Announcements

- HW6?
- Presentation guidelines are up, papers are up on schedule page as well
  1. Select available paper/slot from list
  2. If available, I assign to you, otherwise goto 1.
- 4 broad topics, but let me know if you
  - “Homework” papers on class analysis
  - ML for program analysis tasks
  - Applications of program analysis: smart contracts
  - Dynamic Binary Instrumentation (DBI)
Outline

- Simple type inference, conclusion
  - Let constructs
  - Strategy 2: on-the-fly typing

- Parametric polymorphism

- Hindley Milner type inference. Algorithm W
Simple Type Inference

- Strategy 1 solves constraints offline
  - Use typing rules to generate type constraints
  - Solve type constraints “offline”
  - Essential concepts: equality, unification and substitution

- Strategy 2 solves constraints on the fly
  - Builds the substitution map incrementally
The Let Construct

- In dynamic semantics, \texttt{let x = E_1 in E_2} is equivalent to \((\lambda x. E_2) \ E_1\)
- Typing rule
  \[
  \Gamma \vdash E_1 : \sigma \quad \Gamma; x: \sigma \vdash E_2 : \tau
  \]
  \[
  \Gamma \vdash \texttt{let x = E_1 in E_2} : \tau
  \]
- In static semantics \texttt{let x = E_1 in E_2} is not equivalent to \((\lambda x. E_2) \ E_1\)
  - In \texttt{let}, the type of “argument” \(E_1\) is inferred/checked \textbf{before} the type of function body \(E_2\)
  - \texttt{let} construct enables Hindley Milner style polymorphism!
The Let Construct

- Typing rule

\[
\Gamma \vdash E_1 : \sigma \quad \Gamma; x: \sigma \vdash E_2 : \tau \\
\Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : \tau
\]

- Attribute grammar rule

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

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

\[\Gamma_{E_2} = S_{E_1}(\Gamma_E) + \{x : T_{E_1}\}\]

\[T_E = T_{E_2}\]

\[S_E = S_{E_2} S_{E_1}\]
Typing Let Terms (Strategy 1)

\[
\begin{align*}
\text{let } f &= \lambda x . x, \text{ in } (f \ 1) \\
\text{1. let } t_1 & \in E_2 \\
\text{2. Abs } t_2 & \in E_2 \\
\text{3. App } t_{tf} & = t_1 \rightarrow t_2 \\
\text{C: } & tf = t_2, t_2 = t_x \rightarrow t_x, t_{tf} = \text{Inl} \rightarrow t_3, t_1 = t_3
\end{align*}
\]
The **Letrec Construct**

- **letrec** \( x = E_1 \text{ in } E_2 \)
  - \( x \) can be referenced from within \( E_1 \)
  - Extends calculus with general recursion
    - No need to type **fix** (we can’t!) but we can still type recursive functions like **plus**, **times**, etc.
  - Haskell’s **let** is a **letrec** actually!

- E.g.,

  ```
  letrec plus = \( x \). \( y \). \text{if (x=0) then y else ((plus \ x-1) \ y+1) in …}
  ```

  or in Haskell syntax:

  ```
  let plus x y = if (x=0) then y else plus (x-1) (y+1) in …
  ```
The Letrec Construct

- **letrec** \( x = E_1 \) in \( E_2 \)

Extensions over let rule
1. \( T_{E_1} \) is inferred in augmented environment \( \Gamma_E + \{x:t_x\} \)
2. Must unify \( S_{E_1}(t_x) \) and \( T_{E_1} \)
3. Apply substitution \( S \) on top of \( S_{E_1} \)

Note: Can merge let and letrec, in let

**Attribute grammar rule**

\[
E ::= \text{letrec } x = E_1 \text{ in } E_2
\]

\[
\begin{align*}
\Gamma_{E_1} &= \Gamma_E + \{x:t_x\} \\
S &= \text{Unify}(S_{E_1}(t_x), T_{E_1}) \\
\Gamma_{E_2} &= S S_{E_1}(\Gamma_E) + \{x:T_{E_1}\} \\
T_E &= T_{E_2} \\
S_E &= S_{E_2} S S_{E_1}
\end{align*}
\]
let vs. letrec

\[
\text{let } \text{plus } = \lambda x. \lambda y. \text{if } (x=0) \text{ then } y \text{ else } ((\text{plus } x-1) \ y+1) \text{ in }
\]

...
Algorithm W, Almost There!

def W(Γ, E) = case E of

• c      ->  ( [], TypeOf(c) )

• x      ->  if ( x NOT in Dom(Γ) ) then fail
               else let T_E = Γ(x);
               in ( [], T_E )

• λx.E_1 -> let ( S_{E_1}, T_{E_1} ) = W( Γ + { x : t_x }, E_1 )
               in ( S_{E_1}, S_{E_1}(t_x) → T_{E_1} )

• E_1 E_2 -> let ( S_{E_1}, T_{E_1} ) = W( Γ, E_1 )
               ( S_{E_2}, T_{E_2} ) = W( S_{E_1}(Γ), E_2 )
               \rightarrow S = \text{Unify}( S_{E_2}(T_{E_1}), T_{E_2} → t )
               in ( S S_{E_2} S_{E_1}, S(t) ) \parallel S S_{E_2} S_{E_1} \text{ composes substitutions}

let x = E_1 in E_2  ->  let ( S_{E_1}, T_{E_1} ) = W( Γ, E_1 )
                       ( S_{E_2}, T_{E_2} ) = W( S_{E_1}(Γ) + { x : T_{E_1} }, E_2 )
                       in ( S_{E_2} S_{E_1}, T_{E_2} )
def W(Γ, E) = case E of

  c -> ([]), TypeOf(c))

  x -> if (x NOT in Dom(Γ)) then fail
      else let T_E = Γ(x);
          in ([], T_E)

  λx.E_1 -> let (S_E1, T_E1) = W(Γ+{x:T_E1},E_1)
            in (S_E1, S_E1(t_x)→T_E1)

  E_1 E_2 -> let (S_E1, T_E1) = W(Γ,E_1)
            (S_E2, T_E2) = W(S_E1(Γ),E_2)
            S = Unify(S_E2(T_E1),T_E2→)
            in (S S_E2 S_E1, S(t)) // S S_E2 S_E1 composes substitutions

  let x = E_1 in E_2 -> let (S_E1, T_E1) = W(Γ+{x:T_E1},E_1)
                      S = Unify(S_E1(t_x),T_E1)
                      (S_E2, T_E2) = W(S S_E1(Γ)+{x:T_E1},E_2)
                      in (S_E2 S S_E1, T_E2)
Simple type inference, conclusion
- Let constructs
- Strategy 2: on-the-fly typing

Parametric polymorphism

Hindley Milner type inference. Algorithm W
Motivating Example

- A sound type system rejects some programs that don’t get stuck
- Canonical example

```informal
let f = \x.x
in
if (f true) then (f 1) else 1
```

- Term does not get “stuck”
- Term is NOT TYPABLE in the simply typed lambda calculus. It is typable in Hindley Milner!
Different Styles of (Parametric) Polymorphism

- Impredicative polymorphism (System F)
  \[ \tau ::= b \mid \tau_1 \to \tau_2 \mid T \mid \forall T. \tau \]
  \[ E ::= x \mid \lambda x : \tau. E \mid E_1 E_2 \mid \Lambda T. E \mid E [\tau] \]

- Very powerful
  - Can type self application \( \lambda x. x \ x \)
  - Still cannot type \( \text{fix!} \)

- Type inference is undecidable!
\( \lambda x \cdot x x \) in System F

\( X[\forall T. T \to T] \) instantiates \( T \) with \( \forall T. T \to T \).

\[ \sigma \quad \tau \quad (\text{return type}) \]

\[ [x : \forall T. T \to T] \vdash x [\forall T. T \to T] : (\forall T. T \to T) \to (\forall T. T \to T) \]

\[ [\ ] \vdash \lambda x : \forall T. T \to T. \ x [\forall T. T \to T] x : \forall T. T \to T \]
Different Styles of Polymorphism

- Predicative polymorphism

  \[
  \tau ::= \text{b} \mid \tau_1 \rightarrow \tau_2 \mid T
  \]

  \[
  \sigma ::= \tau \mid \forall T.\sigma \mid \sigma_1 \rightarrow \sigma_2
  \]

  \[
  E ::= x \mid \lambda x:\sigma. E \mid E_1 \ E_2 \mid \Lambda T. E \mid E [\tau]
  \]

- Still very powerful
  - Restricts System F by disallowing instantiation with a polymorphic type: \(E [\tau]\) but not \(E [\sigma]\)

- Type inference is still undecidable!
Different Styles of Polymorphism

- Prenex polymorphism
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid T \]
  \[ \sigma ::= \tau \mid \forall T. \sigma \]
  \[ E ::= x \mid \lambda x : \tau . E \mid E_1 \, E_2 \mid \Lambda T. E \mid E[\tau] \]

- Now type inference is decidable
- But polymorphism is limited
  - You cannot pass polymorphic functions
  - E.g., we cannot pass a sort function as argument
Different Styles of Polymorphism

- Let polymorphism
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid T \]
  \[ \sigma ::= \tau \mid \forall T.\sigma \]
  \[ E ::= x \mid \lambda x:\tau.E \mid E_1 E_2 \mid \Lambda T.E \mid E[\tau] \mid \text{let } x = E_1 \text{ in } E_2 \]

- Like \((\lambda x.E_2) E_1\) but \(x\) can be polymorphic!

- Good engineering compromise
  - Enhance expressiveness
  - Preserve decidability

- This is the Hindley Milner type system
Outline

- Simple type inference, conclusion
  - Let constructs
  - Strategy 2: on-the-fly typing

- Parametric polymorphism

- Hindley Milner type inference. Algorithm W
Towards Hindley Milner

\[
\begin{align*}
\text{let } f &= \lambda x. x \\
\text{in } & \quad \quad \text{if } (f \text{ true}) \text{ then } (f 1) \text{ else } 1
\end{align*}
\]

- Constraints
  \[
  \begin{align*}
  t_f &= t_1 \rightarrow t_1 \\
  t_f &= \text{bool} \rightarrow t_2 \quad // \text{ at call } (f \text{ true}) \\
  t_f &= \text{int} \rightarrow t_3 \quad // \text{ at call } (f 1)
  \end{align*}
  \]

- Doesn’t unify!
Towards Hindley Milner

- Solution:

- Generalize the type variable in type of $f$
  
  $t_f : t_1 \rightarrow t_1$ becomes $t_f : \forall T. T \rightarrow T$

- Different uses of generalized type variables are instantiated differently
  
  - E.g., $(f \text{ true})$ instantiates $t_f$ into $\text{bool} \rightarrow \text{bool}$
  
  - E.g., $(f \text{ 1})$ instantiates $t_f$ into $\text{int} \rightarrow \text{int}$

- When can we generalize?
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):
  - $\tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid t$
  - E.g., $\text{int}$, $\text{bool}$, $\text{int} \rightarrow \text{bool}$, $t_1 \rightarrow \text{int}$, $t_1 \rightarrow t_1$, etc.

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

- Type environment now

\[ \Gamma ::= \text{Identifiers} \rightarrow \text{Type schemes} \]
Instantiations

- Type scheme $\sigma = \forall t_1...t_n.\tau$ 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. Domain($S$) $\supseteq$ BV($\sigma$)
- $\tau'$ is said to be an instance of $\sigma$ ($\sigma \succ \tau'$)
- $\tau'$ is said to be a generic instance when $S$ maps some type variables to new type variables

- E.g., $\sigma = \forall t_1.t_1\rightarrow t_2$
  - $[t_3/t_1] t_1\rightarrow t_2 = t_3\rightarrow t_2$ is a generic instance of $\sigma$
  - $[\text{int}/t_1] t_1\rightarrow t_2 = \text{int}\rightarrow t_2$ is a non-generic instance of $\sigma$
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

\[ \text{let } f = \lambda x.x \text{ in } \begin{cases} \text{if } (f \text{ true}) \text{ then } (f \ 1) \text{ else } 1 \end{cases} \]

- 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 \( \begin{cases} \text{if } (f \text{ true}) \text{ then } (f \ 1) \text{ else } 1 \end{cases} \) and instantiate for each \( f \) in \( \begin{cases} \text{if } (f \text{ true}) \text{ then } (f \ 1) \text{ else } 1 \end{cases} \)
  - E.g., \([u_2/t_1] \ (t_1 \rightarrow t_1)\) where \( u_2 \) is fresh type variable at \( (f \ 1) \)
Generalization, Examples

- \( \lambda f : t_f. \; \lambda x : t_x. \; \text{let } g = f \text{ in } g \; x \)
  - \( \text{Gen}([f : t_f, x : t_x], t_f) \) yields?

Why can’t we generalize \( t_f \)?

Suppose we can generalize to \( \forall t_f \)
- Then \( \forall t_f = t_g \) 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 \( (\lambda f : t_f. \; \lambda x : t_x. \; \text{let } g = f \text{ in } g \; x) \) \( (\lambda y. y+1) \) \text{ true} \) will type-check (unsound!!)

DO NOT generalize variables that are mentioned in type environment \( \Gamma \)!
Hindley Milner Typing Rules

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

(Let)

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

\[ x : \sigma \in \Gamma \quad \tau < \sigma \]
\[ \Gamma \mid- x : \tau \]  

(Var)

- \( x \) in \( E_2 \) of 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 \( F_1 \).)
Hindley Milner Type Inference, Rough Sketch

let x = E₁ in E₂

1. Calculate type $T_{E₁}$ for $E₁$ in $\Gamma;x:t_x$ using simple type inference

2. Generalize free type variables in $T_{E₁}$ to get the type scheme for $T_{E₁}$ (be mindful of caveat!)

3. Extend environment with $x:Gen(\Gamma,T_{E₁})$ and start typing $E₂$

4. Every time we encounter $x$ in $E₂$, instantiate its type scheme using fresh type variables

E.g., id’s type scheme is $\forall t₁.t₁→t₁$ so id is instantiated to $u_k→u_k$ at (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 $\Gamma = []$
   $t_1 = t_3$
   $\Gamma = [f: t_f]$
   $\Gamma = [f: t_f, x: t_x]$
   $t_2 = t_x \rightarrow t_x$

2. Abs

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

4. App

5. App

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

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

$f$ $\text{true}$ $f$ $1$

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

\[ \lambda x. \text{let } f = \lambda y. x \text{ in } (f \text{ true}, f \text{ 1}) \]
def $W(\Gamma, E) = \text{case } E \text{ of}$

- $c \rightarrow ([], \text{TypeOf}(c))$
- $x \rightarrow \text{if } (x \text{ NOT in Domain}(\Gamma)) \text{ then fail}$
  
  else let $T_E = \Gamma(x)$
  
  in case $T_E$ of
  
  - $\forall t_1,...t_n.\tau \rightarrow ([],[u_1/t_1...u_n/t_n] \tau)$
  
  - $\rightarrow ([], T_E)$

- $\lambda x. E_1 \rightarrow \text{let } (S_{E_1}, T_{E_1}) = W(\Gamma+\{x:t_x\}, E_1)$
  
  in $(S_{E_1}, S_{E_1}(t_x)\rightarrow T_{E_1})$

// ...

// continues on next slide!

$u_1$ to $u_n$ are fresh type vars generated at instantiation of polymorphic type
def W(Γ, E) = case E of

    // continues from previous slide
    // ...

    E₁ E₂ -> let (S₁,E₁) = W(Γ,E₁)
    (S₂,E₂) = W(S₁(Γ),E₂)
    S = Unify(S₂(E₁),E₂→t)
    in (S₂ S₁ S₁, S(t))

let x = E₁ in E₂ -> let (S₁,E₁) = W(Γ+{x:t,x},E₁)
    S = Unify( S₁(t,x),E₁ )
    σ = Gen( S₁(Γ), S(T₁) )
    (S₂,E₂) = W(S₂ S₁(Γ)+{x:σ},E₂)
    in (S₂ S₁ S₁, E₂)
Strategy 2 Example

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 \( f \)  
   \( T_2 = t_x \rightarrow t_x \)  
   \( S_2 = [] \)  
   \( \Gamma = [x:t_x] \)

3. if-then-else  
   \( \Gamma = [f: \forall t_x.t_x \rightarrow t_x] \)  
   \( T_3 = \text{int} \)  
   \( S_3 = ... \)  

4. App \( f \)  
   \( \Gamma = [] \)  
   \( T_4 = \text{bool} \)  
   \( S_4 = [\text{bool}/t_4][\text{bool}/u_1] \)  

5. App \( f \)  
   \( \Gamma = [x:t_x] \)  
   \( T_5 = \text{int} \)  
   \( S_5 = [\text{int}/t_5][\text{int}/u_2] \)  

No constraint, types 2. Abs immediately: \( T_2 = t_x \rightarrow t_x : [t_x \rightarrow t_x/t_2] \)  
\( \sigma = \text{Gen}([],t_x \rightarrow t_x) = \forall t_x. t_x \rightarrow t_x \)
Example

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

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

in twice twice succ 4 // let-bound polymorphism
```

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

foo g = g g succ 4 // lambda-bound

in foo twice
```
Hindley Milner Limitations

Quiz example:

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

vs.

\(\text{let } x = (\lambda z. z)\)
\(\text{in}\)
\(x (\lambda y. y) (x 1)\)