Simply Typed Lambda Calculus

From Untyped to Simply Typed Lambda Calculus

Sven Tennie

September 8, 2018

Dream IT
https://dreamit.de

Untyped Lambda Calculus

Untyped Lambda Calculus - Recapitulation

We can boil down computation to a tiny calculus

All we need is:

- Function Definition / Abstraction ($\lambda x.e$)
- Function Application (ee)
- Parameters / Variables (x)

Then we get:

- Booleans
- Numerals
- Data Structures
- Control Flow
- ...
- Turing Completeness (If it can be computed, it can be

Build an Interpreter

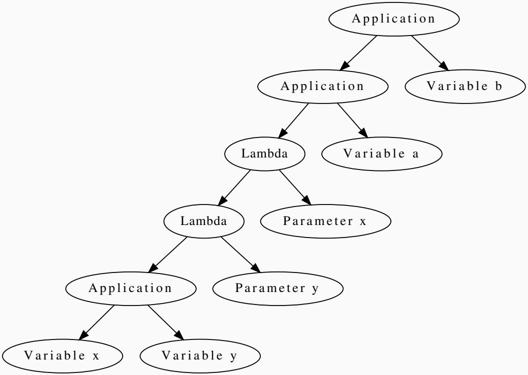
Let's build an interpreter

- Deepen our intiution
- Later move on to the *Simply Typed Lambda Calculus*
  - Why do we need types?
  - How does a type checker work?
  - How does it restrict the programs we might write?
- On our way we'll learn some math mumbo-jumbo: *Natural Deduction*
  - Found in many papers about Type Systems and Programming Language Evaluation

Structure

$e ::=$	Expressions:
x	Variable
$\lambda x.t$	Abstraction
$t\ t$	Application

Abstract Syntax Tree



$(\lambda x.\lambda y.x\ y)\ a\ b$

Interpreter - Syntax

```
module UntypedSyntax where

type Name = String

data Expr
  = Var Name
  | App Expr
^^IExpr
  | Lambda Name
^^I   Expr
  deriving (Eq, Show)
```

Interpreter - Syntax - Examples

```
module UntypedSyntaxExamples where

import UntypedSyntax

-- true ≡ λp. lambdaq.p
true :: Expr
true = Lambda "p" (Lambda "q" (Var "p"))

-- false ≡ λp. lambdaq.q
false :: Expr
false = Lambda "p" (Lambda "q" (Var "p"))

-- if_then_else ≡ λp.λa.λb.pab
if_then_else :: Expr
if_then_else =
  Lambda "p" (Lambda "a" (Lambda "b" (App (App (Var "p") (Var "a")) (Var "b"))))
```

Evaluation Rules - Call by Value

Interpreter - Evaluation

```haskell
module NaiveUntypedEval where

import UntypedSyntax

eval :: Expr -> Expr
eval variable@(Var _) = variable
eval lambda@(Lambda _ _) = lambda
eval (App term1 term2) =
  case eval term1 of
    (Lambda name term1') -> eval $ substitute name term2 term1'
    term -> App term term2
```

Interpreter - Substitution

```haskell
substitute :: String -> Expr -> Expr -> Expr
substitute name substitution var@(Var varName)
  | name  == varName = substitution
  | otherwise = var
substitute name substitution (App term1 term2) =
  App (substitute name substitution term1) (substitute name substitution term2)
substitute name substitution (Lambda varName term) =
  if name == varName
    then Lambda varName term
    else Lambda varName (substitute name substitution term)
```

Interpreter with Environment

```haskell
module UntypedEval where

import UntypedSyntax

import qualified Data.Map.Strict as Map

type Environment = Map.Map Name Expr

eval :: Environment -> Expr -> Maybe Expr
eval env (Var name) = find env name
eval env (App term1 term2) = case eval env term1 of
  Just (Lambda name term) -> eval (Map.insert name term2 env) term
  Just term                -> Just (App term term2)
  Nothing -> Nothing
eval env lambda@(Lambda _ _) = Just lambda

find ::  Environment -> Name -> Maybe Expr
find env name = Map.lookup name env
```

Tests

Simply Typed Lambda Calculus

Interpreter

Type Checker

```
module TypedCheck where

import qualified Data.Map.Strict as Map
import Data.Either.Extra

type Name = String
type Environment = Map.Map Name Type

data Type  = TInt
^^I   | TBool
^^I   | TArr Type Type
^^I   deriving (Eq, Show)

data Term = Variable Name |
^^I      Application Term Term |
^^I      Abstraction Name Type Term
^^I      deriving (Eq, Show)

check :: Environment -> Term -> Either String Type
check env (Variable name) = find env name
check env (Application term1 term2) =
  do
    (TArr ta1 ta2) <- check env term1
    t2 <- check env term2
    if ta1 == t2 then
      Right t2
    else
      Left $ "Expected " ++ (show ta1) ++ " but got : " ++ (show t2)
check env (Abstraction name atype term) = do
```

Tests