# Homework 04 : CS 558

Taylor Berger

November 4, 2015

## 1 Evaluation Updates

With the updated definition of possible term in the Lambda Calculus language, the evaluation procedures had to be updated to reflect the data constructor changes. Below is the updated version of terms in the language of Lambda Calculus extended with Booleans and Naturals:

```haskell
type VarName = String
data Term = Identifier VarName |
  Abstraction VarName Term |
  Application Term Term |
  If Term Term Term |
  Succ Term |
  Pred Term |
  IsZero Term |
  Tru |
  Fls |
  Zero deriving (Eq, Show)
data Type = Function Type Type |
  Boole |
  Nat |
  NullType deriving Eq
instance Show Type where
  show Boole = "Boolean"
  show Nat = "Nat"
  show (Function t1 t2) = "(" ++ show t1 ++ " -> " ++ show t2 ++ ")"
  show NullType = "<NULL>"
type TypeContext = M.Map VarName Type
```

The difference being the introduction of Identifiers, Abstractions, and Applications. Otherwise, the evaluation rules are the same as the last homework. Using the evaluation rules defined in TAPL, we update the definitions for numeric and values as:

```haskell
isNumeric :: Term -> Bool
isNumeric Zero = True
isNumeric (Succ t) = isNumeric t
isNumeric _ = False

isValue :: Term -> Bool
isValue Tru = True
isValue Fls = True
isValue (Identifier _) = True
isValue (Abstraction _ _) = True
isValue t = isNumeric t
```

Then, we can can define the single-step evaluations for the entire STLC extended with Booleans and Nats as:

```
eval1 :: Term → Maybe Term
eval1 t
  | isValue t = Nothing   -- values do not require evaluation
  | otherwise = case t of
    Application t1 t2 → evalApplication t1 t2
    If t1 t2 t3 → evalIf t1 t2 t3
    IsZero t → evalIsZero t
    Succ t → evalSucc t
    Pred t → evalPred t
    otherwise → Nothing
```

Since values (including Abstractions and Identifiers) cannot be rewritten, they return nothing and the function definitions for Application, If, IsZero, Succ, and Pred are:

1. Application:

```
evalApplication :: Term → Term → Maybe Term
evalApplication t1@(Abstraction name t) t2
  | isValue t2 = Just (betaReduc name t2 t)
  | otherwise = eval1 t2 ≫= return ∘ (Application t1)
evalApplication t1 t2
  | isValue t1 = eval1 t2 ≫= return ∘ (Application t1)
  | otherwise = eval1 t1 ≫= return ∘ λt → (Application t t2)
```

2. Beta Reductions:

```
betaReduc :: VarName → Term → Term → Term
betaReduc l r (Identifier name) = if name ≡ l
  then r
  else (Identifier name)
betaReduc l r (Abstraction name term) = Abstraction name $ betaReduc l r term
betaReduc l r (Application t1 t2) = Application (betaReduc l r t1) (betaReduc l r t2)
betaReduc l r (If t1 t2 t3) = If (betaReduc l r t1) (betaReduc l r t2) (betaReduc l r t3)
betaReduc l r (Succ t) = Succ (betaReduc l r t)
betaReduc l r (Pred t) = Pred (betaReduc l r t)
betaReduc l r (IsZero t) = IsZero (betaReduc l r t)
betaReduc l r t = t
```

3. If Statements:

```
evalIf :: Term → Term → Term → Maybe Term
evalIf Tru t2 t3 = Just t2
evalIf Fls t2 t3 = Just t3
evalIf t1 t2 t3 = eval1 t1 ≫= return ∘ λt → (If t t2 t3)
```

4. IsZero:

```
evalIsZero :: Term → Maybe Term
evalIsZero Zero = Just Tru
evalIsZero (Succ t)
  | isNumeric t = Just Fls
  | otherwise = Nothing
evalIsZero t = eval1 t ≫= Just ∘ IsZero
```

5. Succ:

$$
\begin{aligned}
&\mathit{evalSucc} :: \mathit{Term} \rightarrow \mathit{Maybe}\ \mathit{Term}\\
&\mathit{evalSucc}\ t = \mathit{eval1}\ \ t \ggg\!= \mathit{Just} \circ \mathit{Succ}
\end{aligned}
$$

6. Pred:

$$
\begin{aligned}
&\mathit{evalPred} :: \mathit{Term} \rightarrow \mathit{Maybe}\ \mathit{Term}\\
&\mathit{evalPred}\ (\mathit{Succ}\ t) = \mathit{Just}\ t\\
&\mathit{evalPred}\ \mathit{Zero} = \mathit{Just}\ \mathit{Zero}\\
&\mathit{evalPred}\ t = \mathit{eval1}\ \ t \ggg\!= \mathit{Just} \circ \mathit{Pred}
\end{aligned}
$$

Then the repeated application of the single step evaluation becomes:

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Term} \rightarrow \mathit{Term}\\
&\mathit{eval}\ t = \textbf{case}\ \mathit{eval1}\ \ t\ \textbf{of}\\
&\quad \mathit{Just}\ \mathit{t1} \rightarrow \mathit{eval}\ \mathit{t1}\\
&\quad \mathit{Nothing} \rightarrow t
\end{aligned}
$$

Where Nothing represents when no single step evaluation rules apply to the term. We can test this with constructed term from the language of Lambda Calculus extended with Booleans and Naturals to yeild the correct answers seen below.

$$
\begin{aligned}
&\mathit{eval}\ \$\ \mathit{If}\ \ \mathit{Tru}\ \mathit{Tru}\ \mathit{Fls}\\
&\mathit{Tru}\\
&\mathit{eval}\ \$\ \mathit{If}\ (\mathit{IsZero}\ (\mathit{Pred}\ (\mathit{Pred}\ (\mathit{Pred}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero})))))))\ (\mathit{Succ}\ \mathit{Zero})\ \mathit{Zero}\\
&\mathit{Succ}\ \mathit{Zero}\\
&\mathit{eval}\ \$\ \mathit{Application}\ (\mathit{Abstraction}\ \texttt{"x"}\ (\mathit{IsZero}\ (\texttt{"x"})))\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}))\\
&\mathit{Fls}
\end{aligned}
$$

## 2 TLBN Parser

To make parsing easier, I leveraged the parser combinator library (Text.Parsec.*) and used the parser state to combine parsing and type checking into one function call. If the parsing results in a type inconsistency of disparity the parser will monadically fail with an appropriate error message. If the parse completes, the type is guaranteed to be correct and evaluation can then take place.

First a couple important imports:

```
import Prelude hiding (succ, pred)
import Text.Parsec
import Text.Parsec.Char
import Text.ParserCombinators.Parsec.Char
import qualified Data.Map.Strict as M
import LcData   -- contains the Data Constructors for TLBN
```

To make things easier, I defined a couple of constants and a parser combinator that consumes input regardless of how many spaces are in front or following the keyword.

```
returnType = "_"
whitespace :: Monad m => ParsecT String TypeContext m ()
whitespace = spaces >> return ()
```

```
keyword :: Monad m ⇒ String → ParsecT String TypeContext m ()
keyword p = try $ do
  whitespace
  string p <? > ("Expecting keyword: " ++ p)
  whitespace
```

Now to type check at the same time as the parsing, the user state hidden away by the parser is a mapping from string to type which records a typing context for that specific parser. These are quality of life helper methods that make updating and reading the state easier.

```
merge :: VarName → Type → TypeContext → TypeContext
merge name t context = M.insert name t context

getReturnState :: Monad m ⇒ ParsecT String TypeContext m Type
getReturnState = do
  gamma ← getState
  return $ gammaM.! returnType
```

To parse the term true, we look for the keyword 'true' and set the return type to a boolean value.

```
tru :: Monad m ⇒ ParsecT String TypeContext m Term
tru = try $ do
  keyword "true"
  modifyState $ merge returnType Boole
  return Tru
```

Parsing false and zero are identical to parsing true just with the repsective keywords

```
fls :: Monad m ⇒ ParsecT String TypeContext m Term
fls = try $ do
  keyword "false"
  modifyState $ merge returnType Boole
  return Fls

zero :: Monad m ⇒ ParsecT String TypeContext m Term
zero = try $ do
  keyword "0"
  modifyState $ merge returnType Nat
  return Zero
```

It only gets interesting when you parse a function like isZero. Here we parse the correct keywords then use the 'term' parser to parse the possible TLBN term between the parenthesis. After that term is parsed, the type of *returnType* has been set to the type of the newly parsed term. We then check to make sure it is a *Nat*, and if it is, we complete the parse, set the type of *returnType* to the new type (Boole) and, return the properly constructed data. Otherwise, we fail with the typing error in the 'else' clause.

```
iszero :: Monad m ⇒ ParsecT String TypeContext m Term
iszero = try $ do
  keyword "iszero"
  keyword "("
  t ← term <? > "Error parsing: Succ ( Term ), expected Term but failed"
  t_type ← getReturnState
  if t_type ≡ Nat
  then do
    keyword ")"
      -- change the state from Nat to Boole
    modifyState $ merge returnType Boole
```

```
      return (IsZero t)
    else
      fail $ "Expected type 'Nat' in iszero but was " ++ show t_type
```

Similarly, we can parse Succ and Pred terms:

```
succ :: Monad m => ParsecT String TypeContext m Term
succ = try $ do
  keyword "succ"
  keyword "("
  t <- term <?> "Error parsing: Succ ( Term ), expected Term but failed"
  t_type <- getReturnState
  if t_type == Nat
  then do
      -- no need to change the state, it is the same
    keyword ")"
    return (Succ t)
  else
    fail $ "Expected type 'Nat' in 'Succ' but was " ++ show t_type
pred :: Monad m => ParsecT String TypeContext m Term
pred = try $ do
  keyword "pred"
  keyword "("
  t <- term <?> "Error parsing: Pred ( Term ), expected Term but failed"
  t_type <- getReturnState
  if t_type == Nat
  then do
      -- no need to change the state, it is the same
    keyword ")"
    return (Pred t)
  else
    fail $ "Expected type 'Nat' in 'Pred' but was " ++ show t_type
```

If statements require three separate 'term' parses, and it requires the first to be return a boolean and the types of the second and third parse to be equivalent.

```
if_statement :: Monad m => ParsecT String TypeContext m Term
if_statement = try $ do
  keyword "if"
  cond <- term <?> "Expecting 'term' following _if_"
  cond_type <- getReturnState
  if cond_type /= Boole
  then
    fail $ "Expecting Boolean type for if-statement conditional, received: " ++
      show cond_type
  else do
    keyword "then"
    t_then <- term <?> "Expecting 'term' following _then_"
    then_type <- getReturnState
    keyword "else"
    t_else <- term <?> "Expecting 'term' following _else_"
    else_type <- getReturnState
    keyword "fi"
    if then_type == else_type
    then do
```

```
      modifyState $ merge returnType then_type
      return (If cond t_then t_else)
    else
      fail $ "Type inconsistency for then/else parts of if statement\n" ++
        "then type: " ++ (show then_type) ++ "\n" ++
        "else type: " ++ (show else_type) ++ "\n"
```

If the first parsed term is not a boolean term or the second and third term types do not match, this parse fails with the appropriate error message.

For Applications to correctly typecheck, the first term must be a function type and the type of the second term parsed must match the 'argument' value of the function type.

```
application :: Monad m ⇒ ParsecT String TypeContext m Term
application = try $ do
  keyword "app"
  keyword "("
  t1 ← term <?> "Error parsing first 'term' following _app_"
  t1_type ← getReturnState
  keyword ","
  t2 ← term <?> "Error parsing second 'term' following \"_app_ Term\""
  t2_type ← getReturnState
  keyword ")"
  case t1_type of
    (Function t11 t12) → if t11 ≡ t2_type
      then do
        modifyState $ merge returnType t12
        return (Application t1 t2)
      else fail $ "Mismatch types for function application\n"
        ++ "function argument required type: "
        ++ show t11 ++ "\n"
        ++ "actual argument type : "
        ++ show t2_type
    otherwise → fail $ "Expecting Function type for the first term" ++
      "of an application, receieved: " ++ show t1_type
```

Abstractions are typed by setting the *returnType* value to a function type which goes from the parsed identifier type to the resulting type of the parsed term. Upon exiting this parse, we also have to remove any typing information in the *TypeContext* that references the parsed identifier type as that value is not bound to anything outside this typing context.

```
abstraction :: Monad m ⇒ ParsecT String TypeContext m Term
abstraction = try $ do
  keyword "abs"
  keyword "("
  iden ← identifier
  keyword ":"
  iden_type ← identifierType
  keyword "."
  modifyState $ merge iden iden_type
  t ← term
  t_type ← getReturnState
  keyword ")"
  modifyState $ merge returnType (Function iden_type t_type)
```

```
    modifyState $ M.delete iden
    return $ Abstraction iden t
```

To parse an identifier, we just parse any string and make sure the value returned is not a reserved word in our langague or the empty string. If it is, we should fail the parse.

```
identifier :: Monad m => ParsecT String TypeContext m String
identifier = try $ do
    whitespace
    x <- many letter
    case all (x /=) ["succ", "pred", "if", "fi", "arr", "Bool", "Nat",
      "abs", "app", "true", "false", "then", "else",
      "iszero", ""]
      of
      True -> do
        return x
      otherwise -> fail $ "Could not parse an identifier, must not be a reserved" ++
        " word or contain anything but characters: " ++ x
```

To parse a type, we simpley parse the correct keywords for either a Boolean or Nat type. If it is a function type we recursively parse the correct keywords then the two types that make up the fuction type.

```
  -- typing information and ------------------------------------
identifierType :: Monad m => ParsecT String TypeContext m Type
identifierType = boolType <|> natType <|> functionType
   <?> "identifier type parser"
boolType :: Monad m => ParsecT String TypeContext m Type
boolType = try $ keyword "Bool" >> return Boole
natType :: Monad m => ParsecT String TypeContext m Type
natType = try $ keyword "Nat" >> return Nat
functionType :: Monad m => ParsecT String TypeContext m Type
functionType = try $ do
    keyword "arr"
    keyword "("
    t1 <- identifierType
    keyword ","
    t2 <- identifierType
    keyword ")"
    return $ Function t1 t2
```

Parsing the identifier into a term requires us to set the returned type to its correct value. If the identfier is a free variable (does not have a type) then we fail the parse as it is not type-correct.

```
identifier_term :: Monad m => ParsecT String TypeContext m Term
identifier_term = try $ do
    x <- identifier
    context <- getState
    case M.lookup x context of
      Just t -> modifyState $ merge returnType t
      Nothing -> fail $ "Identifier: " ++ x ++ " has no type in current typing context"
    return $ Identifier x
```

Then to bring it all together, any term in this language is the choice of any of the previously defined parsers. If no parser matches, then we fail the parse completely and there must be some syntatic error in the text.

```
term :: Monad m => ParsecT String TypeContext m Term
term =
  identifier_term <|>
  abstraction <|>
  application <|>
  tru <|>
  fls <|>
  if_statement <|>
  zero <|>
  succ <|>
  pred <|>
  iszero <|>
  (try (keyword "(" >> term >>= \k -> keyword ")" >> return k))
  <?> "Basic term parsing"
```

## 3 Compiling it together

To construct the entire program of the parser and evaluator, our main function lazily reads in the contents of the file argument and passes it to the parser. If the parser comes back with an error it prints the error and exits. If the parser comes back with a legitimate parse and type-check, we evaluate the term using the evaluation functions defined in the first section.

```
import System.Environment (getArgs)
import System.IO (openFile, hGetContents, IOMode (ReadMode))
import Data.Map (singleton)
import Data.Either (Either (Left, Right))
import Text.Parsec (runParser)
import LcParser
import LcEvaluator
import LcData
main :: IO ()
main = do
  args <- getArgs
  case length args /= 1 of
    True -> putStrLn help
    otherwise -> parseLC args
parseLC :: [String] -> IO ()
parseLC (filename: _) = do
  contents <- hGetContents =<< openFile filename ReadMode
  case runParser start (singleton returnType NullType) filename contents of
    Left err -> print err
    Right (term, term_type) -> do
      putStrLn $ "Syntax Correct. \n\tResult type: " ++ show (term_type)
      putStrLn $ "Evaluating...\n\tResult: " ++ show (eval term)
help :: String
help = "Program requires only 1 argument. Usage: \n" ++
  "  TLBN <filename>"
```

## 4 Example usages

Examples are pulled directly off of the homework write-up and a more complicated example is found in example4.TLBN (show in the appendix)

```
$ ./TLBN example1.TLBN
  Syntax Correct.
        Result type: Nat
  Evaluating...
        Result: Succ (Succ (Succ Zero))

$ ./TLBN example2.TLBN
  unexpected end of input
  expecting Basic term parsing
  Type inconsistency for then/else parts of if statement
  then type: Nat
  else type: Boolean

$ ./TLBN example3.TLBN
  Syntax Correct.
        Result type: Boolean
  Evaluating...
        Result: Fls

$ ./TLBN example4.TLBN
Syntax Correct.
        Result type: (Boolean -> Boolean)
Evaluating...
        Result: Abstraction "z" (If (Identifier "z") Fls Tru)
```

## Appendix

Example1 Content:

```
app (abs (x: Nat . succ(succ(x))), succ(0))
```

Example2 Content:

```
if iszero (0) then succ (0) else false fi
```

Example3 Content:

```
app (abs (x:Bool.x), false)
```

Example4 Content:

```
app(abs(x:Nat . if app(abs(y:Nat . iszero(y)), x)
                then abs(a:Bool . if a then true else false fi)
                else abs(z:Bool . if z then false else true fi)
                fi), succ (0))
```