Hindley Milner Type Inference, cont.

Announcements

- HW6?
- Please sign up for papers

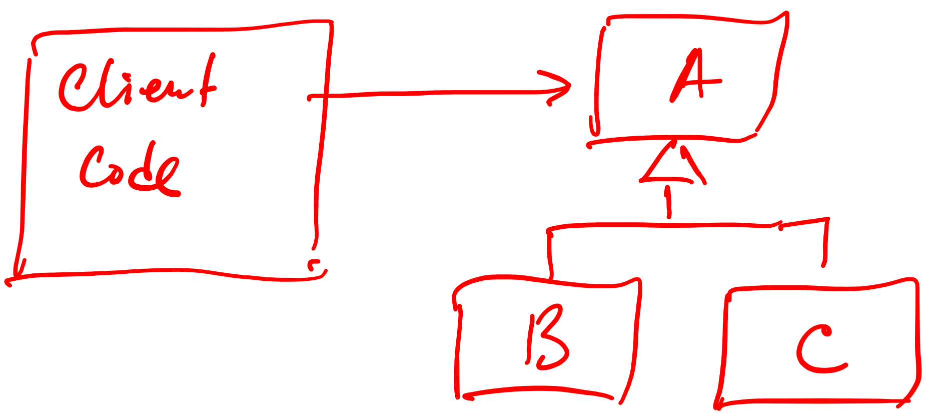
Outline

- 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:

typedef list_node<int> i_list_node;
typedef list<int> i_list;

```
template<class V>
class list_node {
   list_node<V>* prev;
   …
}
```

```
template<class V>
class list {
   list_node<V> header;
   …
}
```

Parametric Polymorphism

- 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

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

MyList2<String> ls = new MyList2<String>(); ✗

MyList2<Integer> li = …    ✓
```

Parametric Polymorphism

- 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
- `(Num t1)` is a predicate in type definition
- `(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

$\lambda x.x$

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

$twice : (t_x \to t_x) \to t_x \to t_x$

```
let
twice f x = f (f x)
in
twice twice (\x -> x+1) 4
```

Towards Hindley Milner

let f = λx.x

in

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

- Constraints

  $t_f = t_1 \rightarrow t_1$

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

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

- Doesn't unify!

DOES NOT WORK IN SIMPLE TYPES.

Expression Syntax (to study Hindley Milner)

- Expressions:

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

- 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)

Just as in simple types!

- Types (aka monotypes):
  - τ ::= **b** | $\tau_1 \rightarrow \tau_2$ | **t**

    t is a type variable
  - E.g., **int, bool, int→bool, t_1→int, t_1→t_1, etc.**
- Type schemes (aka polymorphic types):
  - σ ::= τ | $\forall$**t.**σ

    $\forall t_1\, t_2\, t_3\, t_4 .\ t_3 \rightarrow t_4$
  - E.g., $\forall t_1 . \forall t_2 . (int \rightarrow t_1) \rightarrow t_2 \rightarrow t_3$

    t_3 is a "free" type variable as it isn't bound under $\forall$
  - Note: all quantifiers appear in the beginning, τ cannot contain schemes
- Type environment now

  Γ ::= Identifiers → Type schemes

  $\Gamma = [f : \forall t_x .\ t_x \rightarrow t_x,\ x : int,\ y : bool]$

Instantiations

Turns σ into a τ

- Type scheme $\boldsymbol{\sigma = \forall t_1 \ldots t_n . \tau}$ can be instantiated into a type $\boldsymbol{\tau'}$ by substituting types for the bound variables (**BV**) under the universal quantifier $\forall$
  - $\boldsymbol{\tau' = S\, \tau}$ **S** is a substitution s.t. Domain(**S**) $\supseteq$ **BV**($\boldsymbol{\sigma}$)
  - $\boldsymbol{\tau'}$ is said to be an instance of $\boldsymbol{\sigma}$ ($\boldsymbol{\sigma > \tau'}$)
  - $\boldsymbol{\tau'}$ is said to be a generic instance when **S** maps some type variables to new type variables
- E.g., $\boldsymbol{\sigma = \forall t_1 . t_1 \rightarrow t_2}$

~~$[u / t_1]\,(t_1 \rightarrow t_2) = u \rightarrow t_2$~~

$[int \rightarrow int / t_1]\,(t_1 \rightarrow t_2) = (int \rightarrow int) \rightarrow t_2$

$[bool / t_1]\,(t_1 \rightarrow t_2) = bool \rightarrow t_2$

Generalization (aka Closing)

Turn τ in to σ

- 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., $\mathbf{Gen}([],t_1 \to t_2)$ yields $\forall t_1 t_2.\, t_1 \to t_2$
  - E.g., $\mathbf{Gen}([x{:}t_2],t_1 \to t_2)$ yields $\forall t_1.\, t_1 \to t_2$

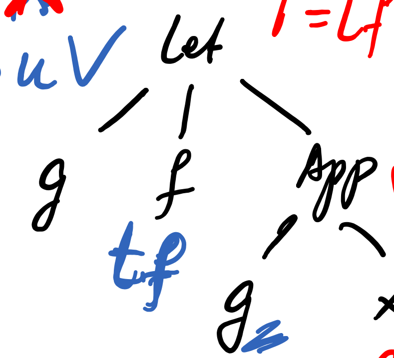
Generalization, Examples

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

- We'll infer type for **λx.x** using simple type inference: $\mathbf{t_1 \rightarrow t_1}$
- Then we'll generalize that type, **Gen**([],$\mathbf{t_1 \rightarrow t_1}$): $\forall \mathbf{t_1.t_1 \rightarrow t_1}$
- 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., $\mathbf{[u_2/t_1]}$ ($\mathbf{t_1 \rightarrow t_1}$) where $\mathbf{u_2}$ is fresh type variable at **(f 1)**

Generalization, Examples

- **λf. λx. let g=f in g x**
  - **Gen**([**f**:t_f,**x**:t_x],t_f) yields?
- Why can't we generalize $\mathbf{t_f}$?
- Suppose we can generalize to ∀$\mathbf{t_f}$.$\mathbf{t_f}$
  - Then ∀$\mathbf{t_f}$.$\mathbf{t_f}$ will instantiate at **g x** to some fresh **u**
  - Then **u** becomes $\mathbf{t_x \rightarrow u'}$ thus losing the important connection between $\mathbf{t_x}$ and $\mathbf{t_f}$!
  - Thus **(λf. λx. let g=f in g x) (λy.y+1) true** will type-check (unsound!!!)
- DO NOT generalize variables that are mentioned in type environment **Γ**!



Hindley Milner Typing Rules

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

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

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

- **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.)

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

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

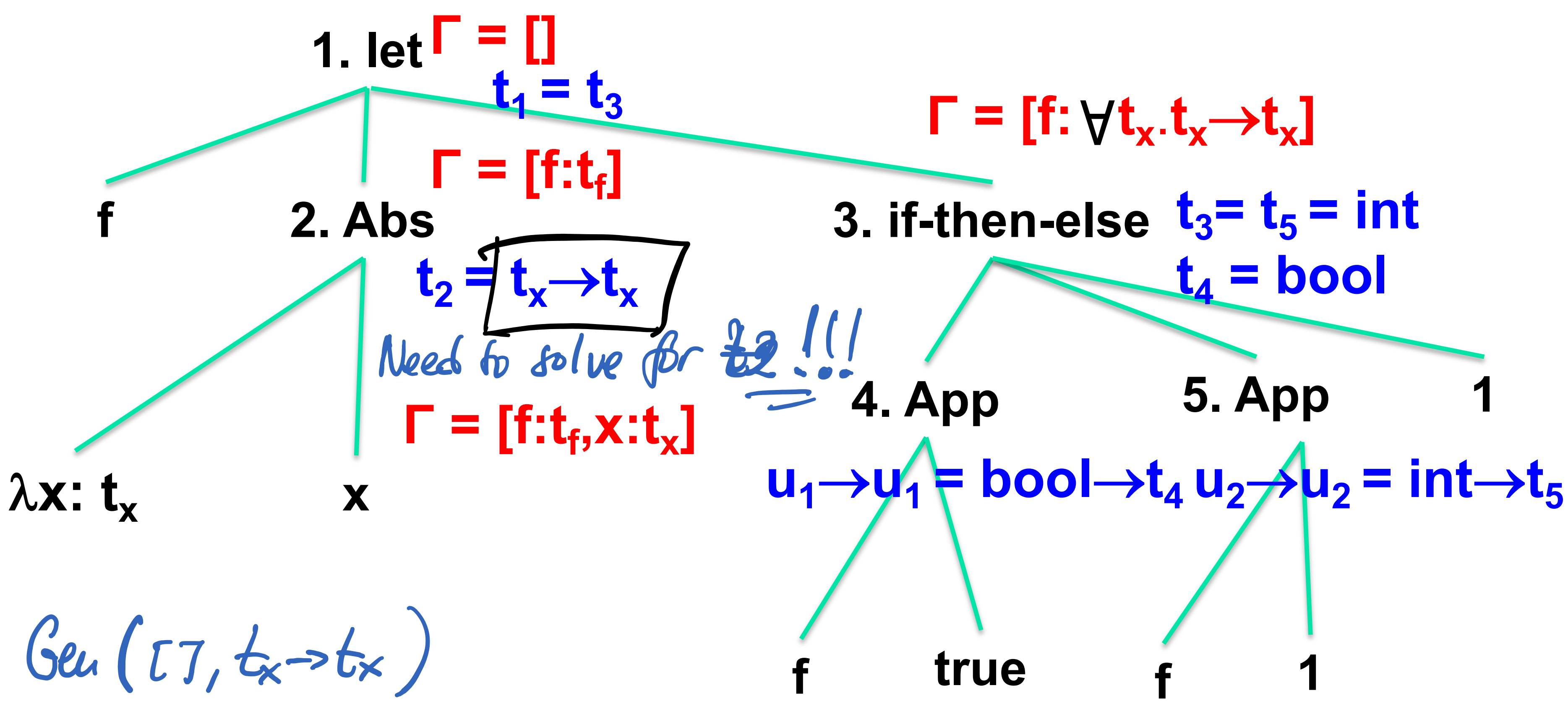
- Two ways:

- Extend Strategy 1 (constraint-based typing)

- Extend Strategy 2 (Algorithm W)

Strategy 1 – like

let f = $\overbrace{\lambda x.x}^{E_1}$ in $\overbrace{\text{if (f true) then (f 1) else 1}}^{E_2}$

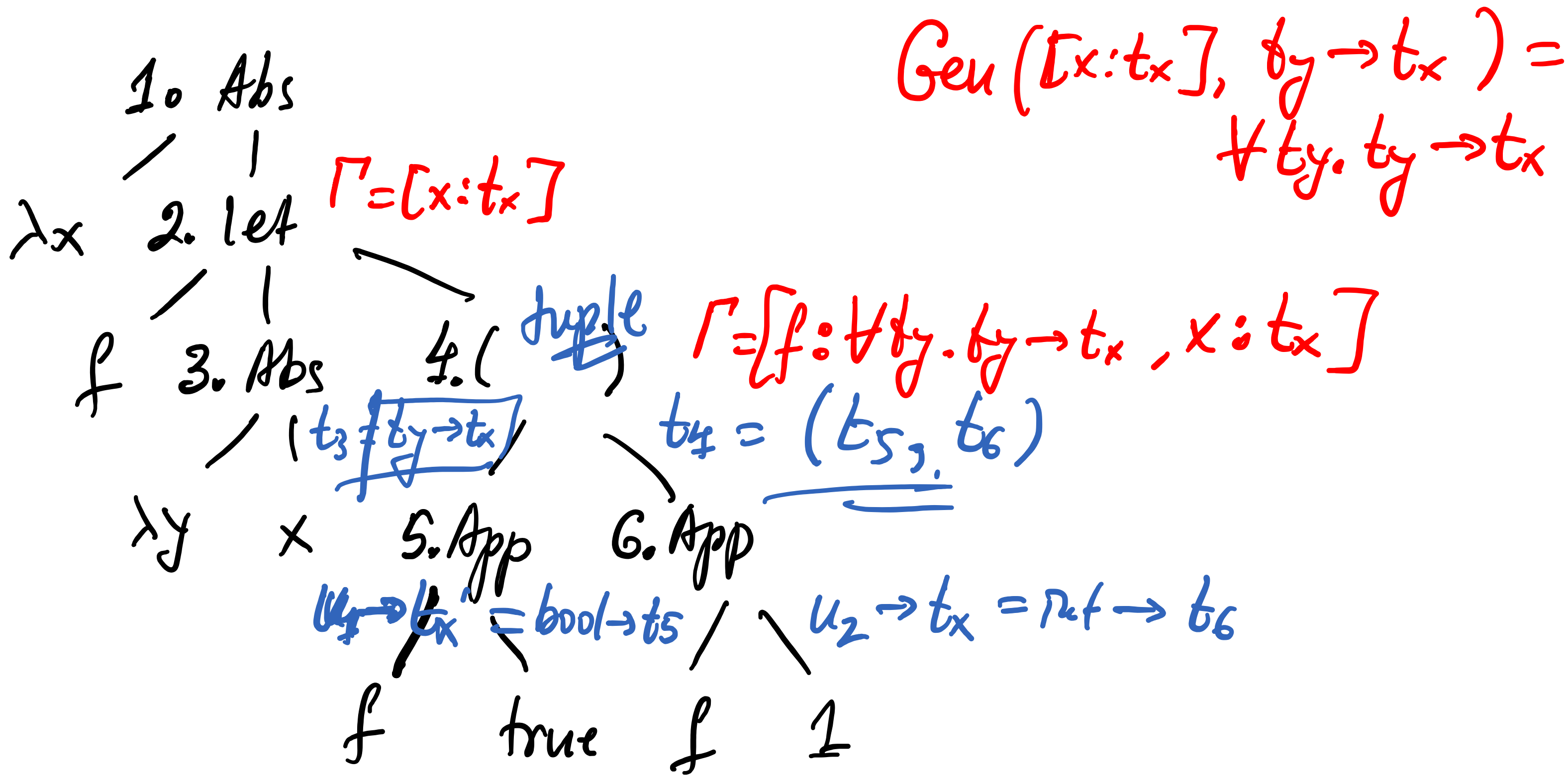


Next, generalize t_f: $\forall t_x.\ t_x \to t_x$

> 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) : $t_x \to (t_x, t_x)$



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 = Γ(x)

in case T_E of

$\forall\, t_1,\ldots t_n.\tau$ -> ([],[$u_1/t_1 \ldots u_n/t_n$] τ) (polytype) — instantiate into brand new monotype

_ -> ([], T_E) (monotype)

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

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

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

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

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

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

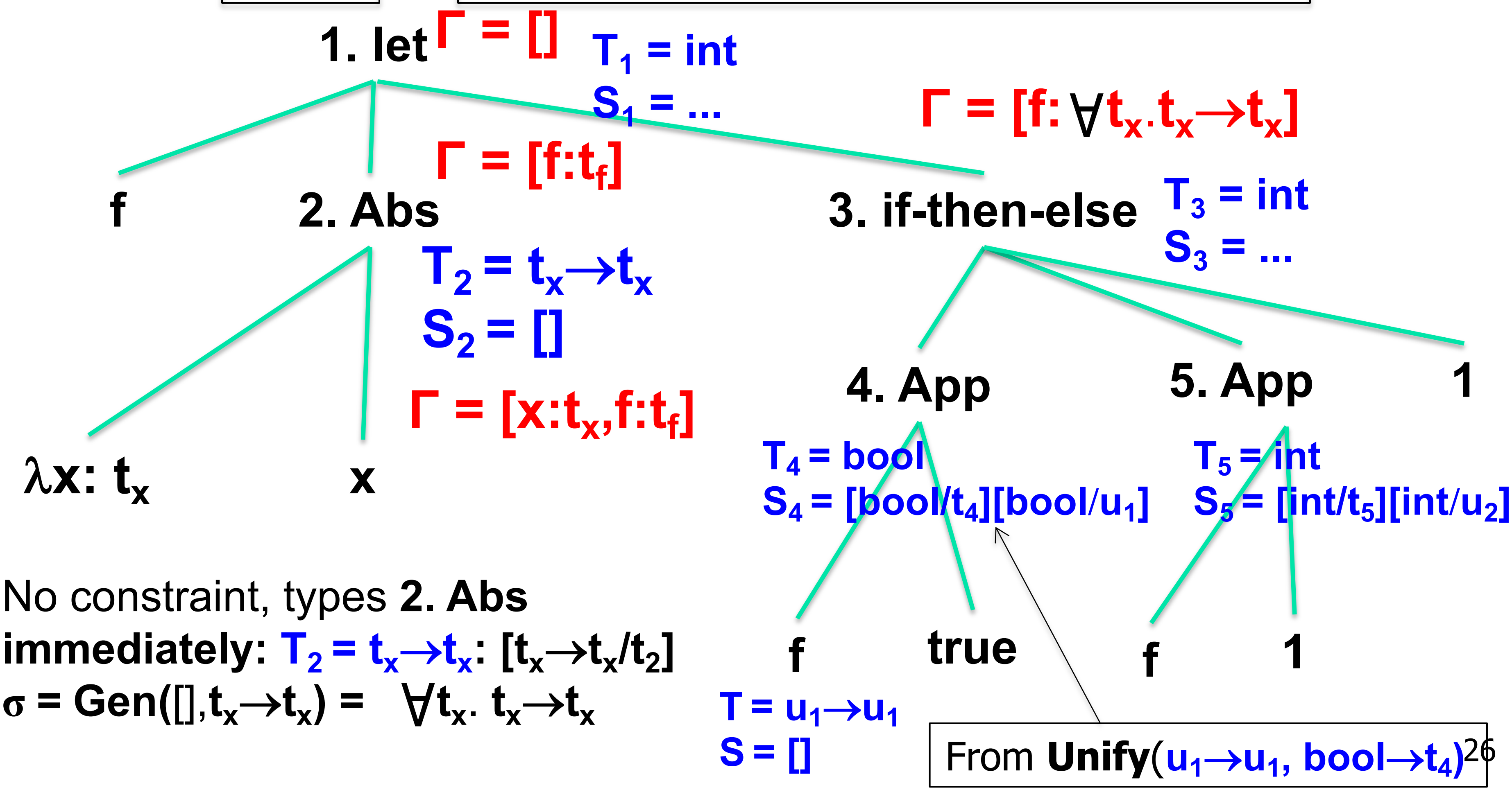
$\sigma = Gen(\ S\ S_{E1}(\Gamma),\ S(T_{E1})\ )$

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

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:** $T_2 = t_x \to t_x$: $[t_x \to t_x / t_2]$

σ = Gen($[], t_x \to t_x$) = $\forall t_x.\ t_x \to t_x$

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

$twice :: (t_x \to t_x) \to t_x \to t_x$

let twice f x = f (f x) $Gen([\,], (t_x \to t_x) \to t_x \to t_x) =$

in twice twice succ 4 // let-bound polymorphism

Hindley Milner Limitations

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

let twice f x = f (f x)

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

in foo twice

Hindley Milner Limitations

- Another example:

(λx. x (λy. y) (x 1)) (λz. z)

vs.

let x = (λz. z)

in

x (λy. 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 accessors
```

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:
  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 → 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!

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 → (a → m b) → m b**

// return encapsulates a value into the monad

return :: **a → 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

 li **>>=** f = concat (map f li)

 return x = [x]

Note: concat::[[**a**]] → [**a**]

e.g., concat [[1,2],[3,4],[5,6]] yields [1,2,3,4,5,6]

- Use **any f** s.t. **f::a→[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 → [Sheep]

parents **s** = MaybeToList (mother **s**) ++ MaybeToList (father **s**)

grandParents :: Sheep → [Sheep]

grandParents **s** = (parents **s**) >>= parents