

# Simply Typed Lambda Calculus

From Untyped to Simply Typed Lambda Calculus

---

Sven Tennie

November 25, 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

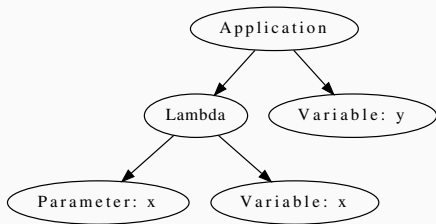
Application

$\lambda x.e$  Function Definition

$e\ e$  Function Application (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 Expr                    --       $\lambda x.e$  Abstraction  
  | App Expr Expr                       --      e e     Application  
  deriving (Eq, Show)
```

---

# Interpreter - Syntax - Examples

---

```
module UntypedSyntaxExamples where
```

```
import UntypedSyntax
```

```
--  $id \equiv \lambda x.x$ 
```

```
id :: Expr
```

```
id = Lambda "x" $ Var "x"
```

---

# Interpreter - Syntax - Examples

---

```
module UntypedSyntaxExamples where
```

```
import UntypedSyntax
```

```
--  $id \equiv \lambda x.x$ 
```

```
id :: Expr
```

```
id = Lambda "x" $ Var "x"
```

---

---

```
--  $true \equiv \lambda p.\lambda q.p$ 
```

```
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"))
```

---

# Interpreter - Syntax - Examples

---

-- *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.

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}$$

**Example**

$$\overbrace{((\lambda x.x) (\lambda y.y))}^{e_1} e_2$$

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}$$

**Example**

$$\overbrace{((\lambda x.x) (\lambda y.y))}^{e_1} e_2$$

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}$$

**Example**

$$\begin{array}{l} \overbrace{((\lambda x.x) (\lambda y.y))}^{e_1} e_2 \\ \rightarrow (\lambda y.y) z \end{array}$$

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

### 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")

## Evaluation Rules - E-App2 - Example

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

### Example

$$\overbrace{(\lambda x.x)}^{v_1} \overbrace{((\lambda y.y) 42)}^{e_2}$$

## Evaluation Rules - E-App2 - Example

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

### Example

$$\overbrace{(\lambda x.x)}^{v_1} \overbrace{((\lambda y.y) 42)}^{e_2}$$



## Evaluation Rules - E-App2 - Example

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

### Example

$$\begin{array}{c} \overbrace{(\lambda x.x)}^{v_1} \quad \overbrace{((\lambda y.y) \ 42)}^{e_2} \\ \rightarrow (\lambda x.x) \ 42 \end{array}$$

## Evaluation Rules - E-App2 - Example

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

### Example

$$\begin{array}{c} \overbrace{(\lambda x.x)}^{v_1} \quad \overbrace{((\lambda y.y) 42)}^{e_2} \\ \rightarrow (\lambda x.x) 42 \end{array}$$

### Note

- We evaluate the parameter before applying the function:  
Eager Evaluation!

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

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

## Evaluation Rules - E-AppLam -Example

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

E-AppLam

### Example

$$\overbrace{(\lambda x.\lambda y.x)}^{\lambda x.e} \overbrace{z}^v$$

# Evaluation Rules - E-AppLam -Example

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

E-AppLam

## Example

$$\overbrace{(\lambda x.\lambda y.x)}^{\lambda x.e} \overbrace{z}^v$$

# Evaluation Rules - E-AppLam -Example

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

E-AppLam

## Example

$$\begin{array}{c} \overbrace{(\lambda x.\lambda y.x)}^{\lambda x.e} \quad \overbrace{z}^v \\ \rightarrow \lambda y.z \end{array}$$

# 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 x e1'_body) -> eval $ substitute x 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@(Lambda varName term) =  
    if name == varName  
    then lambda  
    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"
```

---

# 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

$\tau$  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  $\rightarrow$  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.

- A well typed (typeable) program never gets "stuck".

# 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.

- A well typed (typeable) program never gets "stuck".

**Preservation** : If an expression  $e$  has type  $\tau$ , and is evaluated to  $e'$ , then  $e'$  has type  $\tau$ .

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

# Not all meaningful Programs can be type checked

---

```
id :: a -> a  
id a = a
```

---

- It strongly depends on the type system if this is allowed or not.
- In Simply Typed Lambda Calculus it's not!
  - No polymorphic types ...

## **Evaluation rules stay the same!**

- Type checking is done upfront

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

T-Var

### Meaning

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

- Think of a map:  $\text{Variable} \mapsto \text{Type}$

**Condition** If  $(x, \tau)$  is in the Typing Environment

**Conclusion**  $x$  has type  $\tau$

## Typing Rules - Variables - Example

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \text{T-Var}$$

### Example

$$\underbrace{\lambda x : Int .}_{\Gamma' = \Gamma, x : Int} \underbrace{\lambda y : Bool .}_{\Gamma'' = \Gamma', y : Bool} \underbrace{x}_{\Gamma'' \vdash x : Int}$$

$\lambda x : Int$  Add  $x : Int$  to the Typing Environment ( $\Gamma$ )

$x$  We know from the Typing Environment ( $\Gamma''$ ) that  $x$  has type  $Int$

## 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 $\Gamma$ here?

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

## Typing Rules - Constants - Example

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

### Example

$$\Gamma \equiv \emptyset, \text{True} : \text{Bool}, \text{False} : \text{Bool}, 0 : \text{Int}, 1 : \text{Int}, \dots$$
$$\text{True}$$
$$1$$



## Typing Rules - Constants - Example

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

### Example

$$\Gamma \equiv \emptyset, \text{True} : \text{Bool}, \text{False} : \text{Bool}, 0 : \text{Int}, 1 : \text{Int}, \dots$$

*True*

1

## Typing Rules - Constants - Example

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

### Example

$$\Gamma \equiv \emptyset, \text{True} : \text{Bool}, \text{False} : \text{Bool}, 0 : \text{Int}, 1 : \text{Int}, \dots$$

*True*

*1*

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

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  $\tau_1 \rightarrow \tau_2$

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

## Typing Rules - Lambdas - Example

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

T-Lam

### Example

$$\lambda x : \overbrace{Int}^{\tau_1} . \overbrace{x}^e \quad :?$$

# Typing Rules - Lambdas - Example

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

T-Lam

## Example

$$\lambda x : \overbrace{Int}^{\tau_1} . \overbrace{x}^e$$

∴?

$$\frac{\Gamma, x : \overbrace{Int}^{\tau_1} \vdash e : \overbrace{Int}^{\tau_2}}{\Gamma \vdash \lambda x : \underbrace{Int}_{\tau_1} . \underbrace{e : Int \rightarrow Int}_{\tau_1 \rightarrow \tau_2}}$$

# Typing Rules - Lambdas - Example

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

T-Lam

## Example

$$\lambda x : \overbrace{Int}^{\tau_1} . \overbrace{x}^e$$

$$: \overbrace{Int \rightarrow Int}^{\tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma, x : \overbrace{Int}^{\tau_1} \vdash e : \overbrace{Int}^{\tau_2}}{\Gamma \vdash \lambda x : \underbrace{Int}_{\tau_1} . \underbrace{e : Int \rightarrow Int}_{\tau_1 \rightarrow \tau_2}}$$

## 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  $\tau_1$

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

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

T-App

## Meaning

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

**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
```

---



# Typing Rules - Applications - Example

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

T-App

## Example

$$\overbrace{(\lambda x : Int. True)}^{e_1} \overbrace{42}^{e_2}$$

:?

# Typing Rules - Applications - Example

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

T-App

## Example

$$\overbrace{(\lambda x : \text{Int}. \text{True})}^{e_1} \overbrace{42}^{e_2} \quad :?$$

$$\frac{\Gamma \vdash \overbrace{(\lambda x : \text{Int}. \text{True})}^{e_1} : \overbrace{\text{Int} \rightarrow \text{Bool}}^{\tau_1 \rightarrow \tau_2} \quad \Gamma \vdash \overbrace{42}^{e_2} : \overbrace{\text{Int}}^{\tau_1}}{\Gamma \vdash \underbrace{\overbrace{(\lambda x : \text{Int}. \text{True})}^{e_1}}_{e_1} \underbrace{\overbrace{42}^{e_2}}_{e_2} : \underbrace{\text{Bool}}_{\tau_2}}$$

# Typing Rules - Applications - Example

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

T-App

## Example

$$\overbrace{(\lambda x : \text{Int}. \text{True})}^{e_1} \overbrace{42}^{e_2} : \overbrace{\text{Bool}}^{\tau_2}$$

$$\frac{\Gamma \vdash \overbrace{(\lambda x : \text{Int}. \text{True})}^{e_1} : \overbrace{\text{Int} \rightarrow \text{Bool}}^{\tau_1 \rightarrow \tau_2} \quad \Gamma \vdash \overbrace{42}^{e_2} : \overbrace{\text{Int}}^{\tau_1}}{\Gamma \vdash \underbrace{\overbrace{(\lambda x : \text{Int}. \text{True})}^{e_1}}_{e_1} \underbrace{\overbrace{42}^{e_2}}_{e_2} : \underbrace{\text{Bool}}_{\tau_2}}$$

# 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                         --       $\lambda x: \tau. e$             Abstraction
      Type
      Expr
deriving (Eq, Show)
```

---

# 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 x t1 e) = do  
  t2 <- check (Map.insert x t1 env) e  
  return $ TArr t1 t2
```

---

# 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  
  te1 <- check env e1  
  case te1 of  
    (TArr t1 t2) -> do  
      te2 <- check env e2  
      if t1 == te2  
        then Right t2  
        else Left $ "Expected " ++ (show t1) ++ " but got : " ++ (show te2)  
    _ -> Left $ "Expected TArr but got : " ++ (show te1)
```

---



# Type Checker - Variables

---

```
--  
--  $\frac{x:\tau \in \Gamma}{\Gamma \vdash x:\tau}$  (T-Var)  
--  
check env (Var x) = find env x  
  
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
```

---

# Tests

---

```
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
```

---

# Tests

---

```
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
        --
        -- Does not type check: (λx: Bool.x) 42
        --
        check Map.empty (App (Lambda "x" TBool (Var "x")) (IntValue 5))
          `shouldBe` Left "Expected TBool but got : TInt"
```

---

---

--

-- *Does not type check: 42 False*

--

```
check Map.empty (App (IntValue 42) (BoolValue False)) `shouldBe`  
  Left "Expected TArr but got : TInt"
```

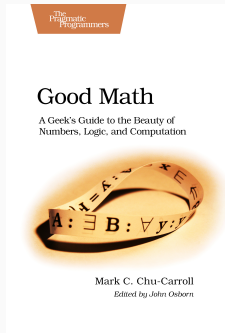
---

**End**



# Thanks

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

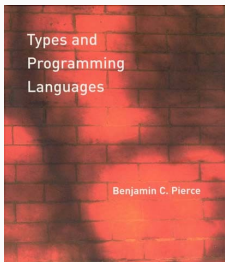


## *A Geek's Guide to the Beauty of Numbers, Logic, and Computation*

- 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