Outline
- The simply typed lambda calculus (System $F_1$)
  - Syntax
  - Static semantics
  - Dynamic semantics
    - Stuck states
  - Type safety = progress + preservation
- Simple type inference
- Quiz 4

Reading
- “Types and Programming Languages” by Benjamin Pierce, Chapters 8 and 9
- Lecture notes based on Pierce and notes by Dan Grossman, UW

Type System
- Syntax
- Dynamic semantics
- ‘Stuck’ states
- Static semantics
  - Type environment
  - Typing rules, also called type judgments
  - This is typically referred to as the type system
- Type safety
  - Connects dynamic and static semantics

Type Expressions
- Introducing type expressions
  - $\tau ::= \text{b} | \tau \to \tau$
  - A type is a basic type b (we will only consider int and bool for simplicity), or a function type
- Examples
  - $(\text{int} \to \text{bool}) \to \text{int}$
  - $\text{int} \to (\text{int} \to \text{int})$ is right-associative, thus can write just $\text{int} \to \text{int} \to \text{int}$
- Syntax of simply typed lambda calculus:
  - $E ::= x \mid (\lambda x: \tau. E_1) \mid (E_1, E_2)$
Type Environment and Type Judgments

- A term in the simply typed lambda calculus is
  - Type correct i.e., well-typed, or
  - Type incorrect i.e., ill-typed
- The rules that judge type correctness are given in the form of type judgments in an environment
  - Environment \( \Gamma |- E : \tau \) (\( |- \) is the turnstile)
  - Read: environment \( \Gamma \) entails that \( E \) has type \( \tau \)
- Type judgment
  \[
  \Gamma |- E_1 : \sigma \rightarrow \tau \\
  \Gamma |- E_2 : \sigma \\
  \Gamma |- (E_1 E_2) : \tau
  \]

Static Semantics

- \( x : \tau \in \Gamma \) looks up the type of \( x \) in environment \( \Gamma \)
- Environment \( \Gamma |- E_1 : \sigma \rightarrow \tau \\
  \Gamma |- E_2 : \sigma \\
  \Gamma |- (E_1 E_2) : \tau \) (Application)
- \( \Gamma |- x : \tau \) binding: augments environment \( \Gamma \) with binding of \( x \) to type \( \tau \)
- Environment \( \Gamma |- (\lambda x: \sigma. E_1) : \sigma \rightarrow \tau \) (Abstraction)

Examples

- Can we deduce the type of this term? \( \lambda f. \lambda x. \text{if } x=1 \text{ then } x \text{ else } (f (f (x-1))) : ? \)
  - \( \Gamma |- E_1 : \text{int} \)
  - \( \Gamma |- E_2 : \text{int} \)
  - \( \Gamma |- b : \text{bool} \)
  - \( \Gamma |- \text{if } b \text{ then } E_1 \text{ else } E_2 : \text{int} \)

Examples

- Can we deduce the type of this term? \( (\lambda x. x (\lambda y. y) (x 1)) (\lambda z. z) : ? \)
  - \( x \) cannot have two “different” types
  - \( (x 1) \) demands \( x : \text{int} \rightarrow ? \)
  - \( (x (\lambda y. y)) \) demands \( x : (\tau \rightarrow \tau) \rightarrow ? \)
  - Program does not reach a “stuck state” but is rejected. A sound type system rejects some programs that don’t get stuck

Examples

- In fact, term \( (\lambda x. x (\lambda y. y) (x 1)) (\lambda z. z) : ? \) cannot be typed with a “practical” type system
  - Term \( \text{let } x = (\lambda z. z) \text{ in } x (\lambda y. y) (x 1) \) can be typed
  - We’ll discuss polymorphic type systems after the break
Outline

The simply typed lambda calculus
- Syntax
- Static semantics
- Dynamic semantics
  - Stuck states
  - Type safety = progress + preservation
- Introduction to simple type inference

Core Dynamic Semantics
- Syntax: $E ::= c | x | (\lambda x. E_1) | (E_1 E_2)$
- $c$ is integer constant
- Values: $V ::= \lambda x. E_1 | c$
- The semantics:

  - Stuck states: terms that are syntactically valid but aren't values and cannot be reduced
  - E.g., $x$, $c c$, $c (\lambda x. E_1)$, etc.

Core Typing Rules

| $\Gamma |- E_1 : \sigma \rightarrow \tau$ | $\Gamma |- E_2 : \sigma$ |
|----------------------------------|--------------------------|
| $\Gamma |- (\lambda x: \tau. E_1) : \sigma \rightarrow \tau$ | $\Gamma |- E_2 : \sigma$ |

Type expressions: $\tau ::= \text{int} | \sigma \rightarrow \tau$

Environment: $\Gamma ::= \text{Nil} | \Gamma, x : \tau$

Soundness Theorem, Formally

Definition: $E$ can get stuck if there exist an $E'$ such that $E \xrightarrow{\star} E'$ and $E'$ is stuck

- Theorem (Soundness): If $\text{Nil} |- E : \tau$ and $E \xrightarrow{n} E'$, then $E'$ is a value, or $E' \xrightarrow{\star}$
- Lemma (Preservation): If $\text{Nil} |- E : \tau$ and $E \xrightarrow{\star} E'$ then $\text{Nil} |- E' : \tau$
- Lemma (Progress): If $\text{Nil} |- E : \tau$ then $E$ is a value or there exist $E'$ such that $E \xrightarrow{\star}$

Progress, Proof Sketch

1. Var: $\text{Nil} |- x : \tau$ --- impossible because $\text{Nil} |- E : \tau$
2. Constant: $\text{Nil} |- c : \text{int} \rightarrow \tau$ --- $E$ is a value
3. Abs: $\text{Nil} |- (\lambda x : \tau. E) : \tau$ --- again, $E$ is a value
4. App: $\text{Nil} |- (E_1 E_2) : \tau$

We have $\text{Nil} |- E_1 : \sigma \rightarrow \tau$ and $\text{Nil} |- E_2 : \sigma$ or otherwise $E$ wouldn't have been well-typed. Continued…

Progress, Proof Sketch

4. App: $\text{Nil} |- E_1 E_2 : \tau$. We have $\text{Nil} |- E_1 : \sigma \rightarrow \tau$ and $\text{Nil} |- E_2 : \sigma$ or $E$ wouldn't have been well-typed
   1. If $E_1$ is not a value, then $E_1 \xrightarrow{\star} E_3$, (Progress holds for $E_1$ by inductive hypothesis.) Thus, $E_1 E_2 \xrightarrow{\star} E_3$
   2. If $E_1$ is a value but $E_2$ is not a value, then $E_2 \xrightarrow{\star} E_3$, (Again, Progress holds for $E_1$ by the inductive hypothesis.) Thus, $V E_3 \xrightarrow{\star} V E_3$
   3. Finally, if $E_1$ and $E_2$ are both values, then $E_1$ must be $\lambda x. E_3$ (this is actually by a lemma, the Canonical Forms lemma). Thus, evaluation rule $(\lambda x. E_3) V \rightarrow E_3[V/x]$ 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: \( x : \text{int} \) --- ...
  2. Constant: \( \text{Nil} \vdash c : \text{int} \) --- ...
  3. Abs: \( \text{Nil} \vdash (\lambda x. E_i) : \tau \) --- ...
  4. App: \( \text{Nil} \vdash (E_1 E_2) : \tau \) --- ... Trickier because need to properly account for substitution!

Soundness

- A well-typed term does not get stuck
  - 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,
  - Reference types,
  - etc., etc.
- Safety = Progress + Preservation

Outline

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

Deducing Types

\[ \lambda x : \text{int}. \lambda y : \text{bool}. x \]

1. Abs \( \Gamma = [] \)
   - \( t_1 = \text{int} \rightarrow \text{bool} \rightarrow \text{int} \)
2. Abs \( \Gamma = [x : \text{int}] \)
   - \( t_2 = \text{bool} \rightarrow \text{int} \)
3. Var \( x \Gamma = [x : \text{int}, y : \text{bool}] \)
   - \( t_3 = \text{int} \)

1,2,3 denote the subcomponents of the term. We will be deducing types for each of these components.

Deducing Types

\[ (\lambda f : \text{int} \rightarrow \text{int}. f \ 5) \ (\lambda x : \text{int}. x + 1) : ? \]

(Example term from MIT 2015 Program Analysis OCW)
Type Inference, Strategy 1

- We can figure out all types even without explicit types for variables
  - \((\lambda f\ 5)\ (\lambda x\ .\ x+1)\) : ?
  - Type inference

Type inference, Strategy 1

- Use typing rules to define type constraints
- Solve type constraints (offline)
- Aka constraint-based typing (e.g., Pierce)

We can infer all types!

- \((\lambda f\ 5)\ (\lambda x\ .\ x+1)\) : ?
- 1. App \(\Gamma = \emptyset\)
  - \(t_2 = t_1\rightarrow t_3\)
  - \(t_3 = \text{int} \rightarrow \text{int}\)
  - \(t_4 = \text{int} \rightarrow \text{int}\)
  - \(t_5 = \text{int} \rightarrow \text{int}\)

- 4. Abs \(\Gamma = [x: t_1]\)

- 5. + \(\Gamma = [x: t_1]\)

Another Example

- twice \(f\ x = f\ (f\ x)\)

- What is the type of twice?
  - It is \(t_2 \rightarrow t_1 \rightarrow t_1\) (\(t_1\) is the type of \(f\ (f\ x)\))
  - Based on the syntax tree of \(f\ (f\ x)\) we have:
    - \(t_2 = t_1 \rightarrow t_1\)
    - \(t_1 = t_2 \rightarrow t_2\)

  Thus, \(t_2 \rightarrow t_1 \rightarrow t_2 \rightarrow t_2\)
  - type of twice is \((t_2 \rightarrow t_1) \rightarrow t_1 \rightarrow t_2\)
  - Note: \(t_2\) is a free type variable! Polymorphism!

Type Constraints from Typing Rules, as Attribute Grammar

- Syntax: \(E ::= x \mid c \mid \lambda x.E \mid E_1 \ E_2 \mid E_1 + E_2\)
  - Grammar rule: 
    - \(E ::= x\)
    - \(C_E = \{ t_2 = \Gamma_E(x) \}\)
    - \(E ::= c\)
    - \(C_E = \{ t_2 = \text{int} \}\)
    - \(E ::= \lambda x.E_1\)
    - \(\Gamma_E = \Gamma_E \times t_2\)
    - \(C_E = C_{E_1} U \{ t_2 = t_2 \rightarrow t_{E_1} \}\)
    - \(E ::= E_1 \ E_2\)
    - \(\Gamma_E = \Gamma_E \Gamma_E \Gamma_E = \Gamma_E\)
    - \(C_E = C_{E_1} U C_{E_2} U \{ t_{E_1} = t_{E_2} \rightarrow t_{E_2} \}\)
    - \(E ::= E_1 + E_2\)
    - \(\Gamma_E = \Gamma_E \Gamma_E \Gamma_E = \Gamma_E\)
    - \(C_E = C_{E_1} U C_{E_2} U \{ t_{E_1} = t_{E_2} \rightarrow t_2 \}\)

Type Constraints from Typing Rules, as Attribute Grammar

- Syntax: \(E ::= \lambda x.E_1\)
  - \(\Gamma_E = \Gamma_E \times t_2\)
  - \(C_E = C_{E_1} U \{ t_2 = t_1 \rightarrow t_{E_1} \}\)

- \(t_2\) is "fresh" type variable for term represented by \(E\)'s subterm.

- \(E ::= E_1 \ E_2\)
  - \(\Gamma_E = \Gamma_E \Gamma_E \Gamma_E = \Gamma_E\)
  - \(C_E = C_{E_1} U C_{E_2} U \{ t_{E_1} = t_{E_2} \rightarrow t_2 \}\)

- \(C\) collects constraints. It is synthesized. Propagates bottom-up the tree.
Example

- $\lambda f. \lambda x. f(f\ x)$

Next Class

- Simple type inference
  - Equality constraints
  - Unification

- Polymorphic types
  - Hindley-Milner type inference
  - Algorithm W

Enjoy Your Break!

- Have a nice break and
- Stay healthy