

Simply Typed Lambda Calculus

From Untyped to Simply Typed Lambda Calculus

Sven Tennie

October 27, 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:

- 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:

- Booleans
- Numerals
- Data Structures
- Control Flow
- ...

Turing Completeness

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

Example - $(\lambda p. \lambda q. p) a b$

$(\lambda p. \lambda q. p) a b$

Meaning

$\lambda p. \lambda q. p$ Is a function that returns a function $(\lambda q. p)$

a, b Some variables (defined somewhere else)

p Is a variable that is bound to the parameter with the same name

Example - $(\lambda p. \lambda q. p) a b$

$(\lambda p. \lambda q. p) a b$ Substitute $p \mapsto a$

Meaning

$\lambda p. \lambda q. p$ Is a function that returns a function $(\lambda q. p)$

a, b Some variables (defined somewhere else)

p Is a variable that is bound to the parameter with the same name

Example - $(\lambda p. \lambda q. p) a b$

$(\lambda p. \lambda q. p) a b$ Substitute $p \mapsto a$

Meaning

$\lambda p. \lambda q. p$ Is a function that returns a function $(\lambda q. p)$

a, b Some variables (defined somewhere else)

p Is a variable that is bound to the parameter with the same name

Example - $(\lambda p. \lambda q. p) a b$

$(\lambda p. \lambda q. p) a b$ Substitute $p \mapsto a$
 $(\lambda q. a) b$

Meaning

$\lambda p. \lambda q. p$ Is a function that returns a function $(\lambda q. p)$

a, b Some variables (defined somewhere else)

p Is a variable that is bound to the parameter with the same name

Example - $(\lambda p. \lambda q. p) a b$

$(\lambda p.$	$\lambda q. p$	$)$	a	b	Substitute $p \mapsto a$
$($	$\lambda q. a$	$)$		b	Substitute $q \mapsto b$

Meaning

$\lambda p. \lambda q. p$ Is a function that returns a function $(\lambda q. p)$

a, b Some variables (defined somewhere else)

p Is a variable that is bound to the parameter with the same name

Example - $(\lambda p. \lambda q. p) a b$

$(\lambda p.$	$\lambda q. p$	$)$	a	b	Substitute $p \mapsto a$
$($	$\lambda q. a$	$)$		b	Substitute $q \mapsto b$

Meaning

$\lambda p. \lambda q. p$ Is a function that returns a function $(\lambda q. p)$

a, b Some variables (defined somewhere else)

p Is a variable that is bound to the parameter with the same name

Example - $(\lambda p. \lambda q. p) a b$

$(\lambda p.$	$\lambda q. p$)	a	b	Substitute $p \mapsto a$
($\lambda q. a$)		b	Substitute $q \mapsto b$
(a)			

Meaning

$\lambda p. \lambda q. p$ Is a function that returns a function $(\lambda q. p)$

a, b Some variables (defined somewhere else)

p Is a variable that is bound to the parameter with the same name

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?
- We'll do *Math Driven Development*
 - Look at the concepts in math first, then translate them to Haskell

$e ::=$

x

$\lambda x.e$

$e\ e$

Expressions:

Variable

Abstraction

Application

$\lambda x.e$ Function Definition

$e\ e$ Function Application

Abstract Syntax Tree

$(\lambda x.x) y$

dot/untyped_lambda_calculus_ast.
dotnix

Meaning:

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

Interpreter - Syntax

```
module UntypedSyntax where
```

```
type Name = String
```

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

Interpreter - Syntax - Examples

```
module UntypedSyntaxExamples where

import UntypedSyntax

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

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

-- and  $\equiv \lambda p. \lambda q. p \ q \ p$ 
and :: Expr
and = Lambda "p" $ Lambda "q" $ App (App (Var "p") (Var "q")) (Var "p")
```

Natural Deduction

$$\frac{}{Axiom} \quad (A1)$$
$$\frac{Antecedent}{Conclusion} \quad (A2)$$

Meaning:

Axiom Rule without Precondition

Antecedent Precondition - if it's fulfilled this rule applies.

Conclusion What follows from this rule.

A1, A2 Names for the rules

Proof: 2 is a Natural Number

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

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

Meaning:

A1 0 is a natural number (by definition)

A2 The successor of a natural number is a natural number

Proof: 2 is a Natural Number

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

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

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

Meaning:

A1 0 is a natural number (by definition)

A2 The successor of a natural number is a natural number

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

Evaluation Rules

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

Meaning:

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

Evaluation Rules - Call by Value - E-App2

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

Meaning:

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

$$(\lambda x.e)v \rightarrow [x/v]e$$

E-AppLam

Meaning:

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

Interpreter - Evaluation

```
module UntypedEval where

import UntypedSyntax

eval :: Expr -> Expr
-- No rule for variables
eval variable@(Var _) = variable
-- No rule for lambdas
eval lambda@(Lambda _ _) = lambda
```

Interpreter - Evaluation

```
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 :: Name -> Expr -> Expr -> Expr
--
-- If the Name matches: Substitute this Var by it's substitution
-- Otherwise: Leave it as is
--
substitute name substitution var@(Var varName)
  | name == varName = substitution
  | otherwise = var
--
-- Recursively substitute in both parts of Applications
--
substitute name substitution (App term1 term2) =
  App (substitute name substitution term1) (substitute name substitution term2)
```

Interpreter - Substitution

```
--  
-- Only substitute in Lambda's body, if the parameter doesn't  
-- redefine the Name in it's scope  
--  
substitute name substitution (Lambda varName term) =  
  if name == varName  
  then Lambda varName term  
  else Lambda varName (substitute name substitution term)
```

Tests

```
module UntypedEvalExamplesSpec where

import NaiveUntypedEval
import Prelude hiding (and)
import Test.Hspec
import UntypedSyntax
import UntypedSyntaxExamples

main :: IO ()
main = hspec spec

spec :: Spec
spec =
  describe "eval" $
    it "should evaluate these terms" $ do
      --
      --  $a \rightarrow a$ 
      --
      eval (Var "a") `shouldBe` Var "a"
```

Tests

```
--  
-- true  $\equiv \lambda p.\lambda q.p$   
--  
-- true  $a\ b \rightarrow a$   
--  
eval (App (App true (Var "a")) (Var "b")) `shouldBe` Var "a"
```

```
--  
-- false  $\equiv \lambda p.\lambda q.q$   
--  
-- and  $\equiv \lambda p.\lambda q.p\ q\ p$   
--  
-- and true false  $\rightarrow false$   
--  
eval (App (App and true) false) `shouldBe`  
  Lambda "p" (Lambda "q" (Var "q"))
```

Simply Typed Lambda Calculus

$e ::=$

x

$\lambda x : \tau . e$

$e e$

Expressions:

Variable

Abstraction

Application

τ Type of the parameter x

- 'Bool', 'Int', ...

What's a Type?

A Type is a set of values that an expression may return:

Bool True, False

Int $[-2^{29}..2^{29} - 1]$ (in Haskell, 'Data.Int')

Simple types don't have parameters, no polymorphism:

Bool, **Int** have no parameters \rightarrow simple types

Maybe a takes a type parameter (*a*) \rightarrow not a simple type

a \rightarrow a is polymorphic \rightarrow not a simple type

Type Safety = Progress + 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 .

- $e \equiv (\lambda x : \text{Int}.x)1$ and $e' \equiv 1$ have both the same type: 'Int'

Evaluation rules stay the same!

- Type checking is done upfront

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

T-Var

Meaning

Γ The Typing Environment, a list of (*Variable* : *Type*) pairs (associations)

$x : \sigma \in \Gamma$ If (x, σ) is in the Typing Environment

$\Gamma \vdash x : \sigma$ x has type σ

Typing Rules - Constants

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

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

$\Gamma \vdash \text{False} : \text{Bool}$ T-False

Meaning

True, False literals / constants are of type `Bool`

n number literals / constants are of `Int`

Why do we need Γ here?

- We handle Type Constructors like variables
- Think: $\Gamma \equiv \emptyset, \text{True} : \text{Bool}, \text{False} : \text{Bool}, 0 : \text{Int}, 1 : \text{Int}, \dots$

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

Meaning

Condition With $x : \tau_1$ in the Typing Environment, e has type τ_2

Conclusion $\lambda x : \tau_1. e$ has type $\tau_1 \rightarrow \tau_2$

Because e has type τ_2 if x has type τ_1

Typing Rules - Applications

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

T-App

Meaning

Condition If e_1 is a function of type $\tau_1 \rightarrow \tau_2$ and e_2 has type τ_1

Conclusion Then the type of $e_1 e_2$ (function application) is τ_2

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

T-App

Meaning

Condition If e_1 is a function of type $\tau_1 \rightarrow \tau_2$ and e_2 has type τ_1

Conclusion Then the type of $e_1 e_2$ (function application) is τ_2

```
id' :: Int -> Int
```

```
id' i = i
```

```
1 :: Int
```

```
(id' 1) :: Int
```

Type Checker - Expressions

```
module TypedSyntax where
```

```
import qualified Data.Map.Strict as Map
```

```
type Name = String
```

```
type Error = String
```

```
data Expr
```

```
  = IntValue Int
```

```
  | BoolValue Bool
```

```
  | Var Name
```

```
  | App Expr
```

```
      Expr
```

```
  | Lambda Name
```

```
      Type
```

```
      Expr
```

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

```
-- e ::=
```

```
--       $[-2^{29}..2^{29} - 1]$ 
```

```
--      True | False
```

```
--      x
```

```
--      e e
```

```
--       $\lambda x: \tau. e$ 
```

Expressions :

Integer Literal

Boolean Literal

Variable

Application

Abstraction

Type Checker - Types

```
type Environment = Map.Map Name Type
```

```
data Type          --  $\tau ::=$           Types :  
  = TInt           --      Int          Integer  
  | TBool          --      Bool         Boolean  
  | TArr Type      --       $\tau_1 \rightarrow \tau_2$   Abstraction / Function  
    Type  
  deriving (Eq, Show)
```

Type Checker - Literals

```
module TypedCheck where

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

import TypedSyntax

check :: Environment -> Expr -> Either Error 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
```

Type Checker - Lambda Abstraction

```
--  
-- 
$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \lambda x:\tau_1. e:\tau_1 \rightarrow \tau_2} \quad (T\text{-}Lam)$$
  
--  
check env (Lambda name atype e) = do  
  t <- check (Map.insert name atype env) e  
  return $ TArr atype t
```

Type Checker - Application

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

```
check env (App e1 e2) = do  
  t1 <- check env e1  
  case t1 of  
    (TArr ta1 ta2) -> do  
      t2 <- check env e2  
      if ta1 == t2  
        then Right ta2  
        else Left $ "Expected " ++ (show ta1) ++ " but got : " ++ (show t2)  
    _ -> Left $ "Expected TArr but got : " ++ (show t1)
```

Type Checker - Variables

```
--  
--  $\frac{x:\sigma \in \Gamma}{\Gamma \vdash x:\sigma} \quad (T\text{-}Var)$   
--  
check env (Var name) = find env name  
  
find :: Environment -> Name -> Either Name Type  
find env name = maybeToEither "Var not found!" (Map.lookup name env)
```

Tests

```
module TypedCheckExamplesSpec where

import Test.Hspec
import TypedCheck
import TypedSyntax

import qualified Data.Map.Strict as Map

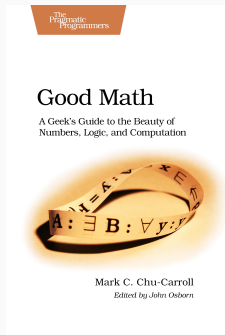
main :: IO ()
main = hspec spec

spec :: Spec
spec = do
  describe "check" $
    it "should type check these terms" $
      --
      -- (λx: Int.x)42 :: Int
      --
      do
        check Map.empty (App (Lambda "x" TInt (Var "x")) (IntValue 5)) `shouldBe`
          Right TInt
```

End

Thanks

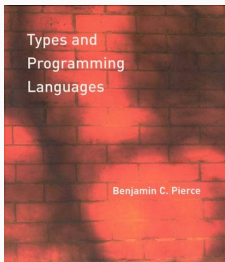
- Hope you enjoyed this talk and learned something new.
- Hope it wasn't too much math and dusty formulas ... :)



*A Geek's Guide to the Beauty of
Numbers, Logic, and Computation*

- Easy to understand

Types and Programming Languages



- Types systems explained by building interpreters / checkers and proving properties
- Very "mathematical", but very complete and self-contained

Write you a Haskell



*Building a modern functional compiler
from first principles.*

- Starts with the Lambda Calculus and goes all the way down to a full Haskell compiler
- Available for free - Not finished, yet