λ-calculus cooked four ways

Lennart Augustsson

1 Introduction

This little paper describes how to implement λ-calculus in four different ways. To be precise, it shows how to implement the functions that compute the (β) normal form of an expression.

2 Preliminaries

2.1 Lambda

The Lambda module implements a simple abstract syntax for λ-calculus together with a parser and a printer for it. It also exports a simple type of identifiers that parse and print in a nice way.

```
module Lambda(LC(..), freeVars, allVars, Id(..)) where
import Data.List(span, union, (\\))
import Data.Char(isAlphaNum)
import Text.PrettyPrint.HughesPJ(Doc, renderStyle, style, text,
           (<>), (<+>), parens)
import Text.ParserCombinators.ReadP
```

The LC type of λ term is parametrized over the type of the variables. It has constructors for variables, λ-abstraction, and application.

```
data LC v = Var v | Lam v (LC v) | App (LC v) (LC v)
   deriving (Eq)
```

Compute the free variables of an expression.

```
freeVars :: (Eq v) => LC v -> [v]
freeVars (Var v) = [v]
freeVars (Lam v e) = freeVars e \\ [v]
freeVars (App f a) = freeVars f `union` freeVars a
```

Compute all variables in an expression.

```
allVars :: (Eq v) => LC v -> [v]
allVars (Var v) = [v]
allVars (Lam _ e) = allVars e
allVars (App f a) = allVars f `union` allVars a
```

The Read instance for the LC type reads λ term with the normal syntax.

```
instance (Read v) => Read (LC v) where
    readsPrec _ = readP_to_S pLC
```

A ReadP parser for λ-expressions.

```
pLC, pLCAtom, pLCVar, pLCLam, pLCApp :: (Read v) => ReadP (LC v)
pLC = pLCLam +++ pLCApp +++ pLCLet

pLCVar = do
    v <- pVar
    return $ Var v

pLCLam = do
    schar '\\'
    v <- pVar
    schar '.'
    e <- pLC
    return $ Lam v e

pLCApp = do
    es <- many1 pLCAtom
    return $ foldl1 App es

pLCAtom = pLCVar +++ (do schar '('; e <- pLC; schar ')'; return e)
```

To make expressions a little easier to read we also allow let expression as a syntactic sugar for λ and application.

```
pLCLet :: (Read v) => ReadP (LC v)
pLCLet = do
    let lcLet (x,e) b = App (Lam x b) e
        pDef = do
          v <- pVar
          schar '='
          e <- pLC
          return (v, e)
    sstring "let"
    bs <- sepBy pDef (schar ';')
    sstring "in"
    e <- pLC
    return $ foldr lcLet e bs


schar :: Char -> ReadP Char
schar c = do skipSpaces; char c

sstring :: String -> ReadP String
sstring c = do skipSpaces; string c

pVar :: (Read v) => ReadP v
pVar = do skipSpaces; readS_to_P (readsPrec 9)
```

Pretty print λ-expressions when shown.

```
instance (Show v) => Show (LC v) where
    show = renderStyle style . ppLC 0
```

```
ppLC :: (Show v) => Int -> LC v -> Doc
ppLC _ (Var v) = text $ show v
ppLC p (Lam v e) = pparens (p>0) $ text ("\\" ++ show v ++ ".") <> ppLC 0 e
ppLC p (App f a) = pparens (p>1) $ ppLC 1 f <+> ppLC 2 a

pparens :: Bool -> Doc -> Doc
pparens True d = parens d
pparens False d = d
```

The Id type of identifiers.

```
newtype Id = Id String
    deriving (Eq, Ord)
```

Identifiers print and parse without any adornment.

```
instance Show Id where
    show (Id i) = i

instance Read Id where
    readsPrec _ s =
        case span isAlphaNum s of
        ("", _) -> []
        (i, s') -> [(Id i, s')]
```

2.2 IdInt

A fast type of identifiers, Ints, for λ-expressions.

```
module IdInt(IdInt(..), firstBoundId, toIdInt) where
import Data.Map as M
import Control.Monad.State
import Lambda
```

An IdInt is just another name for an Int.

```
newtype IdInt = IdInt Int
    deriving (Eq, Ord)

firstBoundId :: IdInt
firstBoundId = IdInt 0
```

It is handly to make IdInt enumerable.

```
instance Enum IdInt where
    toEnum i = IdInt i
    fromEnum (IdInt i) = i
```

We show IdInts so they look like variables. Negative numbers are free variables.

```
instance Show IdInt where
   show (IdInt i) = if i < 0 then "f" ++ show (-i) else "x" ++ show i
```

Any variable type can be converted to IdInt if we can just build a table of them. The conversion assigns a different Int to each different original identifier. Free variables in the expression are translated into negative numbers so they are easily distinguished later.

```
toIdInt :: (Ord v) => LC v -> LC IdInt
toIdInt e = evalState (conv e) (0, fvmap)
  where fvmap = Prelude.foldr (\ (v, i) m -> insert v (IdInt (-i)) m) empty
                              (zip (freeVars e) [1..])
```

The state monad has the next unused Int and a mapping of identifiers to IdInt.

```
type M v a = State (Int, Map v IdInt) a
```

The only operation we do in the monad is to convert a variable. If the variable is in the map the use it, otherwise add it.

```
convVar :: (Ord v) => v -> M v IdInt
convVar v = do
   (i, m) <- get
   case M.lookup v m of
       Nothing -> do
           let ii = IdInt i
           put (i+1, insert v ii m)
           return ii
       Just ii -> return ii

conv :: (Ord v) => LC v -> M v (LC IdInt)
conv (Var v) = liftM Var (convVar v)
conv (Lam v e) = liftM2 Lam (convVar v) (conv e)
conv (App f a) = liftM2 App (conv f) (conv a)
```

3 Naïve Substitution

The Simple module implements the Normal Form function by using a naïve version of substitution.

```
module Simple(nf) where
import Data.List(union, (\\))
import Lambda
import IdInt
```

The normal form is computed by repeatedly performing substitution (β-reduction) on the leftmost redex. Variables and abstractions are easy, but in the case of an application we must compute the function to see if it is an abstraction. The function cannot be computed with the `nf` function since it could perform non-leftmost reductions. Instead we use the `whnf` function.

```
nf :: LC IdInt -> LC IdInt
nf e@(Var _) = e
nf (Lam x e) = Lam x (nf e)
nf (App f a) =
    case whnf f of
        Lam x b -> nf (subst x a b)
        f' -> App (nf f') (nf a)
```

Compute the weak head normal form. It is similar to computing the normal form, but it does not reduce under λ, nor does it touch an application that is not a β-redex.

```
whnf :: LC IdInt -> LC IdInt
whnf e@(Var _) = e
whnf e@(Lam _ _) = e
whnf (App f a) =
    case whnf f of
        Lam x b -> whnf (subst x a b)
        f' -> App f' a
```

Substitution has only one interesting case, the abstraction. For abstraction there are three cases: if the bound variable, `v`, is equal to the variable we are replacing, `x`, then we are done, if the bound variable is in set set of free variables of the substituted expression then there would be an accidental capture and we rename it, otherwise the substitution just continues.

How should the new variable be picked when doing the renaming? The new variable must not be in the set of free variables of `s` since this would case another accidental capture, nor must it be among the free variables of `e'` since this could cause another accidental capture. Conservatively, we avoid all variables occuring in the original `b` to fulfill the second requirement.

```
subst :: IdInt -> LC IdInt -> LC IdInt -> LC IdInt
subst x s b = sub b
 where sub e@(Var v) | v == x = s
                     | otherwise = e
       sub e@(Lam v e') | v == x = e
                        | v `elem` fvs = Lam v' (sub e'')
                        | otherwise = Lam v (sub e')
                            where v' = newId vs
                                  e'' = subst v (Var v') e'
       sub (App f a) = App (sub f) (sub a)

       fvs = freeVars s
       vs = fvs `union` allVars b
```

Get a variable which is not in the given set. Do this simply by generating all variables and picking the first not in the given set.

```
newId :: [IdInt] -> IdInt
newId vs = head ([firstBoundId .. ] \\ vs)
```

4 The Barendregt Convention

The Unique module implements the Normal Form function by using Barendregt's variable convention, i.e., all bound variables are unique.

```
module Unique(nf) where
import Lambda
import qualified Data.Map as M
import Control.Monad.State
import IdInt
```

The first step is to make all variables unique. Then normal form is computed by repeatedly performing substitution (beta reduction) on the leftmost redex. Normalization is run in a State monad with the next free variable.

```
nf :: LC IdInt -> LC IdInt
nf e = evalState (nf' e') i
  where (e', (i, _)) = runState (unique e) (firstBoundId, M.empty)

type N a = State IdInt a

nf' :: LC IdInt -> N (LC IdInt)
nf' e@(Var _) = return e
nf' (Lam x e) = liftM (Lam x) (nf' e)
nf' (App f a) = do
    f' <- whnf f
    case f' of
        Lam x b -> subst x a b >>= nf'
        _ -> liftM2 App (nf' f') (nf' a)
```

Compute the weak head normal form.

```
whnf :: LC IdInt -> N (LC IdInt)
whnf e@(Var _) = return e
whnf e@(Lam _ _) = return e
whnf (App f a) = do
    f' <- whnf f
    case f' of
        Lam x b -> subst x a b >>= whnf
        _ -> return $ App f' a
```

Substitution proceeds by cloning the term that is inserted at every place it is put.

(TODO: No need to clone λ-free terms.)

```
subst :: IdInt -> LC IdInt -> LC IdInt -> N (LC IdInt)
subst x s b = sub b
 where sub e@(Var v) | v == x = clone M.empty s
                     | otherwise = return e
       sub (Lam v e) = liftM (Lam v) (sub e)
       sub (App f a) = liftM2 App (sub f) (sub a)

       clone m e@(Var v) = return $ maybe e Var (M.lookup v m)
       clone m (Lam v e) = do v' <- newVar; liftM (Lam v') (clone (M.insert v v' m) e)
       clone m (App f a) = liftM2 App (clone m f) (clone m a)
```

Create a fresh variable.

```
newVar :: N IdInt
newVar = do
    i <- get
    put (succ i)
    return i
```

Do the actual translation of the term to unique variables. We keep mapping of old variable names to new variable name. Free variables are just left alone since they are already uniquely named.

```
type U a = State (IdInt, M.Map IdInt IdInt) a

unique :: LC IdInt -> U (LC IdInt)
unique (Var v) = liftM Var (getVar v)
unique (Lam v e) = liftM2 Lam (addVar v) (unique e)
unique (App f a) = liftM2 App (unique f) (unique a)
```

Add a variable to the mapping.

```
addVar :: IdInt -> U IdInt
addVar v = do
   (i, m) <- get
   put (succ i, M.insert v i m)
   return i
```

Find an existing variable in the mapping.

```
getVar :: IdInt -> U IdInt
getVar v = do
    (_, m) <- get
    return $ maybe v id (M.lookup v m)
```

5 Higher Order Abstract Syntax

The HOAS module implements the Normal Form function by using Higher Order Abstract Syntax for the λ-expressions. This makes it possible to use the native substitution of Haskell.

```
module HOAS(nf) where
import qualified Data.Map as M
import Lambda
import IdInt
```

With higher order abstract syntax the abstraction in the implemented language is represented by an abstraction in the implementation language. We still need to represent variables for free variables and also during conversion.

```
data HOAS = HVar IdInt | HLam (HOAS -> HOAS) | HApp HOAS HOAS
```

To compute the normal form, first convert to HOAS, compute, and convert back.

```
nf :: LC IdInt -> LC IdInt
nf = toLC . nfh . fromLC
```

The substitution step for HOAS is simply a Haskell application since we use a Haskell function to represent the abstraction.

```
nfh :: HOAS -> HOAS
nfh e@(HVar _) = e
nfh (HLam b) = HLam (nfh . b)
nfh (HApp f a) =
    case whnf f of
        HLam b -> nfh (b a)
        f' -> HApp (nfh f') (nfh a)
```

Compute the weak head normal form.

```
whnf :: HOAS -> HOAS
whnf e@(HVar _) = e
whnf e@(HLam _) = e
whnf (HApp f a) =
    case whnf f of
        HLam b -> whnf (b a)
        f' -> HApp f' a
```

Convert to higher order abstract syntax. Do this by keeping a mapping of the bound variables and translating them as they are encountered.

```
fromLC :: LC IdInt -> HOAS
fromLC = from M.empty
  where from m (Var v) = maybe (HVar v) id (M.lookup v m)
        from m (Lam v e) = HLam $ \ x -> from (M.insert v x m) e
        from m (App f a) = HApp (from m f) (from m a)
```

Convert back from higher order abstract syntax. Do this by inventing a new variable at each λ.

```
toLC :: HOAS -> LC IdInt
toLC = to firstBoundId
  where to _ (HVar v) = Var v
        to n (HLam b) = Lam n (to (succ n) (b (HVar n)))
        to n (HApp f a) = App (to n f) (to n a)
```

6 deBruijn indicies

The DeBruijn module implements the Normal Form function by using de Bruijn indicies.

```
module DeBruijn(nf) where
import Data.List(elemIndex)
import Lambda
import IdInt
```

Variables are represented by their binding depth, i.e., how many λs out the binding λ is. Free variables are represented by negative numbers.

```
data DB = DVar !Int | DLam DB | DApp DB DB

nf :: LC IdInt -> LC IdInt
nf = fromDB . nfd . toDB
```

Computing the normal form proceeds as usual.

```
nfd :: DB -> DB
nfd e@(DVar _) = e
nfd (DLam e) = DLam (nfd e)
nfd (DApp f a) =
    case whnf f of
        DLam b -> nfd (subst 0 a b)
        f' -> DApp (nfd f') (nfd a)
```

Compute the weak head normal form.

```
whnf :: DB -> DB
whnf e@(DVar _) = e
whnf e@(DLam _) = e
whnf (DApp f a) =
    case whnf f of
        DLam b -> whnf (subst 0 a b)
        f' -> DApp f' a
```

Substitution needs to adjust the inserted expression so the free variables refer to the correct binders.

```
subst :: Int -> DB -> DB -> DB
subst o s v@(DVar i) | i == o = adjust 0 s
                     | i >  o = DVar (i-1)
                     | otherwise = v
  where adjust n e@(DVar j) | j >= n = DVar (j+o)
                            | otherwise = e
        adjust n (DLam e) = DLam (adjust (n+1) e)
        adjust n (DApp f a) = DApp (adjust n f) (adjust n a)
subst o s (DLam e) = DLam (subst (o+1) s e)
subst o s (DApp f a) = DApp (subst o s f) (subst o s a)
```

Convert to deBruijn indicies. Do this by keeping a list of the bound variable so the depth can be found of all variables. Do not touch free variables.

```
toDB :: LC IdInt -> DB
toDB = to []
  where to vs (Var v@(IdInt i)) = maybe (DVar i) DVar (elemIndex v vs)
        to vs (Lam x b) = DLam (to (x:vs) b)
        to vs (App f a) = DApp (to vs f) (to vs a)
```

Convert back from deBruijn to the LC type.

```
fromDB :: DB -> LC IdInt
fromDB = from firstBoundId
  where from (IdInt n) (DVar i) | i < 0 = Var (IdInt i)
                                | otherwise = Var (IdInt (n-i-1))
        from n (DLam b) = Lam n (from (succ n) b)
        from n (DApp f a) = App (from n f) (from n a)
```

7 Tasting time

Finally, we want to try out the different implementations. To this end we have a simple main program to pick which normal form function to use.

```
import Misc
import Lambda
import IdInt
import Simple
import Unique
import HOAS
import DeBruijn
import DeBruijnC
```

```
main :: IO ()
main = interactArgs $
       \ args -> (++ "\n") . show . myNF args . toIdInt . f . read . stripComments
  where f :: LC Id -> LC Id  -- just to force the type
       f e = e
       myNF ["U"] = Unique.nf
       myNF ["H"] = HOAS.nf
       myNF ["D"] = DeBruijn.nf
       myNF ["C"] = DeBruijnC.nf
       myNF ["S"] = Simple.nf
```

Timing in seconds on a MacBook processing the file `timing.lam`.

Simple.nf	8.3
Unique.nf	26.6
HOAS.nf	0.13
DeBruijn.nf	41.1
DeBruijnEnv.nf	41.1

The λ-expression in `timing.lam` computes "`factorial 6 == sum [1..37] + 17`", but using Church numerals.

`timing.lam`:

```
let False = \f.\t.f;
    True = \f.\t.t;
    if = \b.\t.\f.b f t;
    Zero = \z.\s.z;
    Succ = \n.\z.\s.s n;
    one = Succ Zero;
    two = Succ one;
    three = Succ two;
    isZero = \n.n True (\m.False);
    const = \x.\y.x;
    Pair = \a.\b.\p.p a b;
    fst = \ab.ab (\a.\b.a);
    snd = \ab.ab (\a.\b.b);
    fix = \ g. (\ x. g (x x)) (\ x. g (x x));
    add = fix (\radd.\x.\y. x y (\ n. Succ (radd n y)));
    mul = fix (\rmul.\x.\y. x Zero (\ n. add y (rmul n y)));
    fac = fix (\rfac.\x. x one (\ n. mul x (rfac n)));
    eqnat = fix (\reqnat.\x.\y. x (y True (const False)) (\x1.y False (\y1.reqnat x1 y1)))
    sumto = fix (\rsumto.\x. x Zero (\n.add x (rsumto n)));
    n5 = add two three;
    n6 = add three three;
    n17 = add n6 (add n6 n5);
    n37 = Succ (mul n6 n6);
    n703 = sumto n37;
```

```
    n720 = fac n6
in  eqnat n720 (add n703 n17)
```

8 Conclusions

This test is too small to draw any deep conclusions, but higher order syntax looks very good. Furthermore, doing the simplest thing is not necessarily bad.