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 $F_1$)
- 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:**
- if true \( M N \rightarrow_\delta M \)
- if false \( M N \rightarrow_\delta N \)
- iszero 0 \( \rightarrow_\delta \) true
- iszero \( \left(\text{succ}^k 0\right) \rightarrow_\delta \) false, \( k>0 \)
- iszero \( \left(\text{pred}^k 0\right) \rightarrow_\delta \) false, \( k>0 \)
- succ \( \left(\text{pred} M\right) \rightarrow_\delta M \)
- pred \( \left(\text{succ} M\right) \rightarrow_\delta M \)
## 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>$M \ N$</td>
<td>$M \ N$</td>
</tr>
<tr>
<td>Abstraction</td>
<td>$\lambda x. M$</td>
<td>$\text{fun } x =&gt; M$</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 $P \ \text{then } M \ \text{else } N$</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:
  \[
  \text{fix } \quad \text{fix } M \rightarrow_{\delta} M (\text{fix } M)
  \]

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

We must show that
\[
\texttt{fix} \ M =_{\delta\beta} \lambda x. \lambda y. \text{ if (iszero } x\text{) } y \ ((\texttt{fix} \ M) \ (\text{pred } x) \ (\text{succ } y))
\]
The Fixed-Point Operator

Define **times** =

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

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 \( M (Y M) \)
Types!

- Constants add power
- But they raise problems because they permit “bad” terms such as
  - `if (λx.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.
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!

- **Variable**
  \[
  \frac{x: \tau \in \Gamma}{\Gamma \vdash x : \tau}
  \]

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

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

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

(Variable)

(Application)

(Abstraction)
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 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
    
    \[
    \begin{align*}
    \text{if} \ true \ E_1 \ E_2 & \rightarrow_\delta E_1 \\
    \text{if} \ false \ E_1 \ E_2 & \rightarrow_\delta E_2
    \end{align*}
    \]
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
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
  - \( \tau ::= b \mid \tau \rightarrow \tau \)
  - A type is a basic type \( b \) (we will only consider \texttt{int}, for simplicity), or a function type

- Examples
  - \texttt{int}
  - \texttt{int \rightarrow (int \rightarrow int)} // \texttt{is right-associative}, thus can write just \texttt{int \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

  \[
  \begin{align*}
  \Gamma |- E_1 : \sigma \rightarrow \tau & \quad \Gamma |- E_2 : \sigma \\
  \end{align*}
  \]

  \[
  \text{Conclusion} \quad \Gamma |- (E_1 E_2) : \tau
  \]
Semantics

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

- **(Application)**
  \[
  \frac{\Gamma |- E_1 : \sigma \rightarrow \tau \quad \Gamma |- E_2 : \sigma}{\Gamma |- (E_1 \ E_2) : \tau}
  \]
  - *binding*: augments environment $\Gamma$ with binding of $x$ to type $\sigma$

- **(Abstraction)**
  \[
  \frac{\Gamma, x : \sigma |- E_1 : \tau}{\Gamma |- (\lambda x : \sigma. \ E_1) : \sigma \rightarrow \tau}
  \]
Examples

- Deduce the type for

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

- Deduce the type for

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

\[
\begin{align*}
\Gamma |- c : \text{int} & \quad \Gamma |- E_1 : \text{int} \quad \Gamma |- E_2 : \text{int} \\
\Gamma |- E_1 + E_2 : \text{int} & \quad (\text{Comparison})
\end{align*}
\]

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

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

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?

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

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

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

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

\[
\begin{align*}
(\lambda x. E) V & \Rightarrow E[V/x] \\
E_1 \Rightarrow E_2 & \quad E_1 E_3 \Rightarrow E_2 E_3 \\
V E_1 \Rightarrow V E_2
\end{align*}
\]

- 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

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

\[ \exists x : \tau \in \Gamma \]

\[ \Gamma |- \ x : \tau \]

\[ \Gamma, x : \sigma |- \ E_1 : \tau \]

\[ \Gamma |- (\lambda x. \ E_1) : \sigma \rightarrow \tau \]

\[ \Gamma |- E_1 : \sigma \rightarrow \tau \quad \Gamma |- E_2 : \sigma \]

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

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

Environment:
\[ \Gamma ::= \text{Nil} | \Gamma, x : \tau \]
Soundness Theorem, Formally

- Definition: E can get stuck if there exist an E' such that E →* E' and E' is stuck

- Theorem (Soundness): If Nil |- E : τ and E →^n E', then E' is a value, or E' → E''
  - Lemma (Preservation): If Nil |- E : τ and E → E' then Nil |- E' : τ
  - Lemma (Progress): If Nil |- E : τ then E is a value or there exist E' such that E → E'
Progress, Proof Sketch

- 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: $\text{Nil} |- x : \tau$ --- impossible because $\text{Nil} |- E : \tau$
2. Constant: $\text{Nil} |- c : \text{int}$ --- $E$ is a value
3. Abs: $\text{Nil} |- (\lambda x. E_1) : \tau$ --- again, $E$ is a value
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. Continued…
Progress, Proof Sketch

4. App: \textbf{Nil |- } E_1 E_2 : \tau. We have \textbf{Nil |- } E_1 : \sigma \rightarrow \tau \text{ and } \textbf{Nil |- } E_2 : \sigma \text{ or otherwise } E \text{ 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, \( \textbf{V } E_2 \rightarrow \textbf{V } E_3 \)

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 \((\lambda x. \ E_3) \ \textbf{V} \rightarrow E_3[V/x]\) 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: $\texttt{Nil} \vdash c : \texttt{int}$ --- ...

3. Abs: $\texttt{Nil} \vdash (\lambda x. E_1) : \tau$ --- ...

4. App: $\texttt{Nil} \vdash (E_1 E_2) : \tau$ --- ... 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
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
Monads

- A way to cleanly compose computations
  - E.g., \( f \) may return a value of type \( a \) or Nothing

Composing computations becomes tedious:

\[
\text{case } (f\ s) \text{ of}
\]

- Nothing \( \rightarrow \) Nothing
- Just \( m \) \( \rightarrow \) case (f m) ...

- In Haskell, monads model IO and other imperative features
An Example: Cloned Sheep

type Sheep = ...
father :: Sheep → Maybe Sheep
father = ...
mother :: Sheep → Maybe Sheep
mother = ...
(Note: a sheep has both parents; a cloned sheep has one)
maternalGrandfather :: Sheep → Maybe Sheep
maternalGrandfather s = case (mother s) of
  Nothing → Nothing
  Just m → father m
An Example

mothersPaternalGrandfather :: Sheep \rightarrow\text{ Maybe Sheep}

mothersPaternalGrandfather s = \text{ case } (\text{ mother } s) \text{ of}
\begin{align*}
\text{Nothing} & \rightarrow \text{ Nothing} \\
\text{Just } m & \rightarrow \text{ case } (\text{ father } m) \text{ of} \\
& \quad \text{Nothing} \rightarrow \text{ Nothing} \\
& \quad \text{Just } gf \rightarrow \text{ father } gf
\end{align*}

- Tedious, unreadable, difficult to maintain
- Monads help!
The Monad Class

- Haskell’s Monad **type class** requires 2 operations, \(\triangleright\triangleright=\) (bind) and **return**

```haskell
class Monad m where

  // \(\triangleright\triangleright=\) (the bind operation) takes a monad // \(m\ a\), and a function that takes \(a\) and turns // it into a monad \(m\ b\), and returns \(m\ b\)

  (\(\triangleright\triangleright=\)) :: m a \(\rightarrow\) (a \(\rightarrow\) m b) \(\rightarrow\) m b

  // **return** encapsulates a value into the monad

  return :: a \(\rightarrow\) m a
```
The **Maybe Monad**

```haskell
instance Monad Maybe where
  Nothing >>= f = Nothing
  (Just x) >>= f = f x
  return n = Just
```

- Back to our example:

```
mothersPaternalGrandfather s =
  (return s) >>= mother >>= father >>= father
```

(Note: if at any point, some function returns Nothing, it gets cleanly propagated.)
The List Monad

- The List type constructor is a monad
  \[ \text{li} \gg f = \text{concat} \ (\text{map} \ f \ \text{li}) \]
- \text{return} \ x = [x]

Note: \text{concat}::[[\text{a}]] \rightarrow [\text{a}]

- e.g., \text{concat} [[1,2],[3,4],[5,6]] yields [1,2,3,4,5,6]

- Use \textbf{any} \ f \ s.t. \ f::\text{a}\rightarrow[\text{b}]. \ f \ may \ return \ a \ list \ of 0,1,2,… \ elements \ of \ type \ b, \ e.g.,
  
  > f \ x = [x+1]
  > [1,2,3] >>= f \ // \ returns \ [2,3,4]
The List Monad

parents :: Sheep \rightarrow [Sheep]
parents \textit{s} = \text{MaybeToList} (\text{mother } \textit{s}) ++
\text{MaybeToList} (\text{father } \textit{s})

grandParents :: Sheep \rightarrow [Sheep]
grandParents \textit{s} = (\text{parents } \textit{s}) >>= \text{parents}