

# Simply-Typed Lambda Calculus

Types and Programming Languages, Spring 2005, SoE UCSC

PRESENTER Jessica Gronski  
SCRIBE Avik Chaudhuri

April 5, 2005

## 1 Why types?

Recall that the untyped lambda calculus has the following syntax and semantics.

Terms  $e ::= x$  (variable)  
          |  $\lambda x.e$  (abstraction)  
          |  $e e$  (application)

Values  $v ::= \lambda x.e$

Reduction

$$\text{(APP-L)} \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \quad \text{(APP-R)} \frac{e_2 \longrightarrow e'_2}{v_1 e_2 \longrightarrow v_1 e'_2} \quad (\beta) \frac{}{\lambda x.e_1 v_2 \longrightarrow e_1[x \mapsto v_2]}$$

A normal form is any term that cannot be reduced. We agreed that one of the goals of typing would be to eliminate the possibility of normal forms that are not values. Such a property would be eventually named *progress*.

A second goal would be to guarantee termination by eliminating terms like  $\lambda x.(xx) \lambda x.(xx)$ .

## 2 Notions of safety

We discussed a couple of definitions.

1.  $\forall e. \exists e'. e \longrightarrow^* e'$  such that  $e'$  is a value in normal form.

This means that there must exist some evaluation path that terminates in a value. If there are several possible evaluation paths, some paths may not terminate, or may terminate in terms that are not values. This did not seem quite appropriate for our goals.

2.  $\forall e. \forall e'. (e \longrightarrow^* e' \text{ and } e' \text{ is in normal form})$  implies  $e'$  is a value.

Observe that this does not require that all evaluation paths terminate; it does require that if an evaluation terminates, it does so at a value.

We adopted (2) as our definition of *safety*.

### 3 Simply-typing an applied $\lambda$ calculus

We wrote down a simply-typed applied calculus with booleans and conditional branches.

Simple types  $T ::= T \rightarrow T$  (function type)  
 $| \mathcal{B}$  (boolean type)

Terms  $e ::= x$  (variable)  
 $| \lambda x : T. e$  (abstraction)  
 $| e e$  (application)  
 $| \text{true}$  (true)  
 $| \text{false}$  (false)  
 $| \text{if } e \text{ then } e \text{ else } e$  (conditional branch)

Values  $v ::= \lambda x. e$   
 $| \text{true}$   
 $| \text{false}$

Reduction

$$\begin{array}{l} \text{(T-APP-L)} \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \quad \text{(T-APP-R)} \frac{e_2 \longrightarrow e'_2}{v_1 e_2 \longrightarrow v_1 e'_2} \quad \text{(T-}\beta\text{)} \frac{}{\lambda x : T. e_1 v_2 \longrightarrow e_1[x \mapsto v_2]} \\ \text{(T-COND-B)} \frac{e_1 \longrightarrow e'_1}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \longrightarrow \text{if } e'_1 \text{ then } e_2 \text{ else } e_3} \\ \text{(T-COND-L)} \frac{}{\text{if true then } e_1 \text{ else } e_2 \longrightarrow e_1} \quad \text{(T-COND-R)} \frac{}{\text{if false then } e_1 \text{ else } e_2 \longrightarrow e_2} \end{array}$$

We then wrote down typing rules for the calculus.

Type environments  $\Gamma ::= \cdot$  (empty)  
 $| \Gamma; x : T$  (augment)

Type judgements  $::= \Gamma \vdash e : T$

The notation  $x : T \in \Gamma$  means that the rightmost binding of  $x$  in  $\Gamma$  is  $x : T$ . We write  $\vdash e : T$  to mean  $\cdot \vdash e : T$ .

$$\begin{array}{l} \text{(AX-VAR)} \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\rightarrow\text{I}) \frac{\Gamma, x : T \vdash e : T'}{\lambda x : T. e : T \rightarrow T'} \quad (\rightarrow\text{E}) \frac{\Gamma \vdash e_1 : T \rightarrow T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 e_2 : T'} \\ \text{(AX-TRUE)} \frac{}{\Gamma \vdash \text{true} : \mathcal{B}} \quad \text{(AX-FALSE)} \frac{}{\Gamma \vdash \text{false} : \mathcal{B}} \end{array}$$

$$(\mathcal{B}E) \frac{\Gamma \vdash e_1 : \mathcal{B} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T}$$

We say  $e$  is well-typed iff  $\vdash e : T$  for some  $T$ .

### 3.1 A digression

We conjectured that it should be possible to encode **true** and **false** in the pure  $\lambda$  calculus, and hence derive the typing rules AX-TRUE, AX-FALSE and  $\mathcal{B}E$ . Initial attempts pointed towards the necessity of universal types in the type system. For example, one could define

$$\text{Types } T ::= T \rightarrow T \\ | \iota$$

$$\text{Boolean types } \mathcal{B}_T = T \rightarrow T \rightarrow T$$

$$\begin{aligned} \text{Boolean values } \text{true}_T &= \lambda t : T. \lambda f : T. t \\ \text{false}_T &= \lambda t : T. \lambda f : T. f \end{aligned}$$

$$\text{Conditional branch } \text{if-then-else}_T = \lambda b : \mathcal{B}_T. \lambda x : T. \lambda y : T. b x y$$

Then AX-TRUE is derived as

$$\frac{\frac{\frac{t : T \in \Gamma, t : T, f : T}{\Gamma, t : T, f : T \vdash t : T}}{\Gamma, t : T \vdash \lambda f : T. t : T \rightarrow T}}{\Gamma \vdash \text{true}_T : \mathcal{B}_T}$$

AX-FALSE is derived as

$$\frac{\frac{\frac{f : T \in \Gamma, t : T, f : T}{\Gamma, t : T, f : T \vdash f : T}}{\Gamma, t : T \vdash \lambda f : T. f : T \rightarrow T}}{\Gamma \vdash \text{false}_T : \mathcal{B}_T}$$

Writing  $\text{if } e_1 \text{ then } e_2 \text{ else } e_3$  for  $\text{if-then-else } e_1 e_2 e_3$ ,  $\mathcal{B}E$  is derived as

$$\frac{\frac{\frac{\frac{b : \mathcal{B}_T \in \Gamma, b : \mathcal{B}_T, x : T, y : T}{\Gamma, b : \mathcal{B}_T, x : T, y : T \vdash b : T \rightarrow T \rightarrow T}}{\Gamma, b : \mathcal{B}_T, x : T, y : T \vdash b x : T \rightarrow T}}{\Gamma, b : \mathcal{B}_T, x : T, y : T \vdash b x y : T}}{\Gamma, b : \mathcal{B}_T, x : T, y : T \vdash \lambda y : T. b x y : T \rightarrow T}}{\Gamma, b : \mathcal{B}_T \vdash \lambda x : T. \lambda y : T. b x y : T \rightarrow T \rightarrow T}}{\Gamma \vdash \text{if-then-else}_T : \mathcal{B}_T \rightarrow T \rightarrow T \rightarrow T} \frac{\Gamma \vdash e_1 : \mathcal{B}_T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{if-then-else}_T e_1 : T \rightarrow T \rightarrow T} \frac{\Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if-then-else}_T e_1 e_2 e_3 : T}$$

## 4 Proving safety

We discussed two lemmas that are sufficient to prove safety in the simply-typed  $\lambda$  calculus.

**Lemma 4.1** (Progress). *If  $e$  is well-typed (i.e.,  $\vdash e : T$  for some  $T$ ) then either  $e$  is a value or  $\exists e'$  such that  $e \longrightarrow e'$ .*

PROOF. By induction on the structure of the derivation  $\vdash e : T$ . ◁

**Lemma 4.2** (Preservation). *If  $\vdash e : T$  for some  $T$  and  $\exists e'$  such that  $e \longrightarrow e'$ , then  $\vdash e' : T$ .*

PROOF. By induction on the structure of the derivation  $\vdash e : T$  and case analysis on  $e \longrightarrow e'$ . ◁

The following theorem asserts the safety property discussed in §2 for well-typed terms.

**Theorem 4.3** (Safety). *If  $e$  is well typed and  $e \longrightarrow^* e'$  such that  $e'$  is in normal form, then  $e'$  is a value.*

PROOF. By induction on the length of  $e \longrightarrow^* e'$ , using Lemmas 4.1 and 4.2. ◁

In fact, a weak version of Lemma 4.2 is sufficient (together with Lemma 4.1) to prove Theorem 4.3.

**Lemma 4.4** (Weak preservation). *If  $e$  is well-typed and  $\exists e'$  such that  $e \longrightarrow e'$ , then  $e'$  is well-typed.*

## 5 Venn diagrams for interesting subsets of terms

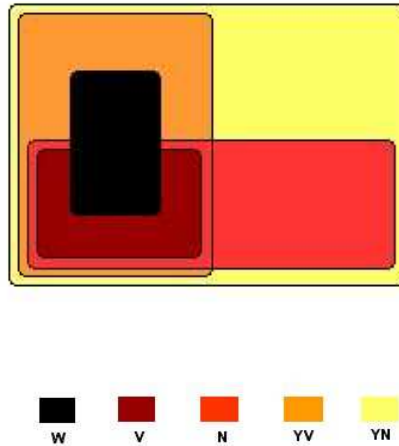


Figure 1: Subsets of terms in the  $\lambda$  calculus

Figure 1 shows containment relationships between the sets  $V$  (values),  $N$  (terms in normal form),  $YV$  (terms that yield values),  $YN$  (terms that yield normal forms) and  $W$  (well-typed terms).

## 6 Erasing type annotations

Finally, we agreed that typing does not affect the operational semantics of the language. The function `erase` erases type annotations on all abstracted variables, giving terms in the untyped language. Thus

$$\begin{aligned}
 \text{erase}(x) &= x \\
 \text{erase}(\lambda x : T.e) &= \lambda x. \text{erase}(e) \\
 \text{erase}(e_1 e_2) &= \text{erase}(e_1) \text{erase}(e_2) \\
 \text{erase}(\text{true}) &= \text{true} \\
 \text{erase}(\text{false}) &= \text{false} \\
 \text{erase}(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) &= \text{if } \text{erase}(e_1) \text{ then } \text{erase}(e_2) \text{ else } \text{erase}(e_3)
 \end{aligned}$$

**Lemma 6.1** (Correspondence of  $\longrightarrow$ ). *For any term  $e$  in the typed language,*

1. *if  $e \longrightarrow e'$  then  $\text{erase}(e) \longrightarrow \text{erase}(e')$*
2. *if  $\text{erase}(e) \longrightarrow e''$  then  $\exists e'. e \longrightarrow e'$  and  $e'' = \text{erase}(e')$*

PROOF. (1) By induction on the structure of the derivation  $e \rightarrow e'$  and (2) by case analysis on  $e$  and induction on the structure of  $\text{erase}(e) \rightarrow e''$ . ◁

In other words, the following diagram commutes.

$$\begin{array}{ccccc}
 e & \longrightarrow & e' & \longrightarrow & \dots \text{ (typed world)} \\
 \downarrow & & \downarrow & & \text{(erase type annotations)} \\
 \text{erase}(e) & \longrightarrow & \text{erase}(e) & \longrightarrow & \dots \text{ (untyped world)}
 \end{array}$$

**Theorem 6.2** (Correspondence of safety). *If  $e$  is well-typed and  $\text{erase}(e) \longrightarrow^* e''$  such that  $e''$  is in normal form, then  $e''$  is a value.*

PROOF. By induction on the length of  $\text{erase}(e) \longrightarrow^* e''$  with Lemma 6.1, and using Theorem 4.3. ◁