CMSC 330: Organization of Programming Languages

Type Inference and Unification

Type Checking vs Type Inference

- Type checking: use declared types to check types are correct

```
let apply (f:('a->'b)) (x:'a):'b = f x
```

- Type inference:

```
let apply f x = f x
```

  - Infer the most general types that could have been declared, and type checks the code without the type information

The Type Inference Algorithm

- Input: A program without types
- Output: A program with type for every expression, which is annotated with its most general type

Why do we want to infer types?

- Reduces syntactic overhead of expressive types
  - // C++ Declare a vector of vectors of integers
    std::vector<std::vector<int>> matrix;
- Guaranteed to produce most general type
- Widely regarded as important language innovation
- Illustrative example of a flow-insensitive static analysis algorithm

History

- Original type inference algorithm
  - Invented by Haskell Curry and Robert Feys for the simply typed lambda calculus in 1958
- In 1969, Hindley
  - extended the algorithm to a richer language and proved it always produced the most general type
- In 1978, Milner
  - independently developed equivalent algorithm, called algorithm W, during his work designing ML
- In 1982, Damas proved the algorithm was complete.
  - Currently used in many languages: ML, Ada, Haskell, C# 3.0, F#, Visual Basic .Net 9.0. Have been plans for Fortress, Perl 6, C++0x,…

Type Inference: Basic Idea

- Example

```
fun x -> 2 + x
-: int -> int = <fun>
```

- What is the type of the expression?
  - + has type: int $\rightarrow$ int $\rightarrow$ int
  - 2 has type: int
  - Since we are applying + to x we need x : int
  - Therefore, **`fun x -> 2 + x`** has type int $\rightarrow$ int

Type Inference: Basic Idea

- Example

```
fun f => f 3
  -:(int → a) → a = <fun>
```

- What is the type of the expression?
  - 3 has type: int
  - Since we are applying f to 3 we need f : int → a and the result is of type a
  - Therefore, **`fun f → f 3`** has type (int → a) →a

Type Inference: Basic Idea

- Example

```
fun f → f (f 3)
```

- What is the type of the expression?

Type Inference: Basic Idea

- Example

```
fun f → f (f “hi”)
```

- What is the type of the expression?

Type Inference: Basic Idea

- Example

```
fun f → f (f 3, f 4)
```

- What is the type of the expression?

Type Inference: Complex Example

```
let square = fun z → z * z in
    fun f → fun x → fun y →
    if (f x y) then (f (square x) y)
    else (f x (f x y))
```

```
* : int → (int → int)
```

```
z : int
```

```
square : int → int
```

```
f : 'a → ('b → bool), x: 'a, y: 'b
```

```
a: int
```

```
b: bool
```

```
(int → bool → bool) →int →bool → bool
```

Unification

- Unification is an algorithmic process of solving equations between symbolic expressions
- Unifies two terms
- Used for pattern matching and type inference
- Simple examples
  - int * x and y * (bool * bool) are **unifiable**
    - y = int
    - x = (bool * bool)
  - int * int and int * bool are **not unifiable**

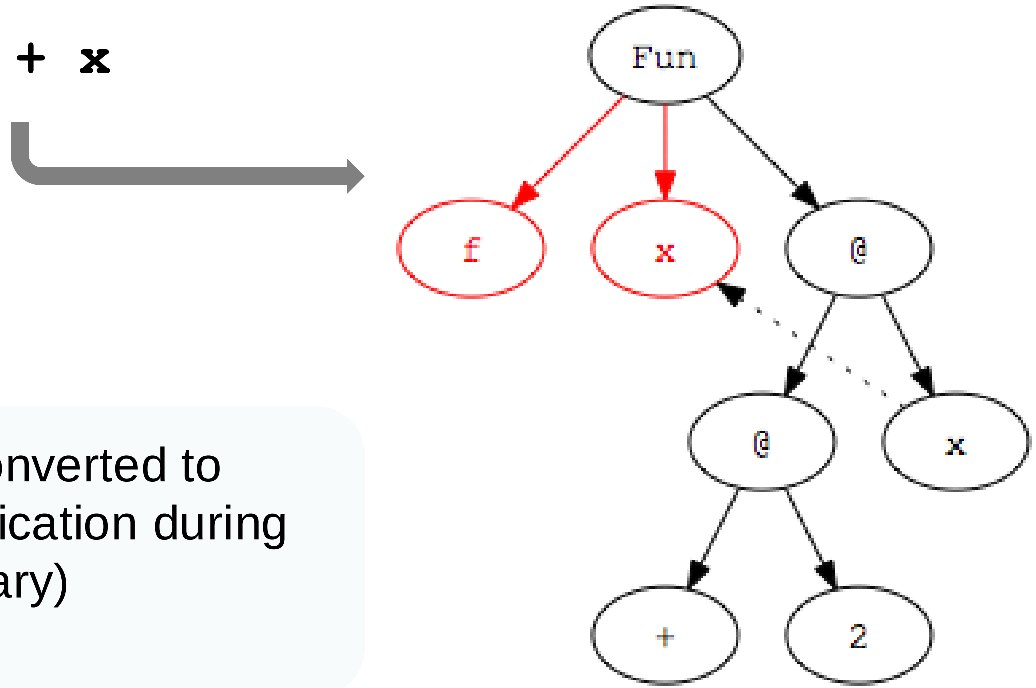
Type Inference Algorithm

- 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 form 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

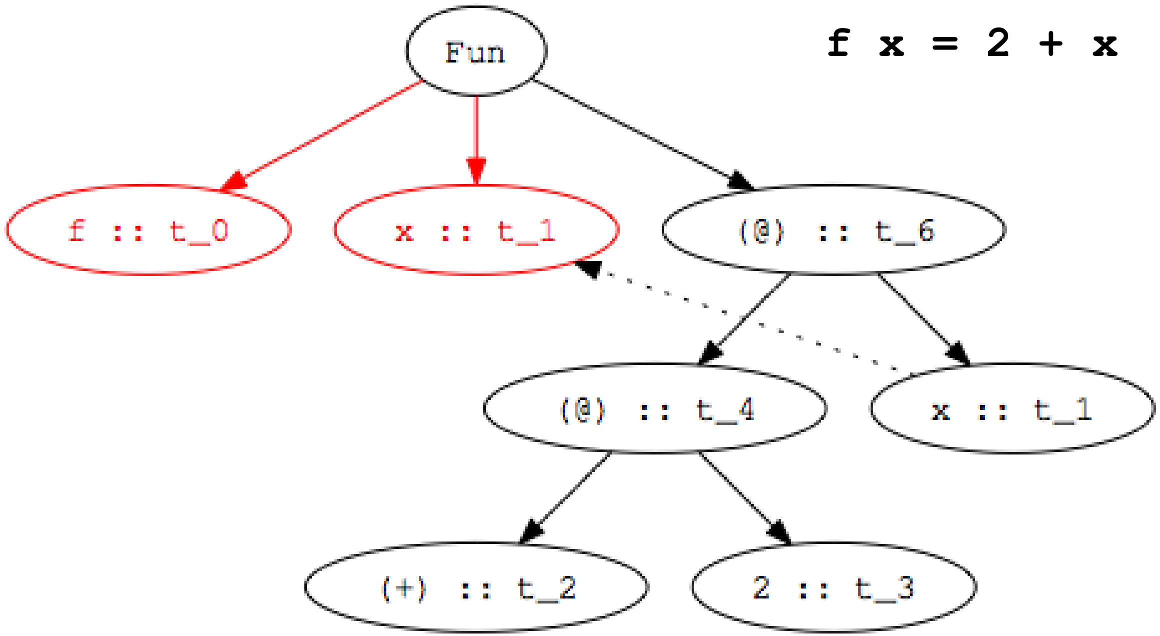
```
let f x = 2 + x
```



Infix operators are converted to Curried function application during parsing: (not necessary)

2 + x → (+) 2 x

Step 2: Assign type variables to nodes



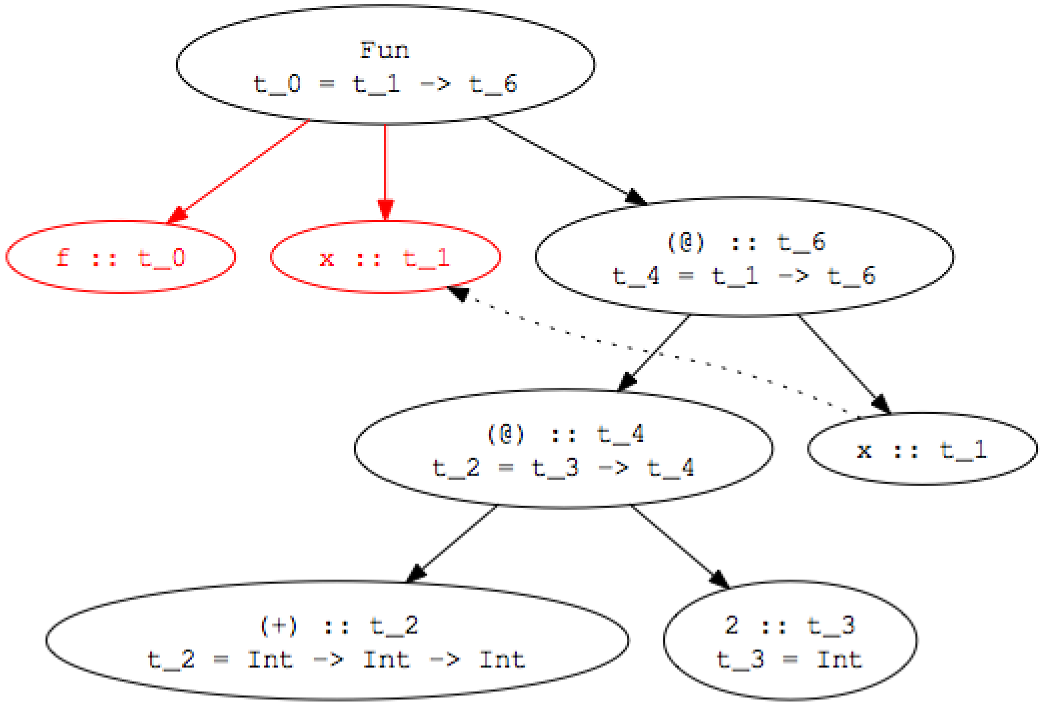
f x = 2 + x

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

```
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
```

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

```
t_3 = int
t_4 = int -> int
```

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

```
t_1 -> t_6 = int -> int
```

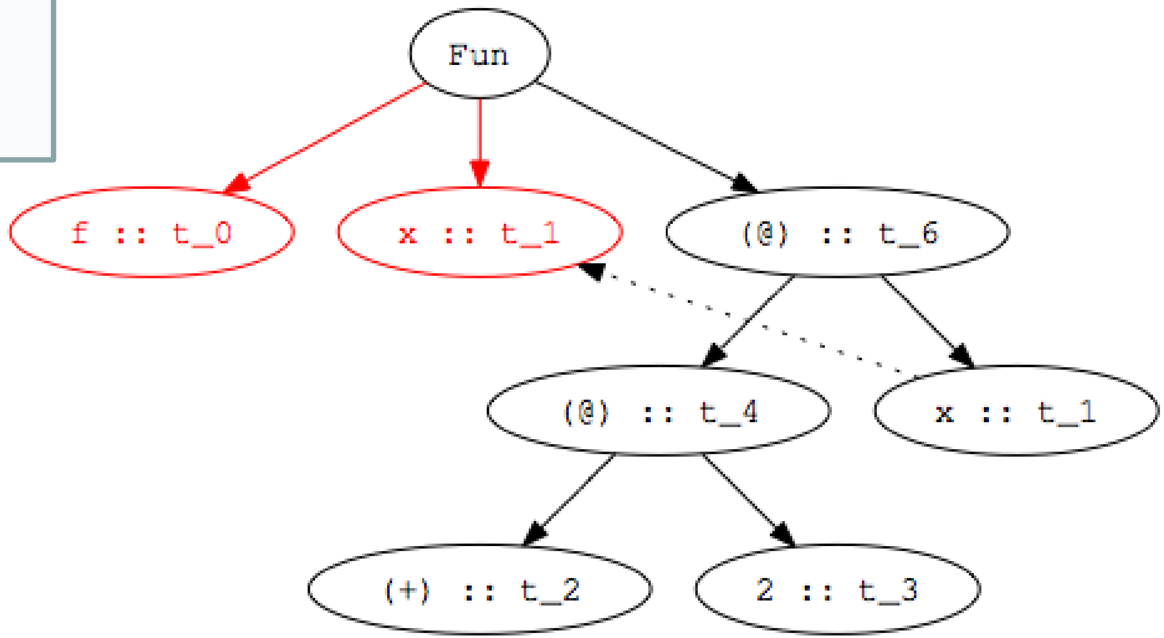
```
t_1 = int
t_6 = int
```

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

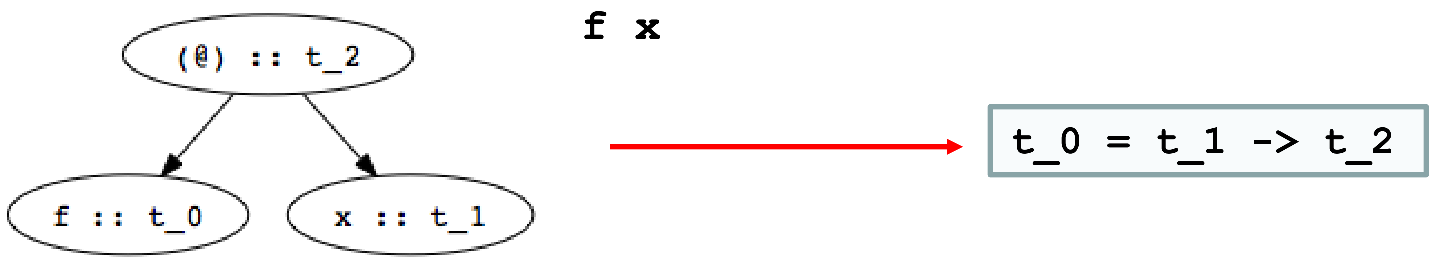
Step 5: Determine type of declaration

```
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>
```

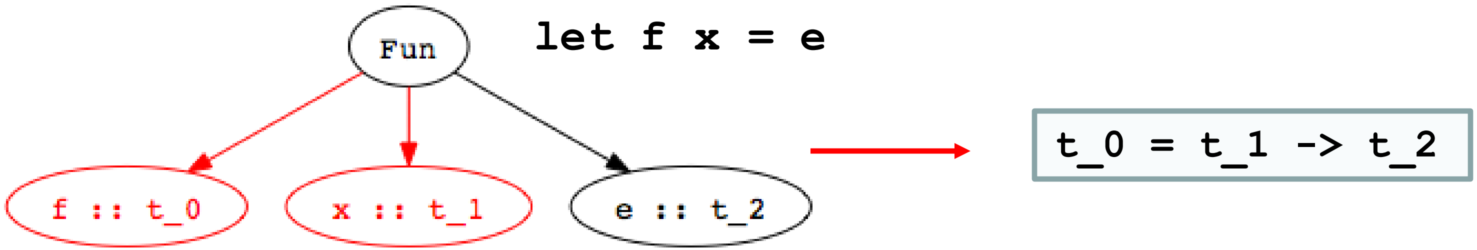


Constraints from Application Nodes



- 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

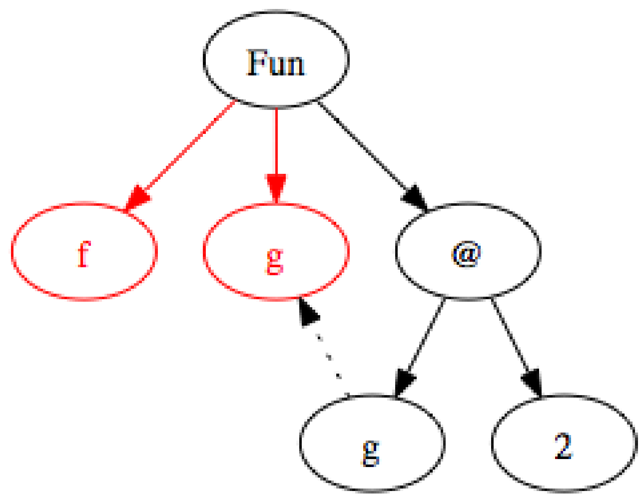
Constraints from Abstractions



- Function declaration:
  - Type of f (t_0 in figure) must domain $\rightarrow$ 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

Inferring Polymorphic Types

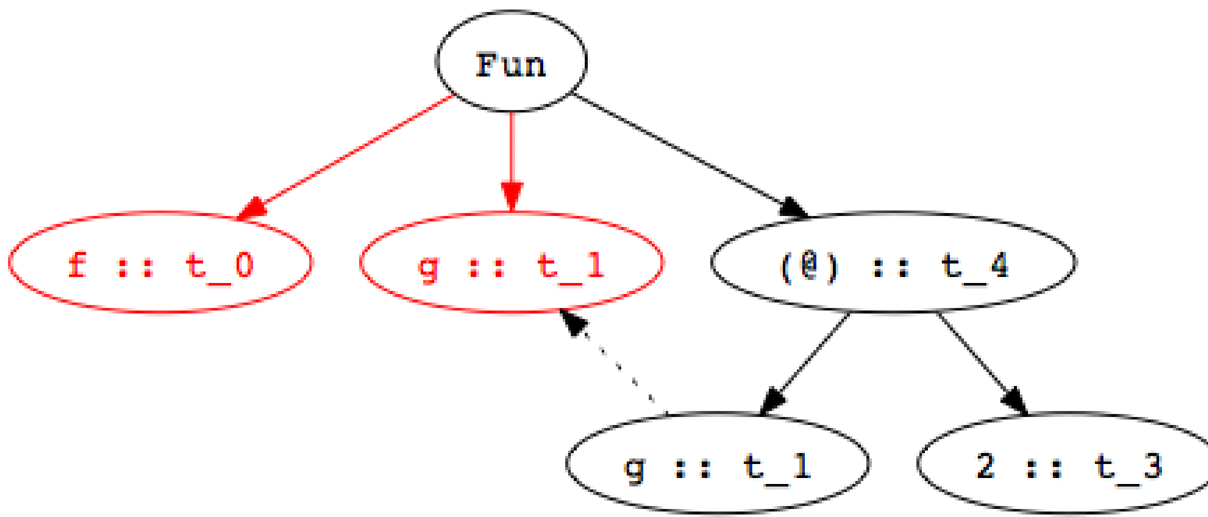
- Example:
  ```
  let f g = g 2
  ```
- Step 1:
  ```
  val f : (int -> t_4) -> t_4 = <fun>
  ```
  - Build Parse Tree



Inferring Polymorphic Types

- Example:
- Step 2:
  - Assign type variables

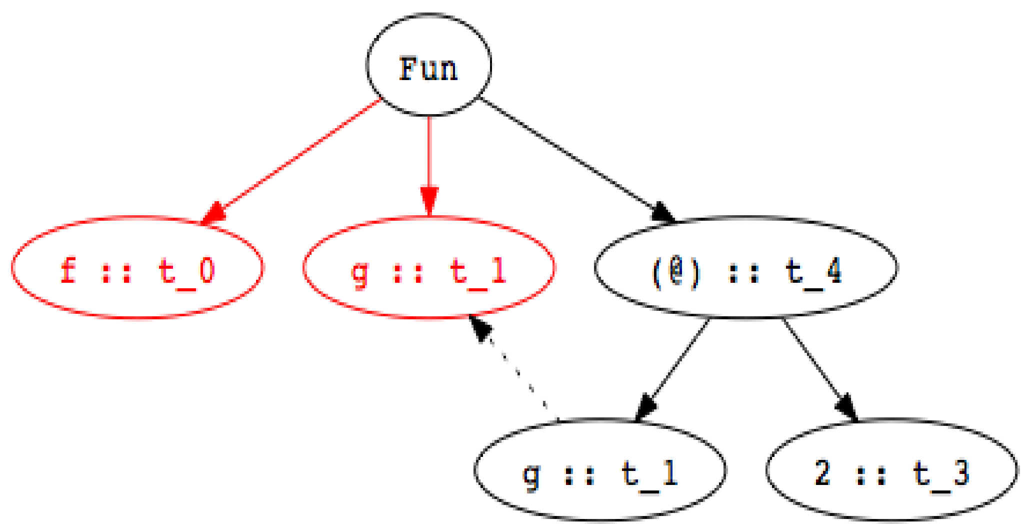
```
let f g = g 2
val f : (int -> t_4) -> t_4 = fun
```



Inferring Polymorphic Types

- Example:
```
let f g = g 2
```
- Step 3:
```
val f : (int -> t_4) -> t_4 = <fun>
```
  - Generate constraints

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



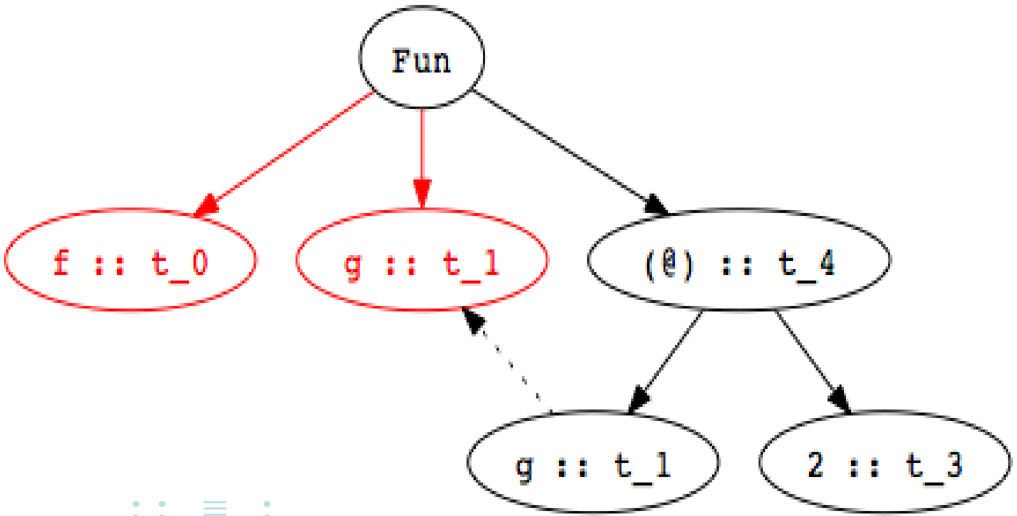
Inferring Polymorphic Types

- Example:
- Step 4:
  - Solve constraints

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

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

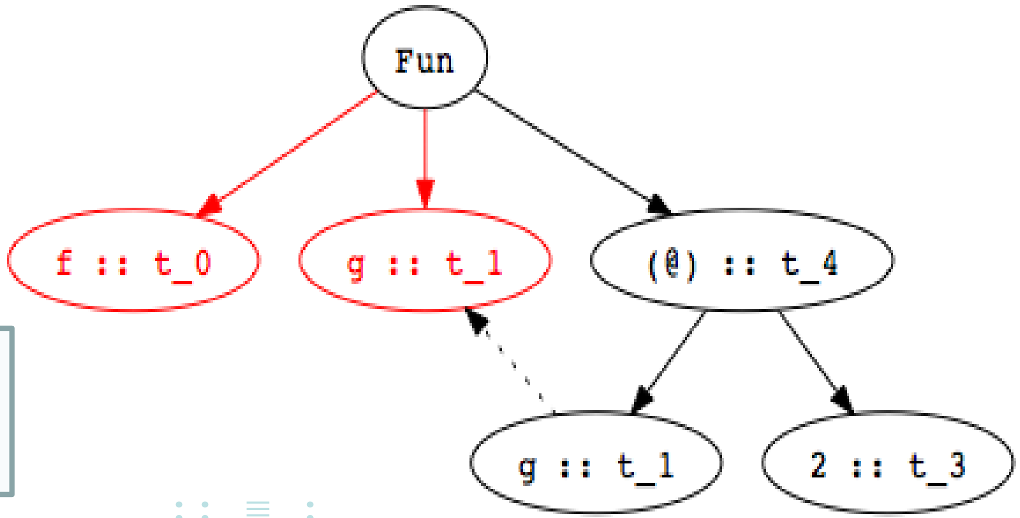
```
t_0 = (int -> t_4) -> t_4
t_1 =  int -> t_4
t_3 =  int
```



Inferring Polymorphic Types

- Example: `let f g = g 2`
- Step 5: `val f : (int -> t_4) -> t_4 = <fun>`
  - 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
```

Using Polymorphic Functions

- Function: `let f g = g 2`

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

- Possible applications:

```
let add x = 2 + x
val add : int -> int = <fun>
f add
:- int = 4
```

```
let isEven x = mod (x, 2) == 0
val isEven: int -> bool = <fun>
f isEven
:- bool= true
```

Recognizing Type Errors

- Function:

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

- Incorrect use

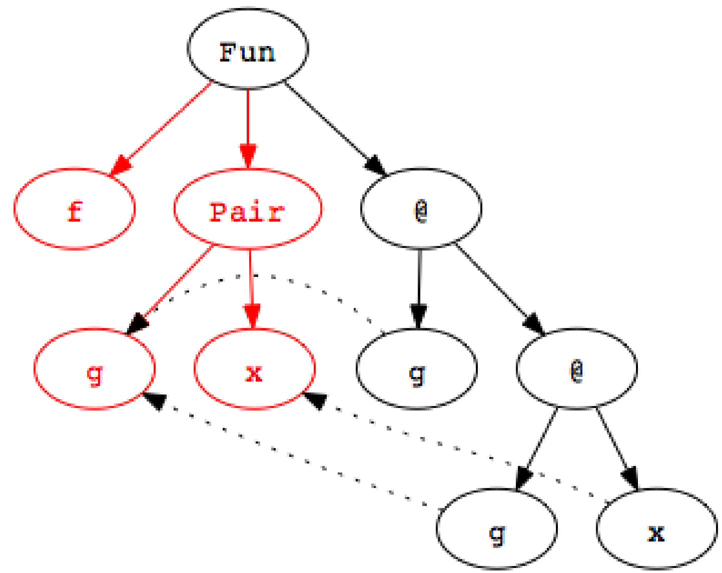
```
let not x = if x then true else false
val not : bool -> bool = <fun>
f not
> Error: operator and operand don't agree
  operator domain: int -> a
  operand:         bool-> bool
```

- Type error:
  cannot unify bool → bool and int → t

Another Example

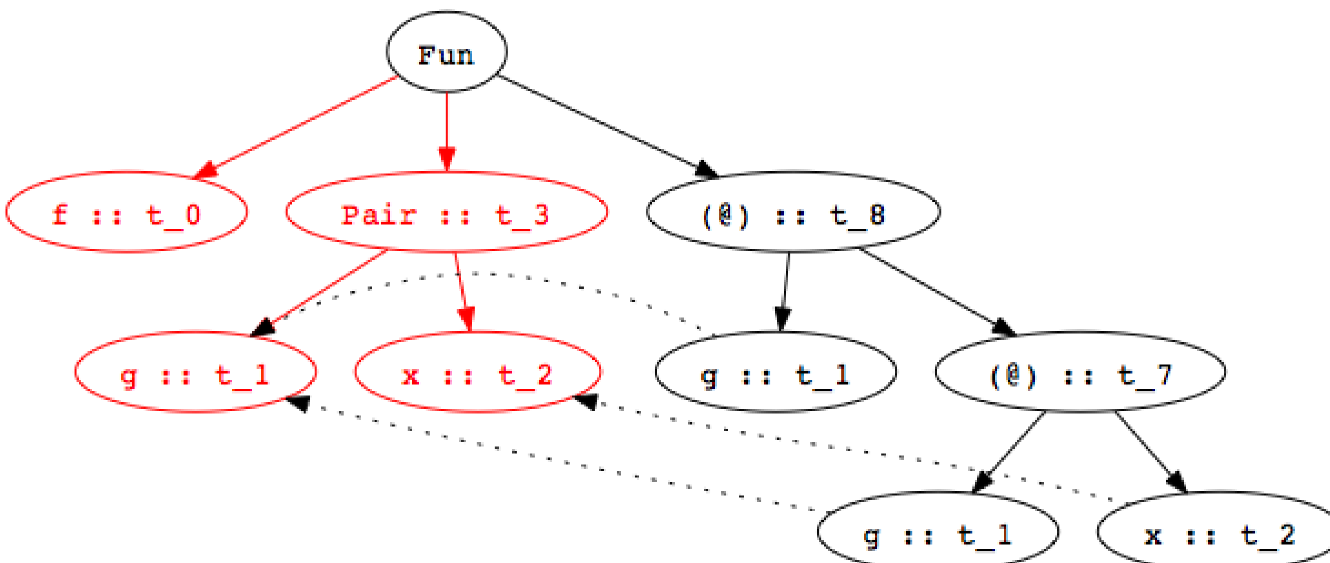
- Example:
- Step 1:
  - Build Parse Tree

```
let f (g,x) = g (g x)
val f : ((t_8 -> t_8) * t_8) -> t_8
```



Another Example

- Example: `let f (g,x) = g (g x)`
- Step 2: `val f : ((t_8 -> t_8) * t_8) -> t_8`
  - Assign type variables

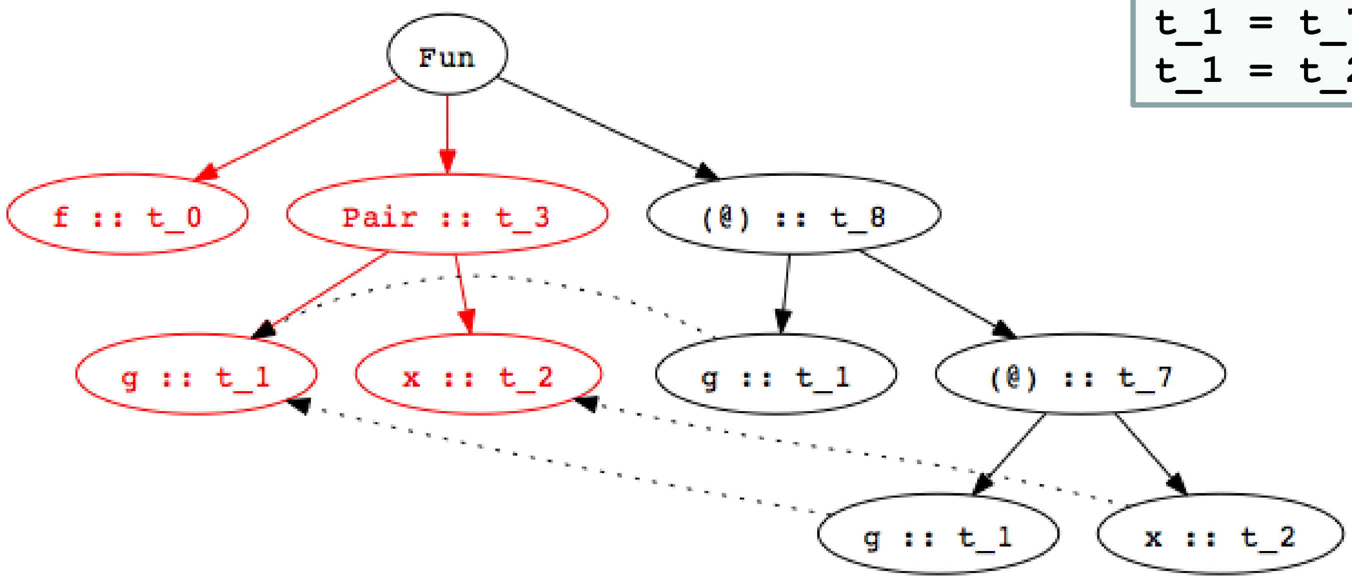


Another Example

- Example:
- Step 3:
  - Generate constraints

```
let f (g,x) = g (g x)
val f : ((t_8 -> t_8) * t_8) -> t_8
```

```
t_0 = t_3 -> t_8
t_3 = (t_1, t_2)
t_1 = t_7 -> t_8
t_1 = t_2 -> t_7
```

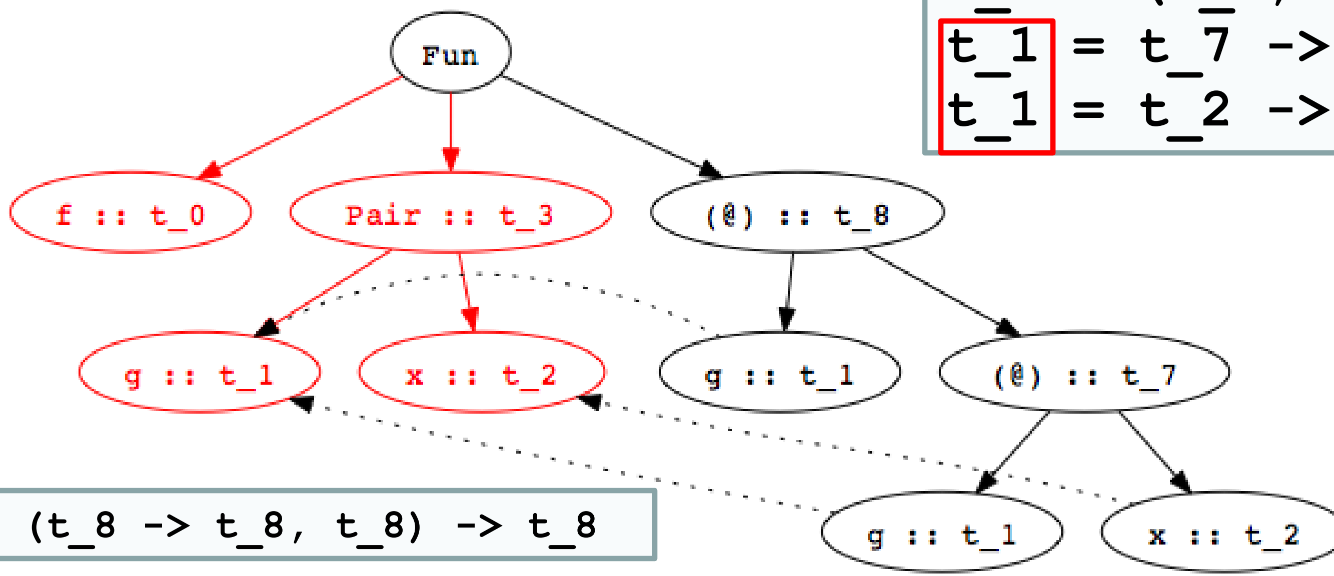


Another Example

- Example:
- Step 4:
  - Solve constraints

```
let f (g,x) = g (g x)
val f : ((t_8 -> t_8) * t_8) -> t_8
```

```
t_0 = t_3 -> t_8
t_3 = (t_1, t_2)
t_1 = t_7 -> t_8
t_1 = t_2 -> t_7
```



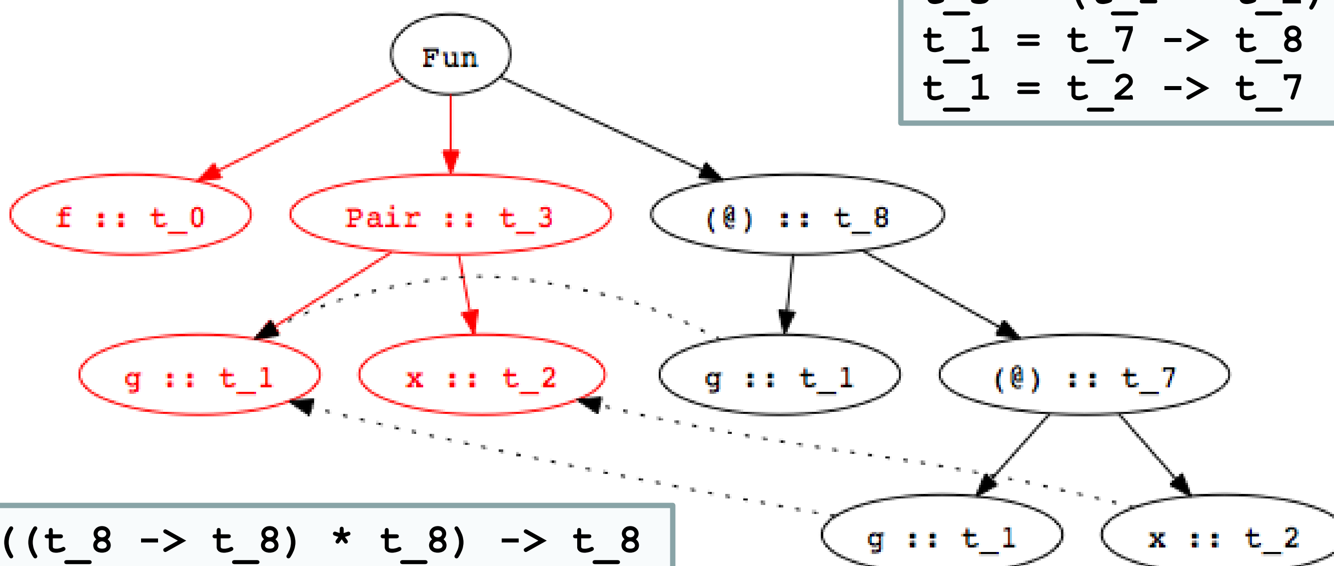

```
t_0 = (t_8 -> t_8, t_8) -> t_8
```

Another Example

- Example:
- Step 5:
  - Determine type of f

```
let f (g,x) = g (g x)
val f : ((t_8 -> t_8) * t_8) -> t_8
```

```
t_0 = t_3 -> t_8
t_3 = (t_1 * t_2)
t_1 = t_7 -> t_8
t_1 = t_2 -> t_7
```




```
t_0 = ((t_8 -> t_8) * t_8) -> t_8
```

Most General Type

- Type inference produces the *most general type*

```
let rec map f lst =
  match lst with
    [] -> []
   | hd :: tl -> f hd :: (map f tl)
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Functions may have many less general types

```
val map : (t_1  -> int, [t_1])  -> [int]
val map : (bool -> t_2, [bool]) -> [t_2]
val map : (char -> int, [cChar]) -> [int]
```

- Less general types are all instances of most general type, also called the *principal type*

Complexity of Type Inference Algorithm

- When Hindley/Milner type inference algorithm was developed, its complexity was unknown
- In 1989, Kanellakis, Mairson, and Mitchell proved that the problem was exponential-time complete
- Usually linear in practice though…
  - Running time is exponential in the depth of polymorphic declarations

Type Inference: Key Points

- Type inference computes the types of expressions
  - Does not require type declarations for variables
  - Finds the most general type by solving constraints
  - Leads to polymorphism
- Sometimes better error detection than type checking
  - Type may indicate a programming error even if no type error
- Some costs
  - More difficult to identify program line that causes error
  - Natural implementation requires uniform representation sizes
- Idea can be applied to other program properties
  - Discover properties of program using same kind of analysis

Example: Swap Two Values

- OCaml

```
let swap (x, y) =
 let temp = !x in
   (x := !y; y := temp)
val swap : 'a ref * 'a ref -> unit = <fun>
```

- C++

```
template <typename T>
void swap(T& x, T& y){
      T tmp = x;  x=y;  y=tmp;
}
```

Declarations both swap two values polymorphically, but they are compiled very differently

Implementation

- OCaml
  - `swap` is compiled into one function
  - Typechecker determines how function can be used
- C++
  - `swap` is compiled differently for each instance
    (details beyond scope of this course …)
- Why the difference?
  - OCaml ref cell is passed by pointer. The local `x` is a pointer to value on heap, so its size is constant
  - C++ arguments passed by reference (pointer), but local `x` is on the stack, so its size depends on the type

Polymorphism vs Overloading

- Parametric polymorphism
  - Single algorithm may be given many types
  - Type variable may be replaced by any type
  - if **`f:t→t`** then **`f:int→int`**, **`f:bool→bool`**, ...
- Overloading
  - A single symbol may refer to more than one algorithm
  - Each algorithm may have different type
  - Choice of algorithm determined by type context
  - Types of symbol may be arbitrarily different
  - In ML, **+** has types **`int*int→int, real*real→real`**, no others
  - Haskel permits more general overloading and requires user assistance

Varieties of Polymorphism

- Parametric polymorphism A single piece of code is typed generically
  - Imperative or first-class polymorphism
  - ML-style or let-polymorphism
- Ad-hoc polymorphism The same expression exhibit different behaviors when viewed in different types
  - Overloading
  - Multi-method dispatch
  - intentional polymorphism
- Subtype polymorphism A single term may have many types using the rule of subsumption allowing to selectively forget information

Summary

- Types are important in modern languages
  - Program organization and documentation
  - Prevent program errors
  - Provide important information to compiler
- Type inference
  - Determine best type for an expression, based on known information about symbols in the expression
- Polymorphism
  - Single algorithm (function) can have many types