Untyped λ-calculus exercises

Adrián Enríquez Ballester

January 9, 2022

The λ-calculus in Haskell

Exercise 11

Using Haskell, define functions boolChurch and boolUnchurch which translate Haskell booleans into Church booleans and vice versa. Use them to check the correctness of your solutions to Exercise 6.

Answer

The implementation is provided in a Haskell source code file named `ChurchBool.hs` within the folder `untyped-church-encodings`.

The script `church-bool-test.sh` simulates a sample session in `ghci` using the defined module and can be executed as follows:

```
chmod u+x church-bool-test.sh && ./church-bool-test.sh
```

Exercise 12

Analogously to the previous exercise, define Haskell functions to convert between Haskell and Church natural numbers, and check the correctness of your solutions to to Exercise 7.

Answer

The implementation is provided in a Haskell source code file named `ChurchNum.hs` within the folder `untyped-church-encodings`.

As in the previous exercise, the script `church-num-test.sh` simulates a sample session in `ghci` using the defined module and can be executed as follows:

```
chmod u+x church-num-test.sh && ./church-num-test.sh
```

Note for the following exercises

The code shown for the following exercises is also provided as a Haskell project managed with Cabal within the folder `untyped-lambda-calculus`.

A simple test suite is also available, which can be executed with the script `test.sh` as follows:

```
chmod u+x test.sh && ./test.sh
```

Exercise 13

Define a Haskell datatype to represent the abstract syntax of the λ-calculus.

Answer

We have used `String` for the set of variable names, but we could have used another set or even parameterize it. The datatype and its value constructor names are self explanatory:

```
data Term
  = Var String
  | Lam String Term
  | App Term Term
```

Exercise 14

Based on the previous exercise, define a Haskell function that obtains the free variables of a lambda term.

Answer

We have used a `Set` data structure from the `containers` package. The function does pattern match on each `Term` value constructor:

```
freeVars :: Term -> S.Set String
freeVars (Var v)     = S.singleton v
freeVars (Lam v t)   = S.delete v $ freeVars t
freeVars (App t1 t2) = freeVars t1 <> freeVars t2
```

Exercise 15

Based on the previous exercises, define a Haskell function that implements capture avoiding substitution.

Answer

First, we provide an auxiliary function for obtaining an infinite stream of variable names:

```
auxVars :: Char -> [String]
auxVars p = map ((p :) . show) [0 :: Integer ..]
```

The following function instantiates one of this infinite streams of variable names, filters some elements which would be already useless and proceeds with a recursive definition case by case for the capture avoiding variable substitution:

```
subs :: String -> Term -> Term -> Term
subs v e =
  go . filter (`S.notMember` freeE) . auxVars $ 'x'
  where
    freeE = freeVars e
    go aux (App t1 t2) = App (go aux t1) (go aux t2)
    go _ t@(Var v')
      | v' == v = e
      | otherwise = t
    go aux t@(Lam v' b)
      | v' == v || not bodyCapture = t
      | bodyCapture && not varCapture = Lam v' $ go aux b
      | otherwise = Lam auxVar . go aux' . subs v' (Var auxVar) $ b
      where
        freeT = freeVars b
        bodyCapture = v `S.member` freeT
        varCapture = v' `S.member` freeE
        (auxVar : aux') = dropWhile (`S.member` freeT) aux
```

Note that, once it requires a fresh variable, it starts consuming the variable names stream until it finds one that is fine and keeps the remaining contents of the stream for later.

Exercise 16

Based on the previous exercises, define a Haskell function that implements β-reduction (one step).

Answer

The following function performs a single step β-reduction only if a β-redex is provided, and returns `Nothing` in any other case:

```
betaRed :: Term -> Maybe Term
betaRed (App (Lam v t) e) = Just $ subs v e t
betaRed _                 = Nothing
```

Now, to perform a reduction like that in just one subterm by following a specific evaluation order, we define a function which takes a transformation an performs it to the first subterm which admits it according to normal order evaluation. It takes advantage of the `Alternative` typeclass instance of `Maybe` to achieve such a concise definition:

```
normalOrder :: Alternative f => (Term -> f Term) -> Term -> f Term
normalOrder red v@(Var _) = red v
normalOrder red l@(Lam v t) =
  red l
    <|> (Lam v <$> normalOrder red t)
normalOrder red a@(App t1 t2) =
  red a
    <|> ((`App` t2) <$> normalOrder red t1)
    <|> ((t1 `App`) <$> normalOrder red t2)
```

With this, we can get a function which performs a single step β-reduction in normal order evaluation as follows:

```
normalOrder betaRed
```

Exercise 17

Based on the previous exercises, define a Haskell function that reduces a lambda term into β-normal form when possible.

Answer

The following function performs recursively a single step β-reduction in normal order evaluation until it does not admit more steps:

```
betaNorm :: Term -> Term
betaNorm t = maybe t betaNorm $ beta t
  where
    beta = normalOrder betaRed
```

Suppose that we had also defined a single step η-reduction for an η-redex named `etaRed`. Note how easy it would be to extend the previous definition to $\beta\eta$-normal forms:

```
betaEtaNorm :: Term -> Term
betaEtaNorm t = maybe t betaEtaNorm $ beta t <|> eta t
  where
    beta = normalOrder betaRed
    eta = normalOrder etaRed
```