# Simply Typed Lambda Calculus

## From Untyped to Simply Typed Lambda Calculus

Sven Tennie

November 15, 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 - $(\lambda p.\lambda q.p)\ a\ b$

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

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

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

$(\lambda p.\quad \lambda q.p\quad )\quad a\quad b\quad$ 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 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?
- We'll do *Math Driven Development*
  - Look at the concepts in math first, then translate them to Haskell

# Structure

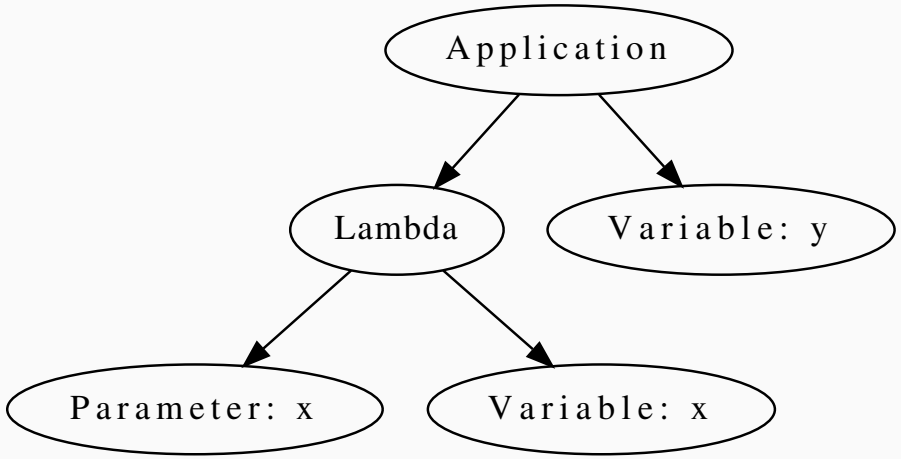
| | | |
|---|---|---|
| $e ::=$ | | Expressions: |
| | $x$ | Variable |
| | $\lambda x.e$ | Abstraction |
| | $e\ e$ | Application |

$\lambda x.e$ Function Definition

$e\ e$ Function Application (Function Call)

# Abstract Syntax Tree

$(\lambda x.x)\ y$




**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                 --      λx.e    Abstraction
           Expr
  | App Expr                    --      e e     Application
        Expr
  deriving (Eq, Show)
```

# Interpreter - Syntax - Examples

# Interpreter - Syntax - Examples

```haskell
module UntypedSyntaxExamples where

import UntypedSyntax

-- id ≡ λx.x
id :: Expr
id = Lambda "x" $ Var "x"
```

# Interpreter - Syntax - Examples

```haskell
module UntypedSyntaxExamples where

import UntypedSyntax

-- id ≡ λx.x
id :: Expr
id = Lambda "x" $ Var "x"
```

```haskell
-- true ≡ λp.λq.p
true :: Expr
true = Lambda "p" (Lambda "q" (Var "p"))

-- false ≡ λp.λq.q
false :: Expr
false = Lambda "p" (Lambda "q" (Var "q"))
```

# Interpreter - Syntax - Examples

```
-- and ≡ λp.λq.p q p
and :: Expr
and = Lambda "p" $ Lambda "q" $ App (App (Var "p") (Var "q")) (Var "p")
```

# Natural Deduction

# Notation

$$\frac{}{Axiom} \quad \text{(A1)}$$

$$\frac{Antecedent}{Conclusion} \quad \text{(A2)}$$

**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

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

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

**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

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

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

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

**Meaning:**

- **A1** $0$ is a natural number (by definition)
- **A2** The successor of a natural number is a natural number

$\rightarrow$ Thus the sucessor of the successor of $0$ $(2)$ must be a natural number

# Evaluation Rules

# Evaluation Rules - Call by Value - E-App1

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

**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.
- "Bare" Untyped Lambda Calculus:
  - Only Lambdas (functions) are values.
  - But you can add Ints, Booleans, etc. ("Enriched Untyped Lambda Calculus")

# 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 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)
--
-- e1 → e1'
-- ------------  (E − App1)
-- e1 e2 → e1' e2
--
 =
  let e1' = eval e1
--
-- e2 → e2'
-- ------------  (E − App2)
-- v1 e2 → v1 e2'
--
   in let e2' = eval e2
       in case e1'
                  of
--
-- (λx.e)v → [x/v]e   (E − AppLam)
--
            (Lambda name e1'_body) -> eval $ substitute name e2' e1'_body
            e1' -> App e1' e2'
```

# Interpreter - Substitution

```haskell
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

```haskell
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 → a
--
      eval (Var "a") `shouldBe` Var "a"
```

# Tests

```
--
-- true ≡ λp.λq.p
--
-- true a b → a
--
      eval (App (App true (Var "a")) (Var "b")) `shouldBe` Var "a"
```

```
--
-- false ≡ λp.λq.q
--
-- and ≡ λp.λq.p q p
--
-- and true false → false"
--
      eval (App (App and true) false) `shouldBe`
        Lambda "p" (Lambda "q" (Var "q"))
```

# Simply Typed Lambda Calculus

# Structure

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

$\tau$ 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 -> 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 $\tau$, and is evaluated to $e'$, then $e'$ has type $\tau$.**

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

# Evaluation

**Evalution rules stay the same!**

- Type checking is done upfront

# Typing Rules - Variables

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

**Meaning**

$\Gamma$ The Typing Environment, a list of $(\mathit{Variable} : \mathit{Type})$ pairs (associations)

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

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

# Typing Rules - Constants

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

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

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

**Meaning**

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

*n* number literals / constants are of `Int`

**Why do we need $\Gamma$ here?**

- We handle Type Constructors like variables
- Think: $\Gamma \equiv \emptyset$, *True* : *Bool*, *False* : *Bool*, $0$ : *Int*, $1$ : *Int*, ...

# Typing Rules - Lambdas

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

**Meaning**

**Condition** With $x : \tau_1$ in the Typing Environment, $e$ has type $\tau_2$

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

Because $e$ has type $\tau_2$ if $x$ has type $\tau_1$

# Typing Rules - Applications

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

**Meaning**

**Condition** If $e_1$ is a function of type $\tau_1 \to \tau_2$ and $e_2$ has type $\tau_2$

**Conclusion** Then the type of $e_1\, e_2$ (function application) is $\tau_2$

# Typing Rules - Applications

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

T-App

## Meaning

**Condition** If $e_1$ is a function of type $\tau_1 \to \tau_2$ and $e_2$ has type $\tau_2$

**Conclusion** Then the type of $e_1\, e_2$ (function application) is $\tau_2$

```
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                       -- e ::=                    Expressions :
  = IntValue Int                --      [−2^29..2^29 − 1]       Integer Literal
  | BoolValue Bool              --      True | False            Boolean Literal
  | Var Name                    --      x                       Variable
  | App Expr                    --      e e                     Application
        Expr
  | Lambda Name                 --      λx: τ.e                 Abstraction
           Type
           Expr
  deriving (Eq, Show)
```

# Type Checker - Types

```
type Environment = Map.Map Name Type

data Type                       -- τ ::=          Types :
  = TInt                        --     Int         Integer
  | TBool                       --     Bool        Boolean
  | TArr Type                   --     τ1 → τ2     Abstraction / Function
         Type
  deriving (Eq, Show)
```

# Type Checker - Literals

```haskell
module TypedCheck where

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

import TypedSyntax

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

# Type Checker - Lambda Abstraction

```
--
-- 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
```

# Type Checker - Application

```
--
-- Γ⊢e1:τ1→τ2   Γ⊢e2:τ1
--  ------------------   (T-App)
--      Γ⊢e1 e2:τ2
--
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

```
--
-- x:σ∈Γ / Γ⊢x:σ   (T-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

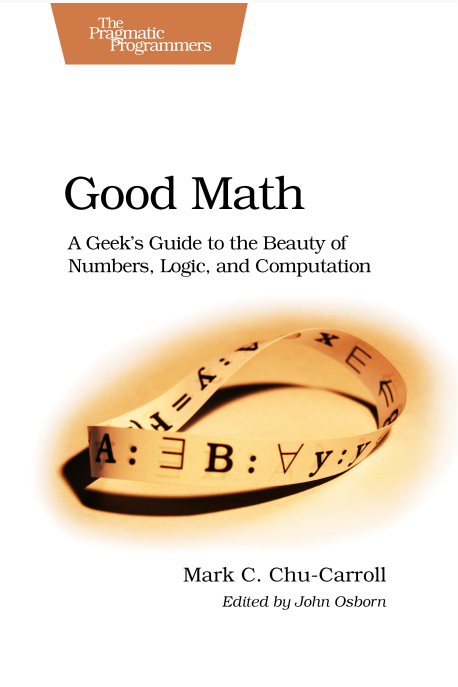
```
module TypedCheckExamplesSpec where

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 ... :)

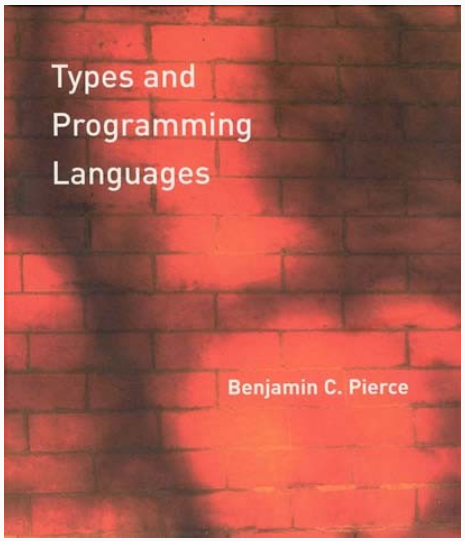
# Good Math




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

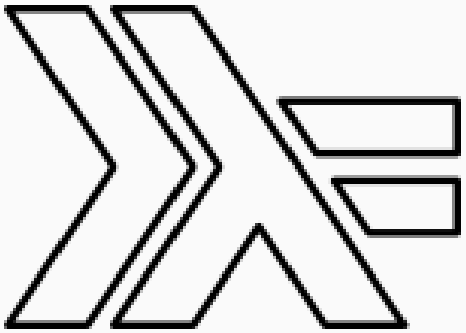
- Easy to understand

# 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