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

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

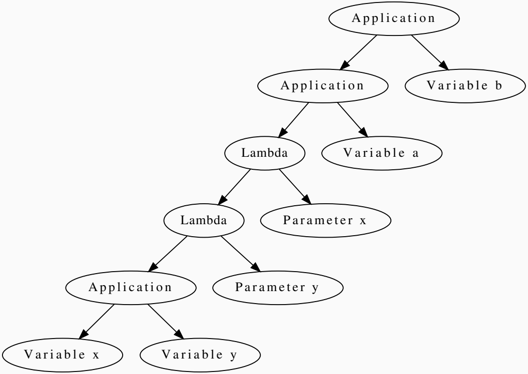
Let's build an interpreter

- 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:
$\quad x$	Variable
$\quad \lambda x.e$	Abstraction
$\quad e\ e$	Application

Abstract Syntax Tree



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

Interpreter - Syntax

```haskell
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 ≡ 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

Some rules...

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

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

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

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)
-- $\frac{e_1 \rightarrow e_1'}{e_1\, e_2 \rightarrow e_1'\, e_2}$  $(E - App1)$
 =
  let e1' = eval e1
-- $\frac{e_2 \rightarrow e_2'}{v_1\, e_2 \rightarrow v_1\, e_2'}$  $(E - App2)$
   in let e2' = eval e2
       in case e1'
                of
-- $(\lambda x.e)v \longrightarrow [x/v]e$  $(E - AppLam)$
            (Lambda name e1'_body) -> eval $ substitute name e2' e1'_body
            e1' -> App e1' e2'
```

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

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

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

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

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

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

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

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

```
--
-- 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)
```

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

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