

# The Typed Lambda Calculus

**COMP3610 – Principles of Programming Languages**

Ben Lippmeier

Australian National University

Semester 2, 2009

## Types

- With Church encoding, the representations of  $c_0$  and  $\text{false}$  are  $\alpha$ -equivalent (ie, differ only in the names of variables).

$$c_0 \stackrel{\text{def}}{=} \lambda s z. z$$

$$\text{false} \stackrel{\text{def}}{=} \lambda x y. y$$

- If a term reduces to  $(\lambda a b. b)$  how will we know whether it should be taken a Boolean, an Integer, or some other type of object?
- Notice how hard it is to define the word “type” without using that word, or a synonym in the definition – this hints at how fundamental the idea is.

## Primitive Values

- The value  $(\lambda a b. b)$  has an ambiguous interpretation. It might be a Bool, Integer, or general function.
- We can extend the  $\lambda$ -calculus with primitive values and types, so that the interpretation of Booleans and Integers is no longer ambiguous.

*true, false* :: Bool

0, 1, 2, 3... :: Int

- Now *false* cannot be mistaken for 0 or *vice-versa*.

## More Primitive Values

Unfortunately, as our primitive values are no longer represented by lambda-terms, we cannot use the Church versions of the functions ‘succ’, ‘plus’, ‘and’, ‘or’, ‘not’ ... etc

We must also add these functions to our new system as primitives.

$$\textit{succ} :: \text{Int} \rightarrow \text{Int}$$
$$\textit{plus} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$
$$\textit{isZero} :: \text{Int} \rightarrow \text{Bool}$$
$$\textit{not} :: \text{Bool} \rightarrow \text{Bool}$$

These type signatures become axioms in our type system.

# The Simply Typed Lambda Calculus

$x$   $\rightarrow$  (value variables)

$a$   $\rightarrow$  (type variables)

## Types

$\tau$  =  $a$

| Int | Bool | Char (base types)

|  $\tau_1 \rightarrow \tau_2$  (function type)

## Expressions

$e$  =  $x$

|  $\lambda(x : \tau). e$  (function abstraction)

|  $e_1 e_2$  (function application)

## Evaluation

$(\lambda(x : \tau). e_1) e_2 \longrightarrow e_1[x := e_2]$

## The Typing Relation

$$\Gamma \vdash e :: \tau$$

“With type environment  $\Gamma$ , the expression  $e$  has type  $\tau$ ”

The *type environment* is a set which contains the types of all variables which are free in the expression. ( $\Gamma =$  “gamma”,  $\tau =$  “tau”)

The type of the expression depends on the types we assign to these free variables, for example:

$$x : \text{Bool} \vdash x :: \text{Bool}$$

VS

$$x : \text{Int} \vdash x :: \text{Int}$$

## Application

When evaluating a function application, the function and argument must have appropriate types.

If they don't, then the evaluation gets *stuck* before we reach normal form.

$isZero :: Int \rightarrow Bool$

$isZero\ 0$	$\longrightarrow true$	OK
$isZero\ true$	$\longrightarrow ???$	STUCK
$isZero\ (succ\ 5)$	$\longrightarrow isZero\ 6 \longrightarrow false$	OK
$0\ isZero$	$\longrightarrow ???$	STUCK
$isZero\ isZero$	$\longrightarrow ???$	STUCK

## The Application Rule

$$\frac{\Gamma \vdash e_1 :: \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 :: \tau_{11}}{\Gamma \vdash e_1 e_2 :: \tau_{12}}$$

- The application rule expresses the typing constraints we wish to place on the two expressions  $e_1$  and  $e_2$ .
- As we're applying  $e_1$  to  $e_2$ , the expression  $e_1$  must evaluate to a function.
- We've named the argument type  $\tau_{11}$ . The function must accept an argument of this type.
- Applying the function will generate a new value as the result. We've named the type of this result  $\tau_{12}$ .

## Type Checking

We check the type of an expression by *proving* that it has some type.

The proofs are usually presented as Gentzen style *proof trees*, whose structure follows the syntax of the expression being checked.

$$\frac{\frac{\frac{\emptyset \vdash isZero :: Int \rightarrow Bool}{\emptyset \vdash isZero (succ\ 5) :: Bool} \quad \frac{\frac{\emptyset \vdash succ :: Int \rightarrow Int \quad \emptyset \vdash 5 :: Int}{\emptyset \vdash succ\ 5 :: Int} (App)}{\emptyset \vdash isZero (succ\ 5) :: Bool} (App)}{\emptyset \vdash isZero (succ\ 5) :: Bool} (App)$$

In this proof each statement is either an axiom, or has been constructed with the Application rule (App).

## Type Errors

If there is a *type error* in the expression then we will not be able to construct a valid proof tree.

The following proof is not valid because function and argument types of (succ true) do not match. The application rule does not apply.

$$\frac{\frac{\emptyset \vdash isZero :: Int \rightarrow Bool}{\emptyset \vdash isZero (succ\ true) :: Bool} \quad \frac{\frac{\emptyset \vdash succ :: Int \rightarrow Int \quad \emptyset \vdash true :: Bool}{\emptyset \vdash succ\ true :: Int} (App)}{\emptyset \vdash isZero (succ\ true) :: Bool} (App)$$

$$\frac{\Gamma \vdash e_1 :: \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 :: \tau_{11}}{\Gamma \vdash e_1\ e_2 :: \tau_{12}}$$

## Type Annotations

What is the type of the following expression?

$$\lambda x. 5$$

Without knowing the type of the argument (and without polymorphism), there is no way to tell. It could be `Int → Int`, `Bool → Int` or some other type.

We will add a *type annotation* to the parameter to make its type explicit.

$$\lambda (x : \text{Bool}). 5$$

## The Abstraction Rule

$$\frac{\Gamma, x : \tau_1 \vdash e_2 :: \tau_2}{\Gamma \vdash \lambda (x : \tau_1). e_2 :: \tau_1 \rightarrow \tau_2}$$

- A  $\lambda$ -abstraction defines a function, so it has a function type.
- The annotation on the parameter gives the type of argument this function can be applied to.
- The parameter variable  $x$  is likely to be free in the function body  $e_2$ , so we must *extend* the type environment with its type.

## Example

$$\frac{\frac{\frac{x : \mathbf{Int} \vdash \mathit{isZero} :: \mathbf{Int} \rightarrow \mathbf{Bool}}{x : \mathbf{Int} \vdash \mathit{isZero} (succ\ x) :: \mathbf{Bool}}{\emptyset \vdash \lambda (x : \mathbf{Int}). \mathit{isZero} (succ\ x) :: \mathbf{Int} \rightarrow \mathbf{Bool}} \text{ (Abs)}}{\frac{\frac{x : \mathbf{Int} \vdash succ :: \mathbf{Int} \rightarrow \mathbf{Int}}{x : \mathbf{Int} \vdash succ\ x :: \mathbf{Int}} \text{ (App)}}{\frac{x : \mathbf{Int} \vdash succ :: \mathbf{Int} \rightarrow \mathbf{Int}}{x : \mathbf{Int} \vdash x :: \mathbf{Int}} \text{ (App)}} \text{ (Var)}}{x : \mathbf{Int} \vdash \mathit{isZero} (succ\ x) :: \mathbf{Bool}} \text{ (App)}$$

## Summary of rules so far

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x :: \tau} \quad (\text{Var})$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 :: \tau_2}{\Gamma \vdash \lambda (x : \tau_1). e_2 :: \tau_1 \rightarrow \tau_2} \quad (\text{Abs})$$

$$\frac{\Gamma \vdash e_1 :: \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 :: \tau_{11}}{\Gamma \vdash e_1 e_2 :: \tau_{12}} \quad (\text{App})$$

## The need for polymorphism

```
let  twiceInt      =  $\lambda f : \text{Int} \rightarrow \text{Int}. \lambda x : \text{Int}. f (f x)$   
in  twiceInt succ 2
```

```
let  twiceBool    =  $\lambda f : \text{Bool} \rightarrow \text{Bool}. \lambda x : \text{Bool}. f (f x)$   
in  twiceBool not true
```

Cutting and pasting code  $\equiv$  We're doing it wrong.  
 $\equiv$  Our language isn't expressive enough.

# The Polymorphic Lambda Calculus

$x$       $\rightarrow$  (value variables)

$a$       $\rightarrow$  (type variables)

## Types

$\sigma$       $::= \forall \bar{a}. \tau$                     (type schemes)

$\tau$       $::= a$

       | Int | Bool | Char            (base types)

       |  $\tau_1 \rightarrow \tau_2$             (function type)

## Expressions

$e$       $::= x$

       |  $\lambda x. e$                     (function abstraction)

       |  $e_1 e_2$                     (function application)

## I'm tired of writing type annotations

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x :: \tau} \quad \text{(Var)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 :: \tau_2}{\Gamma \vdash \lambda (x : \tau_1). e_2 :: \tau_1 \rightarrow \tau_2} \quad \Rightarrow \quad \frac{\Gamma, x : \tau_1 \vdash e_2 :: \tau_2}{\Gamma \vdash \lambda x. e_2 :: \tau_1 \rightarrow \tau_2} \quad \text{(Abs)}$$

$$\frac{\Gamma \vdash e_1 :: \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 :: \tau_{11}}{\Gamma \vdash e_1 e_2 :: \tau_{12}} \quad \text{(App)}$$

- Leaving annotations off bound variables makes life easier for *the programmer*, but now the type checker must “guess” what they are.

## Generalisation

$$\frac{\Gamma \vdash e :: \sigma \quad a \notin \text{fv}(\Gamma)}{\Gamma \vdash e :: \forall a. \sigma} \text{ (Gen)}$$

$$\frac{\frac{\frac{x : a \in x : a}{x : a \vdash x :: a} \text{ (Var)}}{\emptyset \vdash \lambda x. x :: a \rightarrow a} \text{ (Abs)} \quad a \notin \text{fv}(\Gamma)}{\emptyset \vdash \lambda x. x :: \forall a. a \rightarrow a} \text{ (Gen)}$$

## Instantiation

$$\frac{\Gamma \vdash e :: \forall a. \sigma}{\Gamma \vdash e :: \sigma[a := \tau]} \text{ (Inst)}$$

$$\frac{\frac{\frac{id : \forall a. a \rightarrow a \in \Gamma_{id}}{\Gamma_{id} \vdash id :: \forall a. a \rightarrow a} \text{ (Var)}}{\Gamma_{id} \vdash id :: \text{Bool} \rightarrow \text{Bool}} \text{ (Inst)} \quad \Gamma_{id} \vdash \text{True} :: \text{Bool}}{\{id : \forall a. a \rightarrow a\}_{\Gamma_{id}} \vdash id \text{ True} :: \text{Bool}} \text{ (App)}$$

## Sugar is sweeter

$$\frac{\Gamma \vdash e_1 :: \tau_1 \quad \Gamma, e : \tau_1 \vdash e_2 :: \tau_2}{\Gamma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 :: \tau_2} \text{ (Let)}$$

### Exercise:

Convince yourself that this rule flows naturally from the identity:

$$\mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \equiv (\lambda x. e_2) e_1$$

## Example

⋮

$$\frac{\frac{\frac{}{f : (a \rightarrow a), x : a \vdash f (f x) :: a}}{\emptyset \vdash \lambda f. \lambda x. f (f x) :: (a \rightarrow a) \rightarrow a \rightarrow a}}{\emptyset \vdash \lambda f. \lambda x. f (f x) :: \forall a. (a \rightarrow a) \rightarrow a \rightarrow a} \text{ (G)}}{\emptyset \vdash \mathbf{let} \text{ twice} = \lambda f. \lambda x. f (f x) \mathbf{in} \text{ twice succ } 2 :: I}$$

$$\frac{\frac{\frac{\Gamma_{succ} \vdash \text{ twice} :: \forall a. (a \rightarrow a) \rightarrow a \rightarrow a}{\Gamma_{succ} \vdash \text{ twice} :: (I \rightarrow I) \rightarrow I \rightarrow I} \text{ (I)}}{\Gamma_{succ} \vdash \text{ twice succ} :: I \rightarrow I \quad \dots} \quad \Gamma_{succ} \vdash 2 :: I}{\{\text{ twice} : \forall a. (a \rightarrow a) \rightarrow a \rightarrow a\}_{\Gamma_{succ}} \vdash \text{ twice succ } 2 :: I}$$

# The Curry-Howard Isomorphism

Simply typed lambda calculus  $\sim$  Propositional calculus

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

$$\frac{[p]}{p}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 :: \tau_2}{\Gamma \vdash \lambda x. e_2 :: \tau_1 \rightarrow \tau_2}$$

$$\frac{[p]}{p \rightarrow q}$$

$$\frac{\Gamma \vdash e_1 :: \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 :: \tau_{11}}{\Gamma \vdash e_1 e_2 :: \tau_{12}}$$

$$\frac{p \rightarrow q \quad p}{q}$$

# The Curry-Howard Isomorphism $\sim$ deep magic

Polymorphic lambda calculus  $\sim$  Predicate calculus

$$\frac{\Gamma \vdash e :: \sigma \quad a \notin \text{fv}(\Gamma)}{\Gamma \vdash e :: \forall a. \sigma} \quad \frac{P(z) \quad z \text{ arbitrary}}{\forall a. P(a)}$$

$$\frac{\Gamma \vdash e :: \forall a. \sigma}{\Gamma \vdash e :: \sigma[a := \tau]} \quad \frac{\forall a. P(a)}{P(z)}$$