COMP432 Assignment 2

Andrew Childs

October 6, 2007

Contents

1 Combinatory Logic

```
module CL (CLVar, CLTerm(..), drawCLTerm) where

import Data.Tree (drawTree, unfoldTree)
```

In this section we define the structure of a `CLTerm`, and a textual representation. A variable in a `CL` term is represented by a `String` containing its name.

```
type CLVar = String
```

The recursive definition of a `CLTerm` forms a binary tree where the leaves are combinators or variables.

```
data CLTerm = CLVar CLVar -- never "S" or "K"
            | S
            | K
            | I
            | B
            | C
            | S'
            | Bstar
            | C'
            | CLTerm :$ CLTerm
              deriving (Eq)
```

A `show` implementation based on unlambda, which makes the tree structure more obvious.

```
instance Show CLTerm where
    show (CLVar x) = x
    show S = "S"
    show K = "K"
    show I = "I"
    show B = "B"
    show C = "C"
    show S' = "S'"
    show Bstar = "B*"
    show C' = "C'"
    show (t1 :$ t2) =
        "`" ++ show t1 ++ show t2
```

A mapping into generalised rose trees for pretty printing.

```
drawCLTerm :: CLTerm -> String
drawCLTerm = drawTree . unfoldTree clToTree
    where
      clToTree (left :$ right) = ("CLApp", [left,right])
      clToTree x = (show x, [])
```

2 Combinatory Logic Parsing

```
module CLParser ( parseCLTerm) where

import ParserLib
import CL
```

This section defines a very simple parser for combinatory logic terms in the unlambda syntax. Variables are restricted to being single letters.

```
parseCLTerm :: String -> CLTerm
parseCLTerm = some toplevel
```

A term is either a combinator, a single letter variable, or recursively the application of two such terms (written *‘pq*).

```
toplevel :: Parser Char CLTerm
toplevel = (token "S"  <@ const S) <|>
           (token "K"  <@ const K) <|>
           (token "I"  <@ const I) <|>
           (token "B"  <@ const B) <|>
           (token "C"  <@ const C) <|>
           (token "S'" <@ const S') <|>
           (token "B*" <@ const Bstar) <|>
           (token "C'" <@ const C') <|>
           (token "`"  &> (toplevel <&> toplevel) <@ uncurry (:$)) <|>
           (satisfy (`elem` ['a'..'z']) <@ (CLVar . (:[])))
```

3 Lambda Calculus

```
module Lambda where
```

Here we implement the basics of the lambda calculus. Including a recursive definition of lambda terms, a textual representation and a function to determine the free variables of a term.
Variables are represented by a `String` containing their name.

```
type Var = String
```

In the pure lambda calculus, there are only abstractions over a single variable, variables as found in the body of of an abstraction and the application of two terms.

```
data LTerm = Var Var
           | App LTerm LTerm
           | Abs Var LTerm
```

The textual representation uses a blackslash in place of λ

```
instance Show LTerm where
    show (Var x) = x
    show (App t1 t2@(App _ _)) = show t1 ++ " (" ++ show t2 ++ ")"
    show (App t1 t2) = show t1 ++ " " ++ show t2
    show (Abs x t1) = "(\\" ++ x ++ "." ++ show t1 ++ ")"
```

An abstraction binds a variable. A variable is free then if it is not contained in any abstraction over that variable.

```
freeVars :: LTerm -> [Var]
freeVars = freeVars' []

freeVars' :: [Var] -> LTerm -> [Var]
freeVars' bound term =
    case term of
      Var x | x `elem` bound -> []
            | otherwise      -> [x]
      App t1 t2 -> (freeVars' bound t1) ++
                   (freeVars' bound t2)
      Abs x t -> freeVars' (x:bound) t
```

4 Lambda Term Parsing

```
module LambdaParser ( parseLTerm ) where
```

In this section we define a parser for lambda terms, using the parser library.

```
import Control.Arrow as A
import Data.Char (isDigit)

import ParserLib
import Lambda
```

We allow definitions which map from variables to their λ-term values.

```
type Definition = (Var, LTerm)
```

A λ-term consists of a (possibly empty) set of definitions, followed by the "body" of a λ-term. These two are then combined to remove occurrences of free variables from the term.

```
parseLTerm :: [Char] -> LTerm
parseLTerm = some $ ((definitions <|> epsilon) <&> sp body)
                    <& optionally spaces <@ uncurry expand
```

The body of a λ-term is either an entity or a left-associative sequence of applied entities.

```
body :: Parser Char LTerm
body = entity <|> (spacelist entity <@ foldl1 App)
```

An entity is a variable, an abstraction, or a body surrounded in brackets.

```
entity :: Parser Char LTerm
entity = var <@ Var <|> abstraction
         <|> (sptoken "(" &> body <& sptoken ")")
```

An abstraction is a list of variables followed by a body. The list of variables is "unwrapped" to a series of abstractions of a single variable.

```
abstraction :: Parser Char LTerm
abstraction = ((sptoken "(" <&> sptoken "\\") &>
                 sp (varlist <& sptoken "."))
               <&> sp (body <& sptoken ")")
               <@ uncurry abstract
    where
      abstract :: [Var] -> LTerm -> LTerm
      abstract varlist body = foldr Abs body varlist
```

A variable is entirely lower case, starting with an alphabetic character and consisting of alphabetic characters or digits.

```
var :: Parser Char Var
var =            satisfy (`elem` ['a'..'z'])
      <:&> star (satisfy (`elem` (['a'..'z'] ++ ['0'..'9'])))

varlist :: Parser Char [Var]
varlist = spacelist var
```

We generally allow our tokens to have space preceding them in an attempt to be tolerant of whitepsace.

```
sptoken :: String -> Parser Char String
sptoken = sp . token
```

We allow definitions of the form of Haskell-like "let"-statements.

```
definitions :: Parser Char [Definition]
definitions = sptoken "let " &>
              neListOf def (sptoken ";")
              <& sptoken "in "

def :: Parser Char Definition
def = (sp var <& sptoken "=") <&> sp body <@ uncurry (,)
```

Expand all occurrences of *free variables only* with the definitions from the let statement.

```
expand :: [Definition] -> LTerm -> LTerm
expand = expand' []

expand' :: [Var] -> [Definition] -> LTerm -> LTerm
expand' boundVars defs term =
    case term of
      Var x | x `elem` boundVars -> Var x
            | otherwise          -> expand defs $ getVar x
      App t1 t2 -> App (expand' boundVars defs t1)
                       (expand' boundVars defs t2)
      Abs x t -> Abs x (expand' (x:boundVars) defs t)
    where
      getVar x =
          maybe (error $ "Unable to find definition for variable "
                         ++ show x)
                id $ lookup x defs
```

5 Combinatory Logic Zipper

```
{-# LANGUAGE PatternGuards #-}

module CLZipper where

import CL
```

Here we define specialised version of a zipper[1] for CLTerm binary trees.
The functions `focus` and `mkZipper` wrap and unwrap a tree of `CLTerm`s. Navigation is then possible with the primitive navigation functions like`parent`, `leftChild` and `rightChild`.
A `Location` defines a *focus*, which is a subtree of terms, and a *path*, which records how this location was reached.

```
data Path = Top
          | LNode CLTerm Path
          | RNode Path CLTerm

data Location = Location CLTerm Path
```

`mkZipper` records the path as `Top`, indicating this is the top of the tree. `focus` retrieves the focus of the zipper, which may be a subtree.

```
mkZipper :: CLTerm -> Location
mkZipper x = Location x Top

focus :: Location -> CLTerm
focus (Location x _) = x
```

To navigate to a child of a `CLTerm`, we change the focus to point to the child by examining the constructor of the current focus. The child that is not being navigated to is stored in the path. The functions are defined in terms of a monad to allow for failure.

```
leftChildM :: (Monad m) => Location -> m Location
leftChildM (Location (left :$ right) up) =
    return $ Location left (RNode up right)
leftChildM _ = fail "leftChild of bottom"

rightChildM :: (Monad m) => Location -> m Location
rightChildM (Location (left :$ right) up) =
    return $ Location right (LNode left up)
rightChildM _ = fail "rightChild of bottom"
```

To navigate to a parent is to revert the `leftChild` and `rightChild` operations. The current focus becomes a child, while the other child is retrieved from the path. The path then discards the current layer.

```
parentM :: (Monad m) => Location -> m Location
parentM (Location focus Top) = fail "parent of Top"
parentM (Location focus (LNode left up)) =
    return $ Location (left :$ focus) up
parentM (Location focus (RNode up right)) =
    return $ Location (focus :$ right) up
```

A pure version of the above, each of which fails with `error`.

```
parent, leftChild, rightChild  :: Location -> Location
parent = maybe (error "parent of Top") (id) . parentM
leftChild = maybe (error "leftChild of bottom") (id) . leftChildM
rightChild = maybe (error "rightChild of bottom") (id) . rightChildM
```

Repeated applications of `parent`

```
ancestor :: Int -> Location -> Location
ancestor 0 loc = loc
ancestor n loc
    | (Just p) <- parentM loc = ancestor (n-1) p
    | otherwise = loc


root :: Location -> Location
root loc | (Just p) <- parentM loc = root p
         | otherwise = loc
```

Focus value manipulation.

```
changeFocus :: (CLTerm -> CLTerm) -> Location -> Location
changeFocus f (Location x z) = Location (f x) z


putFocus :: CLTerm -> Location -> Location
putFocus = changeFocus . const
```

6 Compilation

module Compile **where**

In this section we describe abstraction elimination for combinatory logic terms.

import CL
import Lambda

The compilation from λ-terms to combinatory logic terms.

```
compile :: LTerm -> CLTerm
```

A variable remains untouched.

```
compile (Var v) = CLVar v
```

Application compiles to the application of the compiled subterms.

```
compile (App t1 t2) = compile t1 :$ compile t2
```

Abstraction of a variable in a body that doesn't contain that variable can be represnted with the constant combinator `K`.

```
compile (Abs v t)
    | not (v `elem` freeVars t) = K :$ compile t
```

A term of the form $\lambda x.x$ is represented as the identity combinator `I`.

```
compile (Abs v1 (Var v2)) | v1 == v2 = I
```

What to say here

```
compile (Abs v1 t2@(Abs v2 e))
    | v1 `elem` freeVars e = compile' v1 (compile t2)
```

By distributivity of substitution over application, and examination of the parallels in the `S` combinator, the following rule applies.

```
compile (Abs v (App t1 t2)) =
    optimise (S :$ compile (Abs v t1) :$ compile (Abs v t2))
```

This is a definition of `compile` for "hybrid" terms, of a lambda abstraction with a combinatory logic body. The results and reasoning are the same

```
compile' :: Var -> CLTerm -> CLTerm
compile' v (t1 :$ t2)
    = optimise (S :$ compile' v t1 :$ compile' v t2)
compile' v (CLVar cv) | v == cv    = I
                      | otherwise = K :$ (CLVar cv)
compile' _ x = K :$ x
```

Each of these rules can be seen from examining the reduction rules in section 7.

```
optimise :: CLTerm -> CLTerm
optimise term =
```

```
  case term of
    S :$ (K :$ p) :$ (K :$ q)       -> K :$ (p :$ q)
    S :$ (K :$ p) :$ I              -> p
    S :$ (K :$ p) :$ (B :$ q :$ r) -> Bstar :$ p :$ q :$ r
    S :$ (K :$ p) :$ q              -> B :$ p :$ q
    S :$ (B :$ p :$ q) :$ (K :$ r) -> C' :$ p :$ q :$ r
    S :$ p :$ (K :$ q)              -> C :$ p :$ q
    S :$ (B :$ p :$ q) :$ r         -> S' :$ p :$ q :$ r
    x                               -> x
```

7 Reduction

{-# LANGUAGE PatternGuards #-}

```
module Reduce (reduce, reductionStep) where
```

Here we define the rules for reduction and the strategy for applying them. This reducer approaches the problem as subtree rewriting.

```
import CL
import CLZipper
```

To manipulate the tree of terms, a zipper as defined in section 5 is used. `reduce` places the terms into the zipper and retrieves the resulting term.

```
reduce :: CLTerm -> CLTerm
reduce = focus . root . zipperReduce . mkZipper
```

The search typically descends into the left subtree. To search the entire tree it is necessary to search the right branches, which can be reached with `forceRightBranch`.

```
forceRightBranch :: Location -> Maybe Location
forceRightBranch loc@(Location _ (LNode _ _))
    = forceRightBranch $ parent loc
forceRightBranch loc@(Location _ (RNode _ _))
    = Just $ rightChild $ parent loc
forceRightBranch loc@(Location _ Top) = Nothing
```

The right branches contain the arguments of a combinator.

```
rightBranches :: Maybe Location -> [CLTerm]
rightBranches Nothing     = []
```

```
rightBranches (Just loc) =
    focus (rightChild loc) : rightBranches (parentM loc)
```

Ideally the reduction rules would be written as

```
reductionStep :: CLTerm -> Maybe CLTerm
reductionStep term =
    case term of
      S :$ x :$ y :$ z          ->
          Just $ (x :$ z) :$ (y :$ z)
      K :$ x :$ _               ->
          Just $ x
      I :$ x                    ->
          Just $ x
      B :$ f :$ g :$ x          ->
          Just $ f :$ (g :$ x)
      C :$ f :$ g :$ x          ->
          Just $ f :$ x :$ g
      S' :$ c :$ f :$ g :$ x    ->
          Just $ c :$ (f :$ x) :$ (g :$ x)
      Bstar :$ c :$ f :$ g :$ x ->
          Just $ c :$ (f :$ (g :$ x))
      C' :$ c :$ f :$ g :$ x    ->
          Just $ c :$ (f :$ x) :$ g
      _                         -> Nothing
```

However this approach requires the algorithm unneeded complexity. The reduction rule that applies is determined by the leftmost child. `reductionStep` looks *ahead* to find the rule that applies. When the tree is rewritten another rule will likely apply, which requires the search to resume above the rewritten tree. Instead we write the rules in terms of the combinator and the arguments, which results in a new subtree and the location to place it in terms of how many arguments were consumed.

```
reductionStep :: [CLTerm] -> Maybe (CLTerm, Int)
reductionStep term =
    case term of
      S : x : y : z : _         ->
          Just $ ((x :$ z) :$ (y :$ z)      , 3)
```

```
K : x : _                    ->
    Just $ (x                          , 2)
I : x : _                    ->
    Just $ (x                          , 1)
B : f : g : x : _            ->
    Just $ (f :$ (g :$ x)              , 3)
C : f : g : x : _            ->
    Just $ (f :$ x :$ g                , 3)
S' : c : f : g : x :_        ->
    Just $ (c :$ (f :$ x) :$ (g :$ x), 4)
Bstar : c : f : g : x : _ ->
    Just $ (c :$ (f :$ (g :$ x))       , 4)
C' : c : f : g : x : _       ->
    Just $ (c :$ (f :$ x) :$ g         , 4)
_                            -> Nothing
```

The reduction strategy is then defined as such

```
zipperReduce :: Location -> Location
zipperReduce loc
```

The algorithm is driven by the leftmost child.

```
  | (Just left) <- leftChildM loc
  = zipperReduce left
```

If the leftmost child results in a reduction, apply it and continue.

```
  | (Just (x, i)) <- reductionStep
                       (focus loc :
                        rightBranches (parentM loc))
  = zipperReduce $ putFocus x $ ancestor i $ loc
```

If the end of a tree is reached, continue in a depth-first manner by descending to a right subtree.

```
  | (Just next) <- forceRightBranch loc
  = zipperReduce next
```

If no more reductions apply the algorithm terminates.

```
  | otherwise
  = loc
```

References

[1] Huet, G. The zipper. *J. Funct. Program. 7*, 5 (1997), 549–554.

A Testing

```
module Test where
```

In this section we define a small set of tests for the entire process.

```
import Test.HUnit

import Lambda
import CL

import LambdaParser
import CLParser

import Compile
import Reduce

s, z :: CLTerm
s = CLVar "s"
z = CLVar "z"
```

A set of definitions for working with Church numerals in the λ-calculus.

```
churchNumericPrelude :: String
churchNumericPrelude =
    "let zero = (\\s z.z) ; " ++
    "    succ = (\\n s z. s (n s z)) ; " ++
    "    one = succ zero ; " ++
    "    two = succ one ; " ++
    "    three = succ two ; " ++
    "    four = succ three ; " ++
    "    five = succ four ; " ++
    "    six = succ five ; " ++
    "    add = (\\x y s z. x s (y s z)) ; " ++
    "    mul = (\\x y. x (add y) zero) " ++
    " in "
```

A set of definitons for dealing with recursion in the λ-calculus, including a fixed point finder and numeric operations.

```
numericPrelude :: String
numericPrelude =
    "let true  = (\\x y.x) ; " ++
    "    false = (\\x y.y) ; " ++
    "    if    = (\\cond btrue bfalse. cond btrue bfalse) ; " ++
    "    pair  = (\\x y proj. proj x y) ; " ++
    "    fst   = (\\x.x true) ; " ++
    "    snd   = (\\x.x false) ; " ++
    "    zero  = (\\x.x) ; " ++
    "    succ  = (\\n. pair false n) ; " ++
    "    pred  = snd ; " ++
    "    one   = succ zero ; " ++
    "    two   = succ one ; " ++
    "    three = succ two ; " ++
    "    four  = succ three ; " ++
    "    five  = succ four ; " ++
    "    six   = succ five ; " ++
    "    zerop = fst ; " ++
    "    add   = (\\r m n. if (zerop m) n (succ (r (snd m) n))) ; " ++
    "    mul   = (\\r m n. if (zerop m) zero (theta add n (r (pred m) n))) ; "
    "    a     = (\\x y. y (x x y)) ; " ++
    "    theta = a a ; " ++
    "    and   = (\\x y. x (y true false) false) ; " ++
    "    or    = (\\x y. x true (y true false)) ; " ++
    "    eq    = (\\r m n. if (and (zerop m) (zerop n)) " ++
                           "true " ++
                           "(if (or (zerop m) (zerop n)) " ++
                               "false " ++
                               "(r (pred m) (pred n)))) ; " ++
    "    church = (\\r n s z. if (zerop n) z (s (r (pred n) s z))) " ++
    " in "
```

A Church numeral-like expression in Combinatory Logic.

```
toChurchNumCL :: Int -> CLTerm
toChurchNumCL 0 = z
```

```
toChurchNumCL n = s :$ toChurchNumCL (n-1)
```

To make testing numeric operations in the above numeric encodings, the following helpers are defined. They "import" the relevant numeric prelude, compile, modify the term to return something Church numeral-like, and finally reduce the term.

```
churchNumBinary :: String -> CLTerm
churchNumBinary =
    let wrap expr = churchNumericPrelude ++ expr
        apply_sz  = (:$ z) . (:$ s)
    in reduce . apply_sz . compile .  parseLTerm . wrap
```

Here we make use of the recursive function `church` to convert from the list-representation of numbers to the Church numeral representation.

```
numBinary :: String -> CLTerm
numBinary =
    let wrap expr = numericPrelude ++
                    "theta church (" ++ expr ++ ")"
        apply_sz  = (:$ z) . (:$ s)
    in reduce . apply_sz . compile .  parseLTerm . wrap
```

Test some very simple arithmetic operations to make sure the reducer is correct for some cases.

```
test_church_numerals
    = TestList $ map (uncurry mkTest) $
      [ ( 1+1
        , "add one one")
      , ( 1+2
        , "add one two")
      , ( 2*2
        , "mul two two")
      , ( (6*6)*(6*6)
        , "mul (mul six six) (mul six six)")
      ]
    where
      describe x y = "church numerals: " ++ show x ++
                     " = " ++ show y
      mkTest x y = TestCase $
```

```
                    assertEqual (describe x y)
                                (toChurchNumCL x)
                                (churchNumBinary y)

test_numerals
    = TestList $ map (uncurry mkTest) $
      [ ( 1+1
        , "theta add one one")
      , ( 1+2
        , "theta add one two")
      , ( 2*2
        , "theta mul two two")
      , ( (6+6)+(6+6)
        , "theta add (theta add six six) (theta add six six)")
      ]
    where
      describe x y = "numerals: " ++ show x ++
                     " = " ++ show y
      mkTest x y = TestCase $
                   assertEqual (describe x y)
                               (toChurchNumCL x)
                               (numBinary y)
```

Bundle the lists together into `tests`

```
tests = TestList [ TestLabel "test_church_numerals" test_church_numerals
                 , TestLabel "test_numerals"        test_numerals
                 ]
```

The entry point runs the tests.

```
main = do runTestTT tests
```

B Parser Library

```
module ParserLib
 ( Parser,
 token,
 symbol,
 satisfy,
```

```
  (<|>),
  (<&>),
  (<@),
  (<&),
  (&>),
  (<:&>),
  failparser,
  succeed,
  epsilon,
  first,
  star,
  plus,
  plus_bang,
  optionally,
  sp,
  just,
  some,
  listOf,
  neListOf,
  spaces,
  spacelist,
  nespacelist,
  pack
    ) where

type Parser symbol result = [symbol] -> [([symbol], result)]
```

Parsing tokens and symbols:

```
token :: Eq s => [s] -> Parser s [s]

token k xs
    | k == take n xs = [(drop n xs, k)]
    | otherwise = []
       where n = length k

symbol :: Eq s => s -> Parser s [s]
```

```
symbol s = token [s]
satisfy :: (s -> Bool) -> Parser s s

satisfy f (x:xs) = if (f x) then [(xs, x)] else []
satisfy f _ = []
```

Selection:

```
infixr 4 <|>
(<|>)  :: (Parser s a) -> (Parser s a) -> Parser s a
(p1 <|> p2) xs  = p1 xs ++ p2 xs
```

Sequence:

```
infixr 6 <&>
(<&>)  :: (Parser s a) -> (Parser s b) -> Parser s (a, b)
(p1 <&> p2) xs = [(xs2, (v1, v2)) |
                  (xs1, v1) <- p1 xs,
                       (xs2, v2) <- p2 xs1]
```

failure and success:

```
failparser :: Parser s r
failparser _ = []

succeed :: r -> Parser s r
succeed v xs = [(xs, v)]

epsilon = succeed []
```

Semantic functions and some derived combinators.

```
infixl 5 <@
(<@)  :: (Parser s a) -> (a -> b) -> Parser s b

(p0 <@ f) xs = [(ys, f v) | (ys, v) <- p0 xs]
```

```
infixr 6 <&
(<&) :: (Parser s a) -> (Parser s b) -> Parser s a
p <& q = p <&> q <@ fst

infixr 6 &>
(&>) :: (Parser s a) -> (Parser s b) -> Parser s b
p &> q = p <&> q <@ snd
```

B.1 Useful things

Allow spaces:

```
sp :: (Parser Char a) -> Parser Char a

sp p = p . dropWhile (==' ')
```

Only take parses which consume all input:

```
just :: (Parser s a) -> Parser s a

just p = filter (null . fst) . p
```

Parse lists:

```
infixr 6 <:&>

(<:&>) :: (Parser s a) -> (Parser s [a]) -> Parser s [a]

p <:&> q = p <&> q <@ (uncurry (:))
```

Kleene *

```
star :: (Parser s a) -> Parser s [a]
star p = p <:&> star p
       <|> epsilon
```

Take the first parse

```
first :: (Parser s a) -> Parser s a

first p = (take 1) . p

some p = snd . head . just p
```

Kleene *

```
plus :: (Parser s a) -> Parser s [a]

plus p = p <:&> star p

plus_bang :: (Parser s a) -> Parser s [a]

plus_bang = first . plus
```

Allow optional items:

```
optionally :: (Parser s a) -> Parser s [a]
optionally p = p <@ (\x -> [x])
            <|> epsilon
```

Parsing lists separated by menaingless separators inside meaningless brackets:

```
pack :: (Parser s a) -> (Parser s b) ->
                (Parser s c) -> Parser s b

pack opensym p closesym = opensym &> p <& closesym

listOf :: (Parser s a) ->
          (Parser s b) ->
           Parser s [a]

listOf p s =       p <:&> star (s &> p)
             <|> epsilon

neListOf p s = p <:&> star (s &> p)

spaces :: Parser Char String

spaces = (first . plus) (satisfy isSpace)

spacelist :: (Parser Char a) -> Parser Char [a]

spacelist p = listOf p spaces
```

```
nespacelist p = neListOf p spaces

isSpace c = c == ' '
```