Simply Typed Lambda Calculus, Progress and Preservation

Announcements
- Grades in Rainbow Grades
- Quizzes 1-3
- Homework 1-2
- HW5, due Thursday after the break
- Quiz 4

Monad Quote
- “A monad is just a monoid in the category of endofunctors, what’s the problem?”
- Monad type class and the monad laws
- Maybe monad
- List monad
- IO monad
- State monad

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 | (λx.E₁) | (E₁ E₂)
  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 λ terms, e.g., true=λx.λy. x)
  - 0, iszero, pred, succ
  Reduction rules:
  - if true M N → M
  - if false M N → N
  - iszero 0 → true
  - iszero (succ 0) → false, k>0
  - iszero (pred 0) → false, k>0
  - succ (pred M) → M
  - pred (succ M) → M
### From an Applied Lambda Calculus to a Functional Language

<table>
<thead>
<tr>
<th>Construct</th>
<th>Applied $\lambda$-Calculus</th>
<th>A 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>$M N$</td>
<td>$M N$</td>
</tr>
<tr>
<td>Abstraction</td>
<td>$\lambda x. M$</td>
<td>$\text{fun } x \Rightarrow M$</td>
</tr>
<tr>
<td>Integer</td>
<td>$\text{succ}^k 0, k &gt; 0$</td>
<td>$k$</td>
</tr>
<tr>
<td></td>
<td>$\text{pred}^k 0, k &gt; 0$</td>
<td>$-k$</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 $\text{val } x = N$ in $M$ end</td>
</tr>
</tbody>
</table>

#### Aside: The Fixed-Point Operator

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

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

#### Aside: The Fixed-Point Operator

- But how do we define $\text{plus}$?

Define $\text{plus } = \text{fix } M$, 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:

\[
Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))
\]

Show that $Y M$ indeed beta-reduces to $M \left(\text{Y } M\right)$

#### Types!

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

- 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

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!

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”
  - More concretely: well-typed terms never reach a “stuck state” (a “bad” term) during evaluation
    - We must give a definition of “stuck state”
    - Each programming language defines its own set of “stuck states”

Stuck States

- Informally, a term is “stuck” if it cannot be further reduced and it is not a value
  - E.g., 0 x
- “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., \( c (\lambda x.x) \), where \( c \) is an int constant, is a "stuck state", i.e., a meaningless state
- E.g., if \( c E_1 E_2 \) 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 \( \text{true} \) \( E_1 \rightarrow \delta E_1 \)
    - if \( \text{false} \) \( E_1 \rightarrow \delta E_2 \)

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

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 \( \Gamma \vdash E : \tau \) (\( \vdash \) is the turnstile)
  - Read: environment \( \Gamma \) entails that \( E \) has type \( \tau \)
  - Type judgment
    - Premises
      - \( \Gamma \vdash E_1 : \sigma \rightarrow \tau \)
      - \( \Gamma \vdash E_2 : \sigma \)
    - Conclusion
      - \( \Gamma \vdash (E_1, E_2) : \tau \)
Semantics

- \( x : \tau \in \Gamma \) \[\Gamma \vdash x : \tau\] (Variable)
- \( \Gamma \vdash E_1 : \sigma \), \( \Gamma \vdash E_2 : \sigma \) \[\Gamma \vdash (E_1 E_2) : \tau\] (Application)
- \( \Gamma \vdash E_1 : \sigma \) \[\Gamma \vdash (\lambda x : \sigma . E_1) : \sigma \rightarrow \tau\] (Abstraction)

Binding: augments environment \( \Gamma \) with binding of \( x \) to type \( \sigma \)

Examples

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

Examples

- Is this a valid type?
  - \( \text{Nil} \vdash \lambda x : \text{int} \cdot \lambda y : \text{bool} . x + y : \text{int} \rightarrow \text{bool} \rightarrow \text{int} \)
  - 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

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

- \( x \) cannot have two “different” types
  - \( (x \ 1) \) demands \( \text{int} \rightarrow ? \)
  - \( (x \ (x \ y)) \) 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
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:
  \[
  \begin{array}{c}
  \text{E}_1 \to \text{E}_3 \quad \text{E}_1 \to \text{E}_2 \\
  \hline
  (\lambda x. E) \to E[V/x] \\
  \end{array}
  \]
- Stuck states: terms that are syntactically valid but aren’t values and cannot be reduced
  - E.g., $x, c, c (\lambda x. E)$, etc.

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 $\Gamma \vdash E : \tau$ and $E \to^* E'$, then $E'$ is a value, or $E'$ is $E''$
  - Lemma (Preservation): If $\Gamma \vdash E : \tau$ and $E \to E'$ then $\Gamma \vdash E' : \tau$
  - Lemma (Progress): If $\Gamma \vdash E : \tau$ then $E$ is a value or there exist $E'$ such that $E \to E'$

Progress, Proof Sketch

4. App: $\text{Nil} \vdash E_1 E_2 : \tau$. We have $\text{Nil} \vdash E_1 : \sigma \to \tau$ and $\text{Nil} \vdash E_2 : \sigma$ or otherwise $E$ wouldn’t have been well-typed
  1. If $E_1$ is not a value, then $E_1 \to E_3$. (Progress holds for $E_1$ by inductive hypothesis.) Thus, $E_1 E_2 \to E_3 E_3$
  2. If $E_1$ is a value but $E_2$ is not a value, then $E_2 \to E_3$. (Again, Progress holds for $E_2$ by the inductive hypothesis.) Thus, $V E_2 \to V E_3$
  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 $(\lambda x. E_1) V \to E_3[V/x]$ 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: $x \to^*$ …
  2. Constant: $\text{Nil} \vdash c : \text{int} \to^*$ ...
  3. Abs: $\text{Nil} \vdash (\lambda x. E_1) : \tau \to^*$ again, $E$ is a value
  4. App: $\text{Nil} \vdash E_1 E_2 : \tau$ or otherwise $E$ wouldn’t have been well-typed. Continued…
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

Next Class

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

Enjoy Your Break!