Simply Typed Lambda Calculus

From Untyped to Simply Typed Lambda Calculus

Sven Tennie

September 13, 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 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

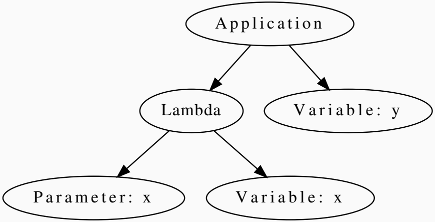
$$
\begin{array}{lr}
e ::= & \text{Expressions:} \\
\quad x & \text{Variable} \\
\quad \lambda x.e & \text{Abstraction} \\
\quad e\ e & \text{Application}
\end{array}
$$

$\lambda x.e$ Function Definition

$e\ e$ Function Application

Abstract Syntax Tree

$(\lambda x.x)\ y$



Meaning:

- Identity function $(\lambda x.x)$ is applied to a variable (y)

Interpreter - Syntax

```haskell
module UntypedSyntax where

type Name = String

data Expr                    -- e ::=        Expressions:
  = Var Name                 --       x      Variable
  | Lambda Name              --       λx.e   Abstraction
           Expr
  | App Expr                 --       e e    Application
        Expr
  deriving (Eq, Show)
```

Interpreter - Syntax - Examples

```haskell
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"))))
```

Natural Deduction

Proof: 2 is a Natural Number

$$\frac{}{0 : \texttt{Nat}} \quad \text{(A1)}$$

$$\frac{n : \texttt{Nat}}{\texttt{succ}(n) : \texttt{Nat}} \quad \text{(A2)}$$

$$\dfrac{\dfrac{\dfrac{}{0 : Nat}\ \text{(A1)}}{succ(0) : Nat}\ \text{(A2)}}{succ(succ(0)) : Nat}\ \text{(A2)}$$

Meaning:

A1 0 is a natural number (by definition)

A2 The successor of a natural number is a natural number

→ Thus the sucessor of the successor of 0 (2) must be a

Evaluation Rules - Call by Value - E-App1

$$\frac{e_1 \rightarrow e_1'}{e_1\, e_2 \rightarrow e_1'\, e_2} \qquad \text{E-App1}$$

Meaning:

- Under the condition that e_1 can be reduced further, do it.

Evaluation Rules - Call by Value - E-App2

$$\frac{e_2 \to e_2'}{v_1\, e_2 \to v_1\, e_2'} \qquad \text{E-App2}$$

Meaning:

- Under the condition that e_2 can be reduced further and v_1 is a value, do it.
- Pure Untyped Lambda Calculus:
  - Only Lambdas (functions) are values.
  - But you can add Ints, Booleans, etc. (and loose purity)

Evaluation Rules - Call by Value - E-AppLam

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

Meaning:

- If a lambda (function) is applied to a value, substitute that value for it's parameter.
- "substitute" : replace it for every occurence in the lambda's body

Interpreter - Evaluation

```haskell
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)
```

-- $\dfrac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \quad (E - App1)$

```haskell
 =
  let e1' = eval e1
```

-- $\dfrac{e_2 \to e_2'}{v_1\, e_2 \to v_1\, e_2'} \quad (E - App2)$

```haskell
   in let e2' = eval e2
       in case e1'
               of
```

-- $(\lambda x.e)v \to [x/v]e \quad (E - AppLam)$

```haskell
            (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

```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

Structure

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

Progress and Preservation

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.

Evaluation

Dynamic rules stay the same!

- Type checking is done upfront

Interpreter

Typing Rules

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \text{T-Var}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \rightarrow \tau_2} \qquad \text{T-Lam}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \qquad \text{T-App}$$

$$\Gamma \vdash n : \mathsf{Int} \qquad \text{T-Int}$$

$$\Gamma \vdash \mathsf{True} : \mathsf{Bool} \qquad \text{T-True}$$

Type Checker

```haskell
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

```haskell
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
--
-- Γ ⊢ n : Int   (T-Int)
--
check _ (IntValue _) = Right TInt
--
-- Γ ⊢ True : Bool   (T-True)
--
check _ (BoolValue True) = Right TBool
--
-- Γ ⊢ False : Bool   (T-False)
--
check _ (BoolValue False) = Right TBool
--
-- x:σ ∈ Γ / Γ ⊢ x:σ   (T-Var)
--
check env (Var name) = find env name
```

Type Checker - Lambda & Application

```haskell
--
-- (Γ,x:τ1 ⊢ e:τ2) / (Γ ⊢ λx:τ1 . e:τ1 → τ2)   (T-Lam)
--
check env (Lambda name atype e) = do
  t <- check (Map.insert name atype env) e
  return $ TArr atype t
--
-- (Γ ⊢ e1 : τ1 → τ2   Γ ⊢ e2 : τ1) / (Γ ⊢ e1 e2 : τ2)   (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)
```

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2} \quad \textit{(T-App)}$$