Simply Typed Lambda Calculus

From Untyped to Simply Typed Lambda Calculus

Sven Tennie

August 26, 2018

Dream IT
https://dreamit.de

Untyped Lambda Calculus

Untyped Lambda Calculus - Retrospection

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

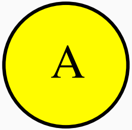
Let's build an intepreter

- 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

$t ::=$	Terms:
x	Variable
$\lambda x.t$	Abstraction
$t\ t$	Application

Abstract Syntax Tree



Evaluation

Naive Interpreter

```
module NaiveUntypedEval where

type Name = String

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

eval  :: Term → Term
eval variable@(Variable _) = variable
eval abstraction@(Abstraction _ _) = abstraction
eval (Application term1 term2) = case eval term1 of
  (Abstraction name term1') → eval $ substitute name term2 term1'
  term                      → Application term term2

substitute :: String → Term → Term → Term
substitute name substitution (Variable varName) = if name == varName then
^^I^^I^^I^^I^^I^^I    substitution
^^I^^I^^I^^I^^I^^I  else
^^I^^I^^I^^I^^I^^I    Variable varName
substitute name substitution (Application term1 term2) = Application (substitute name substitu
substitute name substitution (Abstraction varName term) = if name == varName then
^^I^^I^^I^^I^^I^^I^^I    Abstraction varName term
^^I^^I^^I^^I^^I^^I^^I  else
^^I^^I^^I^^I^^I^^I^^I    Abstraction varName (substitute name substitution term)
```

Interpreter with Environment

```haskell
module UntypedEval where
import qualified Data.Map.Strict as Map

type Name = String
type Environment = Map.Map Name Term

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

eval :: Environment → Term → Maybe Term
eval env (Variable name) = find env name
eval env (Application term1 term2) = case eval env term1 of
  Just (Abstraction name term) → eval (Map.insert name term2 env) term
  Just term                    → Just (Application term term2)
  Nothing → Nothing
eval env abstraction@(Abstraction _ _) = Just abstraction

find ::  Environment → Name → Maybe Term
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