Simply Typed Lambda Calculus, Progress and Preservation
Announcements

- HW5?
- HW6 is posted
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)

- \[ E ::= c | x | ( \lambda x. E_1 ) | ( E_1 E_2 ) \]

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

<table>
<thead>
<tr>
<th>Construct</th>
<th>Applied (\lambda)-Calculus</th>
<th>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>(MN)</td>
<td>(MN)</td>
</tr>
<tr>
<td>Abstraction</td>
<td>(\lambda x.M)</td>
<td>(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>(\text{if } P \ M \ N)</td>
<td>(\text{if } P \ \text{then } M \ \text{else } N)</td>
</tr>
<tr>
<td>Let</td>
<td>((\lambda x.M) \ N)</td>
<td>(\text{let val } x = N \ \text{in } M \ \text{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 **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

$$\text{fix } M =_{\delta \beta} \lambda x. \lambda y. \text{if } (\text{iszero } x) \ y \ ((\text{fix } M) \ (\text{pred } x) \ (\text{succ } y))$$
The Fixed-Point Operator

Define \textbf{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 \textbf{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 \( M (Y \ M) \)
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)
  - \( \text{succ true} \) (undefined in our language)
  - \( \text{plus true 0} \) 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
- 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!

\[
\begin{align*}
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} & \quad \text{(Variable)} \\
\frac{\Gamma \vdash E_1 : \sigma \rightarrow \tau \quad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash (E_1 \ E_2) : \tau} & \quad \text{(Application)} \\
\frac{\Gamma \vdash (\lambda x : \sigma. \ E_1) : \sigma \rightarrow \tau}{\Gamma \vdash (\lambda x : \sigma. \ E_1) : \sigma \rightarrow \tau} & \quad \text{(Abstraction)}
\end{align*}
\]
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”
  - 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 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 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 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 $F_1$)
- 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
  - \( \tau ::= b \mid \tau \to \tau \)
  - A type is a basic type \( b \) (we will only consider \texttt{int}, for simplicity), or a function type

- Examples
  - \texttt{int}
  - \texttt{int \to (int \to int)} // \( \to \) is right-associative, thus can write just \texttt{int \to int \to 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
  - Environment \( \Gamma |- E : \tau \) (|- is the turnstile)
  - Read: environment \( \Gamma \) entails that \( E \) has type \( \tau \)
- Type judgment
  \[
  \frac{\Gamma |- E_1 : \sigma \rightarrow \tau \quad \Gamma |- E_2 : \sigma}{\Gamma |- (E_1 \ E_2) : \tau}
  \]
  - Premises
  - Conclusion
Semantics

\[ \frac{\text{x:}\tau \in \Gamma}{\Gamma \vdash \text{x:}\tau} \]  
(Variable)

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

\[ \Gamma, \text{x:}\sigma \vdash E_1 : \tau \]  
(binding: augments environment \( \Gamma \) with binding of \( \text{x} \) to type \( \sigma \))

\[ \frac{\Gamma \vdash (\lambda x:\sigma. E_1) : \sigma \rightarrow \tau}{\Gamma \vdash (E_1) : \sigma \rightarrow \tau} \]  
(Abstraction)
Examples

- Deduce the type for

\[ \lambda x: \text{int}. \lambda y: \text{bool}. \ x \] in the \textit{nil} environment
Examples

- Deduce the type for

  \[ \lambda x: \text{int}. \lambda y: \text{bool}. \ x \text{ in the nil environment} \]
Extensons

\[
\frac{\Gamma |- c : \text{int}}{\Gamma |- \text{c} : \text{int}}
\]

\[
\frac{\Gamma |- E_1 : \text{int} \quad \Gamma |- E_2 : \text{int}}{\Gamma |- E_1 + E_2 : \text{int}}
\]

\[
\frac{\Gamma |- E_1 : \text{int} \quad \Gamma |- E_2 : \text{int}}{\Gamma |- \text{E}_1 = E_2 : \text{bool}}
\]

\[
\frac{\Gamma |- b : \text{bool} \quad \Gamma |- E_1 : \tau \quad \Gamma |- E_2 : \tau}{\Gamma |- \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau}
\]

(Comparison)
Examples

- Is this a valid type?
  \[ \text{Nil} \vdash \lambda x:\text{int}.\lambda y:\text{bool}.\ x+y\ :\ \text{int} \to \text{bool} \to \text{int} \]
  
  - 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?
  \[ \text{Nil} \vdash \lambda x:\text{bool}.\lambda y:\text{int}.\ \text{if } x \text{ then } y \text{ else } y+1\ :\ \text{bool} \to \text{int} \to \text{int} \]
Examples

Is this a valid type?

\[ \text{Nil} |- \lambda x: \text{bool.} \lambda y: \text{int. if } x \text{ then } y \text{ else } y+1 : \text{bool} \rightarrow \text{int} \rightarrow \text{int} \]
Can we deduce the type of this term?

\( \lambda f. \lambda x. \text{if } x=1 \text{ then } x \text{ else } (f (f (x-1))) : ? \)

\[
\begin{align*}
\Gamma |- E_1 : \text{int} & \quad \Gamma |- E_2 : \text{int} \\
\hline \\
\Gamma |- E_1=E_2 : \text{bool} \\
\hline \\
\Gamma |- E_1 : \text{int} & \quad \Gamma |- E_2 : \text{int} \\
\hline \\
\Gamma |- E_1+E_2 : \text{int} \\
\hline \\
\Gamma |- b : \text{bool} & \quad \Gamma |- E_1 : \tau & \quad \Gamma |- E_2 : \tau \\
\hline \\
\Gamma |- \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau
\end{align*}
\]
Examples

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

- \(x\) cannot have two “different” types
  - \((x\ 1)\) demands \texttt{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
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:

  $$(\lambda x. E) V \Rightarrow E[V/x]$$
  $$E_1 \Rightarrow E_2$$
  $$E_1 E_3 \Rightarrow E_2 E_3$$
  $$V E_1 \Rightarrow V E_2$$

- Stuck states: terms that are syntactically valid but aren’t values and cannot be reduced
  - E.g., $x$, $c$, $c$, $c (\lambda x. E_1)$, etc.
Core Typing Rules

\[
\begin{align*}
\Gamma |- c : \text{int} \\
\chi : \tau \in \Gamma \\
\Gamma |- \chi : \tau \\
\Gamma, \chi : \sigma |- E_1 : \tau \\
\Gamma |- (\lambda \chi. E_1) : \sigma \rightarrow \tau \\
\Gamma |- E_1 : \sigma \rightarrow \tau \quad \Gamma |- E_2 : \sigma \\
\Gamma |- (E_1 \ E_2) : \tau
\end{align*}
\]

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

Environment:
\[\Gamma ::= \text{Nil} | \Gamma, \chi : \tau\]
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 $\text{Nil} \vdash E : \tau$ and $E \Rightarrow^n E'$, then $E'$ is a value, or $E' \Rightarrow E''$
  - Lemma (Preservation): If $\text{Nil} \vdash E : \tau$ and $E \Rightarrow E'$ then $\text{Nil} \vdash E' : \tau$
  - Lemma (Progress): If $\text{Nil} \vdash E : \tau$ then $E$ is a value or there exist $E'$ such that $E \Rightarrow E'$
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: $\text{Nil} \vdash (\lambda x. E_1) : \tau$ --- 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…
4. App: \( \text{Nil} \ |- \ E_1 \ E_2 : \tau \). We have \( \text{Nil} \ |- \ E_1 : \sigma \rightarrow \tau \) and \( \text{Nil} \ |- \ E_2 : \sigma \) or otherwise \( E \) wouldn’t have been well-typed

1. If \( E_1 \) is not a value, then \( E_1 \rightarrow E_3 \). (Progress holds for \( E_1 \) by inductive hypothesis.) Thus, \( E_1 \ E_2 \rightarrow E_3 \ E_2 \)

2. If \( E_1 \) is a value but \( E_2 \) is not a value, then \( E_2 \rightarrow E_3 \). (Again, Progress holds for \( E_2 \) by the inductive hypothesis.) Thus, \( V \ E_2 \rightarrow V \ E_3 \)

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 \( (\lambda x. \ E_3) \ V \rightarrow E_3[V/x] \) 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: $x$ --- ...
2. Constant: $\text{Nil} |- c : \text{int} --- ...
3. Abs: $\text{Nil} |- (\lambda x. E_1) : \tau --- ...
4. App: $\text{Nil} |- (E_1 E_2) : \tau --- ...$ 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 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