Simply Typed Lambda Calculus, Progress and Preservation
Announcements

- HW5?
  - Due March 24, no room for extension

- I will post HW6 next time
  - Simple type inference for the lambda calculus
Outline

- Applied lambda calculus
- Introduction to types and type systems

- Simply typed lambda calculus (System F₁)
  - Syntax
  - Dynamic semantics
  - Static semantics
  - Type safety
“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)

- $E ::= c \mid x \mid (\lambda x.E_1) \mid (E_1E_2)$

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 (\lambda)-Calculus</th>
<th>A Language (ML)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>(x)</td>
<td>(x) (define (fun (x)) (M))</td>
</tr>
<tr>
<td>Constant</td>
<td>(c)</td>
<td>(c) fun (x) (\Rightarrow) (M)</td>
</tr>
<tr>
<td>Application</td>
<td>(M \ N)</td>
<td>(M \ N) if (P) then (M) else (N)</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>if (P) (M) (N)</td>
<td>if (P) then (M) else (N)</td>
</tr>
<tr>
<td>Let</td>
<td>(\text{let}(\lambda x. M) \ N)</td>
<td>(\text{let val} \ x = \ N \text{ in } M \text{ end})</td>
</tr>
</tbody>
</table>

Program Analysis CSCI 4450/6450, A Milanova
The Fixed-Point Operator

- One more constant, and one more rule:

  $\text{fix } M \rightarrow_\delta M (\text{fix } M)$

- 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:

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

- But how do we define plus?

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

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} =

\texttt{fix} \lambda f.\lambda x.\lambda y. \text{if } (\text{iszero } x) 0 \text{ (plus } y \text{ (f (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
  - `if (\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)
  - `succ true` (undefined in our language)
  - `plus true 0` 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 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!

- **Variable**: $\Gamma \vdash x : \tau$
  - Looks up the type of $x$ in environment $\Gamma$.

- **Application**: $\Gamma \vdash (E_1 \ E_2) : \tau$
  - Binding: augments environment $\Gamma$ with binding of $x$ to type $\sigma$.

- **Abstraction**: $\Gamma, x : \sigma \vdash E_1 : \tau$
  - $\Gamma \vdash (\lambda x : \sigma. E_1) : \sigma \rightarrow \tau$
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"

\[ E \rightarrow E_1 \rightarrow E_2 \rightarrow \]
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* which is 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 \textbf{int} constant, is a “stuck state”, i.e., a meaningless state.

- E.g., \( \textbf{if} \ c \ E_1 \ E_2 \) where \( c \) is an \textbf{int} constant, is a “stuck state”:
  - Clearly not a value and clearly no rule applies!
  - Because the evaluation rules for \textbf{if-then-else} are:
    - \( \textbf{if} \ \textbf{true} \ E_1 \ E_2 \ \rightarrow_{\delta} \ E_1 \)
    - \( \textbf{if} \ \textbf{false} \ E_1 \ E_2 \ \rightarrow_{\delta} \ E_2 \)
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.
Type Soundness

Safe

Sound

Complete

Safe
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
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:
  \[ E ::= x \mid (\lambda x: \tau. \, E_1) \mid (E_1 \, E_2) \mid c \]
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

\( \Gamma \vdash E : \tau \) (\( \vdash \) is the turnstile)
- Read: environment \( \Gamma \) entails that \( E \) has type \( \tau \)

Type judgment
\( \Gamma \vdash E_1 : \sigma \rightarrow \tau \) \( \Gamma \vdash E_2 : \sigma \)
\( \Gamma \vdash (E_1 E_2) : \tau \)
Semantics

binding: augments environment $\Gamma$ with binding of $x$ to type $\sigma$

$\Gamma \vdash E_1 : \sigma \rightarrow \tau$

$\Gamma \vdash (E_1 E_2) : \tau$

$\Gamma, x: \sigma \vdash E_1 : \tau$

$\Gamma \vdash (\lambda x: \sigma. E_1) : \sigma \rightarrow \tau$

looks up the type of $x$ in environment $\Gamma$

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

(Variable)

(Application)

(Abstraction)
Examples

- Deduce the type for $\text{int} \rightarrow \text{bool} \rightarrow \text{int}$

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

- Deduce the type for

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

\[ \Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int} \]

\[ \Gamma \vdash E_1 + E_2 : \text{int} \]

\[ \Gamma \vdash E_1 = E_2 : \text{bool} \]

\[ \Gamma \vdash b : \text{bool} \quad \Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau \]

\[ \Gamma \vdash \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau \]

(Comparison)

\[ \not\text{ Assignment} \]
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 $\text{bool}$, $+$ is defined on $\text{ints}$)

- 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

- Is this a valid type?

\( \text{Nil |- } \lambda x: \text{bool.} \lambda y: \text{int. if } x \text{ then } y \text{ else } y+1 : \text{bool \rightarrow int \rightarrow int} \)
Examples

Can we deduce the type of this term?

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

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

\[ \Gamma |- E_1 = E_2 : \text{bool} \]

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

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

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

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