in that regard given that it also supports higher-order functions. In order to better understand how higher-order functions are supported, it would be more illustrative to shift towards a first-order language as the defining language, which would allow a similar meta circular exploration to the one proposed in this paper by building Interpreter I on top of a language like C and then building a meta circular Interpreter of I that supports Interpreter II and explores the denationalization technique showcased by Reynolds.

4 Code Listings

Only the most relevant files are shown below. To be certain of the results demonstrating correct function, a bunch of more test programs were used in bith interpreters and are submitted as a tar.gz despite not being shown below.

4.1 Interpreter I

4.1.1 AST.hs

```
1  {-# LANGUAGE InstanceSigs #-}
2
3  module AST where
4
5  import Control.Monad.Reader (Reader)
6
7  data Exp
8    = Const Val
9    | Var Var
10   | App Exp Exp
11   | Lambda Var Exp
12   | Cond Exp Exp Exp
13   | LetRec Var Var Exp Exp
14
15 type Var = String
16
17 data Val
18   = ValBool Bool
19   | ValInt Int
20   | ValFun (Val -> Val)
21
22 type Env = [(Var, Val)]
23
24 instance Show Val where
25   show :: Val -> String
26   show (ValBool b) = "ValBool " ++ show b
27   show (ValInt i) = "ValInt " ++ show i
28   show (ValFun _) = "ValFun <function>"
```

4.1.2 Lexer.hs

```
1  module Lexer (lexer, parens, integer, reservedOp,
2                reserved, identifier, symbol, whitespace) where
3
4  import Control.Monad (void)
5  import Text.Parsec.Language (haskellStyle)
6  import Text.Parsec.String (Parser)
7  import qualified Text.Parsec.Token as P
8
9  lexer :: P.TokenParser ()
10 lexer = P.makeTokenParser haskellStyle
11
12 parens :: Parser a -> Parser a
13 parens = P.parens lexer
14
15 integer :: Parser Integer
16 integer = P.integer lexer
17
18 reservedOp :: String -> Parser ()
19 reservedOp op = void $ P.reservedOp lexer op
20
21 reserved :: String -> Parser ()
22 reserved = P.reserved lexer
23
24 identifier :: Parser String
25 identifier = P.identifier lexer
26
27 symbol :: String -> Parser String
28 symbol = P.symbol lexer
29
30 whitespace :: Parser ()
31 whitespace = P.whiteSpace lexer
```

4.1.3 Parser.hs

```
1  module Parser (parseExp, parseProgram) where
2
3  import AST (Exp (..), Val (ValBool, ValInt),)
4  import Control.Applicative ((<|>))
5  import Lexer
6    ( identifier,
7      integer,
8      reserved,
9      reservedOp,
10     symbol,
11     whitespace,
12     parens,
13   )
14 import Text.Parsec (ParseError, eof, parse, many, notFollowedBy, noneOf, try)
15 import Text.Parsec.String (Parser)
16 import Control.Monad (void)
17
18 parseExp :: Parser Exp
19 parseExp =
20   parseConst
21     <|> parseLambda
22     <|> parseCond
23     <|> parseLetRec
24     <|> parseApp
25     <|> parseVar
26     <|> parens parseExp
27
28 parseNonAppExp :: Parser Exp
29 parseNonAppExp =
30   parseConst
31     <|> parseLambda
32     <|> parseCond
33     <|> parseLetRec
34     <|> parens parseExp
35     <|> parseVar
36
37 parseApp :: Parser Exp
38 parseApp = do
39   e1 <- parseNonAppExp
40   es <- many (try (notFollowedBy (reserved "in" <|>
41                                   reserved "then" <|>
42                                   reserved "else") *> parseNonAppExp))
43   return $ foldl App e1 es
```

```
44
45 parseConst :: Parser Exp
46 parseConst =
47   Const . ValInt . fromIntegral <$> integer
48     <|> (reserved "True" >> return (Const (ValBool True)))
49     <|> (reserved "False" >> return (Const (ValBool False)))
50
51
52 parseVar :: Parser Exp
53 parseVar = Var <$> identifier
54
55 parseLambda :: Parser Exp
56 parseLambda = do
57   symbol "\\"
58   v <- identifier
59   reservedOp "->"
60   Lambda v <$> parseExp
61
62 parseCond :: Parser Exp
63 parseCond = do
64   reserved "if"
65   e1 <- parseExp
66   reserved "then"
67   e2 <- parseExp
68   reserved "else"
69   Cond e1 e2 <$> parseExp
70
71 parseLetRec :: Parser Exp
72 parseLetRec = do
73   reserved "letrec"
74   f <- identifier
75   x <- identifier
76   reservedOp "="
77   e1 <- parseExp
78   reserved "in"
79   LetRec f x e1 <$> parseExp
80
81 parseProgram :: String -> Either ParseError Exp
82 parseProgram = parse (whitespace *> parseExp <* eof) "<input>"
```

4.1.4 Main.hs

```
1  module Main where
2
3  import AST
4  import Eval (eval)
5  import Parser (parseProgram)
6  import System.Environment (getArgs)
7
8  env :: Env
9  env = [ ("succ",
10             ValFun (\(ValInt n) -> ValInt (n + 1)))
11       , ("equal",
12             ValFun (\ (ValInt n) -> ValFun (\ (ValInt m) -> ValBool (n == m))))
13       , ("add",
14             ValFun (\ (ValInt n) -> ValFun (\ (ValInt m) -> ValInt (n + m))))
15       , ("minus",
16             ValFun (\ (ValInt n) -> ValFun (\ (ValInt m) -> ValInt (n - m))))
17       ]
18
19
20 main :: IO ()
21 main = do
22   args <- getArgs
23   case args of
24     [filename] -> do
25       input <- readFile filename
26       case parseProgram input of
27         Left err -> putStrLn $ "Error: " ++ show err
28         Right exp -> do
29           putStrLn $ "Parsed expression: " ++ show exp
30           let val = eval exp env
31           putStrLn $ "Evaluated value: " ++ show val
32     _ -> putStrLn "Usage: interp1 <filename>"
```

4.1.5 Eval.hs

```
1  module Eval (eval) where
2
3  import AST
4  import Control.Monad.Reader (Reader, ask, asks, local, runReader)
5
6  lookupVar :: Var -> Env -> Maybe Val
7  lookupVar var [] = Nothing
8  lookupVar var ((var', val) : env)
9    | var == var' = Just val
10   | otherwise = lookupVar var env
11
12 eval :: Exp -> Env -> Val
13 eval (Const v) env = v
14 eval (Var var) env =
15   case lookupVar var env of
16     Just val -> val
17     Nothing -> error $ "Unbound variable: " ++ var
18 eval (App e1 e2) env =
19   let v1 = eval e1 env
20   in case v1 of
21       ValFun f -> let v2 = eval e2 env in f v2
22       _ -> error $ "Non-function application: " ++ show v1
23 eval (Lambda var e) env = ValFun (\arg -> eval e ((var, arg) : env))
24 eval (Cond e1 e2 e3) env =
25   let v1 = eval e1 env
26   in case v1 of
27       ValBool b -> if b then eval e2 env else eval e3 env
28       _ -> error $ "Non-boolean in condition: " ++ show v1
29 eval (LetRec fun arg body e) env =
30   let recEnv = (fun, ValFun (\argVal -> eval body ((arg, argVal) : recEnv))) : env
31   in eval e recEnv
```

4.2 Interpreter IV

4.2.1 AST.hs

```
1  {-# LANGUAGE InstanceSigs #-}
2
3  module AST where
4
5  import Control.Monad.Reader (Reader)
6
7  data Exp
8    = Const Val
9    | Var Var
10   | App Exp Exp
11   | Lambda Var Exp
12   | Cond Exp Exp Exp
13   | LetRec Var Var Exp Exp
14   | Let Var Exp Exp
15   | Pair Exp Exp
16   deriving (Show)
17
18 type Var = String
19
20 data Val
21   = ValBool Bool
22   | ValInt Int
23   | ValString String
24   | ValFun (Val -> Cont -> Val)
25   | ValPair (Val,Val)
26
27 type Cont = Val -> Val
28
29 instance Show Val where
30   show :: Val -> String
31   show (ValBool b) = "ValBool " ++ show b
32   show (ValInt i) = "ValInt " ++ show i
33   show (ValString s) = "ValString " ++ show s
34   show (ValFun _) = "ValFun <function>"
35   show (ValPair v) = "ValPair (" ++ show (fst v) ++ ", " ++ show (snd v) ++ ")"
36
37 type Env = [(Var, Val)]
```

4.2.2 Parser.hs

```
1  module Parser (parseExp, parseProgram) where
2
3  import AST (Exp (..), Val (ValBool, ValInt, ValPair, ValString),)
4  import Control.Applicative ((<|>))
5  import Lexer
6    ( identifier,
7      integer,
8      reserved,
9      reservedOp,
10     symbol,
11     whitespace,
12     parens,
13   )
14 import Text.Parsec (ParseError, eof, parse, many, notFollowedBy, noneOf, try)
15 import Text.Parsec.String (Parser)
16 import Control.Monad (void)
17
18 parseLet :: Parser Exp
19 parseLet = do
20   reserved "let"
21   v <- identifier
22   reservedOp "="
23   e1 <- parseExp
24   reserved "in"
25   Let v e1 <$> parseExp
26
27 parseExp :: Parser Exp
28 parseExp =
29   parseConst
30     <|> parseLambda
31     <|> parseCond
32     <|> parseLetRec
33     <|> parseLet
34     <|> parseApp
35     <|> parseVar
36     <|> parens parseExp
37
38 parseNonAppExp :: Parser Exp
39 parseNonAppExp =
40   parseConst
41     <|> parseLambda
42     <|> parseCond
43     <|> parseLetRec
```

```
44      <|> parens parseExp
45      <|> parseVar
46
47  parseApp :: Parser Exp
48  parseApp = do
49    e1 <- parseNonAppExp
50    es <- many (try (notFollowedBy (reserved "in" <|>
51                                    reserved "then" <|>
52                                    reserved "else") *> parseNonAppExp))
53    return $ foldl App e1 es
54
55  parseConst :: Parser Exp
56  parseConst =
57    Const . ValInt . fromIntegral <$> integer
58      <|> (reserved "True" >> return (Const (ValBool True)))
59      <|> (reserved "False" >> return (Const (ValBool False)))
60      <|> parseString
61      <|> parsePairConst
62
63
64  parseString :: Parser Exp
65  parseString = do
66    symbol "\""
67    s <- many (noneOf "\"")
68    symbol "\""
69    return $ Const (ValString s)
70
71  parsePairConst :: Parser Exp
72  parsePairConst = do
73    symbol "["
74    e1 <- parseExp
75    symbol ","
76    e2 <- parseExp
77    symbol "]"
78    return $ Pair e1 e2
79
80
81  parseVar :: Parser Exp
82  parseVar = Var <$> identifier
83
84  parseLambda :: Parser Exp
85  parseLambda = do
86    symbol "\\"
87    v <- identifier
88    reservedOp "->"
```

```
89    Lambda v <$> parseExp
90
91  parseCond :: Parser Exp
92  parseCond = do
93    reserved "if"
94    e1 <- parseExp
95    reserved "then"
96    e2 <- parseExp
97    reserved "else"
98    Cond e1 e2 <$> parseExp
99
100 parseLetRec :: Parser Exp
101 parseLetRec = do
102   reserved "letrec"
103   f <- identifier
104   x <- identifier
105   reservedOp "="
106   e1 <- parseExp
107   reserved "in"
108   LetRec f x e1 <$> parseExp
109
110 parseProgram :: String -> Either ParseError Exp
111 parseProgram = parse (whitespace *> parseExp <* eof) "<input>"
```

4.2.3 Main.hs

```
1  module Main where
2
3  import AST
4  import Eval (eval)
5  import Parser (parseProgram)
6  import System.Environment (getArgs)
7
8  type Cont = Val -> Val
9
10 isFun :: Val -> Bool
11 isFun (ValFun _) = True
12 isFun _ = False
13
14 isBool :: Val -> Bool
15 isBool (ValBool _) = True
16 isBool _ = False
17
18 isPair :: Val -> Bool
19 isPair (ValPair _) = True
20 isPair _ = False
21
22 env :: Env
23 env = [ ("succ",
24              ValFun (\(ValInt n) k -> k (ValInt (n + 1))))
25        , ("equal",
26              ValFun (\ (ValInt n) k ->
27                  k (ValFun (\ (ValInt m) k' -> k' (ValBool (n == m))))))
28        , ("equalB",
29              ValFun (\ (ValBool b) k ->
30                  k (ValFun (\ (ValBool c) k' -> k' (ValBool (b == c))))))
31        , ("equalS",
32              ValFun (\ (ValString s) k ->
33                  k (ValFun (\ (ValString q) k' -> k' (ValBool (s == q))))))
34        , ("fst",
35              ValFun (\ (ValPair (v1,v2)) k -> k v1))
36        , ("snd",
37              ValFun (\ (ValPair (v1,v2)) k -> k v2))
38        , ("add",
39              ValFun (\ (ValInt n) k ->
40                   k (ValFun (\ (ValInt m) k' -> k' (ValInt (n + m))))))
41        , ("minus",
42              ValFun (\ (ValInt n) k ->
43                  k (ValFun (\ (ValInt m) k' -> k' (ValInt (n - m))))))
```

```
44          , ("isValFun",
45                ValFun (\ f k -> k (ValBool (isFun f))))
46          , ("isValBool",
47                ValFun (\ b k -> k (ValBool (isBool b))))
48          , ("isValPair",
49                ValFun (\ p k -> k (ValBool (isPair p))))
50          ]
51
52
53  main :: IO ()
54  main = do
55    args <- getArgs
56    case args of
57      [filename] -> do
58        input <- readFile filename
59        case parseProgram input of
60          Left err -> putStrLn $ "Error: " ++ show err
61          Right exp -> do
62            putStrLn $ "Parsed expression: " ++ show exp
63            let val = eval exp env id
64            putStrLn $ "Evaluated value: " ++ show val
65      _ -> putStrLn "Usage: interp4 <filename>"
```

4.2.4 Eval.hs

```
1  module Eval (eval, runReader) where
2
3  import AST
4  import Control.Monad.Reader (Reader, ask, asks, local, runReader)
5
6  lookupVar :: Var -> Env -> Maybe Val
7  lookupVar var [] = Nothing
8  lookupVar var ((var', val) : env)
9    | var == var' = Just val
10   | otherwise = lookupVar var env
11
12 eval :: Exp -> Env -> Cont -> Val
13 eval (Const v) env k = k v
14 eval (Var var) env k =
15   case lookupVar var env of
16     Just val -> k val
17     Nothing -> error $ "Unbound variable: " ++ var
18 eval (App e1 e2) env k =
19   eval e1 env $ \v1 ->
20     case v1 of
21       ValFun f -> eval e2 env $ \v2 -> f v2 k
22       _ -> error $ "Non-function application: " ++ show v1
23 eval (Lambda var e) env k = k (ValFun (\arg -> eval e ((var, arg) : env)))
24 eval (Cond e1 e2 e3) env k =
25   eval e1 env $ \v1 ->
26     case v1 of
27       ValBool b -> if b then eval e2 env k else eval e3 env k
28       _ -> error $ "Non-boolean in condition: " ++ show v1
29 eval (LetRec fun arg body e) env k =
30   let recEnv =
31     (fun, ValFun (\argVal cont -> eval body ((arg, argVal) : recEnv) cont)) : env
32    in eval e recEnv k
33 eval (Let var e1 e2) env k =
34   eval e1 env $ \v1 ->
35     let newEnv = (var, v1) : env
36      in eval e2 newEnv k
37 eval (Pair e1 e2) env k =
38   eval e1 env $ \v1 ->
39     eval e2 env $ \v2 ->
40       k (ValPair (v1, v2))
```

4.3 MetaCircular

```
1  letrec eval program =
2    let exp = fst program in
3      let env = snd program in
4        if equalS (fst exp) "const"
5          then snd exp
6        else if equalS (fst exp) "var"
7        then
8            letrec lookupVar tup =
9              let var = fst tup in
10             let env = snd tup in
11             if equalS (fst (fst env)) "END"
12                 then "Error: Unbound variable"
13                 else if equalS var (fst (fst env))
14                     then snd (fst env)
15                     else lookupVar [var, snd env]
16           in lookupVar [snd exp, env]
17       else if equalS (fst exp) "cond"
18         then
19           let e1 = fst (snd exp) in
20           let e2 = fst (snd (snd exp)) in
21           let e3 = snd (snd (snd exp)) in
22           let cond = eval [e1, env] in
23           if isValBool cond
24             then if equalB (eval [e1,env]) True
25               then eval [e2,env]
26               else eval [e3,env]
27           else "Error: Non boolean condition"
28       else if equalS (fst exp) "lambda"
29         then
30           let var = fst (snd exp) in
31           let e = snd (snd exp) in
32           letrec f argExp = eval [e, [[var, argExp], env]] in
33           f
34       else if equalS (fst exp) "app"
35         then
36           let e1 = fst (snd exp) in
37           let e2 = snd (snd exp) in
38           let v1 = eval [e1, env] in
39           if isValFun v1 then v1 (eval [e2, env])
40           else "Error: Non function application"
41     else if equalS (fst exp) "letrec"
42       then
43         let fun = (fst (fst (snd exp))) in
```

```
44          let var = (snd (fst (snd exp))) in
45          let body = (fst (snd (snd exp))) in
46          let e = snd (snd (snd exp)) in
47          letrec recEnv x =
48              [[fun, letrec f arg = eval [body, [[var, arg], (recEnv x)]] in f], env]
49          in eval [e, (recEnv 0)]
50        else "Error"
51  in eval
52  [
53    ["letrec", [["fib", "n"],
54      [["cond", [["app", [["app",
55        [["var", "equal"], ["var", "n"]]], ["const", 0]]],
56        [["const", 1],
57        ["cond", [["app", [["app", [["var", "equal"], ["var", "n"]]], ["const", 1]]],
58          [["const", 1],
59          ["app", [["app",
60            [["var", "add"],
61              ["app", [["var","fib"], ["app", [["app",
62                [["var", "minus"], ["var", "n"]]], ["const", 1]]]]]],
63              ["app", [["var","fib"], ["app",
64              [["app", [["var", "minus"], ["var", "n"]]], ["const", 2]]]]]]]]]]]]],
65    ["app", [["var", "fib"], ["const", 6]]]]]]
66  ,
67  [
68    ["succ", \x -> succ x],
69    [["minus", \x -> \y -> minus x y],
70    [["equal", \x -> \y -> equal x y],
71    [["add", \x -> \y -> add x y],
72    [["END", False], False]]]]]
73  ]
```

5 References

References

[1] https://dl.acm.org/doi/10.1145/800194.805852