Simply Typed Lambda Calculus, cont. Simple Type Inference

Announcements

Expr

Var Name : x

Lambda Name Expr : $\lambda x.\ E$

App Expr Expr : $E_1\ E_2$

- HW5?
- Will post HW6 next time
- I am still grading HW4

Type Unsafe C and C++

C :

```
struct str {
  float f;
  short i;
} *s;
...
int *i = &a;
s = (struct str*) i;
s->f = ...
```

FORBIDDEN

C unions!

```
union uni {
  float f; // 4bytes
  short i; // 2 bytes
} u;

u.f = 1.222;
a[u.i] =
```

C++

```
A foo()
|
B foo()
  foo(int)

A *a = new A();
B *b = (B*) a;
b->foo(100);
```

FORBIDDEN ERROR.

Outline

- The simply typed lambda calculus
  - Syntax   PL Syntax
  - Static semantics   Typing rules
  - Dynamic semantics $E_1 \to E_2 \to E_3 \to \dots$
    - Stuck states
  - Type safety = progress + preservation

/ soundness   if $E : \tau$ then either E is value (done) or $E \to E'$ if $E : \tau$ and $E \to E'$ then $E' : \tau$

- Introduction to simple type inference

Putting It All Together, Formally

- Simply typed lambda calculus (System F_1)
  - Syntax
  - The type system: type expressions, environment, and type judgments
  - The dynamic semantics
    - Stuck states
  - Progress and preservation theorem

Type Expressions

- Syntax of simply typed lambda calculus:
  - **E ::= x | (λx$\boxed{: \tau.}$ E$_1$) | (E$_1$ E$_2$) | c**

- Introducing type expressions
  - $\tau ::= b \mid \tau \rightarrow \tau$

    BASE ARROW
  - A type is a basic type **b** (we will only consider **int**, for simplicity), or a function type

- Examples int, int→int, int→int→int, (int→int)→int

  int

  int → (int → int) // → is right-associative, thus can write just **int → int → int**

Type Environment and Type Judgments

- A term in the simply typed lambda calculus is
  - Type correct i.e., well-typed, or
  - Type incorrect

$\Gamma = [x : \tau_1, y : \tau_2, z : \tau_3]$

$\Gamma = [x : int, y : bool, z : \ldots]$

- The rules that judge type correctness are given in the form of type judgments in an environment
  - Environment $\quad \Gamma \vdash E : \tau \quad$ ($\vdash$ is the turnstile)
  - Read: environment Γ entails that E has type τ
  - Type judgment

Premises

$$\frac{\Gamma \vdash E_1 : \sigma \rightarrow \tau \qquad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash (E_1\ E_2) : \tau}$$

Conclusion

Semantics

$$\frac{}{\Gamma \vdash c : int}$$

looks up the type of **x** in environment **Γ**

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

...

$$\frac{[\,] \vdash (\lambda x{:}int.\, x) : int \to int \qquad [\,] \vdash 3 : int}{[\,] \vdash ((\lambda x{:}int.\, x)\ 3) : \tau = int}$$

$$\frac{\Gamma \vdash E_1 : \sigma \to \tau \qquad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash (E_1\ E_2) : \tau} \quad \text{(Application)}$$

binding: augments environment **Γ** with binding of **x** to type **σ**

$$\frac{\Gamma, x{:}\sigma \vdash E_1 : \tau}{\Gamma \vdash (\lambda x{:}\sigma.\, E_1) : \sigma \to \tau} \quad \text{(Abstraction)}$$

$$\frac{\dfrac{x : int \in [x : int]}{[x : int] \vdash x : \tau = int}}{\underset{[\,]}{NIL} \vdash (\lambda x{:}int.\, x) : int \to \tau = int \to int}$$

Examples

- Deduce the type for

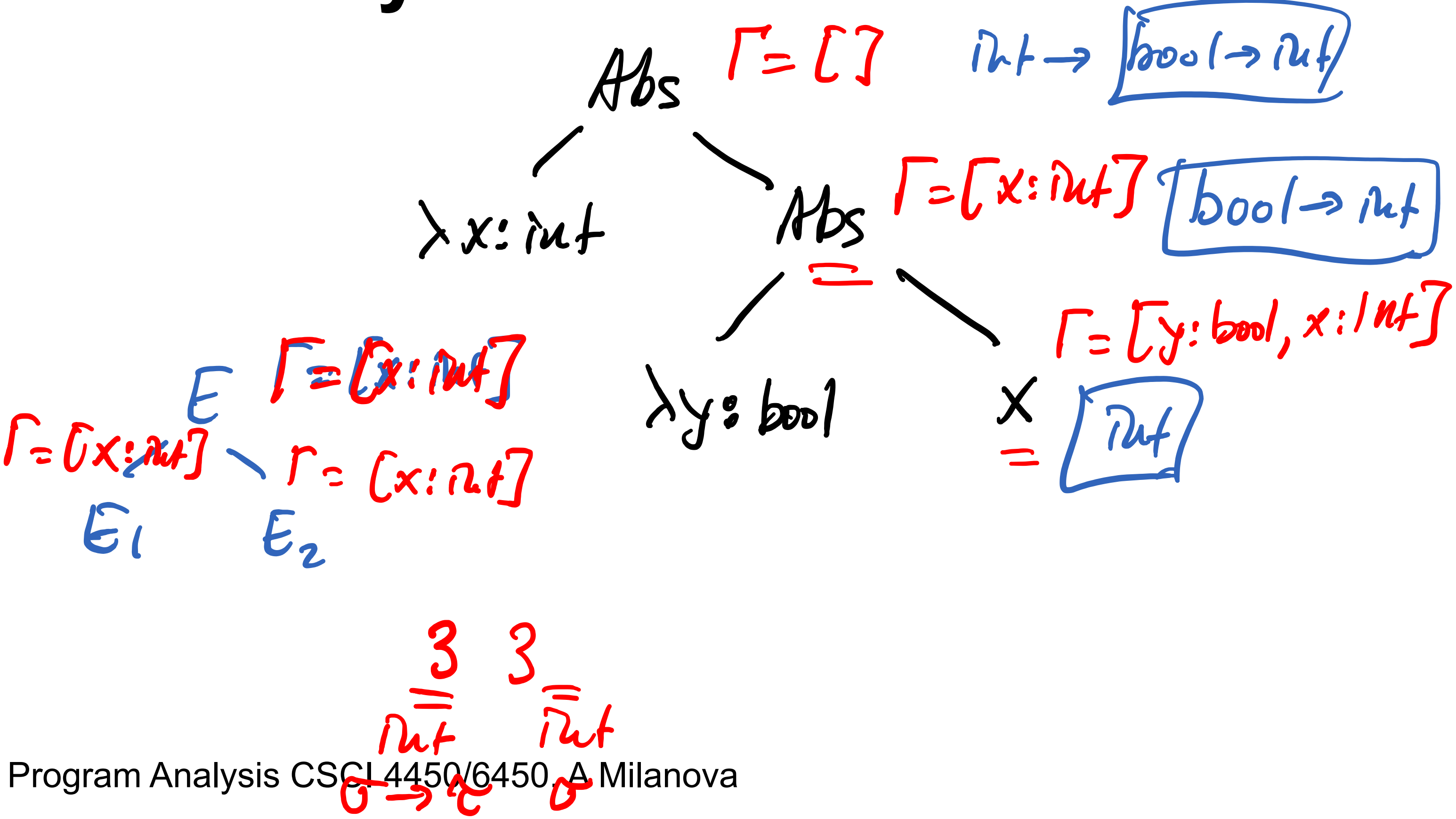
  λx: int.λy: bool. x in the **nil** environment

$$\dfrac{\dfrac{\dfrac{x:int \in [y:bool, x:int] \vdash x = int}{[y:bool, x:int] \vdash x : \tau' = int}}{[x:int] \vdash \lambda y:bool.x : \tau = bool \to \tau' = bool \to int}}{[\,] \vdash \lambda x:int.\underbrace{\lambda y:bool.x}_{\tau} : int \to \tau = int \to (bool \to int) = int \to bool \to int}$$

Examples

- Deduce the type for

  λx: int.λy: bool. x in the **nil** environment



Extensions (of Language and Static Semantics)

$$\frac{}{\Gamma \vdash c : int}$$

$$\frac{\Gamma \vdash E_1 : int \qquad \Gamma \vdash E_2 : int}{\Gamma \vdash E_1 + E_2 : int}$$

(Arithmetic)

$$\frac{\Gamma \vdash E_1 : int \qquad \Gamma \vdash E_2 : int}{\Gamma \vdash E_1 = E_2 : bool}$$

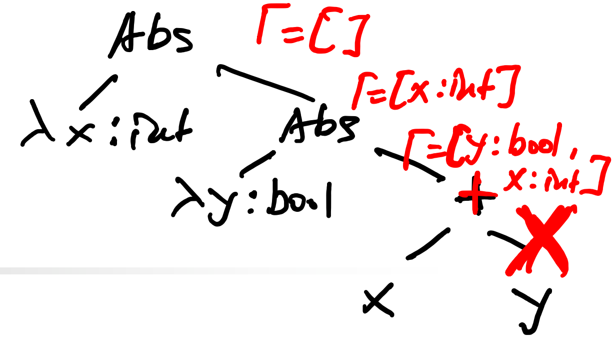
(Comparison)

= is Comparison NOT ASSIGNMENT

$$\frac{\Gamma \vdash b : bool \qquad \Gamma \vdash E_1 : \tau \qquad \Gamma \vdash E_2 : \tau}{\Gamma \vdash \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau}$$

(if-then-else)

Examples



- Is this a valid type?

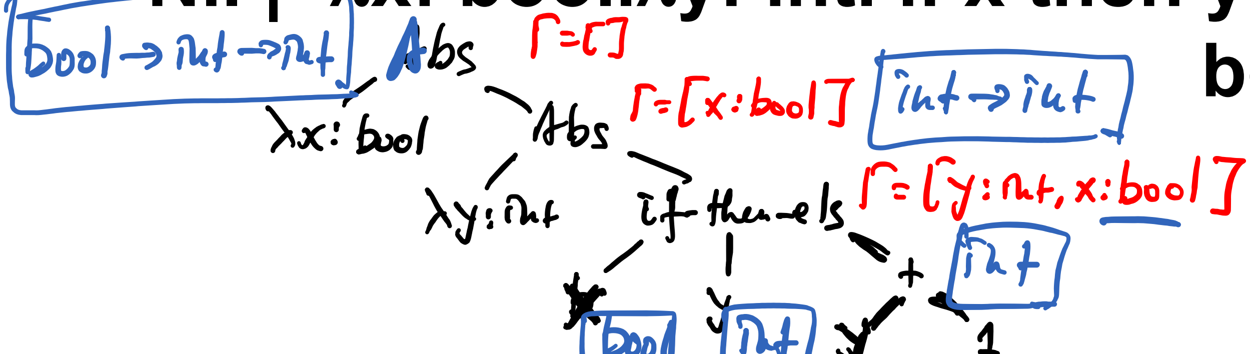
  Nil ⊢ λx: int.λy: bool. x+y : int → bool → int

  TYPE INCORRECT

- Is this a valid type?

  Nil ⊢ λx: bool.λy: int. if x then y else y+1 :
  bool → int → int

  ✓



(type of the if-then-else)

Examples

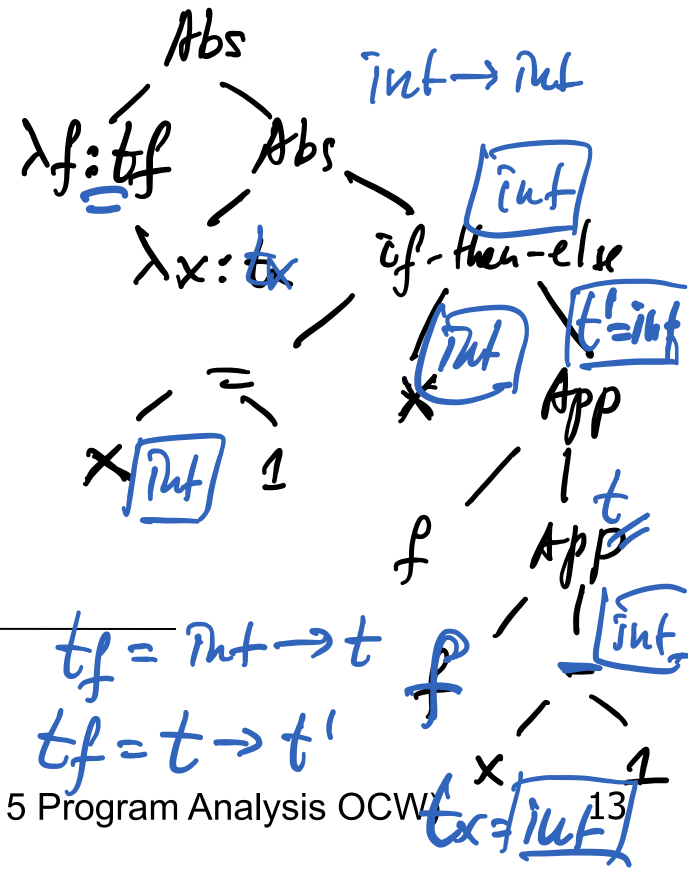
- Can we deduce the type of this term?

λf. λx. if x=1 then x else (f (f (x-1))) : ? $(int \to int) \to int \to int$

$$\frac{\Gamma \vdash E_1 : int \qquad \Gamma \vdash E_2 : int}{\Gamma \vdash E_1 = E_2 : bool}$$

$$\frac{\Gamma \vdash E_1 : int \qquad \Gamma \vdash E_2 : int}{\Gamma \vdash E_1 + E_2 : int}$$

$$\frac{\Gamma \vdash b : bool \qquad \Gamma \vdash E_1 : \tau \qquad \Gamma \vdash E_2 : \tau}{\Gamma \vdash \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau}$$



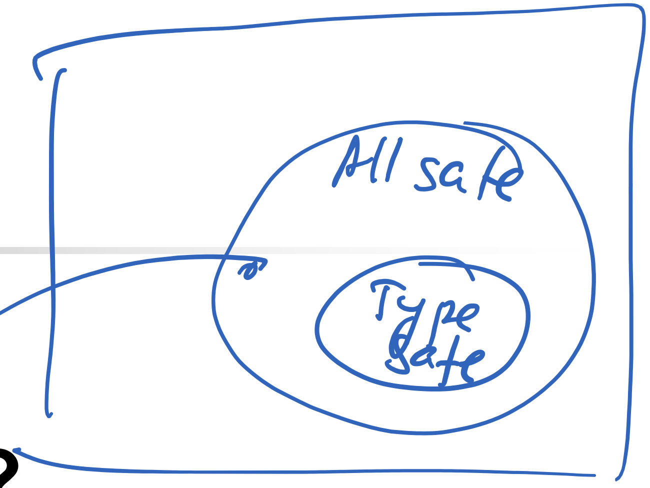
Examples

- How about this

$(\lambda x.\ x\ (\lambda y.\ y)\ (x\ 1))\ (\lambda z.\ z) : ?$

$\rightarrow (\lambda z.\ z)(\lambda y.y)((\lambda z.z)\ 1) \rightarrow$

$(\lambda y.y)((\lambda z.z)\ 1) \rightarrow (\lambda y.y)\ 1 \rightarrow 1$



- **x** cannot have two "different" types
  - **(x 1)** demands **int** → **?**
  - **(x (λy. y))** demands **(τ → τ)** → **?**
- Program does not reach a "stuck state" but is nevertheless rejected. A sound type system typically rejects some correct programs

Putting It All Together, Formally

- Simply typed lambda calculus (System F_1)
  - Syntax of the simply typed lambda calculus
  - The type system: type expressions, environment, and type judgments
  - The dynamic semantics
    - Stuck states
  - Progress and preservation theorem

Core Dynamic Semantics

- Syntax: **E** ::= **c** | **x** | **(** λ**x. E**$_1$ **)** | **(E**$_1$ **E**$_2$ **)**
  - **c** is integer constant
- Values: **V** ::= λ**x. E**$_1$ | **c**
- A "call by value" semantics:

$(\lambda x.x)\ 1 \rightarrow 1$

$$\frac{}{(\lambda x.\ E)\ V \rightarrow E[V/x]} \qquad \frac{E_1 \rightarrow E_2}{E_1\ E_3 \rightarrow E_2\ E_3} \qquad \frac{E_1 \rightarrow E_2}{V\ E_1 \rightarrow V\ E_2}$$

- Stuck states: terms that are syntactically valid but **aren't values** and **cannot be reduced**
  - E.g., **x**, **x ((**λ**x. x) 1)**, **c c**, **c (**λ**x. 1)**, etc.

3 3

Extensions

Core Typing Rules (Again…)

$$\frac{}{\Gamma \mid\!- \; c : \text{int}}$$

$$\frac{x:\tau \in \Gamma}{\Gamma \mid\!- \; x : \tau}$$

$$\frac{\Gamma, x:\sigma \;\mid\!- \; E_1 : \tau}{\Gamma \mid\!- \; (\lambda x.\, E_1) : \sigma \to \tau}$$

$$\frac{\Gamma \mid\!- \; E_1 : \sigma \to \tau \quad \Gamma \mid\!- \; E_2 : \sigma}{\Gamma \mid\!- \; (E_1\; E_2) : \tau}$$

Type expressions:
τ ::= **int** | $\tau \to \tau$

Environment:
Γ ::= **Nil** | $\Gamma, x:\tau$

Soundness Theorem, Formally

- Definition: **E** can get stuck if there exist an **E'** such that **E** $\rightarrow$* **E'** and **E'** is stuck

- Theorem (Soundness): If **Nil** $\vdash$ **E** : τ and **E** $\rightarrow^n$ **E'**, then **E'** is a value, or **E'** $\rightarrow$ **E''**
  - Lemma (Preservation): If **Nil** $\vdash$ **E** : τ and **E** $\rightarrow$ **E'** then **Nil** $\vdash$ **E'** : τ
  - Lemma (Progress): If **Nil** $\vdash$ **E** : τ then **E** is a value or there exist **E'** such that **E** $\rightarrow$ **E'**

Progress, Proof Sketch

- Induction on the structure of the term **E** (as usual). Assuming Progress holds for component terms, prove that it holds for composite term **E**

Progress, Proof Sketch

4. App: **Nil |- E_1 E_2 : τ**. We have **Nil |- E_1: $\sigma \rightarrow \tau$** and **Nil |- E_2 : σ** or otherwise **E** wouldn't have been well-typed

1. If **E_1** is not a value, then **E_1 → E_3**. (Progress holds for **E_1** by inductive hypothesis.) Thus, **E_1 E_2 → E_3 E_2**
2. If **E_1** is a value but **E_2** is not a value, then **E_2 → E_3**. (Again, Progress holds for **E_2** by the inductive hypothesis.) Thus, **V E_2 → V E_3**
3. Finally, if **E_1** and **E_2** are both values, then **E_1** must be **λx. E_3** (this is actually by a lemma, the Canonical Forms lemma). Thus, evaluation rule **(λx. E_3) V → E_3[V/x]** applies. Done!

Preservation, Proof Sketch

- Similarly, by induction on the structure of term **E**. Assuming Preservation holds for component terms, prove that it holds for term **E**

1. Var: **x** --- …
2. Constant: **Nil |- c : int** --- …
3. Abs: **Nil |- (λx. E_1) : τ** --- …
4. App: **Nil |- (E_1 E_2) : τ** --- … Trickier because need to properly account for substitution!

Soundness

- Soundness, worth restating

- For every state (i.e., term **E**) the program reaches, **E** is well-typed (by Preservation)
- Since **E** is well-typed, then it is either a value, or it can be further reduced (by Progress)
- Therefore, no state the program ever reaches is a “stuck” state

Extensions

- Dynamic semantics and static semantics for
  - Arithmetic,
  - Booleans,
  - Records,
  - Unions,
  - Recursive types,
  - Imperative features,
  - etc., etc.
- Safety = Progress + Preservation

Outline

- The simply typed lambda calculus
  - Syntax
  - Static semantics
  - Dynamic semantics
    - Stuck states
  - Type safety = progress + preservation

- Next time: Simple type inference