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!

\[
g \vdash x : \tau \\
g \vdash \text{Variable}
\]

\[
g \vdash E_1 : \sigma \rightarrow \tau \quad g \vdash E_2 : \sigma \\
g \vdash (E_1 \ E_2) : \tau \\
\text{Application}
\]

\[
g \vdash E_1 : \sigma \\
\vdash \text{binding: augments environment } g \\
\vdash \text{with binding of } x \text{ to type } \sigma \\
g \vdash \lambda x : \sigma . E_1 : \sigma \rightarrow \tau \\
\vdash \text{Abstraction}
\]
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**
  - \( \tau ::= b \mid \tau \rightarrow \tau \)
  - A type is a basic type \( b \) (we will only consider `int` and `bool` for simplicity), or a function type

- **Examples**
  - \((\text{int} \rightarrow \text{bool}) \rightarrow \text{int}\)
  - `int` \( \rightarrow \) `(\text{int} \rightarrow \text{int})` // is right-associative, thus can write just `int \rightarrow int \rightarrow 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

- The rules that judge type correctness are given in the form of type judgments in an environment

\[
\Gamma |- E : \tau \quad (\text{-} \text{is the turnstile})
\]

- Read: environment \( \Gamma \) entails that \( E \) has type \( \tau \)

Examples

- Deduce the type for \( \lambda x: \text{int}. \lambda y: \text{bool}. x+y \) in the nil environment

Examples

- Is this a valid type?
  - \( \text{Nil} |- \lambda x: \text{int}. \lambda y: \text{bool}. x+y : \text{int} \rightarrow \text{bool} \rightarrow \text{int} \)
  
  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?
\[ \lambda x. \text{if } x=1 \text{ then } x \text{ else } (f \ (f \ (x-1))) : ? \]

Spring 19 CSCI 4450/6450, A. Milanova (example from MIT 2015 Program Analysis OCW)

\( \Gamma |- E_1 : \text{int} \quad \Gamma |- E_2 : \text{int} \)

\( \Gamma |- E_1; E_2 : \text{bool} \)

\( \Gamma |- E_1 : \text{int} \quad \Gamma |- E_2 : \text{int} \)

\( \Gamma |- \text{if } b \text{ then } E_1 \text{ else } E_2 : \text{int} \)

\( \Gamma |- b : \text{int} \)

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: \( E ::= c \mid x \mid (\lambda x. E_1) \mid (E_1 E_2) \)
  - \( c \) is integer constant
- Values: \( V ::= \lambda x. E_1 \mid c \)
- A “call-by-value” semantics:
  - \( \lambda x. E \to E[V/x] \)
  - \( E_1 \to E_2 \)
  - \( V \to V \)
- Stuck states: terms that are syntactically valid but are not values and cannot be reduced
  - E.g., \( \cdot \), \( c \cdot c \), \( c \ (\lambda x. E_1) \), etc.

Core Typing Rules

| \( \Gamma |- c : \text{int} \) | Type expressions: \( \tau ::= \text{int} \mid \tau \to \tau \) |
|------------------------------|--------------------------------------------------|
| \( x : \tau \in \Gamma \)    | Environment: \( \Gamma ::= \text{Nil} \mid \Gamma, x : \tau \) |
| \( \Gamma |- x : \tau \)     | \( \Gamma |- (\lambda x. E_1) : \sigma \to \tau \) |
| \( \Gamma |- (E_1 E_2) : \tau \) | \( \Gamma |- E_1 : \sigma \to \tau \) \quad \( \Gamma |- E_2 : \sigma \) |

Soundness Theorem, Formally

- Definition: \( E \) can get stuck if there exist an \( E' \) such that \( E \to^{*} E' \) and \( E' \) is stuck

- Theorem (Soundness): If \( \text{Nil} |- E : \tau \) and \( E \to^{*} E' \), then \( E' \) is a value, or \( E' \to E'' \)
  - Lemma (Preservation): If \( \text{Nil} |- E : \tau \) and \( E \to E' \) then \( \text{Nil} |- E' : \tau' \)
  - Lemma (Progress): If \( \text{Nil} |- E : \tau \) then \( E \) is a value or there exist \( E' \) such that \( E \to E' \)
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**: $\text{Nil} \vdash (\lambda x. E_i) : \tau$ --- 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**: $x --- \ldots$
  2. **Constant**: $\text{Nil} \vdash c : \text{int} --- \ldots$
  3. **Abs**: $\text{Nil} \vdash (\lambda x. E_i) : \tau$ --- \ldots
  4. **App**: $\text{Nil} \vdash (E_1 E_2) : \tau$ --- \ldots 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

- λx: int. λy: bool. x

1. Abs $\Gamma = []$
   - $t_1 = \text{int} \rightarrow \text{bool} \rightarrow \text{int}$
   - $\lambda x: \text{int}$
   - $t_2 = \text{bool} \rightarrow \text{int}$
   - $\lambda y: \text{bool}$
   - $t_3 = \text{int}$
   - Var $x \Gamma = [x:]$ int
   - $t_4 = \text{int}$

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)

Deducing Types

$(\lambda f. f) \ (\lambda x. x+1) : ?$

We Can Infer All Types!

We inferred all types!

Another Example

- twice $f \ x = f \ (f \ x)$
- What is the type of twice?
  - It is $t_1 \rightarrow t_2 \rightarrow t_3$ ($t_1$ is the type of $f \ (f \ x)$)
  - Based on the syntax tree of $f \ (f \ x)$ we have:
    - $t_1 = t_2 \rightarrow t_3$
    - $t_2 = t_4 \rightarrow t_5$
    - $t_3 = t_6 \rightarrow t_7$
  - Thus, $t_2 = t_1 = t_2$, $t_3 = t_9 \rightarrow t_8$ and
  - type of twice is $t_9 \rightarrow t_8 \rightarrow t_7$
  - Note: $t_5$ is a free type variable! Polymorphism!
**Type Constraints from Typing Rules, as Attribute Grammar**

- **Syntax:**
  \[ E ::= x \mid c \mid \lambda x. E \mid E_1 \cdot E_2 \mid E_1 + E_2 \]

- **Grammar rule:**
  \[ C_E = \{ t_E = \Gamma_E(x) \} \]

- **Attribute rule:**
  \[ C_E = \{ t_E = \text{int} \} \]

- **Example:**
  \[ \lambda f. \lambda x. f (f x) \]

**Type Constraints from Typing Rules, as Attribute Grammar**

- **Grammar rule:**
  \[ \Gamma_E = \Gamma_{E_1} \cdot \Gamma_{E_2} \]

- **Attribute rule:**
  \[ C_E = C_{E_1} \cup C_{E_2} \cup \{ t_{E_1} = t_{E_2} \} \]

- **Example:**
  \[ \Gamma_f = \Gamma_{E_1} \cdot \Gamma_{E_2} \]

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

---

Spring 19 CSCI 4450/6450, A. Milanova

39

Spring 19 CSCI 4450/6450, A. Milanova

40