Lambda Calculus

Lecture (4): Extending Simply Typed Lambda Calculus to model programming language features

Sriram Rajamani

Microsoft Research

Typed lambda calculus

Syntax:

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

Values:

$$
\begin{aligned}
v :\ & \text{true} \\
\mid\ & \text{false} \\
\mid\ & \lambda x{:}T.\,e
\end{aligned}
$$

Types:

$$
\begin{aligned}
T :\ & \text{bool} \\
\mid\ & T \to T
\end{aligned}
$$

Typing relation
or
Typing judgment

$\Gamma \vdash e : T$ ⇐ under the assumption Γ, e has type T

$$
\begin{aligned}
\Gamma ::=\ & \emptyset \\
\mid\ & \Gamma, x : T
\end{aligned}
$$

⇐ type assumption for free variables

Operational semantics

$$\text{if true then } e_1 \text{ else } e_2 \longrightarrow e_1 \quad \text{[E-IFTRUE]}$$

$$\text{if false then } e_1 \text{ else } e_2 \longrightarrow e_2 \quad \text{[E-IFFALSE]}$$

$$\frac{e_1 \longrightarrow e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \longrightarrow \text{if } e_1' \text{ then } e_2 \text{ else } e_3} \quad \text{[E-IF]}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \quad \text{[E-APP1]}$$

$$\frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'} \quad \text{(E-APP2]}$$

$$(\lambda x : T_{11} .\ e_2)\ v_1 \longrightarrow [x \mapsto v_1]\ e_2$$

[E-APPABS]

Typing rules:

$$\text{true} : \text{Bool} \quad [\text{T-TRUE}] \qquad \text{false} : \text{Bool} \quad [\text{T-FALSE}]$$

$$\frac{e_1 : \text{Bool} \qquad e_2 : T \qquad e_3 : T}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \quad [\text{T-IF}]$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad [\text{T-VAR}]$$

$$\frac{\Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \lambda x : T_1 . e_2 : T_1 \to T_2} \quad [\text{T-ABS}]$$

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2, \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\, e_2 : T_2} \quad [\text{T-APP}]$$

Type safety = Progress + Preservation

Progress: A well-typed term is not stuck (either it is a value, or it can take a step according to operational semantics)

Preservation: If a well-typed term takes a step, the resulting term is also well-typed

Today..

Extensions to simply typed lambda calculus that allow us to model & verify common programming languages

1. Unit Type & sequencing
2. let binding
3. Records and variants
4. Subtyping
5. Pointers
6. Polymorphism

Unit

Syntax

$e ::= \ldots$

$\quad \text{unit}$

$\quad e_1 ; e_2$

Values

$v ::= \ldots$

$\quad \text{unit}$

Types

$T ::= \ldots$

$\quad \text{UNIT}$

Semantics

$$\frac{e_1 \to e_1'}{e_1 ; e_2 \to e_1' ; e_2}$$

$$\text{unit} ; e_1 \to e_1$$

Typing

$$\vdash \text{unit} : \text{UNIT}$$

$$\frac{\Gamma \vdash e_1 : \text{UNIT} \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 ; e_2 : T_2}$$

Unit

Syntax

$e ::= \ldots$
$\quad\quad \text{unit}$
$\quad\quad e_1 ; e_2$

Values

$v ::= \ldots$
$\quad\quad \text{unit}$

Types

$T ::= \ldots$
$\quad\quad \text{UNIT}$

$$\boxed{\begin{array}{l} e_1 ; e_2 \overset{\text{def}}{=} (\lambda x : \text{UNIT}.\, e_2)\ e_1 \\ \quad \text{where } x \notin FV(e_2) \end{array}}$$

Semantics

$$\frac{e_1 \to e_1'}{e_1 ; e_2 \to e_1' ; e_2}$$

$$\text{unit} ; e_1 \to e_1$$

Typing

$$\vdash \text{unit} : \text{UNIT}$$

$$\frac{\Gamma \vdash e_1 : \text{UNIT} \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 ; e_2 : T_2}$$

Let bindings

Syntax:

$$e ::= \ldots \mid \text{let } x = e_1 \text{ in } e_2$$

Semantics

$$\text{let } x = v \text{ in } e \to [x \mapsto v]e$$

$$\frac{e_1 \to e_1'}{\text{let } x = e_1 \text{ in } e_2 \to \text{let } x = e_1' \text{ in } e_2}$$

Typing

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2}$$

Example:

$\lambda f. \lambda x.$
let double $= \lambda g. \lambda y\, (g\,(g(y)))$
in
double f (double f x)

Records

eg:

let $x = \{real : 5, imag : 6\}$ in
square (x.real * x.real + x.imag * x.imag)

Variants

eg:

Addr = <physical : Physical Addr,
virtual : Virtual Addr>

let a = <physical = pa> as Addr;

getname = λa: Addr.
case a of
<physical = x> ⇒ x.firstlast
| <virtual = y> ⇒ y.name

Records

Syntax

$$e ::= \dots \mid \{l_i = e_i \;^{i \in 1..n}\}$$

Values

$$v ::= \dots \mid \{l_i = v_i \;^{i \in 1..n}\}$$

Types

$$T ::= \dots \mid \{l_i : T_i \;^{i \in 1..n}\}$$

Semantics

$$\{l_i = v_i \;^{i \in 1..n}\}.l_j \to v_j$$

$$\frac{e_1 \to e_1'}{e_1.l \to e_1'.l}$$

$$\frac{e_j \to e_j'}{\{l_i = v_i \;^{i \in 1..j-1},\; l_j = e_j,\; l_i = v_i \;^{i \in j+1..n}\} \to \{l_i = v_i \;^{i \in 1..j-1},\; l_j = e_j',\; l_i = v_i \;^{i \in j+1..n}\}}$$

Records

Syntax

$$e ::= \dots$$
$$\{ l_i = e_i^{\,i \in 1..n} \}$$

Values

$$v ::= \dots$$
$$\{ l_i = v_i^{\,i \in 1..n} \}$$

Types

$$T ::= \dots$$
$$\{ l_i : T_i^{\,i \in 1..n} \}$$

Typing rules

$$\frac{\text{for each } i \quad \Gamma \vdash e_i : T_i}{\Gamma \vdash \{ l_i = e_i^{\,i \in 1..n} \} : \{ l_i : T_i^{\,i \in 1..n} \}}$$

$$\frac{\Gamma \vdash e : \{ l_i : T_i^{\,i \in 1..n} \}}{\Gamma \vdash e.l_j : T_j}$$

TODO:

Prove type safety (progress + preservation)

Records

eg:

let $x = \{ real : 5, \; imag : 6 \}$ in

square (x.real * x.real + x.imag * x.imag)

Variants

eg:

Addr = <physical : PhysicalAddr,
virtual : VirtualAddr>

let a = <physical = pa> as Addr;

getname = λa: Addr.
case a of
<physical = x> $\Rightarrow$ x.firstlast
| <virtual = y> $\Rightarrow$ y.name

Variants

Syntax

$e ::= \ldots$

$\quad | \ \langle l = e \rangle \text{ as } T$

$\quad | \ \text{case } e \text{ of } \langle l_i = x_i \rangle \Rightarrow e_i^{\ i \in 1..n}$

Types:

$T ::= \ldots$

$\quad \langle l_i : T_i^{\ i \in 1..n} \rangle$

Semantics

$$\text{case } (\langle l_j = v_j \rangle \text{ as } T) \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i^{\ i \in 1..n} \ \rightarrow \ [x_j \mapsto v_j]\, t_j$$

$$\frac{e \rightarrow e'}{\text{case } e \text{ of } \langle l_i = x_i \rangle \Rightarrow e_i^{\ i \in 1..n} \ \rightarrow \ \text{case } e' \text{ of } \langle l_i = x_i \rangle \Rightarrow e_i^{\ i \in 1..n}}$$

$$\frac{e \rightarrow e'}{\langle l_i = e \rangle \text{ as } T \ \rightarrow \ \langle l_i = e' \rangle \text{ as } T}$$

Variants

Syntax

$e ::= \ldots$

$\mid \langle l = e \rangle \text{ as } T$

$\mid \text{case } e \text{ of } \langle l_i = x_i \rangle \Rightarrow T_i^{\,i \in 1..n}$

Types:

$T ::= \ldots$

$\langle l_i : T_i^{\,i \in 1..n} \rangle$

Typing rules

$$\frac{\Gamma \vdash e_j : T_j}{\Gamma \vdash \langle l_j = e_j \rangle \text{ as } \langle l_i : T_i^{\,i \in 1..n} \rangle : \langle l_i : T_i^{\,i \in 1..n} \rangle}$$

$$\frac{\Gamma \vdash e : \langle l_i : T_i^{\,i \in 1..n} \rangle \qquad \text{for each } i \quad \Gamma, x_i : T_i \vdash e_i : T}{\Gamma \vdash \text{case } e \text{ of } \langle l_i = x_i \rangle \Rightarrow e_i^{\,i \in 1..n} : T}$$

Today..

Extensions to Simply typed lambda calculus that allow us to model & verify common programming languages

1. Unit Type & sequencing
2. let binding
3. Records and Variants
4. Subtyping
5. Pointers
6. Polymorphism

Subtyping:

Consider the term:

$$(\lambda r : \{x : Nat\}.\ r.x)\ \{x = 0, y = 1\}$$

Type $\{x : Nat, y : Nat\}$

Subtyping:

Consider the term:

$$\underbrace{(\lambda r : \{x : Nat\}.\ r.x)}\ \underbrace{\{x=0, y=1\}}$$

Type $\{x : Nat, y : Nat\}$

Cannot type this since:

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\, e_2 : T_2}$$

Idea: A value of type $\{x : Nat, y : Nat\}$ should be usable wherever a value of type $\{x : Nat\}$ is desired!

$S <: T$, read "S is a subtype of T"
and means "A term of type S can be safely used wherever a term of type T is expected"

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \; \text{[T-SUB]}$$

Sub-typing

Types:

$T ::= \ldots$

$\quad \mathrm{Top}$

Typing Rules:

$$S <: S \quad [\text{S-REFL}]$$

$$\frac{S <: U \qquad U <: T}{S <: T}\ [\text{S-TRANS}]$$

$$S <: \mathrm{TOP} \quad [\text{S-TOP}]$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}\ [\text{S-ARROW}]$$

$$\frac{\Gamma \vdash e : S \qquad S <: T}{\Gamma \vdash e : T}\ [\text{T-SUB}]$$

Subtyping

Types:

T ::= ...
  Top

Typing Rules:

$$S <: S \quad \text{[S-REFL]}$$

$$\frac{S <: U \quad U <: T}{S <: T} \text{[S-TRANS]}$$

$$S <: \text{TOP} \quad \text{[S-TOP]}$$

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \text{[S-ARROW]}$$

$$\frac{\Gamma \vdash e : S \quad S <: T}{\Gamma \vdash e : T} \text{[T-SUB]}$$

$$\{l_i : T_i\}^{i \in 1..n+k} <: \{l_i : T_i\}^{i \in 1..n} \quad \text{[S-RCD WIDTH]}$$

$$\frac{\text{foreach } i \quad S_i <: T_i}{\{l_i : S_i\}^{i \in 1..n} <: \{l_i : T_i\}^{i \in 1..m}} \text{[S-RCD DEPTH]}$$

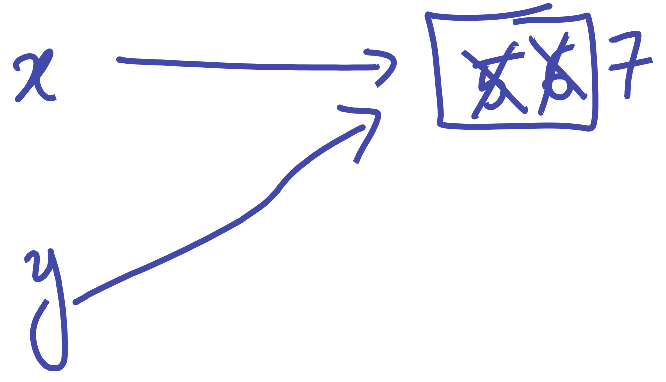
$$\frac{\{k_j \in S_j\}^{j \in 1..n} \text{ is a permutation of } \{l_i : T_i\}^{i \in 1..n}}{\{k_j : S_j\}^{j \in 1..n} <: \{l_i : T_i\}^{i \in 1..n}} \text{[S-RCD PERM]}$$

Pointers (references)

```
let x = ref 5 in
let y = x in
x := !x + 1;
y := !y + 1;
!x
```

Pointers (references)

```
let x = ref 5 in
let y = x in
x := !x + 1;
y := !y + 1;
!x
```



References

Syntax:

$$e ::= \ldots$$

$$\text{ref } e$$

$$e_1 := e_2$$

$$!e$$

Values:

$$v ::= \ldots$$

$$\ell$$ ⟸ store locations (memory address)

Types:

$$T ::= \ldots$$

$$\text{Ref } T$$

Semantics

Introduce:

μ : Locations → values

memory

Operational semantics:

$$e \mid \mu \longrightarrow e' \mid \mu'$$

"Term e evolves to e', changing memory from μ to μ' as a side-effect"

Semantics:

$$\frac{e_1 \mid \mu \to e_1' \mid \mu'}{!e_1 \mid \mu \to !e_1' \mid \mu'} \text{ [E-DEREF]}$$

$$\frac{\mu(l) = v}{!l \mid \mu \to v \mid \mu} \text{ [E-DEREF LOC]}$$

$$\frac{e_1 \mid \mu \to e_1' \mid \mu'}{e_1 := e_2 \mid \mu \to e_1' := e_2 \mid \mu'} \text{ [E-ASSIGN1]}$$

$$\frac{e_2 \mid \mu \to e_2' \mid \mu'}{v := e_2 \mid \mu \to v := e_2' \mid \mu'} \text{ [E-ASSIGN2]}$$

$$l := v_2 \mid \mu \to \text{unit} \mid [l \mapsto v_2]\mu$$

[E-ASSIGN]

$$\frac{e_1 \mid \mu \to e_1' \mid \mu'}{\text{ref } e_1 \mid \mu \to \text{ref } e_1' \mid \mu} \text{ [E-REF]}$$

$$\frac{l \notin dom(\mu)}{\text{ref } v_1 \mid \mu \to l \mid (\mu, l \mapsto v)} \text{ [E-REFV]}$$

Typing relation:

$$\Gamma \mid \Sigma \vdash e : T$$

Store typing: a function from locations to types

Typing rules:

$$\frac{\Sigma(l) = T_1}{\Gamma \mid \Sigma \vdash l : \mathrm{Ref}\ T_1} \quad [\text{T-LOC}]$$

$$\frac{\Gamma \mid \Sigma \vdash e_1 : T_1}{\Gamma \mid \Sigma \vdash \mathrm{ref}\ e_1 : \mathrm{Ref}\ T_1} \quad [\text{T-REF}]$$

$$\frac{\Gamma \mid \Sigma \vdash e_1 : \mathrm{Ref}\ T_1}{\Gamma \mid \Sigma \vdash !e_1 : T_1} \quad [\text{T-DEREF}]$$

$$\frac{\Gamma \mid \Sigma \vdash e_1 : \mathrm{Ref}\ T \qquad \Gamma \mid \Sigma \vdash e_2 : T}{\Gamma \mid \Sigma \vdash e_1 := e_2 : \mathrm{Unit}} \quad [\text{T-ASSIGN}]$$

Interaction between references & subtyping

$$\frac{S <: T}{\text{Ref } S <: \text{Ref } T} \quad [\text{UNSOUND}]$$

Interaction between references & subtyping

$$\frac{S <: T}{\text{Ref } S <: \text{Ref } T}$$ [UNSOUND]

let x = ref {a:5, b:10} in

let foo = λy : ref {a:Nat} in

$y.a$ in

foo (x)

Interaction between references & subtyping

$$\frac{S <: T}{\mathrm{Ref}\ S <: \mathrm{Ref}\ T}\ \text{[UNSOUND]}$$

let x = ref {a:5, b:10} in
let bar = λy : ref {a:Nat} in
y := {a:15} in

bar (x);
$x.b$ ⟵ cannot evaluate!

Sound rule

$$\frac{S <: T \qquad T <: S}{\text{Ref } S <: \text{Ref } T}$$

Sound rule

$$\frac{S <: T \quad T <: S}{\text{Ref } S <: \text{Ref } T}$$

Java design flaw

$$\frac{S <: T}{\text{Array } S <: \text{Array } T}$$ [UNSOUND]

Consequence: • Expensive runtime checks
• Java designers agree that this is a design flaw

let polymorphism

Would like to type:

let double = $\lambda f.\ \lambda a.\ f(f(a))$ in

let a = double $(\lambda x{:}Nat.\ succ(succ(x)))$ 1 in

let b = double $(\lambda x{:}Bool.\ not\ x)$ false in ...

Recall typing rule for let:

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2}$$

Polymorphic let rule [Milner 1978]

Would like to type:

let double = $\lambda f.\ \lambda a.\ f(f(a))$ in

let a = double $(\lambda x{:}Nat.\ succ(succ(x)))$ 1 in

let b = double $(\lambda x{:}Bool.\ not\ x)$ false in

...

Polymorphic typing rule for <u>let</u>:

$$\frac{\Gamma \vdash [x \mapsto e_1]\, e_2 : T_2 \qquad \Gamma \vdash e_1 : T_1}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2}\ [\text{MILNER}]$$

Summary of today's class:

Extensions to simply typed lambda calculus that allow us to model & verify common programming languages

1. Unit Type & sequencing
2. let binding
3. Records and variants
4. Subtyping
5. Pointers
6. Polymorphism