

The Typed Lambda Calculus

COMP3610 – Principles of Programming Languages

Ben Lippmeier

Australian National University
Semester 2, 2009

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 λ -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*.

Types

- With Church encoding, the representations of c_0 and `false` are α -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.

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.

$succ :: Int \rightarrow Int$

$plus :: Int \rightarrow Int \rightarrow Int$

$isZero :: Int \rightarrow Bool$

$not :: Bool \rightarrow Bool$

These type signatures become axioms in our type system.

The Simply Typed Lambda Calculus

x → (value variables)

a → (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

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

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.

$isZero :: \text{Int} \rightarrow \text{Bool}$

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

The Typing Relation

$\Gamma \vdash e :: \tau$

“With type environment Γ , the expression e has type τ ”

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

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

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 τ_{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 τ_{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*, who's structure follows the syntax of the expression being checked.

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

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

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 $\text{Int} \rightarrow \text{Int}$, $\text{Bool} \rightarrow \text{Int}$ or some other type.

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

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

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.

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

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

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 λ -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

$$\begin{array}{c}
 \frac{x : \text{Int} \in \{x : \text{Int}\}}{x : \text{Int} \vdash x :: \text{Int}} \text{ (Var)} \\
 \frac{x : \text{Int} \vdash \text{succ} :: \text{Int} \rightarrow \text{Int} \quad x : \text{Int} \vdash x :: \text{Int}}{x : \text{Int} \vdash \text{succ } x :: \text{Int}} \text{ (App)} \\
 \frac{x : \text{Int} \vdash \text{isZero} :: \text{Int} \rightarrow \text{Bool} \quad x : \text{Int} \vdash \text{succ } x :: \text{Int}}{x : \text{Int} \vdash \text{isZero } (\text{succ } x) :: \text{Bool}} \\
 \frac{}{\emptyset \vdash \lambda (x : \text{Int}). \text{isZero } (\text{succ } x) :: \text{Int} \rightarrow \text{Bool}} \text{ (Abs)}
 \end{array}$$

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.

Summary of rules so far

$$\begin{array}{c}
 \frac{x : \tau \in \Gamma}{\Gamma \vdash x :: \tau} \text{ (Var)} \\
 \\
 \frac{\Gamma, x : \tau_1 \vdash e_2 :: \tau_2}{\Gamma \vdash \lambda (x : \tau_1). e_2 :: \tau_1 \rightarrow \tau_2} \text{ (Abs)} \\
 \\
 \frac{\Gamma \vdash e_1 :: \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 :: \tau_{11}}{\Gamma \vdash e_1 e_2 :: \tau_{12}} \text{ (App)}
 \end{array}$$

The Polymorphic Lambda Calculus

x \rightarrow (value variables)
 a \rightarrow (type variables)

Types

$\sigma ::= \forall \bar{a}. \tau$ (type schemes)
 $\tau ::= a$
 | $\text{Int} \mid \text{Bool} \mid \text{Char}$ (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

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

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

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

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

Instantiation

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

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

Generalisation

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

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

Sugar is sweeter

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

Exercise:

Convince yourself that this rule flows naturally from the identity:

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

Example

$$\begin{array}{c}
 \vdots \\
 \hline
 f : (a \rightarrow a), x : a \vdash f(fx) :: a \\
 \hline
 \emptyset \vdash \lambda f. \lambda x. f(fx) :: (a \rightarrow a) \rightarrow a \rightarrow a \\
 \hline
 \emptyset \vdash \lambda f. \lambda x. f(fx) :: \forall a. (a \rightarrow a) \rightarrow a \rightarrow a \quad (\text{G}) \\
 \hline
 \emptyset \vdash \mathbf{let\ twice = \lambda f. \lambda x. f(fx)\ in\ twice\ succ\ 2} :: I
 \end{array}
 \quad
 \begin{array}{c}
 \Gamma_{succ} \vdash \mathit{twice} :: \forall a. (a \rightarrow a) \rightarrow a \rightarrow a \quad (\text{I}) \\
 \hline
 \Gamma_{succ} \vdash \mathit{twice} :: (I \rightarrow I) \rightarrow I \rightarrow I \\
 \hline
 \Gamma_{succ} \vdash \mathit{twice\ succ} :: I \rightarrow I \quad \dots \quad \Gamma_{succ} \vdash 2 :: I \\
 \hline
 \{\mathit{twice} : \forall a. (a \rightarrow a) \rightarrow a \rightarrow a\} \Gamma_{succ} \vdash \mathit{twice\ succ\ 2} :: I
 \end{array}$$

The Curry-Howard Isomorphism ~ deep magic

Polymorphic lambda calculus ~ Predicate calculus

$$\begin{array}{c}
 \Gamma \vdash e :: \sigma \quad a \notin \text{fv}(\Gamma) \\
 \hline
 \Gamma \vdash e :: \forall a. \sigma
 \end{array}
 \quad
 \begin{array}{c}
 P(z) \quad z \text{ arbitrary} \\
 \hline
 \forall a. P(a)
 \end{array}$$

$$\begin{array}{c}
 \Gamma \vdash e :: \forall a. \sigma \\
 \hline
 \Gamma \vdash e :: \sigma[a := \tau]
 \end{array}
 \quad
 \begin{array}{c}
 \forall a. P(a) \\
 \hline
 P(z)
 \end{array}$$

The Curry-Howard Isomorphism

Simply typed lambda calculus ~ Propositional calculus

$$\begin{array}{c}
 \frac{x : \tau \in \Gamma \quad [p]}{\Gamma \vdash x :: \tau} \\
 \\
 \frac{\Gamma, x : \tau_1 \vdash e_2 :: \tau_2 \quad [p]}{\Gamma \vdash \lambda x. e_2 :: \tau_1 \rightarrow \tau_2} \quad \frac{q}{p \rightarrow q} \\
 \\
 \frac{\Gamma \vdash e_1 :: \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 :: \tau_{11}}{\Gamma \vdash e_1 e_2 :: \tau_{12}} \quad \frac{p \rightarrow q \quad p}{q}
 \end{array}$$