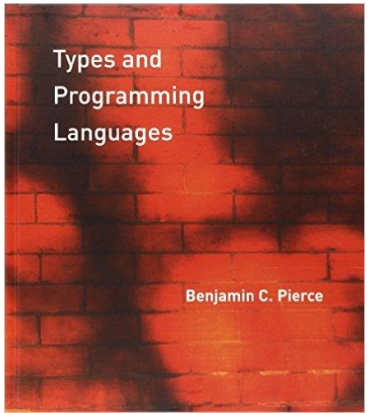
Concepts in Programming Languages
Recitation 11:

Hindley-Milner Type Inference

Yotam Feldman
(original slides by Kathleen Fisher, John Mitchell, Oded Padon, Mooly Sagiv)

Reference:
Types and Programming Languages
by Benjamin C. Pierce, Chapter 22



The Type Inference Problem

- Input: A program without types (e.g., Lambda calculus)
- Output: A program with type for every expression (e.g., typed Lambda calculus)
  - Every expression is annotated with its most general type

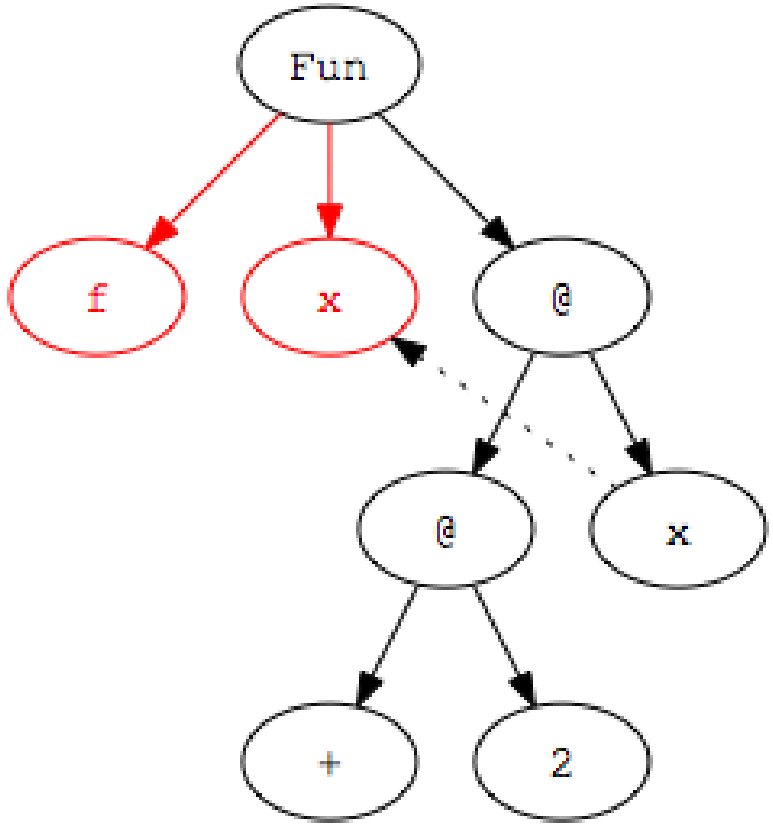
Type Inference Algorithm

- Parse program to build parse tree
- Assign type variables to nodes in tree
- Generate constraints:
  - From environment: literals (2), built-in operators (+), known functions (tail)
  - From structure of parse tree: e.g., application and abstraction nodes
- Solve constraints using *unification*
- Determine types of top-level declarations

Step 1: Parse Program

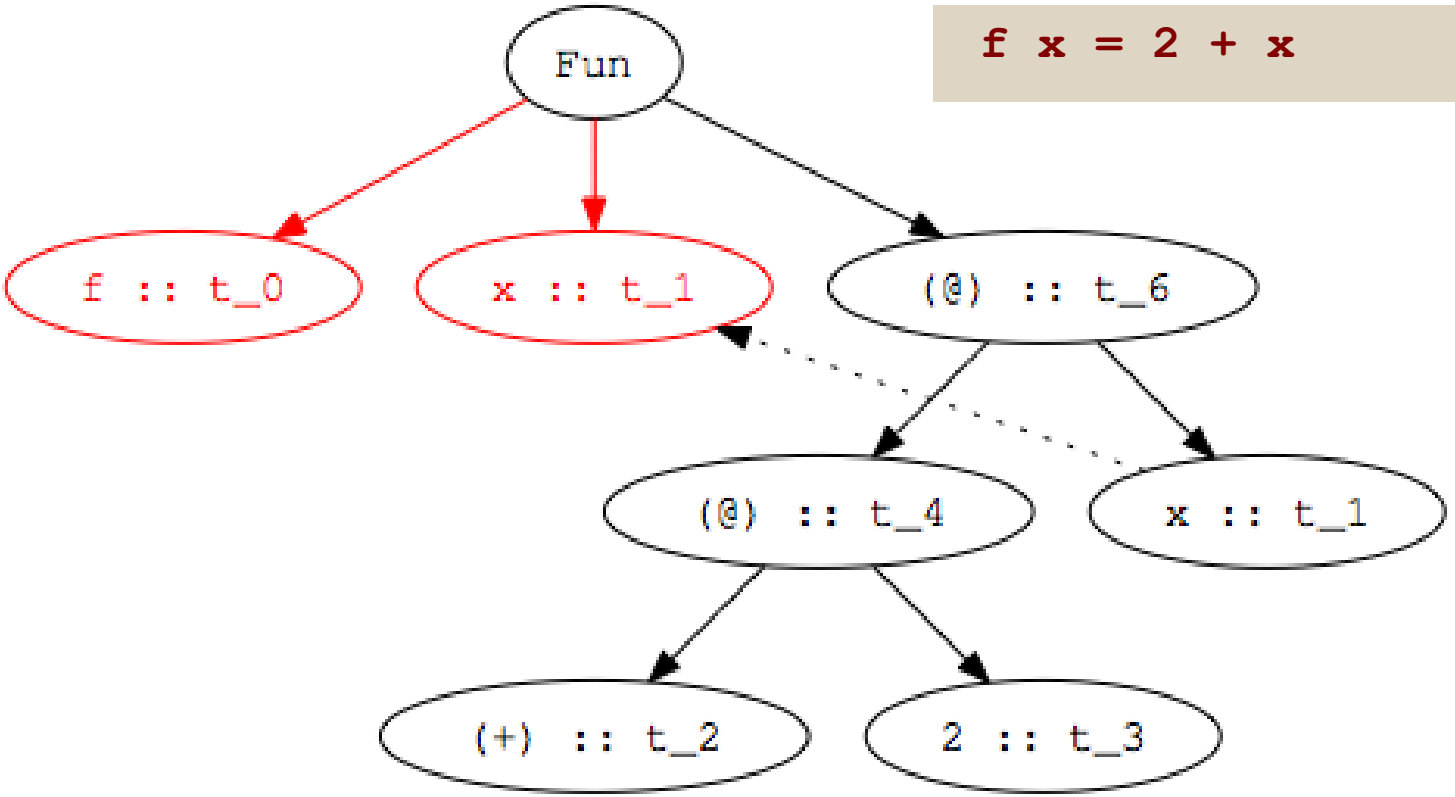
- Parse program text to construct parse tree

```
let f x = 2 + x
```



Step 2: Assign type variables to nodes

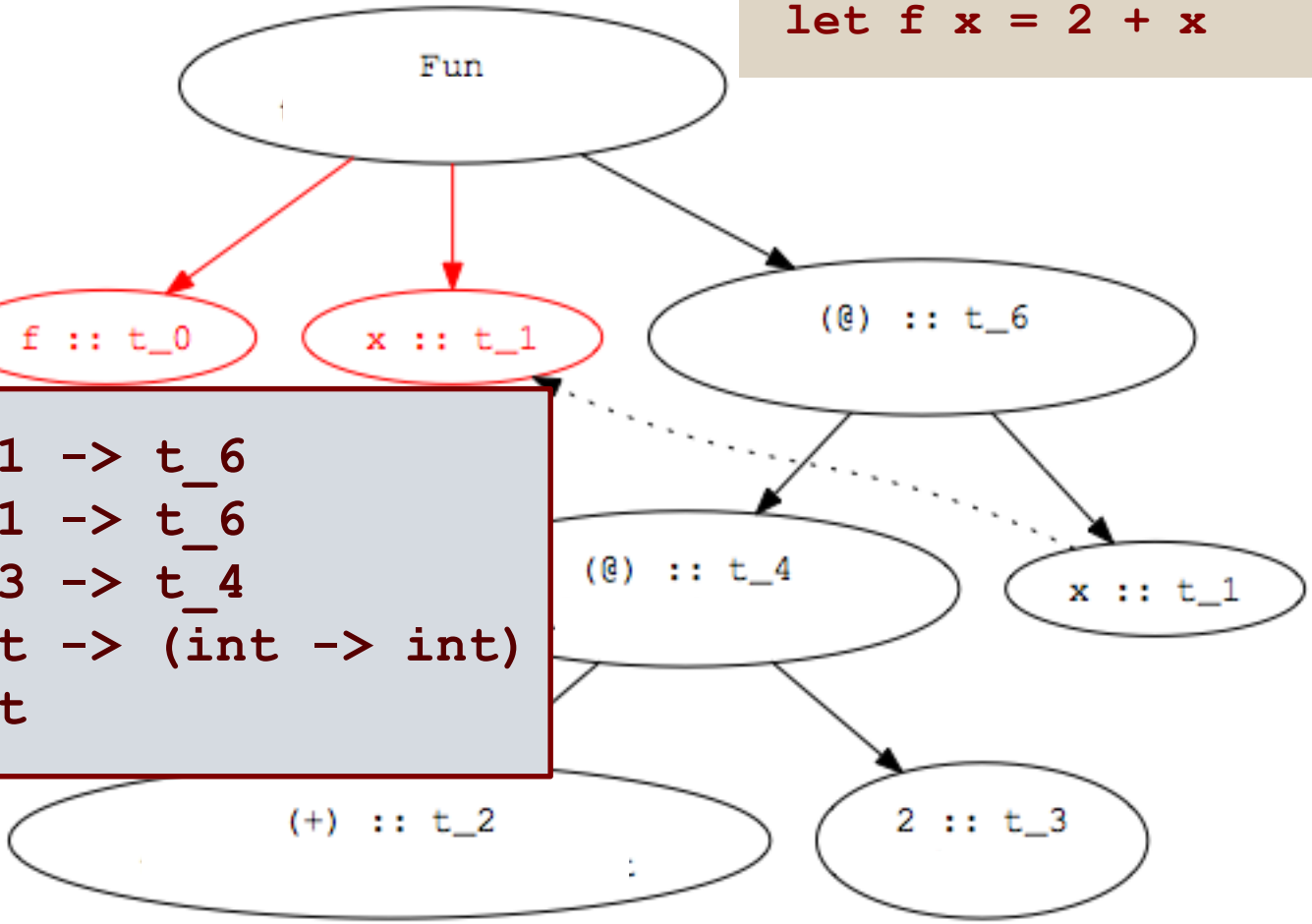
```
f x = 2 + x
```



Variables are given same type as binding occurrence

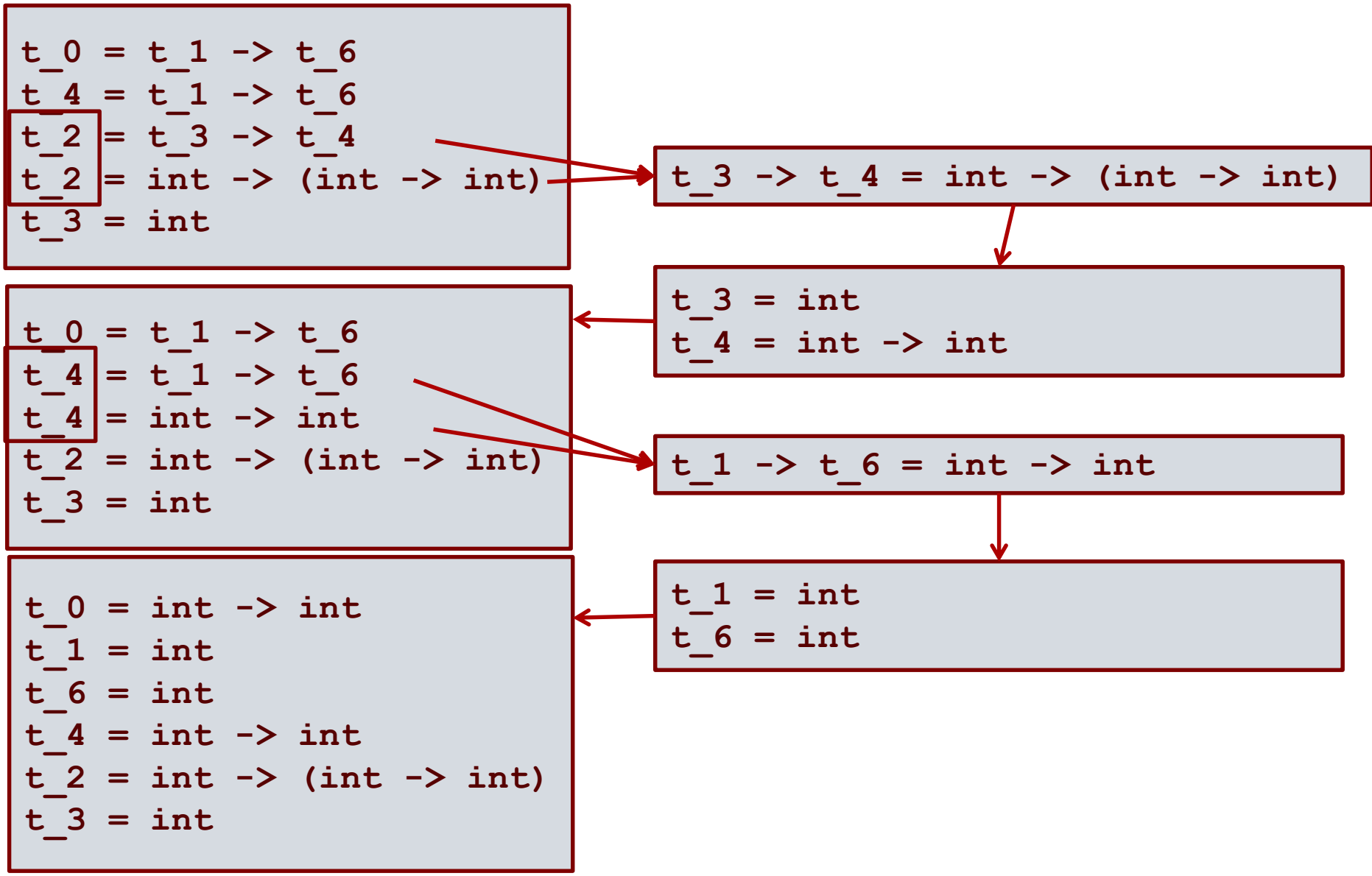
Step 3: Add Constraints

```
let f x = 2 + x
```




```
t_0 = t_1 -> t_6
t_4 = t_1 -> t_6
t_2 = t_3 -> t_4
t_2 = int -> (int -> int)
t_3 = int
```

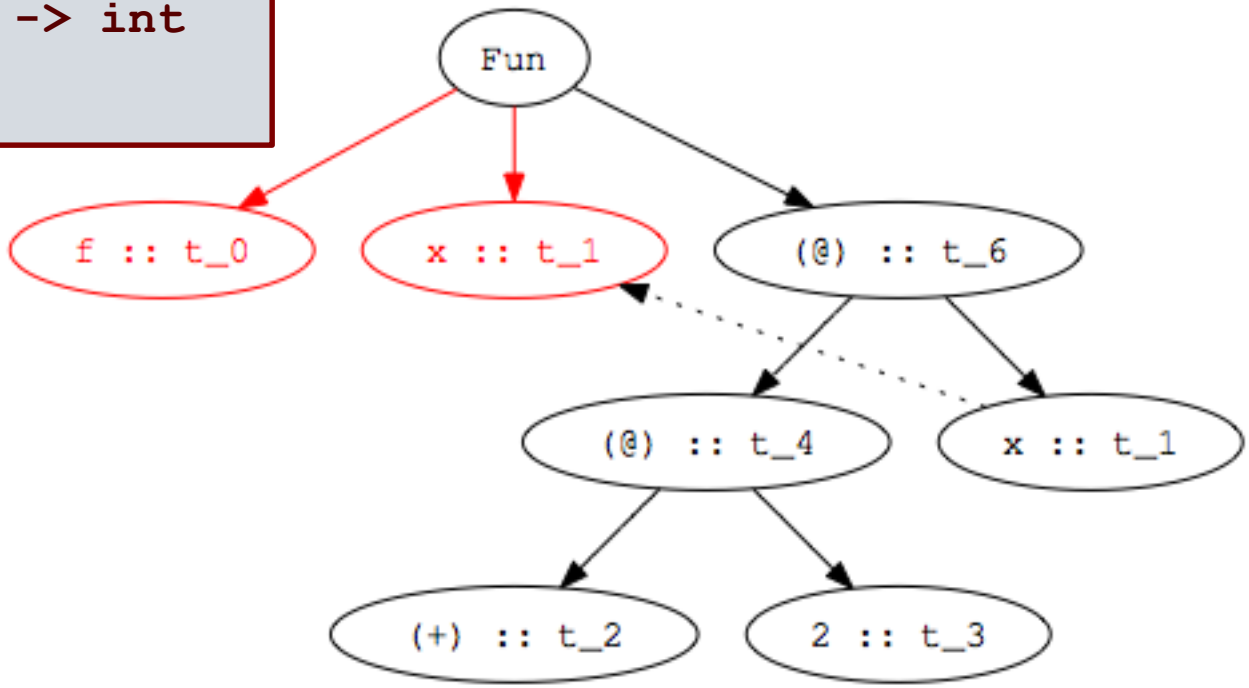
Step 4: Solve Constraints



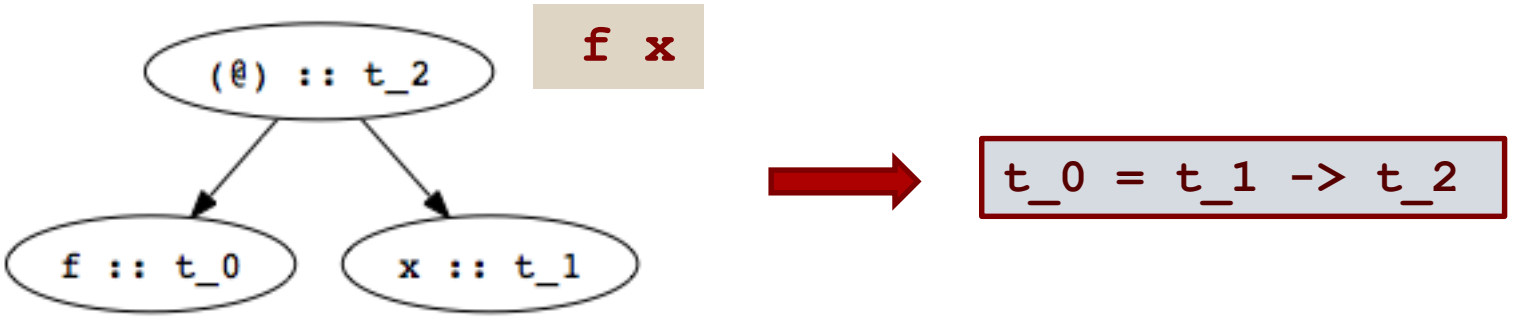
Step 5:
Determine type of declaration

```
t_0 = int -> int
t_1 = int
t_6 = int -> int
t_4 = int -> int
t_2 = int -> int -> int
t_3 = int
```

```
let f x = 2 + x
val f : int -> int =<fun>
```

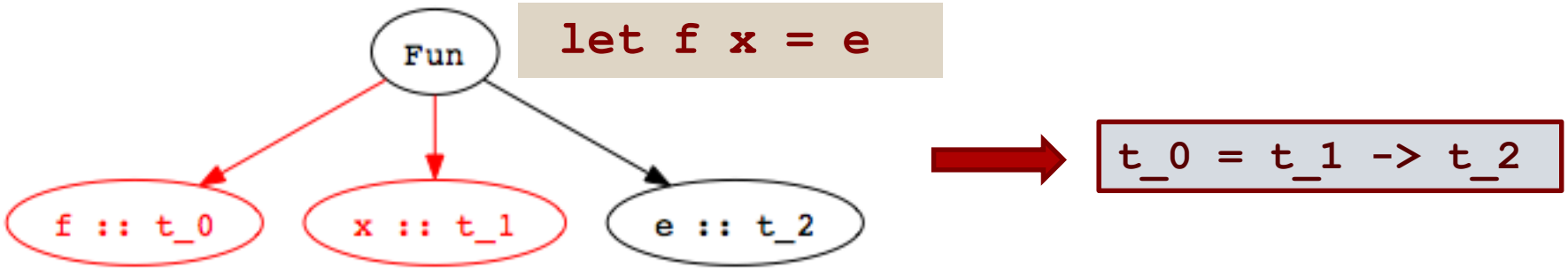


Constraints from Application Nodes



- Function application (apply f to x)
  - Type of f (t_0 in figure) must be domain → range
  - Domain of f must be type of argument x (t_1 in fig)
  - Range of f must be result of application (t_2 in fig)
  - Constraint: t_0 = t_1 -> t_2

Constraints from Abstractions



- Function declaration:
  - Type of f (t_0 in figure) must domain → range
  - Domain is type of abstracted variable x (t_1 in fig)
  - Range is type of function body e (t_2 in fig)
  - Constraint: t_0 = t_1 -> t_2

Example

let f x = (x + 5)

let f:t_1 x:t_2 = (x:t_2 +:t_3 5:t_4):t_5

Constraints:

- $t_1 = t_2 \rightarrow t_5$
- $t_3 = t_2 \rightarrow t_4 \rightarrow t_5$
- $t_4 = int$
- $t_3 = int \rightarrow int \rightarrow int$

Solution:

- $t_2 = int$
- $t_4 = int$
- $t_3 = int \rightarrow int \rightarrow int$
- $t_5 = int$
- $t_1 = int \rightarrow int$

f: $int \rightarrow int$

Example

```
let f    x    = (x    +    5.   )
let f:t1 x:t2 = (x:t2 +:t3 5.:t4):t5
```

Constraints:

- $t_1 = t_2 \rightarrow t_5$
- $t_3 = t_2 \rightarrow t_4 \rightarrow t_5$
- $t_4 =$ float
- $t_3 =$ int $\rightarrow$ int $\rightarrow$ int

Solution:

None,

Unification error

Type error

Example

```
fun f     x     = (x    +    y    )
fun f:t1  x:t2  = (x:t2 +:t3 y:?):t5
```

Type error

Example: Recursive Unification

fun f:t_1 x:t_2 = (x:t_2 x:t_2):t_3

Constraints:

- $t_1 = t_2 \rightarrow t_3$
- $t_2 = t_2 \rightarrow t_3$

Solution:
None,
Unification error

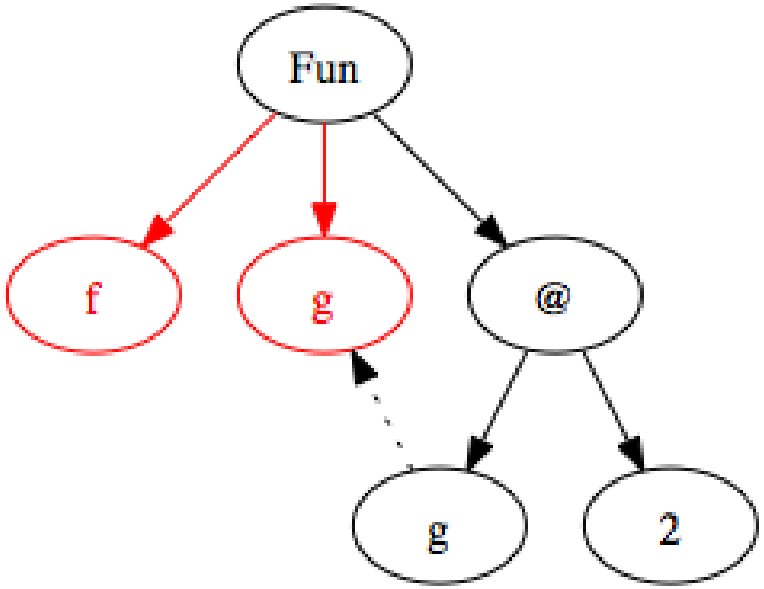
Type error

Inferring Polymorphic Types

- Example:

```
let f g = g 2
val f : (int -> t_4) -> t_4 = <fun>
```

- Step 1:
  Build Parse Tree

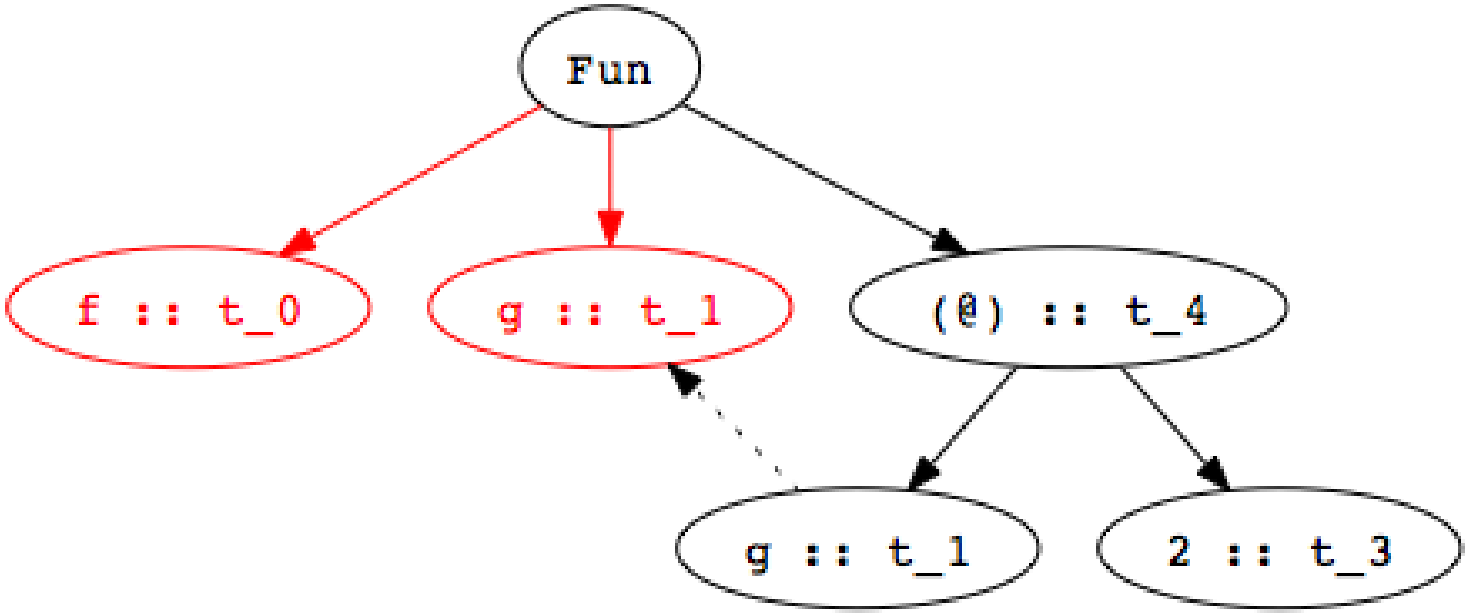


Inferring Polymorphic Types

- Example:

```
let f g = g 2
val f : (int -> t_4) -> t_4 = fun
```

- Step 2:
  Assign type variables



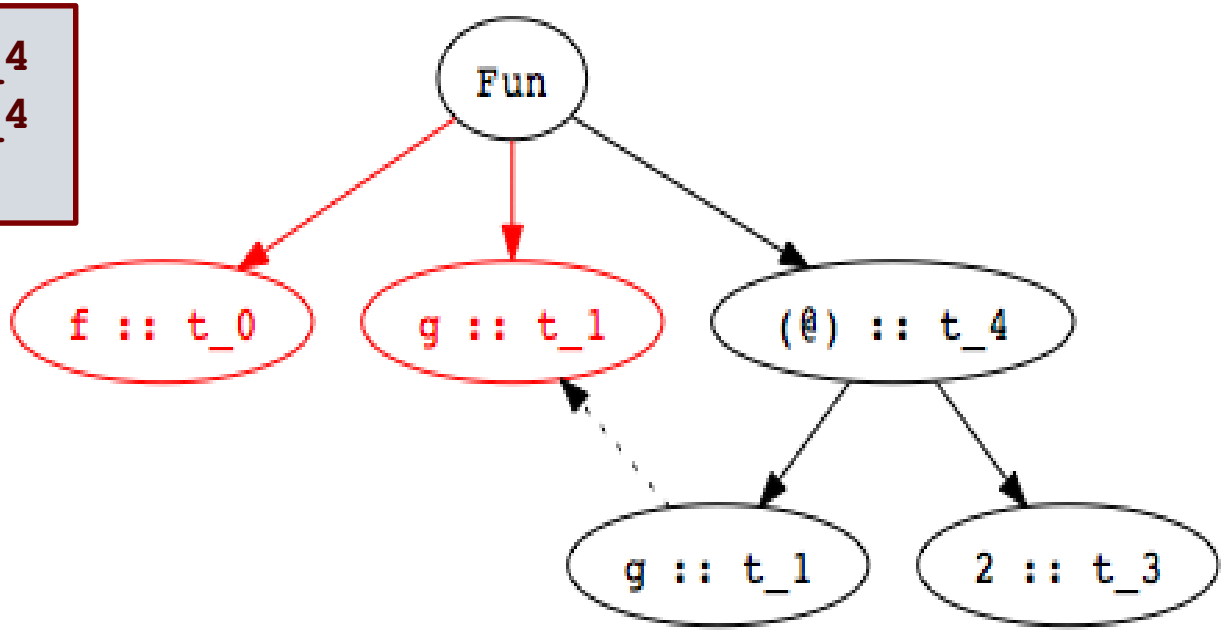
Inferring Polymorphic Types

- Example:

```
let f g = g 2
val f : (int -> t_4) -> t_4 = <fun>
```

- Step 3:
  Generate constraints

```
t_0 = t_1 -> t_4
t_1 = t_3 -> t_4
t_3 = int
```



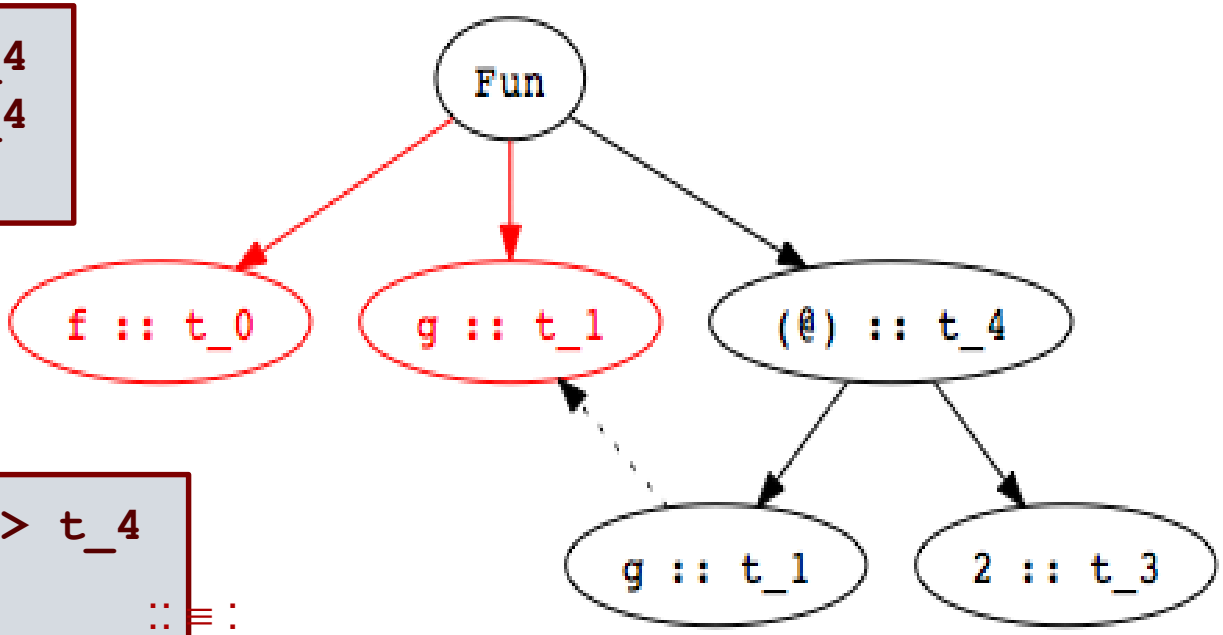
Inferring Polymorphic Types

- Example:

```
let f g = g 2
val f : (int -> t_4) -> t_4 = <fun>
```

- Step 4:
  Solve constraints

```
t_0 = t_1 -> t_4
t_1 = t_3 -> t_4
t_3 = int
```




```
t_0 = (int -> t_4) -> t_4
t_1 =  int -> t_4
t_3 =  int
```

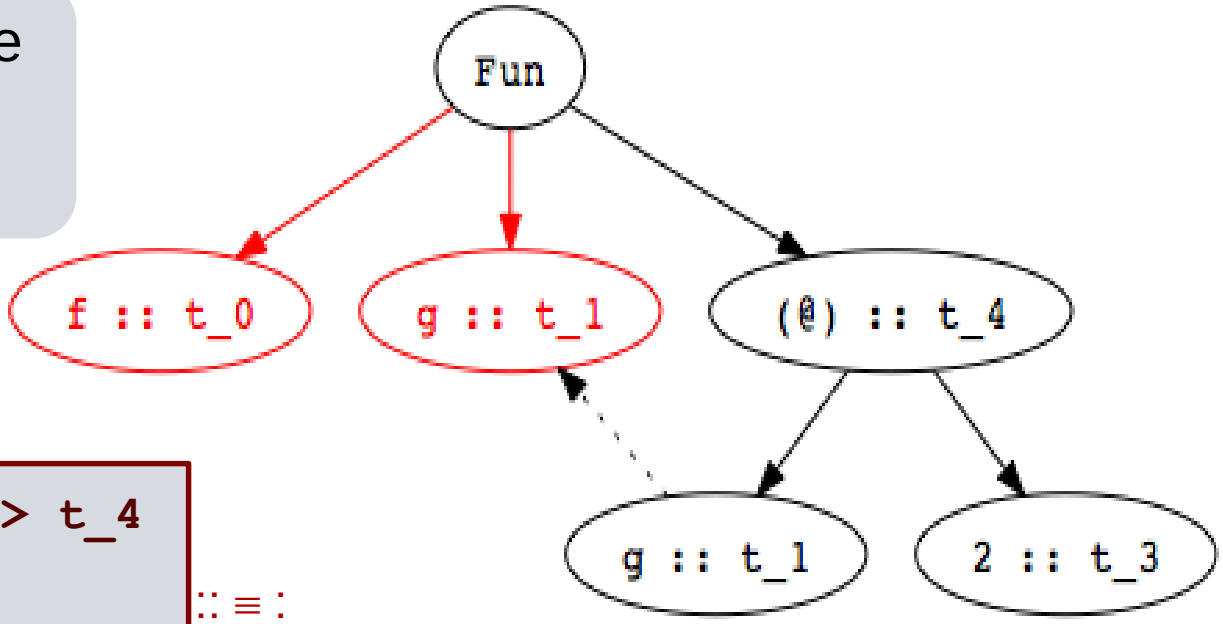
Inferring Polymorphic Types

- Example:

```
let f g = g 2
val f : (int -> t_4) -> t_4 = <fun>
```

- Step 5:
  Determine type of top-level declaration

Unconstrained type variables become polymorphic types




```
t_0 = (int -> t_4) -> t_4
t_1 =  int -> t_4
t_3 =  int
```

:: ≡ :

Example: Parametric Polymorphism

let f:t_1 x:t_2 y:t_3 = (x:t_2 y:t_3):t_4

Constraints:

- $t_1 = t_2 \rightarrow t_3 \rightarrow t_4$
- $t_2 = t_3 \rightarrow t_4$

Solution:

$t_2 = t_3 \rightarrow t_4$

$t_1 = (t_3 \rightarrow t_4) \rightarrow t_3 \rightarrow t_4$

f: $(a \rightarrow b) \rightarrow a \rightarrow b$

Pattern Matching

- Matching with multiple cases

```
let isempty l = match l with
    |[] -> true
    | _ -> false
```

- Infer type of each case
  - First case:

```
[t_1] -> bool
```

  - Second case:

```
t_2 -> bool
```

- Combine by unification of the types of the cases

```
val isempty : [t_1] -> bool = <fun>
```

Bad Pattern Matching

- Matching with multiple cases

```
let isempty l = match l with
    |[] -> true
    | _ -> 0
```

- Infer type of each case
  - First case:

```
[t_1] -> bool
```

  - Second case:

```
t_2 -> int
```

- Combine by unification of the types of the cases

```
Type Error: cannot unify bool and int
```

Recursion

```
let rec concat a b = match a with
    | [] -> b
    | x::xs -> x :: concat xs b
```

- To handle recursion, introduce type variables for the function:

```
concat : t_1 -> t_2 -> t_3
```

- Use these types to conclude the type of the body:
  - Pattern matching first case:

```
[t_4] -> t_5 -> t_5
unify [t_4] with t_1,t_5 with t_2,
                        t_5 with t_3
t_1 =[t_4] and t_2 = t_3 = t_5
```

  - Pattern matching second case:

```
[t_6] -> t_7 -> [t_6]
unify [t_6] with t_1, t_7 with t_2,
                        [t_6] with t_3
```

```
unify [t_6] with t_1, t_7 with t_2,
 t_3  with [t_6]
```

Recursion

```
let rec concat a b = match a with
    | [] -> b
    | x::xs -> x :: concat xs b
```

- To handle recursion, introduce type variables for the function:

```
concat : t_1 -> t_2 -> t_3
```

- Conclude the type of the function:

```
val concat : [t_4] -> [t_4] -> [t_4] = <fun>
```

Example: Recursion on Lists

```
let rec length xs =
  (match xs with
    | [] -> 0
    | (h :: t)->
        1 + (length t)
   )
```

Example: Recursion on Lists

```
let rec length:t1 xs:t2 =
  (match xs:t2 with
    | []:[t4] -> 0:int
    | (h:t5 :: t:t6):t7 ->
        1:int +:int->int->int
          (length:t1 t:t6):t8
  ):t3
```

Constraints and Unification

$t_1 = t_2 \rightarrow t_3$	`lambda`
$t_2 = [t_4]$	`match`
$t_2 = t_7$	`match`
$t_3 = int$	`match`
$t_3 = t_8$	`match`
$t_6 = [t_5]$	`apply (::)`
$t_7 = [t_5]$	`apply (::)`
$t_1 = t_6 \rightarrow t_8$	`apply (length)`
$t_8 = int$	`apply (+)`

---

length : $[t_4] \rightarrow int$

```
let rec length:t1 xs:t2 =
  (match xs:t2 with
    | []:[t4] -> 0:int
    | (h:t5 :: t:t6):t7 ->
        1:int +:int->int->int
          (length:t1 t:t6):t8
  ):t3
```

Hindley-Milner Implementation

Let Polymorphism

- let g = (fun f -> 5) in (g g)
- **Let**: (1) analyse definition
  - **Lambda**:
    - f: t_1
    - Body: **identifier**, int
  - g: $t_1 \rightarrow$int
- (2) Analyse body, **Application**. Context: {g: $t_1 \rightarrow$int}
  - Function: **identifier**, g: $t_1 \rightarrow$int
  - Argument: **identifier**, g: $t_1 \rightarrow$int
  - **Unify**: function = argument $\rightarrow t_2$
    - $t_1 \rightarrow$int = $(t_1 \rightarrow$int$) \rightarrow t_2$
    - Thus, $t_1 = (t_1 \rightarrow$int) and int=t_2
- Type of entire expression: t_2=int

Recursive unification error!

Let Polymorphism

- let g = (fun f -> 5) in (g g)
- **Let**: (1) analyse definition
  - **Lambda**:
    - f: t_1
    - Body: **identifier**, int
  - g: $t_1 \rightarrow$int
- (2) Analyse body, **Application**. Context: {g: $t_1 \rightarrow$int}
  - Function: **identifier**, g: $t_{11} \rightarrow$int
  - Argument: **identifier**, g: $t_{12} \rightarrow$int
  - **Unify**: function = argument $\rightarrow t_2$
    - $\mathbf{t_{11}} \rightarrow$int = ($\mathbf{t_{12}} \rightarrow$int)$\rightarrow t_2$
    - Thus, $\mathbf{t_{11}}$= ($\mathbf{t_{12}} \rightarrow$int) and int=$t_2$
- Type of entire expression: t_2=int

Lambda Polymorphism?

- fun f -> f f
- **Lambda**:
  - f: t_1
  - Body: **Application**, Context {f: t_1}
    - Function: **identifier**, f: t_{11}
    - Argument: **identifier**, f: t_{12}
    - **Unify**: function = argument → t_2
      - $t_{11} = t_{12} \rightarrow t_2$
- All is well (?)
- **No**: **Lambda**-bound parametric polymorphism not supported.
  - Higher-rank types make type inference undecidable
  - Non-trivial tradeoff with the need to warn on programmer errors