Typed Lambda Calculus

Zhiyan Yao, Yiqing Zhang, Zhongyu Shi

Saturday 1st June, 2024

Abstract

We implemented a simplest version of lambda calculus in the first place, including variable declaration, lambda abstraction, application, α conversion and β reduction. Based on it, we further extended it to typed lambda calculus that support Boolean, Integer and function boolean -> integer types using η reduction.

Contents

1 A brief introduction to lambda calculus

Lambda calculus (also written as λ-calculus) is a formal system for expressing computation, which is invented by Alonzo Church. Lambda calculus defined the notion of computable function via this system from a function-based perspective, serving as the foundation of today's functional programming. Later his PhD student Alan Turing invented Turing machines and defined the notion of computable function via these machines from a state-based perspective, serving as the foundation of today's imperative programming.

1.1 Basic Rules

In the simplest form of lambda calculus, terms are built using only the following three rules:

- x: A variable is a character or string representing a parameter.
- $(\lambda x.M)$: A lambda abstraction is a function definition, taking as input the bound variable x and returning the body M.
- $(M\ N)$: An application, applying a function M to an argument N. Both M and N are lambda terms.

1.2 Reduce operation

In the simplest form of lambda calculus, the reduction operations include:

- $(\lambda x.M[x]) \to (\lambda y.M[y])$: α-conversion, renaming the bound variables in the expression. Used to avoid name collisions.
- $((\lambda x.M)\ N) \to (M[x := N])$: β-reduction, replacing the bound variables with the argument expression in the body of the abstraction.

2 Implementation

This section describes a module which we will import later on.

```
module Basics where

import Control.Monad
import System.Random
import qualified Data.Map as Map
import Control.Monad.State
import Data.List (union)
import Data.Functor.Identity (Identity, runIdentity)
```

```haskell
-- Define VarMap as a type alias for Map.Map String Int
type VarMap = Map.Map String Int

-- Define the data type for lambda calculus expressions
data Expr = Var Int            -- Using Int to encode variables
          | Abs Int Expr       -- Abstractions also use Ints for variable names
          | App Expr Expr      -- Application remains unchanged
          deriving (Eq, Show)

-- Function to perform substitution
substitute :: Int -> Expr -> Expr -> Expr
substitute x with expr = case expr of
    Var y -> if y == x then with else Var y
    Abs y body -> if y == x then Abs y body else Abs y (substitute x with body)
    App e1 e2 -> App (substitute x with e1) (substitute x with e2)

-- Function to perform eta reduction
etaReduce :: Expr -> (Expr, Bool)
etaReduce (Abs x (App f (Var y)))
    | x == y && x `notElem` freeVars f = (f, True)
    | otherwise = let (e, changed) = etaReduce (App f (Var y))
                  in (Abs x e, changed)
etaReduce (Var x) = (Var x, False)
etaReduce (Abs x e) = let (e', changed) = etaReduce e in (Abs x e', changed)
etaReduce (App e1 e2) = let (e1', changed1) = etaReduce e1
                            (e2', changed2) = etaReduce e2
                        in (App e1' e2', changed1 || changed2)

-- Function to perform beta reduction with a limit to recursion depth
betaReduce :: Expr -> Int -> (Expr, Bool)
betaReduce expr 0 = (expr, False) -- Stop reducing after depth limit reached
betaReduce expr depth = case expr of
  Var _ -> (expr, False)
  Abs x e -> let (e', changed) = betaReduce e (depth - 1) in (Abs x e', changed)
  App (Abs x body) arg -> let result = substitute x arg body
                              (reducedResult, changed) = betaReduce result (depth - 1)
                          in (reducedResult, True)
  App e1 e2 -> let (e1', changed1) = betaReduce e1 (depth - 1)
                   (e2', changed2) = betaReduce e2 (depth - 1)
               in (App e1' e2', changed1 || changed2)

-- Generates a new variable name not in the list
freshVar :: [Int] -> Int -> Int
freshVar vars base = head $ filter (`notElem` vars) $ map (\i -> base + i) [1..]

-- -- Function to perform alpha conversion to avoid capture
-- alphaConvert :: Int -> Expr -> Expr
-- alphaConvert x (Abs y body)
--     | x == y = let newY = freshVar (freeVars body) y
--                in Abs newY (substitute y (Var newY) body)
--     | otherwise = Abs y (alphaConvert x body)
-- alphaConvert x (App e1 e2) = App (alphaConvert x e1) (alphaConvert x e2)
-- alphaConvert _ expr = expr

alphaConvert :: Int -> Expr -> StateT VarMap Identity Expr
alphaConvert x (Abs y body)
    | x == y = do
        let newY = freshVar (freeVars body) y
        modify (Map.insert (show y) newY)  -- Assuming y is an Int and needs to be
            converted to String
        body' <- alphaConvert x body
        return $ Abs newY body'
    | otherwise = do
        body' <- alphaConvert x body
        return $ Abs y body'
alphaConvert x (App e1 e2) = do
    e1' <- alphaConvert x e1
    e2' <- alphaConvert x e2
    return $ App e1' e2'
alphaConvert _ expr = return expr

-- Example of running alphaConvert with an initial empty VarMap
```

```
runAlphaConvert :: Expr -> Expr
runAlphaConvert expr = runIdentity (evalStateT (alphaConvert 1 expr) Map.empty)


-- Collects all free variables in an expression, now returns a list of Int
freeVars :: Expr -> [Int]
freeVars (Var x) = [x]
freeVars (Abs x e) = filter (/= x) (freeVars e)
freeVars (App e1 e2) = freeVars e1 `union` freeVars e2

-- Helper function to print and reduce expressions
printAndReduce :: Expr -> IO ()
printAndReduce expr = do
    putStrLn "Original Expression:"
    print expr
    let (reducedExprBeta, betaChanged) = betaReduce expr 10  -- Reduced depth for safety in
         Omega
    when betaChanged $ do
      putStrLn "After Beta Reduction:"
      print reducedExprBeta
    let (reducedExprEta, etaChanged) = etaReduce reducedExprBeta
    when etaChanged $ do
      putStrLn "After Eta Reduction:"
      print reducedExprEta
```

3 Wrapping it up in an exectuable

We will now use the library form Section 2 in a program.

```
module Main where

import Basics

main :: IO ()
main = do
  -- Example 1: Identity Function
  putStrLn "Example 1: Identity Function"
  let identity = Abs 1 (Var 1)
  let expr1 = App identity (Var 2)
  printAndReduce expr1

  -- Example 2: Combination of Functions
  putStrLn "\nExample 2: Combination of Functions"
  let function = Abs 1 (Abs 2 (App (Var 1) (Var 2)))
  let expr2 = App (App function (Var 3)) (Var 4)
  printAndReduce expr2

  -- Example 3: Omega Combinator (non-terminating)
  putStrLn "\nExample 3: Omega Combinator"
  let omega = App (Abs 1 (App (Var 1) (Var 1))) (Abs 1 (App (Var 1) (Var 1)))
  printAndReduce omega

  -- Example 4: Church Numerals (Two and Three)
  putStrLn "\nExample 4: Church Numerals (Two applied to Three)"
  let two = Abs 1 (Abs 2 (App (Var 1) (App (Var 1) (Var 2))))
  let three = Abs 1 (Abs 2 (App (Var 1) (App (Var 1) (App (Var 1) (Var 2)))))
  let expr4 = App two three
  printAndReduce expr4

  -- Example 5: S and K Combinators
  putStrLn "\nExample 5: S and K Combinators"
  let sComb = Abs 1 (Abs 2 (Abs 3 (App (App (Var 1) (Var 3)) (App (Var 2) (Var 3)))))
  let kComb = Abs 1 (Abs 2 (Var 1))
  let expr5 = App (App sComb kComb) identity
  printAndReduce expr5
```

We can run this program with the commands:

```
stack build
stack exec myprogram
```

The output of the program is something like this:

```
Hello!
[1,2,3,4,5,6,7,8,9,10]
[100,100,300,300,500,500,700,700,900,900]
[1,3,0,1,1,2,8,0,6,4]
[100,300,42,100,100,100,700,42,500,300]
GoodBye
```

4 Simple Tests

We now use the library QuickCheck to randomly generate input for our functions and test some properties.

```haskell
module Main where

import Basics

import Test.Hspec
import Test.QuickCheck
```

The following uses the HSpec library to define different tests. Note that the first test is a specific test with fixed inputs. The second and third test use QuickCheck.

```haskell
main :: IO ()
main = hspec $ do
  describe "Basics" $ do
    it "somenumbers should be the same as [1..10]" $
      somenumbers `shouldBe` [1..10]
    it "if n > - then funnyfunction n > 0" $
      property (\n -> n > 0 ==> funnyfunction n > 0)
    it "myreverse: using it twice gives back the same list" $
      property $ \str -> myreverse (myreverse str) == (str::String)
```

To run the tests, use `stack test`.

To also find out which part of your program is actually used for these tests, run `stack clean && stack test` Then look for "The coverage report for ... is available at ... .html" and open this file in your browser. See also: `https://wiki.haskell.org/Haskell_program_coverage`.