Hindley Milner Type Inference, cont.
Announcements

- HW6?
- Please sign up for papers
Outline

- Hindley Milner type inference
  - Expression and type syntax
  - Instantiations and generalization
  - Typing rules
  - Type inference
    - Strategy 1 or
    - Strategy 2 as known as Algorithm W
  - Observations and examples

- Haskell records and monads
Varieties of Polymorphism

- Subtype polymorphism
  - Code can use a subclass $B$ where a superclass $A$ is expected; discussed earlier, gives rise to class analysis
  - Standard in object-oriented languages
Varieties of Polymorphism

- Parametric polymorphism
  - Code has a type as parameter
  - Type parameter can be explicit or implicit
  - Standard in functional programming languages

- Ad-hoc polymorphism (overloading)
Parametric Polymorphism

- Ada, Clu, C++, Java, Haskell (type classes)
- Explicit parametric polymorphism is also known as genericity
- C++ templates:

```cpp
template<class V>
class list_node {
    list_node<v>* prev;
    ...
}
```

```cpp
template<class V>
class list {
    list_node<v> header;
    ...
}
```
Java generics, e.g., bounded polymorphism:

class MyList1<E extends Object> {
    void m(E p) {
        p.intValue(); //compile-time error; Object
                      //does not have intValue()
    }
}

class MyList2<E extends Number> {
    void m(E p) {
        p.intValue(); //OK. Number has intValue()
    }
}
Parametric Polymorphism

- Instantiations respect the bound

```java
class MyList2<E extends Number> {
    void m(E arg) {
        arg.intValue(); //OK. Number has intValue()
    }
}
```

```java
MyList2<String> ls = new MyList2<String>(); // X
MyList2<Integer> li = ... // V
```
Haskell type classes:

```
sum :: (Num t1) => t1 -> [t1] -> t1
sum n [] = n
sum n (x:xs) = sum (n+x) xs
```

- \( t1 \) is a type parameter
- \((\text{Num } t1)\) is a predicate in type definition
- \((\text{Num } t1)\) constrains the types we can instantiate the generic function with
Let Polymorphism

- Haskell and ML
  - Known as ML-style polymorphism, or
  - Hindley Milner polymorphism

```
\x.x
let f = \x -> x in if (f True) then (f 1) else 0
```

\[
\text{let } \text{twice : } (t_x \rightarrow t_x) \rightarrow t_x \rightarrow t_x
\]

```
let twice f x = f (f x)
```

```
\text{twice twice } (\x \rightarrow x+1) \ 4
```
Towards Hindley Milner

```latex
let f = \lambda x.x

in

if (f true) then (f 1) else 1
```

- Constraints
  
  \[ t_f = t_1 \to t_1 \]
  
  \[ t_f = \text{bool} \to t_2 \] // at call \( f \ true \)
  
  \[ t_f = \text{int} \to t_3 \] // at call \( f \ 1 \)

- Doesn’t unify!
Expression Syntax (to study Hindley Milner)

- Expressions:

\[ E ::= c \mid x \mid \lambda x. E_1 \mid E_1 E_2 \mid \text{let } x = E_1 \text{ in } E_2 \]

- There are no types in the syntax

- The type of each sub-expression is derived by the Hindley Milner type inference algorithm
Type Syntax
(to study Hindley Milner)

- Types (aka monotypes):
  - \( \tau ::= b \mid \tau_1 \to \tau_2 \mid t \)
  - E.g., \texttt{int}, \texttt{bool}, \texttt{int\to bool}, \texttt{t_1\to int}, \texttt{t_1\to t_1}, etc.

- Type schemes (aka polymorphic types):
  - \( \sigma ::= \tau \mid \forall t. \sigma \)
  - \( \forall \tau_1 \cdot \sigma \mid t_4 \to t_3 \to t_4 \)
  - E.g., \( \forall t_1 . \forall t_2 . (\texttt{int\to t_1}) \to t_2 \to t_3 \)
  - Note: all quantifiers appear in the beginning, \( \tau \) cannot contain schemes

- Type environment now
  - \( \Gamma ::= \text{Identifiers} \to \text{Type schemes} \)
  - \( \Gamma ::= \{ f:\forall \tau. \tau \to \tau, \forall \} : \text{bool} \} \)
Instantiations

Type scheme \( \sigma = \forall t_1 \ldots t_n. \tau \) can be instantiated into a type \( \tau' \) by substituting types for the bound variables (BV) under the universal quantifier \( \forall \)

- \( \tau' = S \tau \) \( S \) is a substitution s.t. \( \text{Domain}(S) \supseteq \text{BV}(\sigma) \)
- \( \tau' \) is said to be an instance of \( \sigma \) (\( \sigma > \tau' \))
- \( \tau' \) is said to be a generic instance when \( S \) maps some type variables to new type variables

E.g., \( \sigma = \forall t_1. t_1 \rightarrow t_2 \)

\[
\begin{align*}
\left[ u / t_2 \right] (t_2 \rightarrow t_2) &= u \rightarrow t_2 \\
\left[ \text{null} \rightarrow \text{null} / t_1 \right] (t_1 \rightarrow t_2) &= (\text{null} \rightarrow \text{null}) \rightarrow t_2 \\
\left[ \text{bool} / t_1 \right] (t_1 \rightarrow t_2) &= \text{bool} \rightarrow t_2
\end{align*}
\]
Generalization (aka Closing)

- We can generalize a type $\tau$ as follows

$$\text{Gen}(\Gamma, \tau) = \forall t_1, \ldots, t_n. \tau$$

where $\{t_1 \ldots t_n\} = \text{FV}(\tau) - \text{FV}(\Gamma)$

- Generalization introduces polymorphism

- Quantify type variables that are free in $\tau$ but are not free in the type environment $\Gamma$
  - E.g., $\text{Gen}([], t_1 \rightarrow t_2)$ yields $\forall t_1 t_2. t_1 \rightarrow t_2$
  - E.g., $\text{Gen}([x:t_2], t_1 \rightarrow t_2)$ yields $\forall t_1. t_1 \rightarrow t_2$
Generalization, Examples

\[
\begin{align*}
\Gamma &= \left\{ f : \forall t_1. t_1 \rightarrow t_1 \right\} \\
\text{let } f &= \lambda x. x \text{ in } \text{if } (f \text{ true}) \text{ then } (f \text{ 1}) \text{ else 1 } \\
\end{align*}
\]

- We’ll infer type for \( \lambda x. x \) using simple type inference: \( t_1 \rightarrow t_1 \)
- Then we’ll generalize that type, \( \text{Gen}([], t_1 \rightarrow t_1) : \forall t_1. t_1 \rightarrow t_1 \)
- Then we’ll pass the polymorphic type into \textit{if (f true) then (f 1) else 1} and instantiate for each \( f \) in \textit{if (f true) then (f 1) else 1}
  - E.g., \([u_2/t_1] (t_1 \rightarrow t_1)\) where \( u_2 \) is fresh type variable at \( (f \ 1) \)
Generalization, Examples

- \(\lambda f. \lambda x. \text{let } g = f \text{ in } g \ x\)
  - \(\text{Gen}([f : t_f, x : t_x], t_f)\) yields?
- Why can’t we generalize \(t_f\)?
- Suppose we can generalize to \(\forall t_f. t_f\)
  - Then \(\forall t_f. t_f\) will instantiate at \(g \ x\) to some fresh \(u\)
  - Then \(u\) becomes \(t_x \rightarrow u'\) thus losing the important connection between \(t_x\) and \(t_f\)!
  - Thus \((\lambda f. \lambda x. \text{let } g = f \text{ in } g \ x) \ (\lambda y. y + 1) \text{ true}\) will type-check (unsound!!!)
- DO NOT generalize variables that are mentioned in type environment \(\Gamma\)!
Hindley Milner Typing Rules

\[
\frac{\Gamma; x : \tau \vdash E_1 : \tau \quad \Gamma; x : \text{Gen}(\Gamma, \tau) \vdash E_2 : \tau'}{\Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : \tau'} \quad \text{(Let)}
\]

- Type of \( x \) as inferred for \( E_1 \) is \( \tau \). Type of \( x \) in \( E_2 \) is the generalized type scheme \( \sigma = \text{Gen}(\Gamma, \tau) \)

\[
\frac{x : \sigma \in \Gamma \quad \tau < \sigma \quad (\text{Var})}{\Gamma \vdash x : \tau}
\]

- \( x \) in \( E_2 \) of \text{let}: \( x \) is of type \( \tau \) if its type \( \sigma \) in the environment can be instantiated to \( \tau \)

(Note: remaining rules, \( c, \text{App}, \text{Abs} \) are as in F\(_1\).)
Outline

- Hindley Milner type inference
  - Expression and type syntax
  - Instantiations and generalization
  - Typing rules
  - Type inference
    - Strategy 1 or
    - Strategy 2 as known as Algorithm W
  - Observations and examples

- Haskell records and monads
Hindley Milner Type Inference, Rough Sketch

\[
\text{let } x = E_1 \text{ in } E_2
\]

1. Calculate type \( T_{E_1} \) for \( E_1 \) in \( \Gamma;x:t_x \) using simple type inference
2. Generalize free type variables in \( T_{E_1} \) to get the type scheme for \( T_{E_1} \) (be mindful of caveat!)
3. Extend environment with \( x:Gen(\Gamma,T_{E_1}) \) and start typing \( E_2 \)
4. Every time we encounter \( x \) in \( E_2 \), instantiate its type scheme using fresh type variables

E.g., \( \text{id} \)'s type scheme is \( \forall t_1. t_1 \rightarrow t_1 \) so \( \text{id} \) is instantiated to \( u_k \rightarrow u_k \) at \((\text{id} 1)\)
Hindley Milner Type Inference

Two ways:

- Extend Strategy 1 (constraint-based typing)
- Extend Strategy 2 (Algorithm W)
Strategy 1  

let $f = \lambda x. x$ in if (f true) then (f 1) else 1

1. let $\Gamma = []$
   $t_1 = t_3$
   $\Gamma = [f: t_f]$ 

2. Abs
   $t_2 = t_x \rightarrow t_x$

3. if-then-else
   $t_3 = t_5 = \text{int}$
   $t_4 = \text{bool}$

4. App
   $u_1 \rightarrow u_1 = \text{bool} \rightarrow t_4$
   $u_2 \rightarrow u_2 = \text{int} \rightarrow t_5$

5. App
   $f \rightarrow \text{true}
   f \rightarrow 1

Next, generalize $t_f: \forall t_x. t_x \rightarrow t_x$

$u_1$ and $u_2$ are fresh type vars generated at instantiation of polymorphic type.
Example

\( \lambda x. \text{let } f = \lambda y. x \text{ in } (f \text{ true}, f \text{ 1}) : t_x \rightarrow (t_x, t_x) \)

1. Abs
2. let
3. Abs
4. tuple
5. App
6. App

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

\( \Gamma = [f : \forall y. y \rightarrow t_x, x : t_x] \)

\( t_4 = (t_5, t_6) \)

\( u_4 \rightarrow t_x = \text{bool} \rightarrow t_5 \)

\( u_2 \rightarrow t_x = n \rightarrow t_6 \)

Gen (\( t_4 \rightarrow t_x \)) = \( \forall y. y \rightarrow t_x \)
Strategy 2: Algorithm W

\[ \text{def } W(\Gamma, E) = \text{ case } E \text{ of} \]

\[ \begin{align*}
\text{c} & \rightarrow ([], \text{TypeOf(c)}) \\
\text{x} & \rightarrow \text{if (x NOT in Domain(}\Gamma\text{)) then fail} \\
& \quad \text{else let } T_E = \Gamma(x) \\
& \quad \text{in case } T_E \text{ of} \\
& \quad \quad \forall t_1, \ldots, t_n. \tau \rightarrow ([], [u_1/t_1\ldots u_n/t_n] \tau) \quad \text{(polytype)} \\
& \quad \quad _{-} \rightarrow ([], T_E) \quad \text{(mono type)} \\
\lambda x. E_1 & \rightarrow \text{let } (S_{E_1}, T_{E_1}) = W(\Gamma + \{x : t_x\}, E_1) \\
& \quad \text{in } (S_{E_1}, S_{E_1}(t_x) \rightarrow T_{E_1}) \\
\end{align*} \]

// ...

// continues on next slide!

\text{u}_1 \text{ to } \text{u}_n \text{ are fresh type vars generated at instantiation of polymorphic type}
def W(Γ, E) = case E of

    // continues from previous slide
    // ...
    E₁ E₂  -> let (S₁₁, T₁₁) = W(Γ,E₁)
    (S₂₁, T₂₁) = W(S₁₁(Γ),E₂)
    S = Unify(S₂₁(T₁₁), T₂₁ → t)
    in (S S₂₁ S₁₁, S(t))

let x = E₁ in E₂  -> let (S₁₁, T₁₁) = W(Γ+{x:t},E₁)
    S = Unify( S₁₁(t), T₁₁ )
    σ = Gen( S S₁₁(Γ), S(T₁₁) )
    (S₂₁, T₂₁) = W(S S₁₁(Γ)+{x:σ},E₂)
    in (S₂₁ S S₁₁, T₂₁)
let \( f = \lambda x.x \) in if (f true) then (f 1) else 1

1. let \( \Gamma = [] \) \( T_1 = \text{int} \) \( S_1 = ... \)
2. Abs \( \lambda x: t_x \)
   \( \Gamma = [f:t_f] \)
   \( T_2 = t_x \rightarrow t_x \) \( S_2 = [] \)
   \( \Gamma = [x:t_x,f:t_f] \)
3. if-then-else \( \Gamma = [f: \forall t_x.t_x \rightarrow t_x] \)
4. App \( T_3 = \text{int} \) \( S_3 = ... \)
5. App \( T_4 = \text{bool} \) \( S_4 = [\text{bool}/t_4][\text{bool}/u_1] \)
   \( T_5 = \text{int} \) \( S_5 = [\text{int}/t_5][\text{int}/u_2] \)

No constraint, types
immediately: \( T_2 = t_x \rightarrow t_x \cdot [t_x \rightarrow t_x/t_2] \)
\( \sigma = \text{Gen}([],t_x \rightarrow t_x) = \forall t_x. t_x \rightarrow t_x \)
\( T = u_1 \rightarrow u_1 \) \( S = [] \)
From \( \text{Unify}(u_1 \rightarrow u_1, \text{bool} \rightarrow t_4)^{26} \)
Hindley Milner Observations

- Do not generalize over type variables mentioned in type environment (they are used elsewhere)

- `let` is the only way of defining polymorphic constructs

- Generalize the types of let-bound identifiers only after processing their definitions
Hindley Milner Observations

- Generates the **most general type** (principal type) for each term/subterm
- Type system is sound

Complexity of Algorithm W

- PSPACE-Hard
- Because of nested let blocks
Hindley Milner Limitations

- Only let-bound constructs can be polymorphic and instantiated differently

\[
\text{twice} :: (\text{tx} \to \text{tx}) \to \text{tx} \to \text{tx}
\]

\[
\text{let twice f x = f (f x) } \quad \text{Ben (EJ, } (\text{tx} \to \text{tx}) \to \text{tx} \to \text{tx}) =
\]

\[
\text{in twice twice succ 4 } // \text{ let-bound polymorphism}
\]
Hindley Milner Limitations

- Only let-bound constructs can be polymorphic and instantiated differently

```plaintext
let twice f x = f (f x)

foo g = g g succ 4 // lambda-bound
in foo twice
```
Hindley Milner Limitations

- Another example:

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

vs.

let\ x = (\lambda z. z)

in

\[x (\lambda y. y) (x\ 1)\]
Outline

- Hindley Milner type inference
  - Expression and type syntax
  - Instantiations and generalization
  - Typing rules
  - Type inference
    - Strategy 1 or
    - Strategy 2 as known as Algorithm W
  - Observations and examples

- Haskell records and monads
Haskell Records

{- Constraint environment. -}
type Constraints = [(Type, Type)]
data ConstraintEnv = CEnv
   { constraints :: Constraints
   , var :: Int
   , tenv :: TEnv
   }
cenv = Cenv { constraints=[] , var=0 , tenv=[] } ;; new environment
… constraints cenv … var cenv … tenv cenv … ;; field accessor
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  \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 → 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!

Program Analysis CSCI 4450/6450, A Milanova (Example from All About Monads Tutorial)
Haskell’s Monad type class requires 2 operations, \( \triangleright\triangleright= \) (bind) and return

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**

```
instance Monad Maybe where
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
    return     = 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
  \[ \text{li} >>= f = \text{concat} \ (\text{map} \ f \ \text{li}) \]

- \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 \texttt{s} = \textbf{MaybeToList} (\textbf{mother} \texttt{s}) ++
\textbf{MaybeToList} (\textbf{father} \texttt{s})

grandParents :: Sheep \rightarrow [Sheep]
grandParents \texttt{s} = (\textbf{parents} \texttt{s}) >>= \textbf{parents}