Announcements

- Welcome back!
- Grades in Rainbow Grades
  - Quizzes 1-4
  - Homework 1-3
- HW5 is due Thursday
- HW6, Simple type inference, coming up Thu
- Quiz 4: I apologize!

Outline

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

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 (aka concrete semantics!). In type theory, it is
  - A sequence of reductions
- Static semantics (aka abstract semantics!). In type theory, it is defined in terms of
  - Type environment
  - Typing rules, also called type judgments
  - This is typically referred to as the type system

Example, The Static Semantics. More On This Later!

\[
\begin{align*}
&\text{Variable} & \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \\
&\text{Application} & \frac{\Gamma \vdash E_1 : \sigma \rightarrow \tau \quad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash (E_1 E_2) : \tau} \\
&\text{Abstraction} & \frac{\Gamma \vdash \tau \quad \Gamma, x : \sigma \vdash E_1 : \tau}{\Gamma \vdash (\lambda x. E_1) : \sigma \rightarrow \tau}
\end{align*}
\]
Type System

- A type system either accepts a term (i.e., term is "well-typed"), or rejects it
- Type soundness, also called type safety
  - Well-typed terms never "go wrong"
  - Program never executes an operation on arguments of the wrong type (i.e., not supported)
  - More concretely: well-typed terms never reach a "stuck state" (a "bad" term) during evaluation
    - We must give a definition of "stuck state"
    - A programming language defines its "stuck states"

Stuck States

- "Stuck states" characterize runtime errors
- In real programming languages "stuck states" are called forbidden errors. (Forbidden error is the application of an operation on arguments not supported by the operation.)
  - "Stuck states"/forbidden errors defined for language
  - We will define "stuck states" formally for the simply typed lambda calculus, in just awhile
  - Informally, a term is "stuck" if it is not a value, and it cannot be further reduced
    - E.g. 0 x

Stuck States Examples

- E.g., c (λx.x), where c is an int constant, is a "stuck state", i.e., a meaningless state
- E.g., if c E₁ E₂ where c is an int constant, is a "stuck state"
  - Clearly not a value and clearly no rule applies!
  - Because the evaluation rules for if-then-else are
    - if true E₁ E₂ → E₁
    - if false E₁ E₂ → E₂

Type Soundness

- Remember, a type system accepts or rejects terms
- A sound type system never accepts a term that can get stuck
- A complete type system never rejects a term that cannot get stuck
  - Typically, whether a term gets stuck is undecidable
  - Type systems choose type soundness

Safety = Progress + Preservation

- Progress: A well-typed term is not stuck (i.e., either it is a value, or there is an evaluation step that applies)
- Preservation: If a well-typed term takes a step of evaluation, then the resulting term is also well-typed
- Soundness follows:
  - Each state reached by program is well-typed (by Preservation)
  - A well-typed state is not stuck (by Progress)
  - Thus, each state reached by the program is not stuck

Putting It All Together, Formally

- Simply typed lambda calculus (System F₁)
- Syntax of the simply typed lambda calculus
- The type system (i.e., static semantics): type expressions, environment, and type judgments
- The dynamic semantics
  - Stuck states
  - Progress and preservation, safety theorem
### Type Expressions

- **Introducing type expressions**
  - $\tau ::= b | \tau \rightarrow \tau$
  - A type is a basic type $b$ (we will only consider `int` and `bool` for simplicity), or a function type

- **Examples**
  - `(int → bool) → int`
  - `int → (int → int) //` right-associative, thus can write just `int → int → int`

- **Syntax of simply typed lambda calculus:**
  - $E ::= x | (\lambda x : \tau. E_1) | (E_1 E_2)$

### Type Environment and Type Judgments

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

- The rules that judge type correctness are given in the form of type judgments in an environment.
  - **Environment** $\Gamma |- E : \tau$ ($\Gamma$ 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

- $\Gamma | x : \tau$ looks up the type of $x$ in environment $\Gamma$
  - **Variable**
  - $\Gamma |- x : \tau$

- $\Gamma |- E_1 : \sigma \rightarrow \tau$ $\Gamma |- E_2 : \sigma$ $\Gamma |- (E_1 E_2) : \tau$
  - **Application**
  - binding: augments environment $\Gamma$
  - $\Gamma, x : \sigma |- E_1 : \tau$
  - $\Gamma |- (\lambda x : \sigma. E_1) : \sigma \rightarrow \tau$
  - **Abstraction**

### Examples

- Deduce the type for $\lambda x : int. \lambda y : bool. x + y$ in the `nil` environment
  - No. It gets rightfully rejected. Term reaches a “stuck state” as it applies + on a value of the wrong type ($y$ is `bool`, + is defined on `ints`)

- Is this a valid type? $\text{Nil} |- \lambda x : bool. y : int \rightarrow bool \rightarrow int$
  - No. It gets rightfully rejected. Term reaches a “stuck state” as it applies + on a value of the wrong type ($y$ is `bool`, + is defined on `ints`)

### Extensions

- **Is this a valid type?**
  - $\text{Nil} |- \lambda x : bool. y : \text{int} \rightarrow \text{bool} \rightarrow \text{int}$
  - No. It gets rightfully rejected. Term reaches a “stuck state” as it applies + on a value of the wrong type ($y$ is `bool`, + is defined on `ints`)

- **Is this a valid type?**
  - $\text{Nil} |- \lambda x : \text{bool}. \lambda y : \text{int}. \text{if } x \text{ then } y \text{ else } y + 1 : \text{bool} \rightarrow \text{int} \rightarrow \text{int}$
Examples

Can we deduce the type of this term?
\( \lambda f. \lambda x. \text{if } x=1 \text{ then } x \text{ else } (f (f (x-1))) : ? \)

Γ |- E₁ : int  Γ |- E₂ : int
Γ |- E₁ E₂ : bool
Γ |- E₁ : int  Γ |- E₂ : int
Γ |- E₁ E₂ : int
Γ |- E₁ : τ  Γ |- E₂ : τ
Γ |- if b then E₁ else E₂ : τ

How about this
\((\lambda x. (\lambda y. y) (x 1)) (\lambda z. z) : ? \)

x cannot have two “different” types
- \((x 1)\) demands \(\text{int} \rightarrow ?\)
- \((x (\lambda y. y))\) demands \((\tau \rightarrow \tau) \rightarrow ?\)

Program does not reach a “stuck state” but is nevertheless rejected. A sound type system typically rejects some correct programs

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: \(V ::= \lambda x. E_1 | c\)
- A “call-by-value” semantics:
  \[
  \begin{array}{c}
  (\lambda x. E) V \rightarrow [E[V/x]] \\
  E_1 \rightarrow E_2 \\
  E_1 E_2 \rightarrow E_3 \\
  V E_1 \rightarrow V E_2
  \end{array}
  \]
- Stuck states: terms that are syntactically valid but aren’t values and cannot be reduced
  - E.g., \(x, c c, c (\lambda x. E_1), \text{ etc.}\)

Core Typing Rules

\[
\begin{array}{c}
\Gamma |- c : \text{int} \\
\vdots \Gamma |- x : \tau \\
\vdots \Gamma, x : \tau |- E_1 : \tau \\
\vdots \Gamma |- (\lambda x. E_1) : \sigma \rightarrow \tau \\
\vdots \Gamma |- E_1 : \sigma \rightarrow \tau \\
\vdots \Gamma |- E_2 : \sigma \\
\vdots \Gamma |- (E_1 E_2) : \tau
\end{array}
\]

Type expressions:
\[
\begin{array}{c}
\tau ::= \text{int} | \tau \rightarrow \tau
\end{array}
\]

Environment:
\[
\begin{array}{c}
\Gamma ::= \text{Nil} | \Gamma, x : \tau
\end{array}
\]

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 \(\text{Nil} |- E : \tau\) and \(E \rightarrow^* E’\), then \(E’\) is a value, or \(E’ \rightarrow E”\)
    - Lemma (Preservation): If \(\text{Nil} |- E : \tau\) and \(E \rightarrow E’\) then \(\text{Nil} |- E’ : \tau\)
    - Lemma (Progress): If \(\text{Nil} |- E : \tau\) then \(E\) is a value or there exist \(E’\) such that \(E \rightarrow E’\)
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 $E$
  1. Var: $\text{Nil} \vdash x : \tau$ --- impossible because $\text{Nil} \vdash E : \tau$
  2. Constant: $\text{Nil} \vdash c : \text{int}$ --- $E$ is a value
  3. Abs: $\text{Nil} \vdash (\lambda x. E) : \tau$ --- again, $E$ is a value
  4. App: $\text{Nil} \vdash (E_1 E_2) : \tau$

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

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: $\text{Nil} \vdash c : \text{int}$ --- …
  3. Abs: $\text{Nil} \vdash (\lambda x. E) : \tau$ --- …
  4. App: $\text{Nil} \vdash (E_1 E_2) : \tau$ --- … 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,
  - Reference types,
  - etc., etc.
- Safety = Progress + Preservation

Outline

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

Type Inference, Strategy 1

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

Type inference, Strategy 1

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

Type Constraints

- We constructed a system of type constraints
- Let's solve the system of constraints

\[
\begin{align*}
t_2 &= t_4 + t_3 & t_4 &= \text{int} \rightarrow t_3 \quad t_5 &= \text{int} \rightarrow \text{int} \\
t_2 &= t_4 + t_3 & t_3 &= \text{int} \\
t_3 &= t_2 + t_5 & t_4 &= \text{int} \rightarrow \text{int} \\
t_3 &= t_2 + t_5 & t_5 &= \text{int} \rightarrow t_3 \\
t_5 &= \text{int} \\
\end{align*}
\]

We inferred all \( t \)'s!

\( t_4 = \text{int} \rightarrow \text{int} \rightarrow \text{int} \)
\( t_5 = \text{int} \rightarrow \text{int} \)
\( t_6 = \text{int} \rightarrow \text{int} \)

\( (\lambda f: \text{int} \rightarrow \text{int}. f) 5) (\lambda x: \text{int}. x+1) : \text{int} (t_4) \)

We Can Infer All Types!

\[
\begin{align*}
(\lambda f. f) 5 & \quad (\lambda x. x+1) : ? \\
\text{Var } f & \quad \text{Const } 5 \\
\text{Γ } = [f: \text{int}] & \quad \text{Γ } = [x: \text{int}] \\
\text{Γ } = [\text{int}] & \quad \text{Γ } = [x: \text{int}] \\
\text{Γ } = \[] & \quad \text{Γ } = \[] \\
\end{align*}
\]

Another Example

- \( \text{twice } f \ x = f \ (f \ x) \)
- What is the type of \( \text{twice} \)?
  - It is \( t_1 \rightarrow t_2 \rightarrow t_1 \) (\( t_1 \) is the type of \( f \ (f \ x) \))
- Based on the syntax tree of \( f \ (f \ x) \) we have:
     \[
     \begin{align*}
     t_2 &= t_4 + t_3 \\
     t_4 &= t_2 + t_1 \\
     t_3 &= t_2 + t_1 \\
     \end{align*}
     \]
- Thus, \( t_2 = t_1 = t_2 \)
- \( t_1 \rightarrow t_2 \rightarrow t_1 \)

Note: \( t_2 \) is a free type variable! Polymorphism!
Type Constraints from Typing Rules, as Attribute Grammar

- Syntax: $E ::= x | c | \lambda x. E | E_1 E_2 | E_1 + E_2$
- Grammar rule: $E ::= \lambda x. E_1$
  - $\Gamma_{E_1} = \Gamma_E; x:t$
  - $C_E = C_{E_1} U C_{E_2} U \{ t_{E_1} = t_{E_2} \rightarrow t_E \}$
  - $C$ collects constraints. It is synthesized. Propagates bottom-up the tree.

$E ::= E_1 E_2$
  - $\Gamma_{E_1} = \Gamma_E$
  - $\Gamma_{E_2} = \Gamma_E$
  - $C_E = C_{E_1} U C_{E_2} U \{ t_{E_1} = t_{E_2} \rightarrow t_E \}$

$E ::= E_1 + E_2$
  - $\Gamma_{E_1} = \Gamma_E$
  - $\Gamma_{E_2} = \Gamma_E$
  - $C_E = C_{E_1} U C_{E_2}$

Example

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

Next Class

- Simple type inference
  - Equality constraints
  - Unification
- Polymorphic types
- Hindley-Milner type inference
  - Algorithm W

Γ is inherited. Propagates top-down the tree.

$E ::= \lambda x. E_1$
  - $\Gamma_{E_1} = \Gamma_E; x:t$
  - $C_E = C_{E_1} U \{ t_{E_1} = \text{int}, t_{E_2} = \text{int}, t_E = \text{int} \}$

$E ::= E_1 E_2$
  - $\Gamma_{E_1} = \Gamma_E$
  - $\Gamma_{E_2} = \Gamma_E$
  - $C_E = C_{E_1} U C_{E_2} U \{ t_{E_1} = t_{E_2} \rightarrow t_E \}$

$E ::= E_1 + E_2$
  - $\Gamma_{E_1} = \Gamma_E$
  - $\Gamma_{E_2} = \Gamma_E$
  - $C_E = C_{E_1} U C_{E_2}$

$t_E$ is “fresh” type variable for term represented by $E$’s subtree.