Simply Typed Lambda Calculus, Progress and Preservation

Announcements
- Grades in Rainbow Grades
- Quizzes 1-4
- Homework 1-3
- HW5
  - Will extend couple of days
- Change in schedule
  - Moved all paper presentations to the end of term
  - List of papers, schedule and instructions coming up. If you have specific requests, please email!

Outline
- Applied lambda calculus
- Introduction to types and type systems
- The simply typed lambda calculus
- Syntax
- Dynamic semantics
- Static semantics
- Type safety

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

Applied Lambda Calculus (from Sethi)
- $E ::= c | x | (\lambda x.E_1) | (E_1 E_2)$
  - Augments the pure lambda calculus with constants.
  - An applied lambda calculus defines its set of constants and reduction rules. For example:
    - Constants:
      - if, true, false
        - (all these are $\lambda$ terms, e.g., true=$\lambda x.\lambda y. x$)
      - 0, iszero, pred, succ
    - Reduction rules:
      - if true $MN \rightarrow \delta M$
      - if false $MN \rightarrow \delta N$
      - iszero 0 $\rightarrow \delta$ true
      - iszero (succ 0) $\rightarrow \delta$ false, k>0
      - iszero (pred 0) $\rightarrow \delta$ false, k>0
      - succ (pred M) $\rightarrow \delta M$
      - pred (succ M) $\rightarrow \delta M$

From an Applied Lambda Calculus to a Functional Language

<table>
<thead>
<tr>
<th>Construct</th>
<th>Applied $\lambda$-Calculus</th>
<th>Our Language (ML)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>$x$</td>
<td>$x$</td>
</tr>
<tr>
<td>Constant</td>
<td>$c$</td>
<td>$c$</td>
</tr>
<tr>
<td>Application</td>
<td>$MN$</td>
<td>$fun \ x \Rightarrow M$</td>
</tr>
<tr>
<td>Abstraction</td>
<td>$\lambda x.M$</td>
<td>$k$</td>
</tr>
<tr>
<td>Integer</td>
<td>succ 0, k&gt;0</td>
<td>pred 0, k&gt;0</td>
</tr>
<tr>
<td>Conditional</td>
<td>if P M N</td>
<td>if P then M else N</td>
</tr>
<tr>
<td>Let</td>
<td>($\lambda x.M$ N)</td>
<td>let val $x = N$ in M end</td>
</tr>
</tbody>
</table>
Aside: The Fixed-Point Operator

- One more constant, and one more rule:
  \[ \text{fix} \quad \text{fix } \text{M} \rightarrow \delta \text{M} (\text{fix M}) \]

- Needed to define recursive functions:
  \[ \text{plus } x \ y = \begin{cases} y & \text{if } x = 0 \\ \text{plus} (\text{pred} x) (\text{succ} y) & \text{otherwise} \end{cases} \]

- Therefore:
  \[ \text{plus} = \lambda x. \lambda y. \text{if} (\text{iszero} \ x) y (\text{plus} (\text{pred} x) (\text{succ} y)) \]

Aside: The Fixed-Point Operator

- But how do we define \textit{plus}?

Define \textit{plus} = \textit{fix M}, where
\[ M = \lambda f. \lambda x. \lambda y. \text{if} (\text{iszero} \ x) y (f (\text{pred} x) (\text{succ} y)) \]

We must show that
\[ \text{fix M} = \delta \lambda x. \lambda y. \text{if} (\text{iszero} \ x) y ((\text{fix M}) (\text{pred} x) (\text{succ} y)) \]

Aside: The Y Combinator

- \text{fix} is, of course, a lambda expression!

- One possibility, the famous Y-combinator:
  \[ Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \]

Exercise: define \textit{factorial} = ?

Show that \[ Y \ M \] indeed beta-reduces to \[ M (Y \ M) \]

Think of \textit{fix}, or \text{Y} as the function that takes a function \text{M} and returns \text{M}((\text{M}(\text{M}(\text{M}(\text{M}(\text{M}(\text{M}(...)))))))

Types!

- Constants add power
- But they raise problems because they permit “bad” terms such as
  - \[ \text{if} (\lambda x. x) \ y \ z \] (arbitrary function values are not permitted as predicates, only true/false values)
  - \[ 0 \ x \] (0 does not apply as a function)
  - \text{succ} \text{true} (undefined in our language)
  - \text{plus} \text{true} (\lambda x. x) etc.

Why types?
- Safety. Catch semantic errors early
- Data abstraction. Simple types and ADTs
- Documentation (statically-typed languages only)
  - Type signature is a form of specification!
- Statically typed vs. dynamically typed languages
- Type annotations vs. type inference
- Type safe vs. type unsafe
Types!

- Important subarea of programming languages, program analysis
- Related to abstract interpretation, although...
  - AI is framework of choice for reasoning about imperative languages
  - Type systems are framework of choice for reasoning about functional languages
- Type systems and extensions to reason about imperative programs

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!

\[
\begin{align*}
\Gamma &\vdash \text{x : } \tau \\
\Gamma &\vdash \text{(E₁ E₂) : } \tau
\end{align*}
\]

(Variable)

\[
\begin{align*}
\Gamma &\vdash \text{x : } \sigma \\
\Gamma &\vdash \text{E₁ : } \sigma \to \tau \\
\Gamma &\vdash \text{E₂ : } \sigma
\end{align*}
\]

(Application)

\[
\begin{align*}
\Gamma &\vdash (\lambda x : \sigma. \text{E₁}) : \sigma \to \tau
\end{align*}
\]

(Abstraction)

Stuck States

- Informally, a term is “stuck” if it cannot be further reduced (i.e., no reduction rule applies) and it is not a value (an int constant or a function value)
- “Stuck states” characterize runtime errors
- In real programming languages “stuck states” correspond to forbidden errors such as seg faults, execution of operation on illegal arguments, etc.
- We will define “stuck states” formally for the simply typed lambda calculus, in just awhile

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: type expressions, environment, and type judgments
  - The dynamic semantics
  - Stuck states
  - Progress and preservation theorem

Type Expressions

- Introducing type expressions
  - τ ::= b | τ → τ
  - A type is a basic type b (we will only consider int), or a function type
- Examples
  - int
  - int → (int → int) // → is right-associative, thus can write just int → int → int
- Syntax of simply typed lambda calculus:
  - E ::= x | (λx:τ. E₁) | (E₁ E₂)

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 Γ |- E : τ
  - Read: environment Γ entails that E has type τ
  - Type judgment
    Γ |- E₁ : σ → τ
    Γ |- E₂ : σ
    Premises
    Γ |- (E₁ E₂) : τ
    Conclusion

Semantics

- x:τ ∈ Γ
  - looks up the type of x in environment Γ
  - (Variable)
- Γ |- E₁ : σ → τ
  - Γ |- E₂ : σ
  - Γ |- (E₁ E₂) : τ
  - (Application)
- Γ, x:σ |- E₁ : τ
  - binding: augments environment Γ with binding of x to type σ
  - Γ, x:σ |- (λx:σ. E₁) : σ → τ
  - (Abstraction)
Examples

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

Examples

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

Examples

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

Examples

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

Examples

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

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:

Core Dynamic Semantics

- Stuck states: terms that are syntactically valid but are not \text{values} and cannot be \text{reduced}
  - \( E.g., \ c \ c \ c \ c \ (\lambda x. E_1) \), etc.
Core Typing Rules

\[ \begin{align*}
\Gamma | c & : \text{int} \\
\Gamma | x & : \tau \\
\Gamma, x : \tau & | E_1 : \tau \\
\Gamma | \lambda x. E & : \tau \\
\Gamma | E_1 : \sigma \rightarrow \tau, \Gamma | E_2 : \sigma & | E : \tau
\end{align*} \]

Type expressions:
\[ \tau ::= \text{int} | \tau \rightarrow \tau \]

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

Soundness Theorem, Formally

- Definition: E can get stuck if there exist an E' such that E \rightarrow^* E' and E' is stuck.
- Theorem (Soundness): If Nil |- E : \tau and E \rightarrow^* E', then E' is a value or E' \rightarrow E''.
- Lemma (Preservation): If Nil |- E : \tau and E \rightarrow E' then Nil |- E' : \tau.
- Lemma (Progress): If 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: Nil |- x : \tau --- impossible because Nil |- E : \tau.
2. Constant: Nil |- c : \text{int} --- E is a value.
3. Abs: Nil |- \lambda x. E : \tau --- again, E is a value.
4. App: Nil |- E_1 E_2 : \tau

We have Nil |- E_1 : \sigma \rightarrow \tau and Nil |- 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: Nil |- c : \text{int} --- …
3. Abs: Nil |- \lambda x. E : \tau --- …
4. App: Nil |- 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 reaches is “stuck”
Extensions

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

Next Class

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

Enjoy Your Break!