Hindley Milner Type Inference

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, **let x = E_1 in E_2** is equivalent to **(λx.E_2) E_1**
- Typing rule

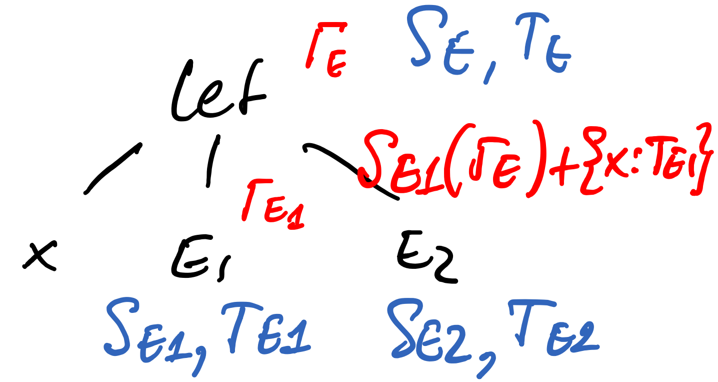
$$\frac{\Gamma \vdash E_1 : \sigma \qquad\qquad \Gamma;x{:}\sigma \vdash E_2 : \tau}{\Gamma \vdash \textbf{let } x = E_1 \textbf{ in } E_2 : \tau}$$

- In static semantics **let x = E_1 in E_2** is not equivalent to **(λx.E_2) E_1**
  - In **let**, the type of "argument" **E_1** is inferred/checked **before** the type of function body **E_2**
  - **let** construct enables Hindley Milner style polymorphism!

The Let Construct

- Typing rule

$$\frac{\Gamma \vdash E_1 : \sigma \qquad \Gamma;x{:}\sigma \vdash E_2 : \tau}{\Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : \tau}$$



- Attribute grammar rule

E ::= let x = E_1 in E_2

$\Gamma_{E1} = \Gamma_E$

$\Gamma_{E2} = S_{E1}(\Gamma_E) + \{x:T_{E1}\}$

$T_E = T_{E2}$ $\qquad$ $S_E = S_{E2}\ S_{E1}$

Typing Let Terms (Strategy 1)

let $f = \underbrace{\lambda x.x}_{E_1}$ in $\underbrace{(f\ 1)}_{E_2}$ [int]

1. let t_1 $\Gamma = [\,]$ $\{t_1 = t_3\}$

- f $\{t_f = t_2\}$
- 2. Abs t_2 $\Gamma = [\,]$
  - $\lambda x : t_x$
  - x
  - $\{t_2 = t_x \to t_x\}$
- 3. App $\Gamma = [f : t_f]$ $\{t_f = int \to t_3\}$
  - f
  - 1

$C = \{ t_f = t_2,\ t_2 = t_x \to t_x,\ t_f = int \to t_3,\ t_1 = t_3 \}$

The Letrec Construct

- **letrec x = E_1 in E_2**
  - **x** can be referenced from within $\mathbf{E_1}$
  - 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 = λx.λy. 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

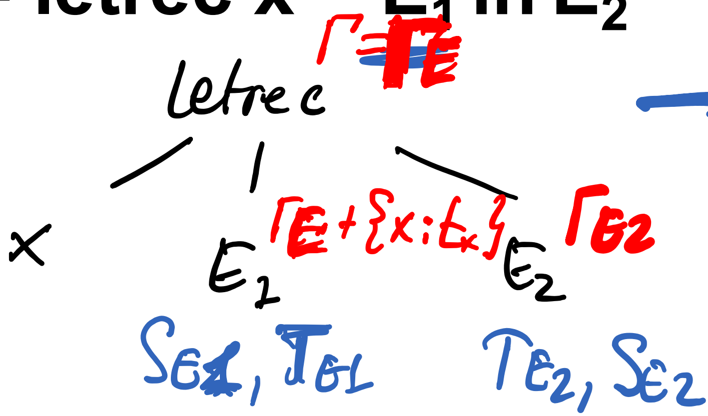
- **letrec x = E_1 in E_2**

Extensions over let rule
1. $\mathbf{T_{E1}}$ is inferred in augmented environment Γ_E + {**x**:t_x}
2. Must unify $\mathbf{S_{E1}(t_x)}$ and $\mathbf{T_{E1}}$
3. Apply substitution **S** on top of $\mathbf{S_{E1}}$

Note: Can merge **let** and **letrec**, in **let** **Unify** and **S** have no impact

- Attribute grammar rule

E ::= letrec x = E_1 in E_2



$$\Gamma_{E1} = \Gamma_E + \{x:t_x\}$$

$$S = Unify(S_{E1}(t_x), T_{E1})$$

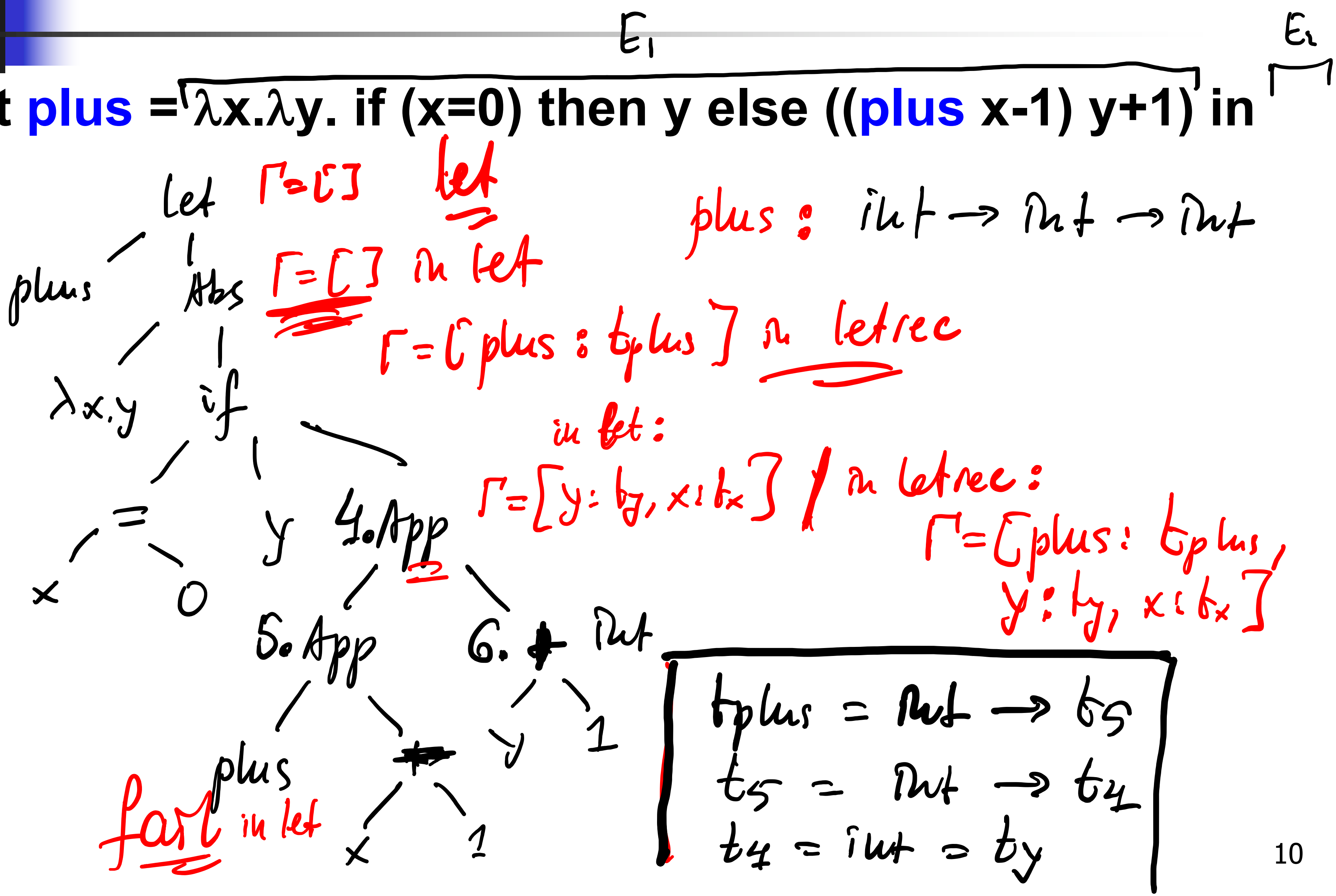
$$\Gamma_{E2} = S\ S_{E1}(\Gamma_E) + \{x:T_{E1}\}$$

$$T_E = T_{E2} \qquad S_E = S_{E2}\ S\ S_{E1}$$

let vs. letrec

let plus = λx.λy. if (x=0) then y else ((plus x-1) y+1) 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 $T_E = \Gamma(x)$;
  in ([], T_E)
- $\lambda x.E_1$ -> let $(S_{E1},T_{E1}) = W(\Gamma+\{x:t_x\},E_1)$
  in $(S_{E1}, S_{E1}(t_x)\to T_{E1})$
- $E_1\ E_2$ -> let $(S_{E1},T_{E1}) = W(\Gamma,E_1)$
  $(S_{E2},T_{E2}) = W(S_{E1}(\Gamma),E_2)$
  $S = Unify(S_{E2}(T_{E1}),T_{E2}\to t)$
  in $(S\ S_{E2}\ S_{E1}, S(t))$ // $S\ S_{E2}\ S_{E1}$ composes substitutions

let x = E_1 in E_2 -> let $(S_{E1},T_{E1}) = W(\Gamma,E_1)$
$(S_{E2},T_{E2}) = W(S_{E1}(\Gamma)+\{x:T_{E1}\},E_2)$
in $(S_{E2}\ S_{E1}, T_{E2})$

let not letrec

Algorithm W, Almost There! (merges let and letrec)

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.E1      ->  let (S_E1,T_E1) = W(Γ+{x:t_x},E1)
                       in (S_E1, S_E1(t_x)→T_E1)
        E1 E2      ->  let (S_E1,T_E1) = W(Γ,E1)
                           (S_E2,T_E2) = W(S_E1(Γ),E2)
                           S = Unify(S_E2(T_E1),T_E2→t)
                       in (S S_E2 S_E1, S(t)) // S S_E2 S_E1 composes substitutions
        let x = E1 in E2 -> let (S_E1,T_E1) = W(Γ+{x:t_x},E1)
                                S = Unify(S_E1(t_x),T_E1)
                                (S_E2,T_E2) = W(S S_E1(Γ)+{x:T_E1},E2)
                             in (S_E2 S S_E1, T_E2)
```

Outline

- 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

let f = λx.x

$t_f = t_x \to t_x$

in

$t_f = bool \to bool$ $\quad$ $t_f = int \to int$

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

$\forall T.\ T \to T$

- Impredicative polymorphism (System F)

  $\tau ::= b \mid \tau_1 \to \tau_2 \mid T \mid \forall T.\tau$

  $E ::= x \mid \lambda x{:}\tau.E \mid E_1\ E_2 \mid \Lambda T.E \mid E\ [\tau]$

  Can instantiate with polymorphic type!

- Very powerful
  - Can type self application **λx. x x**

    $\lambda x{:}\forall T.T \to T.\ x\,[\forall T.T \to T]\,x$
  - Still cannot type **fix**!

- Type inference is undecidable!

$\lambda x.\ x\ x$ in System F

$x[\forall T.\ T \to T]$ instantiates T with $\forall T.\ T \to T$.

$$\frac{[x:\forall T.\,T\to T] \vdash x[\forall T.\,T\to T] : \overbrace{(\forall T.\,T\to T)}^{\sigma} \to \overbrace{(\forall T.\,T\to T)}^{\tau\ \text{(return type)}} \qquad [x:\forall T.\,T\to T] \vdash x : \overbrace{\forall T.\,T\to T}^{\sigma}}{[\,] \vdash \lambda x:\forall T.\,T\to T.\ x[\forall T.\,T\to T]\ x : \overbrace{\forall T.\,T\to T}^{\tau}}$$

Different Styles of Polymorphism

- Predicative polymorphism

  $\tau ::= b \mid \tau_1 \to \tau_2 \mid T$

  $\sigma ::= \tau \mid \forall T.\sigma \mid \sigma_1 \to \sigma_2$

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

  We cannot type $\lambda x.\ x\ x$

- 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

  $\tau ::= b \mid \tau_1 \to \tau_2 \mid T$

  $\sigma ::= \tau \mid \forall T.\sigma$

  $E ::= x \mid \lambda x{:}\tau.E \mid E_1\ E_2 \mid \Lambda T.E \mid E\ [\tau]$

- 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

  $\tau ::= b \mid \tau_1 \to \tau_2 \mid T$

  $\sigma ::= \tau \mid \forall T.\sigma$

  $E ::= x \mid \lambda x{:}\tau.E \mid E_1\ E_2 \mid \Lambda T.E \mid E[\tau] \mid$ **let** $x = E_1$ **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

E_1

$$\text{Gen}([\,], t_x \rightarrow t_x) = \forall t_x .\, t_x \rightarrow t_x$$

let f = λx.x

in $E_2 \quad f : \forall t_x .\, t_x \rightarrow t_x$

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

- Constraints

  $\mathbf{t_f = t_1 \rightarrow t_1}$

  $\mathbf{t_f = bool \rightarrow t_2}$ // at call **(f true)**

  $\mathbf{t_f = int \rightarrow t_3}$ // at call **(f 1)**

- 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 true)** instantiates **t_f** into **bool→bool**
  - E.g., **(f 1)** instantiates **t_f** into **int→int**
- When can we generalize?

Expression Syntax (to study Hindley Milner)

- Expressions:

$$E ::= c \mid x \mid \lambda x.E_1 \mid E_1\ E_2 \mid \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2$$

- 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., **int, bool, int→bool,** $t_1 \rightarrow$**int,** $t_1 \rightarrow t_1$**, etc.**

As a simple types

t is a type variable

- 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, τ cannot contain schemes

t_3 is a "free" type variable as it isn't bound under $\forall$

- Type environment now

Gamma ::= Identifiers → Type schemes

Instantiations

- Type scheme $\sigma = \forall t_1 \ldots t_n . \tau$ can be instantiated into a type τ' 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**(σ)
  - τ' is said to be an instance of σ ($\sigma > \tau'$)
  - τ' 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 σ
  - $[int/t_1]\ t_1 \rightarrow t_2 = int \rightarrow t_2$ is a non-generic instance of σ

Generalization (aka Closing)

- We can generalize a type τ as follows

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

  where $\{t_1 \ldots t_n\} = \mathbf{FV}(\tau) - \mathbf{FV}(\Gamma)$
- Generalization introduces polymorphism
- Quantify type variables that are free in τ but are not free in the type environment Γ
  - E.g., $\mathbf{Gen}([],t_1 \rightarrow t_2)$ yields $\forall t_1,t_2.t_1 \rightarrow t_2$
  - E.g., $\mathbf{Gen}([x:t_2],t_1 \rightarrow t_2)$ yields $\forall t_1.t_1 \rightarrow t_2$

Generalization, Examples

let f = **λx.x** **in** **if (f true) then (f 1) else 1**

- We'll infer type for **λx.x** using simple type inference: $\mathbf{t_1 \to t_1}$
- Then we'll generalize that type, **Gen**([],$\mathbf{t_1 \to t_1}$): $\forall \mathbf{t_1 . t_1 \to t_1}$
- Then we'll pass the polymorphic type into **if (f true) then (f 1) else 1** and instantiate for each **f** in **if (f true) then (f 1) else 1**
  - E.g., $\mathbf{[u_2/t_1]}$ ($\mathbf{t_1 \to t_1}$) where $\mathbf{u_2}$ is fresh type variable at **(f 1)**

Generalization, Examples

- **$\lambda f{:}t_f.\ \lambda x{:}t_x.$ let g=f in g x**
  - **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 **($\lambda f{:}t_f.\ \lambda x{:}t_x.$ let g=f in g x) ($\lambda y.y+1$) true** will type-check (unsound!!!)
- DO NOT generalize variables that are mentioned in type environment **Γ**!

Hindley Milner Typing Rules

$$\frac{\Gamma;x{:}\tau \mid\!- E_1 : \tau \qquad \Gamma;x{:}Gen(\Gamma,\tau) \mid\!- E_2 : \tau'}{\Gamma \mid\!- \text{let } x = E_1 \text{ in } E_2 : \tau'} \quad \text{(Let)}$$

- Type of **x** as inferred for $\mathbf{E_1}$ is τ. Type of **x** in $\mathbf{E_2}$ is the generalized type scheme σ = **Gen(Γ,τ)**

$$\frac{x{:}\sigma \in \Gamma \qquad \tau < \sigma}{\Gamma \mid\!- x : \tau} \quad \text{(Var)}$$

- **x** in $\mathbf{E_2}$ of **let: x** is of type τ if its type σ in the environment can be instantiated to τ

(Note: remaining rules, **c**, **App**, **Abs** are as in F_1.)

Hindley Milner Type Inference, Rough Sketch

let x = E_1 in E_2

1. Calculate type $\mathbf{T_{E1}}$ for $\mathbf{E_1}$ in $\mathbf{\Gamma;x:t_x}$ using simple type inference
2. Generalize free type variables in $\mathbf{T_{E1}}$ to get the type scheme for $\mathbf{T_{E1}}$ (be mindful of caveat!)
3. Extend environment with $\mathbf{x:Gen(\Gamma,T_{E1})}$ and start typing $\mathbf{E_2}$
4. Every time we encounter **x** in $\mathbf{E_2}$, instantiate its type scheme using fresh type variables

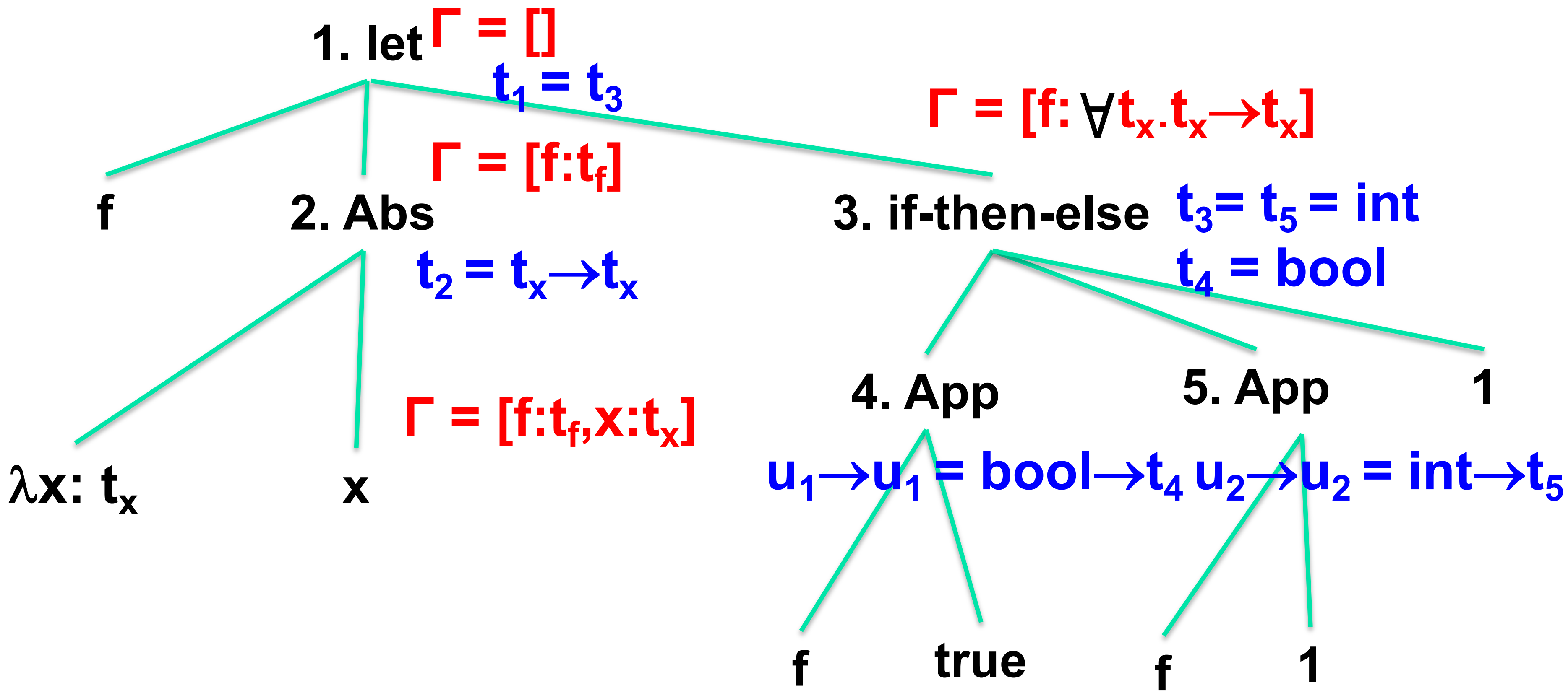
E.g., **id**'s type scheme is $\forall \mathbf{t_1.t_1 \rightarrow t_1}$ so **id** is instantiated to $\mathbf{u_k \rightarrow u_k}$ at **(id 1)**

Hindley Milner Type Inference

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

Strategy 1

let f = λx.x in if (f true) then (f 1) else 1



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

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

Example

- **λx. let f = λy.x in (f true, f 1)**

Strategy 2: Algorithm W

u_1 to u_n are fresh type vars generated at instantiation of polymorphic type

def W(Γ, E) = case E of

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

x -> if (x NOT in Domain(Γ)) then *fail*

else let $T_E = \Gamma(x)$

in case T_E of

$\forall\, t_1,\ldots t_n.\tau$ -> ([],$[u_1/t_1 \ldots u_n/t_n]\ \tau$)

_ -> ([], T_E)

$\lambda x.E_1$ -> let $(S_{E1},T_{E1}) = W(\Gamma+\{x{:}t_x\},E_1)$

in $(S_{E1}, S_{E1}(t_x) \rightarrow T_{E1})$

// ...

// continues on next slide!

Strategy 2: Algorithm W

def W(Γ, E) = case E of

// continues from previous slide

// ...

$E_1\ E_2$ -> let $(S_{E1},T_{E1}) = W(\Gamma,E_1)$

$(S_{E2},T_{E2}) = W(S_{E1}(\Gamma),E_2)$

$S = Unify(S_{E2}(T_{E1}),T_{E2}\rightarrow t)$

in $(S\ S_{E2}\ S_{E1},\ S(t))$

let $x = E_1$ in E_2 -> let $(S_{E1},T_{E1}) = W(\Gamma+\{x:t_x\},E_1)$

$S = Unify(\ S_{E1}(t_x),T_{E1}\ )$

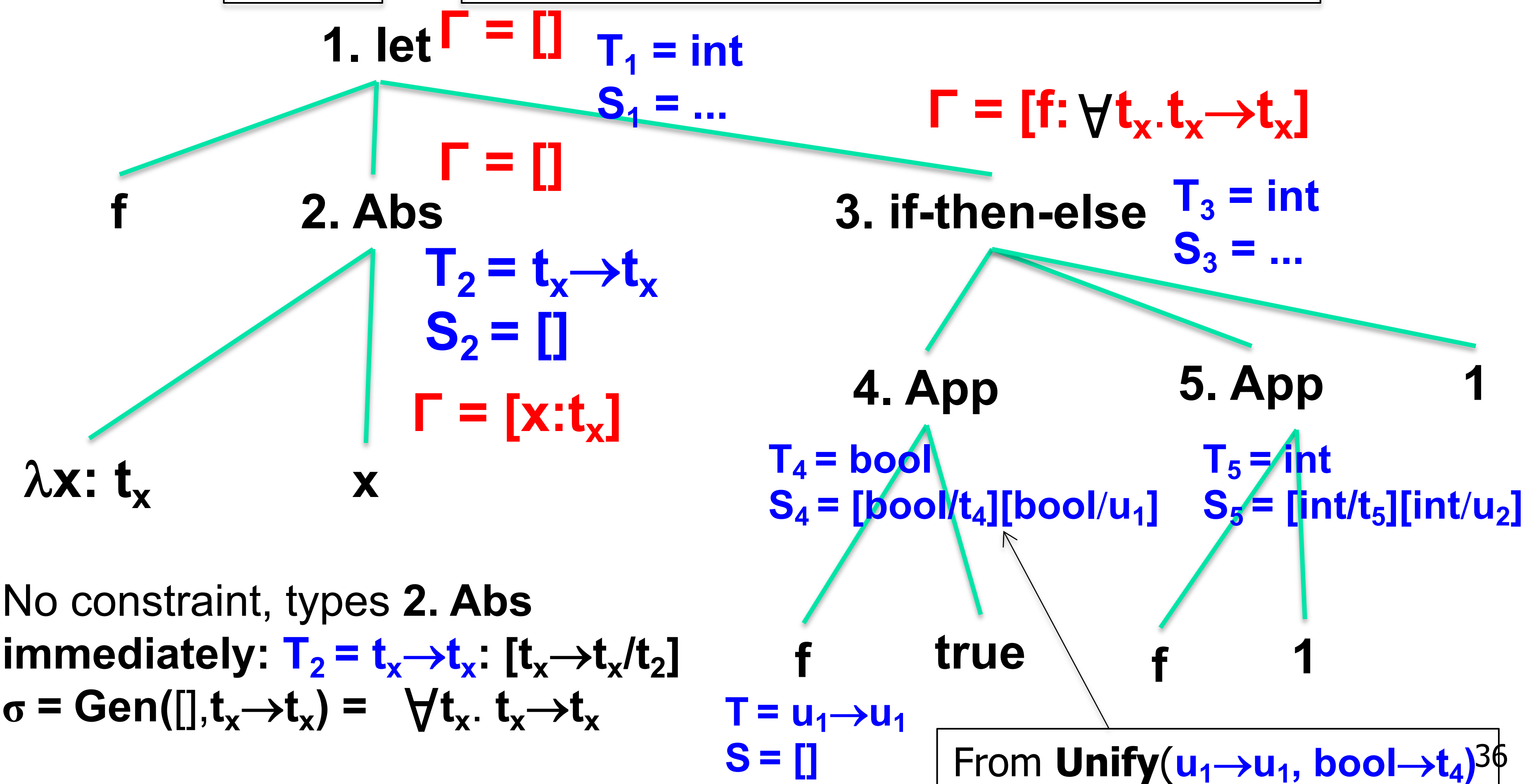
$\sigma = Gen(\ S\ S_{E1}(\Gamma),\ S(T_{E1})\ )$

$(S_{E2},T_{E2}) = W(S\ S_{E1}(\Gamma)+\{x:\sigma\},E_2)$

in $(S_{E2}\ S\ S_{E1},\ T_{E2})$

Strategy 2 Example

let f = λx.x in if (f true) then (f 1) else 1



No constraint, types **2. Abs immediately:** $T_2 = t_x \rightarrow t_x$: $[t_x \rightarrow t_x / t_2]$

$\sigma = Gen([], t_x \rightarrow t_x) = \forall t_x.\ t_x \rightarrow t_x$

Example

- **λx. let f = λy.x in (f true, f 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:

(λx. x (λy. y) (x 1)) (λz. z)

vs.

let x = (λz. z)

in

x (λy. y) (x 1)