

## 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

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!

1

### 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

2

### 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

3

### Structure

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

$\lambda x. e$  Function Definition

$e\ e$  Function Application (Function Call)

4

### Abstract Syntax Tree

$(\lambda x. x)\ y$

#### Meaning

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

5

### Interpreter - Syntax

```
module UntypedSyntax where

type Name = String

data Expr
  = Var Name
  | Lambda Name Expr
  | App Expr Expr
  deriving (Eq, Show)

-- e ::=
-- x      Variable
-- λx.e   Abstraction
-- e e    Application
```

6

---

```
module UntypedSyntaxExamples where
```

```
import UntypedSyntax
```

```
-- id ≡ λx.x
```

```
id :: Expr
```

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

---

```
-- 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"))
```

---

7

---

```
-- and ≡ λp.λq.p q p
```

```
and :: Expr
```

```
and = Lambda "p" $ Lambda "q" $ App (App (Var "p") (Var "q")) (Var "p")
```

---

8

## Natural Deduction

---

## Notation

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

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

### 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

9

## Proof: 2 is a Natural Number

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

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

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

### Meaning

**A1** 0 is a natural number (by definition)

**A2** The successor of a natural number is a natural number

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

10

## Evaluation Rules

---

## Evaluation Rules - Call by Value - E-App1

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

### Meaning

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

11

## Evaluation Rules - E-App1 - Example

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

### Example

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

12

## 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")

13

## 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

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

### Note

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

14

## Evaluation Rules - Call by Value - E-AppLam

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

### Meaning

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

15

## Evaluation Rules - E-AppLam -Example

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

### Example

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

16

## 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
```

17

## 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'
```

18

## 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)
```

19

## 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)
```

20

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

21

## 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"))
```

22

## 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, ...

23

## 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 → simple types  
**Maybe a** takes a type parameter ( $a$ ) → not a simple type  
 $a \rightarrow a$  is polymorphic → not a simple type

24

## 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: Int

25

## 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 ...

26

## Evaluation

Evaluation rules stay the same!

- Type checking is done upfront

27

## Typing Rules - Variables

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

### Meaning

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

- Think of a map:  $Variable \mapsto Type$

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

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

28

## Typing Rules - Variables - Example

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

### Example

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

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

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

29

## Typing Rules - Constants

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

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

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

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

30

## Typing Rules - Constants - Example

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

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

### Example

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

*True*

*1*

31

## Typing Rules - Lambdas

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

### Meaning

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

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

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

32

## Typing Rules - Lambdas - Example

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

### Example

$$\lambda x : \underbrace{Int}_{\tau_1} . \underbrace{x}_{\tau_2} \quad \quad \quad \tau_1 \rightarrow \tau_2$$

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

33

## Typing Rules - Applications

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

### 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
```

34

## Typing Rules - Applications - Example

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

### Example

$$\underbrace{(\lambda x : Int. True)}_{e_1} \underbrace{42}_{e_2} \quad \quad \quad \tau_2$$

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

35

## Type Checker - Expressions

```
module TypedSyntax where

import qualified Data.Map.Strict as Map

type Name = String
type Error = String

data Expr      -- e ::=
  = IntValue Int      -- [-229..229 - 1]      Integer Literal
  | BoolValue Bool    -- True | False        Boolean Literal
  | Var Name          -- x                    Variable
  | App Expr Expr     -- e e                Application
  | Lambda Name       -- λx. τ.e             Abstraction
    Type
    Expr
  deriving (Eq, Show)
```

36

## Type Checker - Types

```
type Environment = Map.Map Name Type

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

37

## 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
--
-- 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
```

38

## Type Checker - Lambda Abstraction

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

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

39

## Type Checker - Application

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

--
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)
```

40

## Type Checker - Variables

```
--
-- 
$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x:\tau} \quad (T-Var)$$

--
check env (Var x) = find env x

find :: Environment -> Name -> Either Name Type
find env name = maybeToEither "Var not found!" (Map.lookup name env)
```

41

## Tests

```
module TypedCheckExamplesSpec where

import Test.Hspec
import TypedCheck
import TypedSyntax

import qualified Data.Map.Strict as Map

main :: IO ()
main = hspec spec
```

42

## 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"
```

43

## Tests

```
--  
-- Does not type check: 42 False  
--  
check Map.empty (App (IntValue 42) (BoolValue False)) `shouldBe`  
  Left "Expected TArr but got : TInt"
```

44

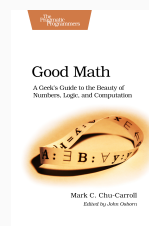
## End

## Thanks

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

45

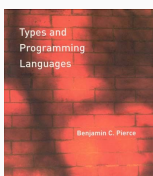
## Good Math



*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