Simply Typed Lambda Calculus, cont. Simple Type Inference
Announcements

- HW5?
- HW6 is posted
Evaluation of Recursive Function, Revisit
Evaluation of Recursive Function, Revisit
Outline

- The simply typed lambda calculus
  - Syntax
  - Static semantics
  - Dynamic semantics
    - Stuck states
  - Type safety = progress + preservation

- Introduction to simple type inference
Putting It All Together, Formally

- Simply typed lambda calculus ($\text{System F}_1$)
  - Syntax
  - 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 (\texttt{int} \rightarrow \texttt{int}) // \rightarrow \texttt{is right-associative}, thus can write just \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{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

Environment \( \Gamma |- E : \tau \) (\( |- \) is the turnstile)

Read: environment \( \Gamma \) entails that \( E \) has type \( \tau \)

Type judgment

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

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

\[ \frac{\Gamma \vdash \text{E}_1 : \sigma \rightarrow \tau \quad \Gamma \vdash \text{E}_2 : \sigma}{\Gamma \vdash (\text{E}_1 \text{E}_2) : \tau} \] (Application)

\[ \frac{\Gamma, \text{x:}\sigma \vdash \text{E}_1 : \tau}{\Gamma \vdash (\lambda \text{x:}\sigma. \text{E}_1) : \sigma \rightarrow \tau} \] (Abstraction)

binding: augments environment \( \Gamma \) with binding of \( \text{x} \) to type \( \sigma \)
Examples

- Deduce the type for

\[ \lambda x: \text{int}. \lambda y: \text{bool}. \ x \] in the \text{nil} environment
# Extensions (to Static Semantics)

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma</td>
</tr>
<tr>
<td>$\Gamma</td>
</tr>
<tr>
<td>$\Gamma</td>
</tr>
<tr>
<td>$\Gamma</td>
</tr>
</tbody>
</table>

(Comparison)
Examples

Is this a valid type?

```
Nil |- \(\lambda x: \text{int}. \lambda y: \text{bool}. \; x + y : \text{int} \rightarrow \text{bool} \rightarrow \text{int}
```

Is this a valid type?

```
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

Can we deduce the type of this term?

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

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

- How about this

\[(\lambda x. \ x \ (\lambda y. \ y) \ (x \ 1)) \ (\lambda z. \ z) : ?\]

- \(x\) cannot have two “different” types
  - \((x \ 1)\) demands \(\text{int} \rightarrow ?\)
  - \((x \ (\lambda y. \ y))\) 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
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
Core Dynamic Semantics

- Syntax: \( E ::= c | x | (\lambda x. E_1) | (E_1 E_2) \)
  - \( c \) is integer constant
- Values: \( V ::= \lambda x. E_1 | c \)
- 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., \( x, x ((\lambda x. x) 1), c c, c (\lambda x. 1) \), etc.
Extensions
Core Typing Rules (Again…)

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

\[
\begin{align*}
\text{x} : \tau & \in \Gamma \\
\Gamma |- \text{x} : \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma , x : \sigma |- E_1 : \tau \\
\Gamma |- (\lambda x. E_1) : \sigma \rightarrow \tau
\end{align*}
\]

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

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

Environment:
\[
\Gamma ::= \text{Nil} | \Gamma , x : \tau
\]
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 $E \rightarrow E'$.
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 $E$
Progress, Proof Sketch

4. App: \textbf{Nil |- E}_1 \textbf{ E}_2 : \tau. We have \textbf{Nil |- E}_1 : \sigma \rightarrow \tau and \textbf{Nil |- E}_2 : \sigma or otherwise \textbf{E} wouldn’t have been well-typed

1. If \textbf{E}_1 is not a value, then \textbf{E}_1 \rightarrow \textbf{E}_3. (Progress holds for \textbf{E}_1 by inductive hypothesis.) Thus, \textbf{E}_1 \textbf{ E}_2 \rightarrow \textbf{E}_3 \textbf{ E}_2

2. If \textbf{E}_1 is a value but \textbf{E}_2 is not a value, then \textbf{E}_2 \rightarrow \textbf{E}_3. (Again, Progress holds for \textbf{E}_2 by the inductive hypothesis.) Thus, \textbf{V E}_2 \rightarrow \textbf{V E}_3

3. Finally, if \textbf{E}_1 and \textbf{E}_2 are both values, then \textbf{E}_1 must be \lambda x. \textbf{E}_3 (this is actually by a lemma, the Canonical Forms lemma). Thus, evaluation rule \((\lambda x. \textbf{E}_3) \textbf{ V} \rightarrow \textbf{E}_3[\textbf{V}/x]\) 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
The simply typed lambda calculus
- Syntax
- Static semantics
- Dynamic semantics
  - Stuck states
- Type safety = progress + preservation

Introduction to simple type inference
Deducing Types

- \( \lambda x: \text{int}. \lambda y: \text{bool}. \ x \)

1. Abs \( \Gamma = [] \)
   \( t_1 = \text{int} \rightarrow \text{bool} \rightarrow \text{int} \)

2. Abs \( \Gamma = [x: \text{int}] \)
   \( t_2 = \text{bool} \rightarrow \text{int} \)

3. Var \( x \) \( \Gamma = [x: \text{int}, y: \text{bool}] \)
   \( t_3 = \text{int} \)

1, 2, 3 denote the subcomponents of the term. We will be deducing types for each of these components.
Deducing Types

\((\lambda f:\text{int} \to \text{int}. \ f \ 5) \ (\lambda x:\text{int}. \ x+1) : ?\)
Type Inference, Strategy 1

- We can figure out all types even without explicit types for variables
  - \((\lambda f. f \, 5) \, (\lambda x. x+1) : ?\)
  - Type inference

Type inference, Strategy 1

- Use typing rules to derive type constraints
- Solve type constraints (offline)
- Aka constraint-based typing (e.g., Pierce)
We Can Infer All Types!

\[(\lambda f. f 5) \ (\lambda x. x+1) : ?\]

1. App \[\Gamma = []\]
2. Abs \[\Gamma = []\]
3. App \[t_f = \text{int} \to t_3\]
4. Abs \[t_4 = t_x \to t_5\]
5. + \[t_5 = \text{int}\]

\[\Gamma = [f: t_f]\]
\[\Gamma = [f: t_f]\]
\[\Gamma = [f: t_f]\]

\[\lambda x: t_x\]

\[\lambda x: t_x\]

\[\Gamma = [x: t_x]\]

\[\Gamma = [x: t_x]\]

\[\Gamma = [x: t_x]\]
Type Constraints

- We constructed a system of type constraints
- Let’s solve the system of constraints

\[
t_2 = t_4 \rightarrow t_1 \hspace{1cm} t_f = \text{int} \rightarrow t_3 = t_4 = \text{int} \rightarrow \text{int}
\]
\[
t_2 = t_f \rightarrow t_3 \hspace{1cm} t_3 = \text{int} \hspace{1cm} t_1 = t_3 = \text{int}
\]
\[
t_4 = t_x \rightarrow t_5 \hspace{1cm} t_4 = \text{int} \rightarrow \text{int}
\]
\[
t_f = \text{int} \rightarrow t_3
\]
\[
t_5 = \text{int}, \hspace{1cm} t_x = \text{int}
\]

- \((\lambda f: \text{int} \rightarrow \text{int}. \hspace{0.1cm} f \hspace{0.1cm} 5) \hspace{0.1cm} (\lambda x: \text{int}. \hspace{0.1cm} x+1) : \text{int} \hspace{0.1cm} (t_1)\)

We inferred all t’s!
\[
t_1 = \text{int}
\]
\[
t_2 = (\text{int} \rightarrow \text{int}) \rightarrow \text{int}
\]
\[
t_3 = \text{int}
\]
\[
t_4 = \text{int} \rightarrow \text{int}
\]
\[
t_f = \text{int} \rightarrow \text{int}
\]
Another Example

- \( \text{twice } f \ x = f \ (f \ x) \)
- What is the type of \texttt{twice}?
Another Example

- \( \text{twice } f \; x = f \; (f \; x) \)
- What is the type of \texttt{twice}?
  - It is \( t_f \rightarrow t_x \rightarrow t_1 \) (\( t_1 \) is the type of \( f \; (f \; x) \))
- Based on the syntax tree of \( f \; (f \; x) \) we have:
  \[
  t_f = t_2 \rightarrow t_1 \\
  t_f = t_x \rightarrow t_2
  \]
  Thus, \( t_x = t_1 = t_2 \), \( t_f = t_x \rightarrow t_x \) and the type of \texttt{twice} is \( (t_x \rightarrow t_x) \rightarrow t_x \rightarrow t_x \)

Note: \( t_x \) is a free type variable! \textbf{Polymorphism!}
Type Constraints from Typing Rules, as Attribute Grammar

Syntax:  \[ E ::= x \mid c \mid \lambda x.E \mid E_1 E_2 \mid E_1 + E_2 \]

Grammar rule:  

Attribute rule:  

- \( E ::= x \)  
  \( C_E = \{ t_E = \Gamma_E(x) \} \)

- \( E ::= c \)  
  \( C_E = \{ t_E = \text{int} \} \)

- \( E ::= \lambda x.E_1 \)  
  \( \Gamma_{E_1} = \Gamma_E ; x : t_x \)
  \( C_E = C_{E_1} \cup \{ t_E = t_x \rightarrow t_{E_1} \} \)

- \( E ::= E_1 E_2 \)  
  \( \Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = \Gamma_E \)
  \( C_E = C_{E_1} \cup C_{E_2} \cup \{ t_{E_1} = t_{E_2} \rightarrow t_E \} \)

- \( E ::= E_1 + E_2 \)  
  \( \Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = \Gamma_E \)
  \( C_E = C_{E_1} \cup C_{E_2} \cup \{ t_{E_1} = \text{int}, t_{E_2} = \text{int}, t_E = \text{int} \} \)
Type Constraints from Typing Rules, as Attribute Grammar

\[ \Gamma \text{ is inherited. Propagates top-down the tree.} \]

\[ E ::= \lambda x. E_1 \]

\[ \Gamma_{E_1} = \Gamma_{E}; x: t_x \]
\[ C_E = C_{E_1} \cup \{ t_E = t_x \rightarrow t_{E_1} \} \]

\[ t_E \text{ is “fresh” type variable for term represented by } E \text{’s subtree.} \]

\[ E ::= E_1 \ldots E_2 \]

\[ \Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = \Gamma_E \]
\[ C_E = C_{E_1} \cup C_{E_2} \cup \{ t_{E_1} = t_{E_2} \rightarrow t_E \} \ldots \]

\[ C \text{ collects constraints. It is synthesized. Propagates bottom-up the tree.} \]
Example

\[ \lambda f. \lambda x. f (f x) \]
Next Week

- 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:

```haskell
    case (f s) of
        Nothing → Nothing
        Just m → case (f m) ...
```

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

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

mothersPaternalGrandfather :: Sheep → Maybe Sheep

mothersPaternalGrandfather s = case (mother s) of
  Nothing → Nothing
  Just m → case (father m) of
    Nothing → Nothing
    Just gf → father gf

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

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

```haskell
class Monad m where

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

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

// \text{return} encapsulates a value into the monad

\text{return} :: a \to m a
```

41
The **Maybe Monad**

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

- 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

\[
\text{li >>= f} = \text{concat} \ (\text{map} \ f \ \text{li})
\]

\[
\text{return} \ x = [x]
\]

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

E.g., \(\text{concat} \ [[1,2],[3,4],[5,6]] \) yields \([1,2,3,4,5,6]\)

- Use \textbf{any} \(f\) s.t. \(f : : a \rightarrow [b]\). \(f\) may return a list of 0,1,2,… elements of type \(b\), e.g.,

\[
> f \ x = [x+1]
\]

\[
> [1,2,3] >>= f \ // \text{returns} \ [2,3,4]
\]
The List Monad

parents :: Sheep \rightarrow [Sheep]
parents s = MaybeToList (mother s) ++
            MaybeToList (father s)

grandParents :: Sheep \rightarrow [Sheep]
grandParents s = (parents s) >>= parents