Simple Type Inference
Announcements

- Quiz 5, pushed back to Thursday
An aside… Monads

- The 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 \( \texttt{a} \) or Nothing

  Composing computations becomes tedious:
  
  ```
  case (f s) of
      Nothing  \rightarrow Nothing
      Just m  \rightarrow \text{case } (f \cdot m) \ldots
  ```

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

type Sheep = ...

father :: Sheep \rightarrow Maybe Sheep
father = ...

mother :: Sheep \rightarrow Maybe Sheep
mother = ...

(Note: a sheep has both parents; a cloned sheep has one)

maternalGrandfather :: Sheep \rightarrow Maybe Sheep
maternalGrandfather s = case (mother s) of
  Nothing \rightarrow Nothing
  Just m \rightarrow father m
An Example

mothersPaternalGrandfather :: Sheep \rightarrow\ Maybe Sheep

mothersPaternalGrandfather s = \text{case } (\text{mother } s) \text{ of}
  Nothing \rightarrow\ Nothing

  Just m \rightarrow\ \text{case } (\text{father } m) \text{ of}
  Nothing \rightarrow\ Nothing

  Just gf \rightarrow\ \text{father } gf

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

- Haskell’s Monad `type class` requires 2 operations, `>>= (bind)` and `return`

class Monad m where

  // `>>=` (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`
  `>>=) :: 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 n
```

- Back to our example:

  ```haskell
  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: \(\text{concat}::[[a]] \to [a]\)

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

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

\[
\begin{align*}
> f x &= [x+1] \\
> [1,2,3] &>>= f \quad // \text{returns } [2,3,4]
\end{align*}
\]
The List Monad

parents :: Sheep \rightarrow [Sheep]
parents \( s \) = MaybeToList (mother \( s \)) ++
\hspace{2cm} MaybeToList (father \( s \))

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

- Introduction to types and type systems
- Simply typed lambda calculus (System F₁)
  - Language syntax, type expression syntax
  - Static semantics
  - Dynamic semantics
  - Type soundness: Safety = Progress + Preservation
    - Proved for the simply typed lambda calculus
- Intro to simple type inference
Outline

- Simple type inference
  - Equality constraints
  - Unification
  - Substitution
  - Strategy 1: Constraint-based typing
  - Strategy 2: On-the-fly typing: Algorithm W, almost
- Parametric polymorphism (next time…)
- Hindley Milner type inference. Algorithm W
Reading

- “Types and Programming Languages”, by Benjamin Pierce, Chapter 22, 23

- Lecture notes based partially on MIT 2015 Program Analysis OCW
Core Typing Rules

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

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

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

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

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

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

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

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

Environment:
\[ \Gamma ::= \text{Nil} \mid \Gamma, x : \tau \]
Extensions to Core Typing Rules

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

\[ \begin{align*}
\Gamma |- E_1 : \text{int} & \quad \Gamma |- E_2 : \text{int} \\
\Gamma |- E_1 = E_2 : \text{bool}
\end{align*} \]

\[ \begin{align*}
\Gamma |- b : \text{bool} & \quad \Gamma |- E_1 : \tau & \quad \Gamma |- E_2 : \tau \\
\Gamma |- \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau
\end{align*} \]
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 define type constraints
- Solve type constraints
- Aka constraint-based typing (e.g., Pierce)
We Can Infer All Types!

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

1. **App**
   \[ \Gamma = [] \]
   \[ t_2 = t_4 \rightarrow t_1 \]

2. **Abs**
   \[ \Gamma = [] \]
   \[ t_2 = t_f \rightarrow t_3 \]
   \[ \lambda f: \; t_f \]
   \[ \Gamma = [f: t_f] \]
   \[ \text{Var } f \]

3. **App**
   \[ \Gamma = [f: t_f] \]
   \[ t_f = \text{int} \rightarrow t_3 \]
   \[ \lambda x: \; t_x \]
   \[ \text{Const } 5 \]

4. **Abs**
   \[ \Gamma = [] \]
   \[ t_4 = t_x \rightarrow t_5 \]
   \[ \lambda x: \; t_x \]
   \[ \text{Var } x \]

5. **+**
   \[ \Gamma = [x: t_x] \]
   \[ t_5 = \text{int} \]
   \[ t_x = \text{int} \]

\( \frac{\Gamma \mid E_1 : \text{int} \quad \Gamma \mid E_2 : \text{int}}{\Gamma \mid E_1 + E_2 : \text{int}} \)

\( \frac{\Gamma \mid E_1 : \sigma \rightarrow \tau \quad \Gamma \mid E_2 : \sigma}{\Gamma \mid (E_1 \; E_2) : \tau} \)
Type Constraints

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

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

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

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

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

\begin{itemize}
  \item \texttt{Abs} \\
  \texttt{t}_1 = \texttt{tf} \rightarrow \texttt{t}_2 \\
  \Gamma = [f : \texttt{tf}] \\
  \texttt{t}_2 = \texttt{tx} \rightarrow \texttt{t}_3 \\
  \lambda x : \texttt{tx} \\
  \texttt{App} \\
  \texttt{tf} = \texttt{t}_4 \rightarrow \texttt{t}_3 \\
  \Gamma = [x : \texttt{tx}, f : \texttt{tf}] \\
  \texttt{tf} = \texttt{t}_x \rightarrow \texttt{t}_4 \\
  \texttt{f} \\
  \texttt{App} \\
  \texttt{tf} = \texttt{t}_x \rightarrow \texttt{t}_4 \\
  \Gamma = [x : \texttt{tx}, f : \texttt{tf}] \\
  \texttt{t}_1 = (\texttt{tx} \rightarrow \texttt{tx}) \rightarrow \texttt{tx} \rightarrow \texttt{tx}
\end{itemize}
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:
  \[
  t_f = \left( t_2 \rightarrow t_1 \right)
  \]
  \[
  t_f = \left( t_x \rightarrow t_2 \right)
  \]
  Thus, \( t_x = t_1 = t_2 \), \( t_f = t_x \rightarrow t_x \) and type of \texttt{twice} is \( (t_x \rightarrow t_x) \rightarrow t_x \rightarrow t_x \)

Note: \( t_x \) is a free type variable! 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:  
- \( E ::= x \)  
- \( E ::= c \)  
- \( E ::= \lambda x.E_1 \)  
- \( E ::= E_1 E_2 \)  
- \( E ::= E_1 + E_2 \)

Attribute rule:  
- \( C_E = \{ t_E = \Gamma_E(x) \} \)  
- \( C_E = \{ t_E = \text{int} \} \)  
- \( \Gamma_{E1} = \Gamma_E; x: t_x \)  
- \( C_E = C_{E1} \cup \{ t_E = t_x \rightarrow t_{E1} \} \)  
- \( \Gamma_{E1} = \Gamma_E \quad \Gamma_{E2} = \Gamma_E \)  
- \( C_E = C_{E1} \cup C_{E2} \cup \{ t_{E1} = t_{E2} \rightarrow t_E \} \)  
- \( \Gamma_{E1} = \Gamma_E \quad \Gamma_{E2} = \Gamma_E \)  
- \( C_E = C_{E1} \cup C_{E2} \cup \{ t_{E1} = \text{int}, t_{E2} = \text{int}, t_E = \text{int} \} \)
Type Constraints from Typing Rules, as Attribute Grammar

\[ (t_E, C_E) \leftarrow \text{interpret}(E, \Gamma) \]

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

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

\[ \Gamma_{E_1} = \Gamma_{E}; x: t_x \]

\[ C_E = C_{E_1} U \{ t_E = t_x \rightarrow t_{E_1} \} \]

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

\[ E ::= E_1 E_2 \]

\[ \Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = \Gamma_E \]

\[ C_E = C_{E_1} U C_{E_2} U \{ t_{E_1} = t_{E_2} \rightarrow t_E \} \ldots \]

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

- Two key concepts
- Equality
  - What does it mean for two types to be equal?
  - Structural equality (aka structural equivalence)
- Unification
  - Can two types be made equal by choosing appropriate substitutions for their type variables?
  - Robinson’s unification algorithm (which you already know from Prolog!)
Equality and Unification

- What does it mean for two types $\tau_a$ and $\tau_b$ to be equal?

- Structural equality
  - Suppose $\tau_a = t_1 \rightarrow t_2$
  - $\tau_b = t_3 \rightarrow t_4$
  - Structural equality entails
    $\tau_a = \tau_b$ means $t_1 \rightarrow t_2 = t_3 \rightarrow t_4$ iff $t_1 = t_3$ and $t_2 = t_4$
Equality and Unification

Can two types be made equal by choosing appropriate substitutions for their type variables?

Robinson’s unification algorithm

- Suppose $\tau_a = \text{int} \rightarrow t_1$
  $\tau_b = t_2 \rightarrow \text{bool}$
  - Can we unify $\tau_a$ and $\tau_b$? Yes, if $\text{bool}/t_1$ and $\text{int}/t_2$

- Suppose $\tau_a = \text{int} \rightarrow t_1$
  $\tau_b = \text{bool} \rightarrow \text{bool}$
  - Can we unify $\tau_a$ and $\tau_b$? No.
Example

Yes, if \texttt{int} \rightarrow t_2/t_1 \text{ and } \texttt{bool} \rightarrow t_3
Simple Type Substitution
(essential to define unification)

- Language of types
  \[ \tau ::= b \quad // \text{primitive type, e.g., int, bool} \]
  \[ | t \quad // \text{type variable} \]
  \[ | \tau \rightarrow \tau \quad // \text{function type} \]

- A substitution is a map
  - \[ S : \text{Type Variable} \rightarrow \text{Type} \]
  - \[ S = [\tau_1/t_1, \ldots, \tau_n/t_n] \quad // \text{substitute type } \tau_i \text{ for type var } t_i \]

- A substitution instance \[ \tau' = S \tau \]
  - \[ S = [ t_0 \rightarrow \text{bool} / t_1 ] \quad \tau = t_1 \rightarrow t_1 \quad \text{then} \]
  - \[ S(\tau) = S(t_1 \rightarrow t_1) = (t_0 \rightarrow \text{bool}) \rightarrow (t_0 \rightarrow \text{bool}) \]
Simple Type Substitution
(essential to define unification)

- Substitutions can be composed
  - \( S_1 = [ t_0 \rightarrow \text{bool} / t_1 ] \)
  - \( S_2 = [ \text{int} / t_0 ] \)
  - \( \tau = t_1 \rightarrow t_1 \)
  - \( S_2 S_1(\tau) = S_2( S_1(t_1 \rightarrow t_1) ) = \)
    \[
    S_2 \left( (t_0 \rightarrow \text{bool}) \rightarrow (t_0 \rightarrow \text{bool}) \right) = (\text{int} \rightarrow \text{bool}) \rightarrow \text{int} \rightarrow \text{bool}
    \]

Program Analysis CSCI 4450/6450, A Milanova (based on MIT 2015 Program Analysis OCW)
Examples

- Substitutions can be composed
  - $S_1 = \left[ \frac{t_x}{t_1} \right]$
  - $S_2 = \left[ \frac{t_x}{t_2} \right]$

- $\tau = t_2 \rightarrow t_1$

- $S_2 S_1 (\tau) = ?$
  - $S_2 \left( t_2 \rightarrow t_x \right) = t_x \rightarrow t_x$
Examples

Substitutions can be composed

- $S_1 = [t_1 / t_2]$
- $S_2 = [t_3 / t_1]$
- $S_3 = [t_4 \rightarrow \text{int} / t_3]$

- $\tau = t_1 \rightarrow t_2$
- $S_3 S_2 S_1 (\tau) = ?$

$$S_3 S_2 (t_1 \rightarrow t_1) = S_3 (t_3 \rightarrow t_2) = (t_4 \rightarrow \text{int}) \rightarrow t_4 \rightarrow \text{int}$$
Some Terminology...

- A substitution $S_1$ is **less specific** (i.e., more general) than substitution $S_2$ if $S_2 = S \cdot S_1$ for some substitution $S$
  - E.g., $S_1 = [ t_1 \rightarrow t_1 / t_2 ]$ is more general than $S_2 = [ \text{int} \rightarrow \text{int} / t_2 ]$ because $S_2 = S \cdot S_1$ for $S = [ \text{int} / t_1 ]$

- A **principal unifier** of a constraint set $C$ is a substitution $S_1$ that satisfies $C$, and $S_1$ is more general than any $S_2$ that satisfies $C$
Examples

- Find principal unifiers (when they exist) for
  - \( \{ \text{int} \rightarrow \text{int} = t_1 \rightarrow t_2 \} \) \( \left[ \text{int} / t_1, \text{int} / t_2 \right] \)
  - \( \{ \text{int} = \text{int} \rightarrow t_2 \} \) DOES NOT EXIST
  - \( \{ t_1 = \text{int} \rightarrow t_2 \} \) \( \left[ \text{int} / t_2 / t_1 \right] \)
  - \( \{ t_1 = \text{int}, t_2 = t_1 \rightarrow t_1 \} \) \( \left[ \text{int} / t_1, \text{int} / \text{int} / t_2 \right] \)
  - \( \{ t_1 \rightarrow t_2 = t_2 \rightarrow t_3, t_3 = t_4 \rightarrow t_5 \} \) \( \left[ t_1 / t_2, t_1 / t_3, t_4 \rightarrow t_5 / t_1 \right] \)
**Unification**

(essential for type inference!)

- **Unify**: tries to unify $\tau_1$ and $\tau_2$ and returns a **principal unifier for** $\tau_1 = \tau_2$ if unification is successful

```python
def Unify(\tau_1, \tau_2) =
    case (\tau_1, \tau_2)
    (\tau_1, t_2) = [\tau_1/t_2] provided $t_2$ does not occur in $\tau_1$
    (t_1, \tau_2) = [\tau_2/t_1] provided $t_1$ does not occur in $\tau_2$
    (b_1, b_2) = if (eq? b_1 b_2) then [ ] else fail
    (\tau_{11} \rightarrow \boxed{\tau_{12}}, \tau_{21} \rightarrow \boxed{\tau_{22}}) = let
        $S_1 = \text{Unify}(\tau_{11}, \tau_{21})$
        $S_2 = \text{Unify}(S_1(\tau_{12}), S_1(\tau_{22}))$
    in $S_2 S_1$ // compose substitutions
    otherwise = fail
```

This is the **occurs check**!
Examples

- **Unify** (int\(\rightarrow\)int, \(t_1\rightarrow t_2\)) yields ?
  \[
  [\text{int} / t_1] \leftarrow \text{Unify} (\text{int}, t_1) \\
  [\text{int} / t_2] \leftarrow \text{Unify} (\text{int}, t_2) \\
  [\text{int} / t_1, \text{int} / t_2] \leftarrow
  \]

- **Unify** (int, int\(\rightarrow\)t_2) yields ?
  \[
  \text{fail} \\
  (\text{int and int} \rightarrow t_2 \text{ cannot unify}) \\
  \text{all 4 cases miss}
  \]

- **Unify** (\(t_1\), int\(\rightarrow\)t_2) yields ?
  \[
  [\text{int} \rightarrow t_2 / t_1] 
  \]
Unify Set of Constraints $C$

- **UnifySet**: tries to unify $C$ and returns a **principal unifier for $C$** if unification is successful

```python
def UnifySet (C) =
    if C is Empty Set then []
    else let
        C = \{ $\tau_1=$$\tau_2$ \} U C'
        S = Unify ($\tau_1$,$\tau_2$) // Unify returns a substitution $S$
        in
        UnifySet ( S(C') ) ° S
        // Composition of substitutions
```

Program Analysis CSCI 4450/6450, A Milanova
Examples

- \{ t_1 = \text{int}, t_2 = t_1 \rightarrow t_1 \} \rightarrow [\text{int} / t_1, \text{int} \rightarrow \text{int} / t_2]$

- \{ t_1 \rightarrow t_2 = t_2 \rightarrow t_3, t_3 = t_4 \rightarrow t_5 \}$

- \{ t_f = t_2 \rightarrow t_1, t_f = t_x \rightarrow t_2 \}$

- \{ t_2 = t_4 \rightarrow t_1, t_2 = t_f \rightarrow t_3, t_4 = t_x \rightarrow t_5, t_f = \text{int} \rightarrow t_3, t_5 = \text{int}, t_x = \text{int} \}$
Type Inference, Strategy 1

- Aka constraint-based typing (e.g., Pierce)

- Traverse parse tree to derive a set of type constraints $C$
  - These are equality constraints
  - (Pseudo code in earlier slides)

- Solve type constraints offline
  - Use unification algorithm
  - (Pseudo code in slide 36)