Simply Typed Lambda Calculus, Progress and Preservation
Announcements

- HW5 on Submitty
  - Questions?

- Grading HW4

- Check your Rainbow grades
Outline

- Applied lambda calculus
- Introduction to types and type systems
  - Simply typed lambda calculus (System $F_1$)
  - 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)

- $E ::= c \mid x \mid (\lambda x.E_1) \mid (E_1 E_2)$

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 $M N \rightarrow_{\delta} M$
- if false $M N \rightarrow_{\delta} N$
- iszero 0 $\rightarrow_{\delta} true$
- iszero $(\text{succ}^k 0) \rightarrow_{\delta} false$, $k>0$
- iszero $(\text{pred}^k 0) \rightarrow_{\delta} false$, $k>0$
- succ $(\text{pred} M) \rightarrow_{\delta} M$
- pred $(\text{succ} M) \rightarrow_{\delta} M$
From an Applied Lambda Calculus to a Functional Language

<table>
<thead>
<tr>
<th>Construct</th>
<th>Applied λ-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 ( \lambda x. M )</td>
</tr>
<tr>
<td>Abstraction</td>
<td>( \lambda x. M )</td>
<td>fun x =&gt; 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 val x = N in M end</td>
</tr>
</tbody>
</table>
The Fixed-Point Operator

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

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

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

Define \texttt{plus} = \texttt{fix} \ M, where

\( M = \lambda f. \lambda x. \lambda y. \text{if} \ (\text{iszero} \ x) \ \text{y} \ (f \ (\text{pred} \ x) \ (\text{succ} \ y)) \)

Then show that

\[
\text{fix} \ M =_{\delta \beta} \lambda x. \lambda y. \text{if} \ (\text{iszero} \ x) \ \text{y} \ ((\text{fix} \ M) \ (\text{pred} \ x) \ (\text{succ} \ y))
\]

\[
(fix \ M) \rightarrow M (fix \ M) = \\
(\lambda f. \lambda x. \lambda y. \cdots) (fix \ M) \rightarrow \cdots
\]
The Fixed-Point Operator

Define **times** =

\[ \text{fix } \lambda f. \lambda x. \lambda y. \text{if} \ (\text{iszero} \ x) \ 0 \ \text{plus} \ y \ (f \ (\text{pred} \ x) \ y) \]

Exercise: define **factorial** = ?
The Y Combinator

- **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 reduces into \( M (Y M) \)

\[ Y M \rightarrow? M (Y M) \]

\[ (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) M \rightarrow \]

\[ (\lambda x. M (x x)) (\lambda x. M (x x)) \rightarrow \]

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

Program Analysis CSCI 4430/6450, A Milanova
Types!

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

- Important subarea of programming languages and program analysis

- Related to abstract interpretation, although…
  - AI is framework of choice for reasoning about imperative languages
  - Type systems is framework of choice for reasoning about functional languages
Type System

- Syntax
- Dynamic semantics (aka concrete semantics!). In type theory, it is
  - A sequence of reductions  \( E \rightarrow E_1 \rightarrow E_2 \rightarrow \ldots \rightarrow E_n \)
- 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!

\[ \Gamma \vdash x : \tau \]

- \( x : \tau \in \Gamma \)
- looks up the type of \( x \) in environment \( \Gamma \)

\[ \Gamma = [x : \text{int}, y : \text{int} \rightarrow \text{int}] \]

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

\[ \Gamma \vdash E_1 : \sigma \rightarrow \tau \quad \Gamma \vdash E_2 : \sigma \]

- (Application)

\[ \Gamma \vdash (E_1 \; E_2) : \tau \]

\[ \Gamma, x : \sigma \vdash E_1 : \tau \]

- (Abstraction)

\[ \Gamma \vdash (\lambda x : \sigma. \; E_1) : \sigma \rightarrow \tau \]

\[ \llbracket \Gamma \rrbracket \vdash \lambda x : \text{int}, \lambda y : \text{bool}. \; x \]

Program Analysis CSCI 4450/6450, A Milanova
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
Informally, a term is “stuck” if it cannot be further reduced, and it is not a value.

- E.g, \( 0 \times \text{True} + 5 \)

In real programming languages stuck states correspond to forbidden errors which is execution of operation on illegal arguments.

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 true $E_1 \ E_2 \rightarrow_\delta E_1$
    - if false $E_1 \ E_2 \rightarrow_\delta E_2$
Type Soundness

- Remember, a type system either accepts a term $M$ or rejects $M$
- A **sound type system** never accepts a term that can get stuck $E \rightarrow E_1 \rightarrow E_2 \rightarrow \ldots$
- 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**
Type Soundness

Sound:

- All terms

Terms that do not reach a stuck state

Accepted

Terms that don't get stuck

Accepted by C++

Complete:

- All terms

Terms that don't get stuck

Accepted
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 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