



# UNTYPED LAMBDA CALCULUS (II)

# RECALL: CALL-BY-VALUE O.S.

- Basic rule

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

- Search rules:

$$\frac{e1 \rightarrow e1'}{e1 e2 \rightarrow e1' e2}$$

$$\frac{e2 \rightarrow e2'}{v e2 \rightarrow v e2'}$$

Quiz: Write the rules for Right-to-Left call-by-value O.S.?

# CALL-BY-VALUE EVALUATION EXAMPLE

$(\lambda x. x x) (\lambda y. y)$

$\rightarrow x x [\lambda y. y / x]$

$= (\lambda y. y) (\lambda y. y)$

$\rightarrow y [\lambda y. y / y]$   Note  $y$  is free in the body of  $\lambda y. y$ , i.e.,  $y!$

$= \lambda y. y$

## ANOTHER EXAMPLE

$$(\lambda x. x x) (\lambda x. x x)$$
$$\rightarrow x x [\lambda x. x x/x]$$
$$= (\lambda x. x x) (\lambda x. x x)$$

- In other words, it is simple to write non-terminating computations in the lambda calculus
- what else can we do?

# WE CAN DO EVERYTHING

- The lambda calculus can be used as an “assembly language”
- We can show how to compile useful, high-level operations and language features into the lambda calculus
  - Result = adding high-level operations is **convenient** for programmers, but **not a computational necessity**
    - Concrete syntax vs. abstract syntax
    - “Syntactic sugar”
  - Result = make your compiler intermediate language simpler

# BOOLEANS

- we can encode booleans
- we will represent “true” and “false” as functions named “tru” and “fls”
- how do we define these functions?
- think about how “true” and “false” can be used
- they can be used by a testing function:
  - “test b then else” returns “then” if b is true and returns “else” if b is false
  - i.e., test tru then else →\* then; test fls then else →\* else
  - the only thing the implementation of test is going to be able to do with b is to apply it
  - the functions “tru” and “fls” must distinguish themselves when they are applied

# BOOLEANS

$\text{tru} = \lambda t.\lambda f. t$        $\text{fls} = \lambda t.\lambda f. f$

$\text{test} = \lambda x.\lambda \text{then}.\lambda \text{else}. x \text{ then else}$

- E.g. (underlined are redexes):

$\text{test tru a b}$

$= \underline{(\lambda x.\lambda \text{then}.\lambda \text{else}. x \text{ then else}) \text{ tru}} a b$

$\rightarrow \underline{(\lambda \text{then}.\lambda \text{else}. \text{tru then else})} a b$

$\rightarrow \underline{(\lambda \text{else}. \text{tru a else})} b$

$\rightarrow \text{tru a b}$

$= \underline{(\lambda t.\lambda f. t)} a b$

$\rightarrow \underline{(\lambda f. a)} b$

$\rightarrow a$

Remember  
applications are  
left associative:  
 $(((\text{test tru}) a) b)$

Quiz: Step-by-step, evaluate  
 $\text{test fls a b}$ ?

## BOOLEANS

$\text{tru} = \lambda t.\lambda f. t$                        $\text{fls} = \lambda t.\lambda f. f$

$\text{and} = \lambda b.\lambda c. b\ c\ \text{fls}$

$\text{and}\ \text{tru}\ \text{tru}$

$\rightarrow^* \text{tru}\ \text{tru}\ \text{fls}$

$\rightarrow^* \text{tru}$

( $\rightarrow^*$  stands for multi-step evaluation)



## BOOLEANS

$\text{tru} = \backslash t.\backslash f. t$                        $\text{fls} = \backslash t.\backslash f. f$

$\text{and} = \backslash b.\backslash c. b c \text{ fls}$

$\text{and fls tru}$

$\rightarrow^* \text{fls tru fls}$

$\rightarrow^* \text{fls}$

What will be the definition of “or” and “not”?

# BOOLEANS

`tru = \t.\f. t`            `fls = \t.\f. f`

`or = \b.\c. b tru c`

`or fls tru`

`→* fls tru tru`

`→* tru`

`or fls fls`

`→* fls tru fls`

`→* fls`

# PAIRS

$\text{pair} = \lambda f.\lambda s.\lambda b. b f s$  (\*pair is a constructor: pair x y\*)

$\text{fst} = \lambda p. p \text{tru}$

$\text{snd} = \lambda p. p \text{fls}$

$\text{fst} (\text{pair } v \ w)$   
 $= \text{fst} ((\lambda f.\lambda s.\lambda b. b f s) \ v \ w)$   
 $\rightarrow \text{fst} ((\lambda s.\lambda b. b \ v \ s) \ w)$   
 $\rightarrow \text{fst} (\lambda b. b \ v \ w)$   
 $= (\lambda p. p \ \text{tru}) (\lambda b. b \ v \ w)$   
 $\rightarrow (\lambda b. b \ v \ w) \ \text{tru}$   
 $\rightarrow \text{tru } v \ w$                      $/* \ \text{tru} = \lambda t.\lambda f. t \ */$   
 $\rightarrow * \ v$

## AND WE CAN GO ON...

- numbers
- arithmetic expressions (+, -, \*,...)
- lists, trees and datatypes
- exceptions, loops, ...
- ...
- the general trick:
  - values will be functions – construct these functions so that they return the appropriate information when called by an operation (applied by another function)

## QUIZ:

Suppose the numbers can be encoded in lambda calculus as:

$$0 = \lambda f. \lambda x. x$$

$$1 = \lambda f. \lambda x. f x$$

$$2 = \lambda f. \lambda x. f (f x)$$

...

Define `succ` in lambda calculus such that

$$\text{succ } 0 \rightarrow^* 1$$

$$\text{succ } 1 \rightarrow^* 2$$

...



# SIMPLY-TYPED LAMBDA CALCULUS



14

# SIMPLY TYPED LAMBDA-CALCULUS

- Goal: construct a similar system of language that combines the pure lambda-calculus with the basic types such as bool and num.
- A new type:  $\rightarrow$  (arrow type)
- Set of simple types over the type bool is
$$t ::= \text{bool}$$
$$| t_1 \rightarrow t_2$$
- Note: type constructor  $\rightarrow$  is right associative:
  - $t_1 \rightarrow t_2 \rightarrow t_3 == t_1 \rightarrow (t_2 \rightarrow t_3)$

# SYNTAX (I)

$e ::=$

- $x$
- |  $\text{true}$
- |  $\text{false}$
- |  $\text{if } e_1 \text{ then } e_2 \text{ else } e_3$
- |  $\lambda x : t . e$
- |  $e_1 \ e_2$

$v ::=$

- $\text{true}$
- |  $\text{false}$
- |  $\lambda x : t . e$

expressions:

- (variable)
- (true value)
- (false value)
- (conditional)
- (abstraction)
- (application)

values:

- (true value)
- (false value)
- (abstraction value)



## SYNTAX (II)

$t ::=$

$\text{bool}$

|  $t_1 \rightarrow t_2$

types:

(base boolean type)

(type of functions)

$\Gamma ::=$

$\cdot$

|  $\Gamma, x: t$

contexts:

(empty context)

(variable binding)

# TYPING RULES

- The type system of a language consists of a set of inductive definitions with judgment form:

$$\Gamma \vdash e : t$$

- “If the current typing context is  $\Gamma$ , then expression  $e$  has type  $t$ .”
- This judgment is known as *hypothetical judgment* ( $\Gamma$  is the hypothesis).
- $\Gamma$  (sometimes written as “G”) is a typing context (type map) which is mapping between  $x$  and  $t$  of the form  $x : t$
- $x$  is the variable name appearing in  $e$
- $t$  is a type that’s bound to  $x$

# EVALUATION (O.S.)

[ $e \rightarrow e'$ ]

$$\frac{e_1 \rightarrow e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow \text{if } e_1' \text{ then } e_2 \text{ else } e_3} \quad (\text{E - if0})$$

$$\frac{}{\text{if } \textit{true} \text{ then } e_2 \text{ else } e_3 \rightarrow e_2} \quad (\text{E - if1})$$

$$\frac{}{\text{if } \textit{false} \text{ then } e_2 \text{ else } e_3 \rightarrow e_3} \quad (\text{E - if2})$$

$$\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2} \quad (\text{E - App1})$$

$$\frac{e_2 \rightarrow e_2'}{v_1 e_2 \rightarrow v_1 e_2'} \quad (\text{E - App2})$$

$$\frac{}{(\lambda x : t.e) v \rightarrow e[v/x]} \quad (\text{E - AppAbs})$$

# TYPING

$[\Gamma \vdash e : t]$

$$\frac{x:t \in \Gamma}{\Gamma \mid -x:t} \quad \text{(T-Var)}$$

$$\frac{}{\Gamma \mid -true:bool} \quad \text{(T-True)}$$

$$\frac{}{\Gamma \mid -false:bool} \quad \text{(T-False)}$$

$$\frac{\Gamma \mid -e_1:bool \quad \Gamma \mid -e_2:t \quad \Gamma \mid -e_3:t}{\Gamma \mid -\text{if } e_1 \text{ then } e_2 \text{ else } e_3:t} \quad \text{(T-If)}$$

$$\frac{\Gamma, x:t_1 \mid -e_2:t_2}{\Gamma \mid -\lambda x:t_1. e_2 : t_1 \rightarrow t_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \mid -e_1:t_{11} \rightarrow t_{12} \quad \Gamma \mid -e_2:t_{11}}{\Gamma \mid -e_1 e_2:t_{12}} \quad \text{(T-App)}$$

# PROPERTIES OF SIMPLY-TYPED LAMBDA CALCULUS

**Lemma 1 (Uniqueness of Typing).** For every typing context  $\Gamma$  and expression  $e$ , there exists *at most* one  $t$  such that  $\Gamma \vdash e : t$ .

*(note: we don't consider sub-typing here)*

**Proof:**

By induction on the derivation of  $\Gamma \vdash e : t$ .

Case t-var: since there's at most one binding for  $x$  in  $\Gamma$ ,  $x$  has either no type or one type  $t$ . Case proved

Case t-true and t-false: obviously true.

Case t-if: 
$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t}$$

(1)  $t$  is unique

(By I.H.)

Case proved.

# PROPERTIES OF SIMPLY-TYPED LAMBDA CALCULUS

Case t-abs: 
$$\frac{\Gamma, x:t_1 \mid -e_2:t_2}{\Gamma \mid -\lambda x:t_1.e_2:t_1 \rightarrow t_2}$$

(1)  $t_2$  is unique

(By I.H.)

(2)  $\Gamma$  contains just one  $(x, t)$  pair so  $t_1$  is unique

(By (1) and assumption of t-abs)

(3)  $t_1 \rightarrow t_2$  is unique

(By (2) and t-abs)

Case t-app: 
$$\frac{\Gamma \mid -e_1:t_{11} \rightarrow t_{12} \quad \Gamma \mid -e_2:t_{11}}{\Gamma \mid -e_1 e_2:t_{12}}$$

(1)  $e_1$  and  $e_2$  satisfies Lemma 1

(By I.H.)

(2) There's at most one instance of  $t_{11}$

(By (1))

(3)  $t_{12}$  is unique, too

(By (2) & I.H.)

Quiz: Why does  $\Gamma$  contain just one instance of  $(x, t)$ , for any  $t$ ?  
In other words, each variable appears only once in  $\Gamma$ .

# PROPERTIES OF SIMPLY-TYPED LAMBDA CALCULUS

## Lemma 2 (Inversion for Typing).

- If  $\Gamma \vdash x : t$  then  $x : t \in \Gamma$
- If  $\Gamma \vdash (\lambda x : t_1. e) : t$  then there is a  $t_2$  such that
$$t = t_1 \rightarrow t_2 \text{ and } \Gamma, x : t_1 \vdash e : t_2$$
- If  $\Gamma \vdash e_1 e_2 : t$  then there is a  $t'$  such that
$$\Gamma \vdash e_1 : t' \rightarrow t \text{ and } \Gamma \vdash e_2 : t'$$

## Proof:

From the definition of the typing rules, there is only one rule for each type of expression, hence the result.

- **Well-typedness:** An expression  $e$  in the language  $L$  is said to be *well-typed*, if there exists some type  $t$ , such that  $e : t$ .

# PROPERTIES OF SIMPLY-TYPED LAMBDA CALCULUS

## Canonical Forms Lemma

(Idea: Given a type, want to know something about the shape of the value)

If  $\cdot \vdash v : t$  then

If  $t = \text{bool}$  then  $v = \text{true}$  or  $v = \text{false}$ ;

If  $t = t_1 \rightarrow t_2$  then  $v = \lambda x : t_1. e$

Proof:

By inspection of the typing rules.



# PROPERTIES OF SIMPLY-TYPED LAMBDA CALCULUS

## Exchange Lemma

If  $G, x:t_1, y:t_2, G' \vdash e:t$ ,  
then  $G, y:t_2, x:t_1, G' \vdash e:t$ .

Proof by induction on derivation of

$G, y:t, x:t, G' \vdash e:t$

(Practice!)

## Weakening Lemma

If  $G \vdash e:t$  then  $G, x:t' \vdash e:t$  (provided  $x$  not in  $\text{Dom}(G)$ )

(Practice!)

# TYPE SAFETY OF A LANGUAGE

- Safety of a language = Progress + Preservation
- Progress: A well-type term is not stuck (either it is a value or it can take a step according to the evaluation rules)
- Preservation: If a well-typed term (with type  $t$ ) takes a step of evaluation, then the resulting term is also well typed with type  $t$ .
- **Type-checking**: the process of verifying *well-typedness* of a program (or a term).

# PROGRESS THEOREM

- Suppose  $e$  is a closed and well-typed term (that is  $e : t$  for some  $t$ ). Then either  $e$  is a value or else there is some  $e'$  for which  $e \rightarrow e'$ .

Proof: By induction on the derivation of typing:  $[\Gamma \vdash e : t]$

Case T-Var: doesn't occur because  $e$  is closed.

Case T-True, T-False, T-Abs: immediate since these are values.

Case T-App:

- (1)  $e_1$  is a value or can take one step evaluation. Likewise for  $e_2$ . (By I.H.)
- (2) If  $e_1$  can take a step, then E-App1 can apply to  $(e_1 e_2)$ . (By (1))
- (3) If  $e_2$  can take a step, then E-App2 can apply to  $(e_1 e_2)$  (By (1))
- (4) If both  $e_1$  and  $e_2$  are values, then  $e_1$  must be an abstraction, therefore E-AppAbs can apply to  $(e_1 e_2)$  (By (1) and canonical forms v)
- (5) Hence  $(e_1 e_2)$  can always take a step forward. (By (2,3,4))

# PROGRESS THEOREM (CONT'D)

Case T-if:

1.  $e_1$  can either take a step or is a value (By I.H.)
2. Subcase 1:  $e_1$  can take a step (By I.H.)
  1. if  $e_1$  then  $e_2$  else  $e_3$  can take a step (By E-if0)
3. Subcase 2:  $e_1$  is a value (By I.H.)
  1. If  $e_1 = \text{true}$ , if  $e_1$  then  $e_2$  else  $e_3 \rightarrow e_2$  (By E-if1)
  2. If  $e_1 = \text{false}$ , if  $e_1$  then  $e_2$  else  $e_3 \rightarrow e_3$  (By E-if2)
4. In both subcases,  $e$  can take a step. Case proved.

# PRESERVATION THEOREM

- If  $G \vdash e : t$  and  $e \rightarrow e'$ , then  $G \vdash e' : t$ .

**Proof:** By induction on the derivation of  $G \vdash e : t$ .

Case T-Var, T-Abs, T-True, T-False:

Case doesn't apply because variable or values can't take one step evaluation.

Case T-If:  $e = \text{if } e_1 \text{ then } e_2 \text{ else } e_3$ .

If  $e \rightarrow e'$  there are two subcases cases:

Subcase 1:  $e_1$  is not a value.

(1)  $e_1 : \text{bool}$

(By assumption and inversion of T-if)

(2)  $e_1 \rightarrow e_1'$  and  $e_1' : \text{bool}$

(By IH)

(3)  $G \vdash e' : t$

(By T-If and (2))

Subcase 2:  $e_1$  is a value, i.e. either true or false.

(4)  $e \rightarrow e_2$  or  $e \rightarrow e_3$  and  $e' : t$  ( $e' = e_2$  or  $e_3$ )

(By E-If1, E-If2 and IH)

Case proved.

# PRESERVATION THEOREM (CONT'D)

Case T-App:  $e = e_1 e_2$ . Need to prove,  $G \mid - e' : t_{12}$

If  $e_1$  is not a value then:

(5)  $e_1 \rightarrow e_1'$ , and  $e_1' : t_{11} \rightarrow t_{12}$ .

(By IH)

(6)  $e_1' e_2 : t_{12}$

(By T-App)

If  $e_1$  is a value then:

(7)  $e_1$  is an abstraction.

(By assumption and T-Abs)

There are two subcases for  $e_2$ .

Subcase 1:  $e_2$  is a value. Let's call it  $v$ .

(8)  $e = \lambda x. e'' v$ , and

$G \mid - \lambda x. e'' : t_{11} \rightarrow t_{12}$ ,

(By assumption of T-App)

$G, x: t_{11} \mid - e'' : t_{12}$ ,

$G \mid - v : t_{11}$

(By (7) and inversion of T-Abs)

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

(By E-AppAbs)

(10)  $G \mid - e'' [v / x] : t_{12}$ .

(By (8), (9) and **substitution lemma**)

(11)  $G \mid - e' : t_{12}$

(By (10) & assumption)

Subcase 2:  $e_2$  is not a value.

(12) Suppose  $e_2 \rightarrow e_2'$ . Then  $e \rightarrow e_1 e_2'$ , i.e.,  $e' = e_1 e_2'$ .

(By E-App2)

(13)  $G \vdash e_2' : t_{11}$

(By I.H., T-App)

(14)  $G \vdash e_1 e_2' : t_{12}$ .

(By (13))

(15)  $G \vdash e' : t_{12}$ .

(By (12) & (14))

Case proved.

QED.

## SUBSTITUTION LEMMA

If  $G, x : t' \vdash e : t$ , and  $G \vdash v : t'$ , then  $G \vdash e [v / x] : t$ .

Proof left as an exercise.



# CURRY-HOWARD CORRESPONDENCE

- A.k.a *Curry-Howard Isomorphism*
- Connection between type theory and logic

Logic	Programming Languages
Propositions	Types
Proposition $P \supset Q$	Type $P \rightarrow Q$
Proposition $P \wedge Q$	Type $P \times Q$ (product/pair type)
Proof of proposition $P$	Expression $e$ of type $P$
Proposition $P$ is provable	Type $P$ is inhabited (by some expression)