

Simply Typed Lambda Calculus

From Untyped to Simply Typed Lambda Calculus

Sven Tennie

September 9, 2018

Dream IT

<https://dreamit.de>

Untyped Lambda Calculus

Untyped Lambda Calculus - Recapitulation

We can boil down computation to a tiny 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 ($e e$)
- Parameters / Variables (x)

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 ($e e$)
- Parameters / Variables (x)

Then we get:

PL constructs

- Booleans
- Numerals
- Data Structures
- Control Flow

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 ($e e$)
- Parameters / Variables (x)

Then we get:

PL constructs

- Booleans
- Numerals
- Data Structures
- Control Flow

Turing Completeness

- Turing Completeness
 - If it can be computed, it can be computed in Lambda Calculus!

Build an Interpreter

Let's build an interpreter

- Deepen our intuition
- 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

$e ::=$

x

$\lambda x.e$

$e\ e$

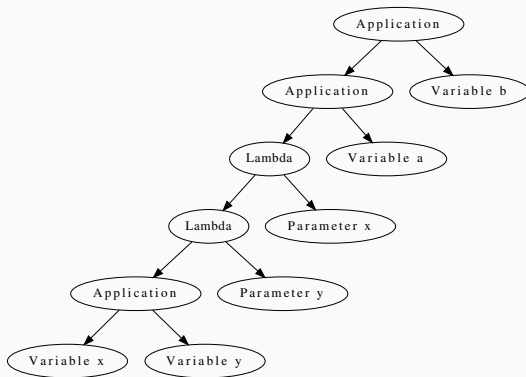
Expressions:

Variable

Abstraction

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
```

```
      Expr
```

```
  | Lambda Name
```

```
      Expr
```

```
  deriving (Eq, Show)
```

Interpreter - Syntax - Examples

```
module UntypedSyntaxExamples where

import UntypedSyntax

-- true  $\equiv \lambda p. \text{lambda}q.p$ 
true :: Expr
true = Lambda "p" (Lambda "q" (Var "p"))

-- false  $\equiv \lambda p. \text{lambda}q.q$ 
false :: Expr
false = Lambda "p" (Lambda "q" (Var "p"))

-- if_then_else  $\equiv \lambda p. \lambda a. \lambda b. pab$ 
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

Some rules...

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \quad \text{E-App1}$$

$$\frac{e_2 \rightarrow e'_2}{v_1 e_2 \rightarrow v_1 e'_2} \quad \text{E-App2}$$

$$(\lambda x. e) v \rightarrow [x/v]e \quad \text{E-AppLam}$$

Interpreter - Evaluation

```
module NaiveUntypedEval where

import UntypedSyntax

eval :: Expr -> Expr
-- No rule for variables
eval variable@(Var _) = variable
-- No rule for lambdas
eval lambda@(Lambda _ _) = lambda
eval (App e1 e2)
  -- 
$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \quad (E - App1)$$

  =
    let e1' = eval e1
    -- 
$$\frac{e_2 \rightarrow e'_2}{v_1 e_2 \rightarrow v_1 e'_2} \quad (E - App2)$$

    in let e2' = eval e2
       in case e1'
         of
           -- 
$$(\lambda x. e) v \rightarrow [x/v]e \quad (E - AppLam)$$

           (Lambda name e1'_body) -> eval $ substitute name e2' e1'_body
           e1' -> App e1' e2'
```

Interpreter - Substitution

```
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

```
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
```

Simply Typed Lambda Calculus

$e ::=$

x

$\lambda x : \tau. e$

$e e$

Expressions:

Variable

Abstraction

Application

Progress : If an expression is well typed then either it is a value, or it can be further evaluated by an available evaluation rule.

Preservation : If an expression e has type τ , and is evaluated to e' , then e' has type τ .

Dynamic rules stay the same!

- Type checking is done upfront

Typing Rules

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

T-Var

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2}$$

T-Lam

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

T-App

$$\Gamma \vdash n : \text{Int}$$

T-Int

$$\Gamma \vdash \text{True} : \text{Bool}$$

T-True

Type Checker

```
module TypedSyntax where

import qualified Data.Map.Strict as Map

type Name = String

type Environment = Map.Map Name Type

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

data Expr
  = IntValue Int
  | BoolValue Bool
  | Var Name
  | App Expr
      Expr
  | Lambda Name
      Type
      Expr
  deriving (Eq, Show)
```

Type Checker - Literals & Variables

```
module TypedCheck where

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

import TypedSyntax

find :: Environment -> Name -> Either String Type
find env name = maybeToEither "Var not found!" (Map.lookup name env)

check :: Environment -> Expr -> Either String Type
--
--  $\Gamma \vdash n : \text{Int} \quad (T\text{-Int})$ 
--
check _ (IntValue _) = Right TInt
--
--  $\Gamma \vdash \text{True} : \text{Bool} \quad (T\text{-True})$ 
--
check _ (BoolValue True) = Right TBool
--
--  $\Gamma \vdash \text{False} : \text{Bool} \quad (T\text{-False})$ 
--
check _ (BoolValue False) = Right TBool
--
--  $\frac{x:\sigma \in \Gamma}{\Gamma \vdash x:\sigma} \quad (T\text{-Var})$ 
--
check env (Var name) = find env name
```


Type Checker - Lambda & Application

```
--  
--       $\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \lambda x:\tau_1. e:\tau_1 \rightarrow \tau_2}$    (T-Lam)  
--
```

```
check env (Lambda name atype e) = do  
  t <- check (Map.insert name atype env) e  
  return $ TArr atype t
```

```
--  
--       $\frac{\Gamma \vdash e_1:\tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2:\tau_1}{\Gamma \vdash e_1 e_2:\tau_2}$    (T-App)  
--
```

```
check env (App e1 e2) = do  
  (TArr ta1 ta2) <- check env e1  
  t2 <- check env e2  
  if ta1 == t2  
    then Right ta2  
    else Left $ "Expected " ++ (show ta1) ++ " but got : " ++ (show t2)
```
