Simply Typed Lambda Calculus
Lecture Outline

- Applied lambda calculus
- Introduction to types and type systems

- The simply typed lambda calculus (System F₁)
  - Syntax
  - Dynamic semantics
  - Static semantics
  - Type safety
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 (succ\(^k\) 0) \( \rightarrow_\delta \) false, \( k>0 \)
- iszero (pred\(^k\) 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 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</td>
</tr>
<tr>
<td>Abstraction</td>
<td>λx. M</td>
<td>fun x =&gt; M</td>
</tr>
<tr>
<td>Integer</td>
<td>succ^k 0, k&gt;0</td>
<td>k</td>
</tr>
<tr>
<td></td>
<td>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>
</tbody>
</table>

Let 

(λx. M) N

let val x = N in M end
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:

\[ \text{plus} \ x \ y = \begin{cases} 
  y & \text{if } x = 0 \\
  \text{plus} \ (\text{pred} \ x) \ (\text{succ} \ y) & \text{otherwise}
\end{cases} \]

\[ x-1 \quad y+1 \]

- 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

We have to show

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

$$\text{fix } M =_{\delta} M ( \text{fix } M ) =$$

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

Define `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 to \( M (Y M) \)
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 first argument, only true/false values)
  - `(0 x)` (0 does not apply as a function)
  - `succ true` (undefined in our language)
  - `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
Type System

- Syntax

- Dynamic semantics (i.e., how the program works). In type theory, it is
  - A sequence of reductions

- Static semantics (i.e., typing rules). 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)**
  \[ \Gamma \vdash x : \tau \]
  \( \text{x:}\tau \in \Gamma \)

- **(Application)**
  \[ \Gamma \vdash (E_1 E_2) : \tau \]
  \( \Gamma \vdash E_1 : \sigma \rightarrow \tau \), \( \Gamma \vdash E_2 : \sigma \)

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

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

Programming Languages CSCI 4430, 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”
- A sound type system never accepts a term that can “go wrong”
- A complete type system never rejects a term that cannot “go wrong”
- Whether a term can “go wrong” is undecidable
- Type systems choose type soundness (i.e., safety)
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
- Type soundness theorem: progress and preservation theorem
Type Expressions

- Introducing type expressions
  - $\tau ::= b \mid \tau \rightarrow \tau$
  - A type is a basic type $b$ (we will only consider `int` and `bool`, for simplicity), or a function type

- Examples
  - int
  - bool $\rightarrow$ (int $\rightarrow$ int) // $\rightarrow$ is right-associative, thus can write just `bool $\rightarrow$ int $\rightarrow$ 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

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

\[
\frac{x: \tau \in \Gamma}{\Gamma |- x : \tau}
\]  
(Variable)

$\Gamma |- E_1 : \sigma \rightarrow \tau \quad \Gamma |- E_2 : \sigma$

\[
\Gamma |- (E_1 \ E_2) : \tau
\]  
(Application)

$\Gamma, x: \sigma |- E_1 : \tau$

\[
\Gamma |- (\lambda x: \sigma. E_1) : \sigma \rightarrow \tau
\]  
(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 \textit{nil} environment
Extensions

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

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

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

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

(Comparison)
Examples

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

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

Can we deduce the type of this term?

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

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

Can we deduce the type of this term?

```
foldl =

\lambda f. \lambda x. \lambda y. \text{if } x=() \text{ then } y \text{ else } (\text{foldl } f \ (\text{cdr } x) \ (f \ y \ (\text{car } x)))
```

\[
\begin{align*}
\Gamma |- E : \text{list } \tau \\
\Gamma |- \text{car } E : \tau \\
\Gamma |- \text{cdr } E : \text{list } \tau
\end{align*}
\]
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
The End