

Types and Type Inference

Edward Z. Yang

previously revised by J Mitchell and K Fisher

Reading: “Concepts in Programming Languages”,
Revised Chapter 6 - handout on Web!!

Outline

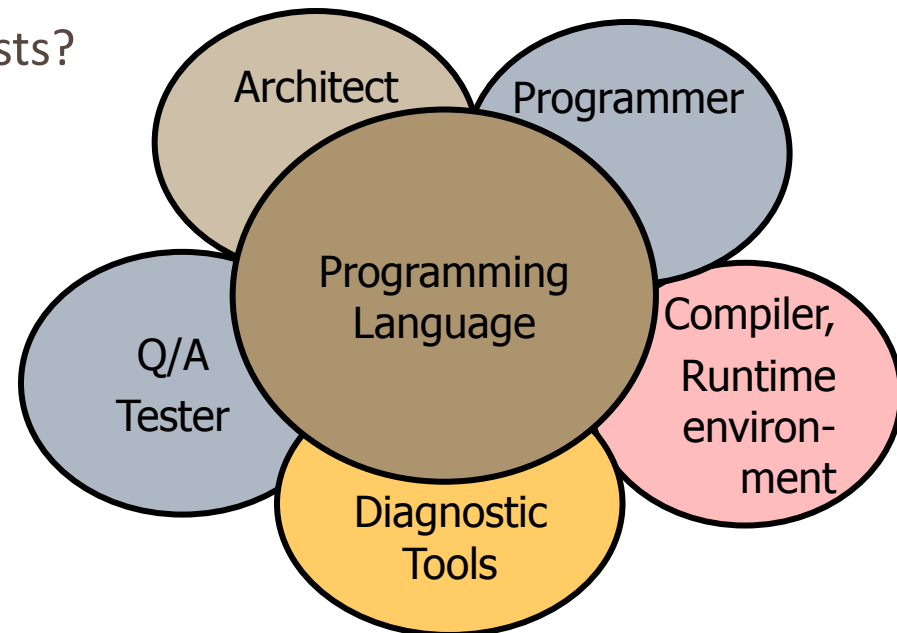
- General discussion of types
 - What is a type?
 - Compile-time versus run-time checking
 - Conservative program analysis
- Type inference
 - Discuss algorithm and examples
 - Illustrative example of static analysis algorithm
- Polymorphism
 - Uniform versus non-uniform implementations

“The world’s most lightweight and widely used formal method”

TYPES, GENERALLY SPEAKING

Language Goals and Trade-offs

- Thoughts to keep in mind
 - What features are convenient for programmer?
 - What other features do they prevent?
 - What are design tradeoffs?
 - Easy to write but harder to read?
 - Easy to write but poorer error messages?
 - What are the implementation costs?



What is a type?

- A type is a collection of computable values that share some structural property.

Examples

```
Integer  
String  
Int → Bool  
(Int → Int) → Bool
```

Non-examples

```
{3, True, \x->x}  
Even integers  
{f:Int → Int | x>3 =>  
  f(x) > x * (x+1)}
```

(Most of the time!)

Distinction between sets of values that are types and sets that are not types is *language dependent*.

Advantages of Types

- Identify and prevent errors
 - Compile-time or run-time checking can prevent meaningless computations such as `3 + true` – “Bill”
- Program organization and documentation
 - Separate types for separate concepts
 - Represent concepts from problem domain
 - Document intended use of declared identifiers
 - Types can be checked, unlike program comments
- Support optimization
 - Example: short integers require fewer bits
 - Access components of structures by known offset

What is a type error?

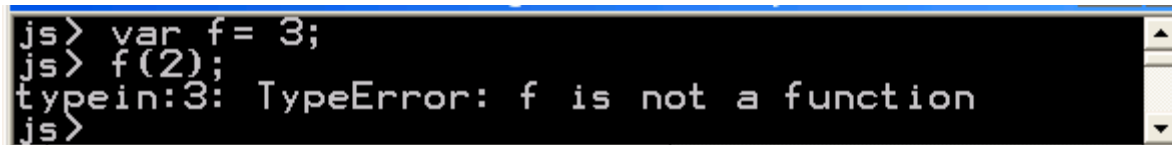
- Whatever the compiler/interpreter says it is?
- Something to do with bad bit sequences?
 - Floating point representation has specific form
 - An integer may not be a valid float
- Something about programmer intent and use?
 - A type error occurs when a value is used in a way that is inconsistent with its definition
 - Example: declare as character, use as integer

Type errors are language dependent

- Array out of bounds access
 - C/C++: runtime errors.
 - Haskell/Java: dynamic type errors.
- Null pointer dereference
 - C/C++: run-time errors
 - Haskell/ML: pointers are hidden inside datatypes
 - Null pointer dereferences would be incorrect use of these datatypes, therefore static type errors

Compile-time vs Run-time Checking

- JavaScript and Lisp use run-time type checking
 - $f(x)$ Make sure f is a function before calling f



```
js> var f= 3;  
js> f(2);  
typein:3: TypeError: f is not a function  
js>
```

- Haskell and Java use compile-time type checking
 - $f(x)$ Must have $f :: A \rightarrow B$ and $x :: A$
- Basic tradeoff
 - Both kinds of checking prevent type errors
 - Run-time checking slows down execution
 - Compile-time checking restricts program flexibility
 - JavaScript array: elements can have different types
 - Haskell list: all elements must have same type

Expressiveness

- In JavaScript, we can write a function like

```
function f(x) { return x < 10 ? x : x(); }
```

Some uses will produce type error, some will not.

- Static typing always conservative

```
if    (complicated-boolean-expression)
      then  f(5);
      else  f(15);
```

Relative Type-Safety of Languages

- **Not safe:** BCPL family, including C and C++
 - Casts, pointer arithmetic
- **Almost safe:** Algol family, Pascal, Ada.
 - Dangling pointers.
 - Allocate a pointer p to an integer, deallocate the memory referenced by p, then later use the value pointed to by p.
 - No language with explicit deallocation of memory is fully type-safe.
- **Safe:** Lisp, Smalltalk, ML, Haskell, Java, JavaScript
 - Dynamically typed: Lisp, Smalltalk, JavaScript
 - Statically typed: ML, Haskell, Java

If code accesses data, it is handled with the type associated with the creation and previous manipulation of that data

“What do you mean, I don’t have to write all my types down? Sweet!”

TYPE INFERENCE

Type Checking vs Type Inference

- Standard type checking:

```
int f(int x) { return x+1; };  
int g(int y) { return f(y+1)*2; };
```

- Examine body of each function
- Use declared types to check agreement

- Type inference:

```
int f(int x) { return x+1; };  
int g(int y) { return f(y+1)*2; };
```

- Examine code without type information. Infer the most general types that could have been declared.

ML and Haskell are *designed* to make type inference feasible.

Why study type inference?

- Types and type checking
 - Improved steadily since Algol 60
 - Eliminated sources of unsoundness.
 - Become substantially more expressive.
 - Important for modularity, reliability and compilation
- Type inference
 - Reduces syntactic overhead of expressive types.
 - 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,...

uHaskell

- Subset of Haskell to explain type inference.
 - Haskell and ML both have overloading
 - Will not cover type inference with overloading

```
<decl> ::= [<name> <pat> = <exp>]  
<pat>  ::= Id | (<pat>, <pat>) | <pat> :  
<pat> | []  
<exp>  ::= Int | Bool | [] | Id | (<exp>)  
          | <exp> <op> <exp>  
          | <exp> <exp> | (<exp>, <exp>)  
          | if <exp> then <exp> else <exp>
```


Type Inference: Basic Idea

- Example

```
f x = 2 + x  
> f :: Int -> Int
```

- What is the type of f?

+ has type: $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

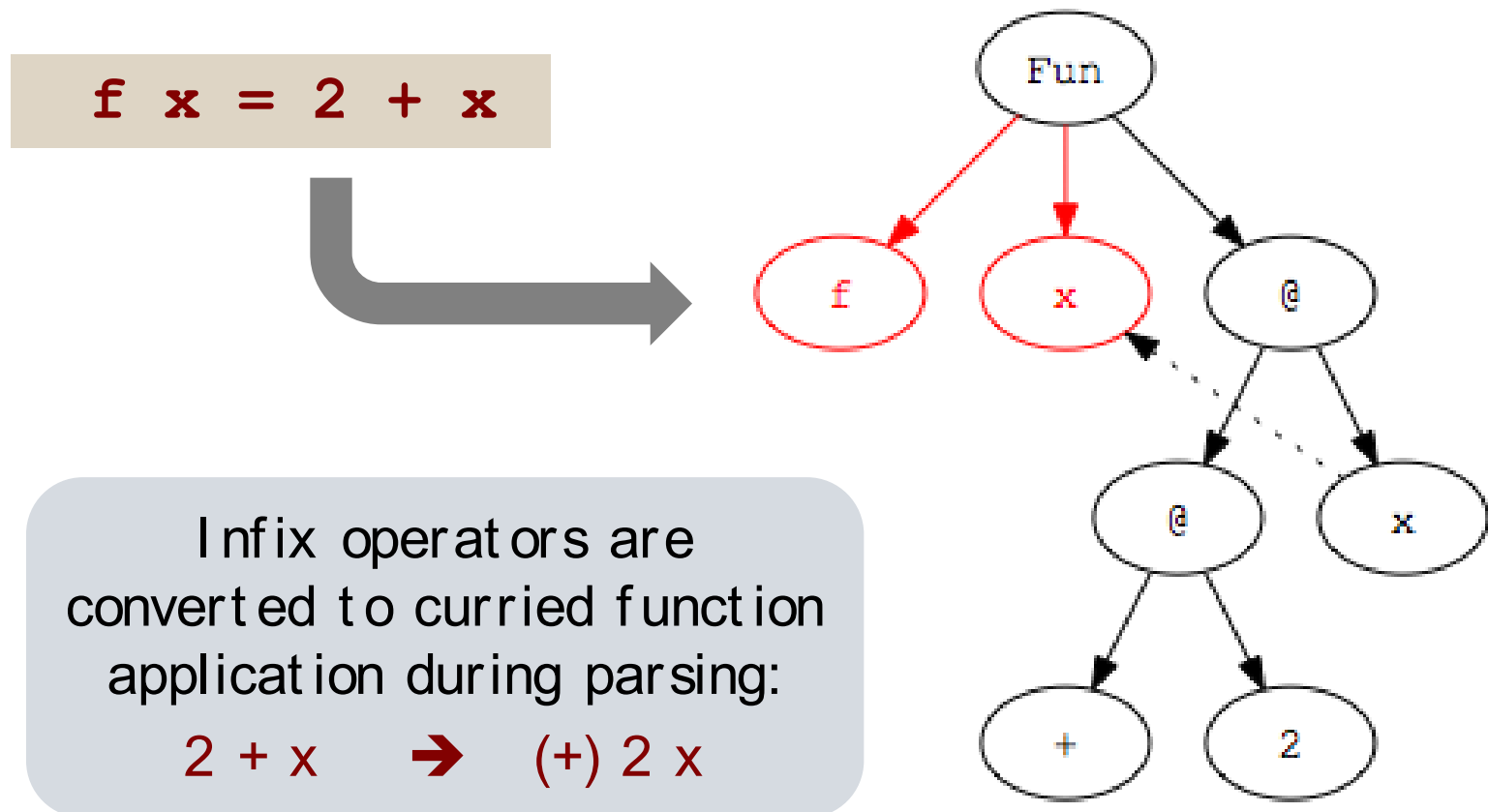
2 has type: Int

Since we are applying + to x we need $x :: \text{Int}$

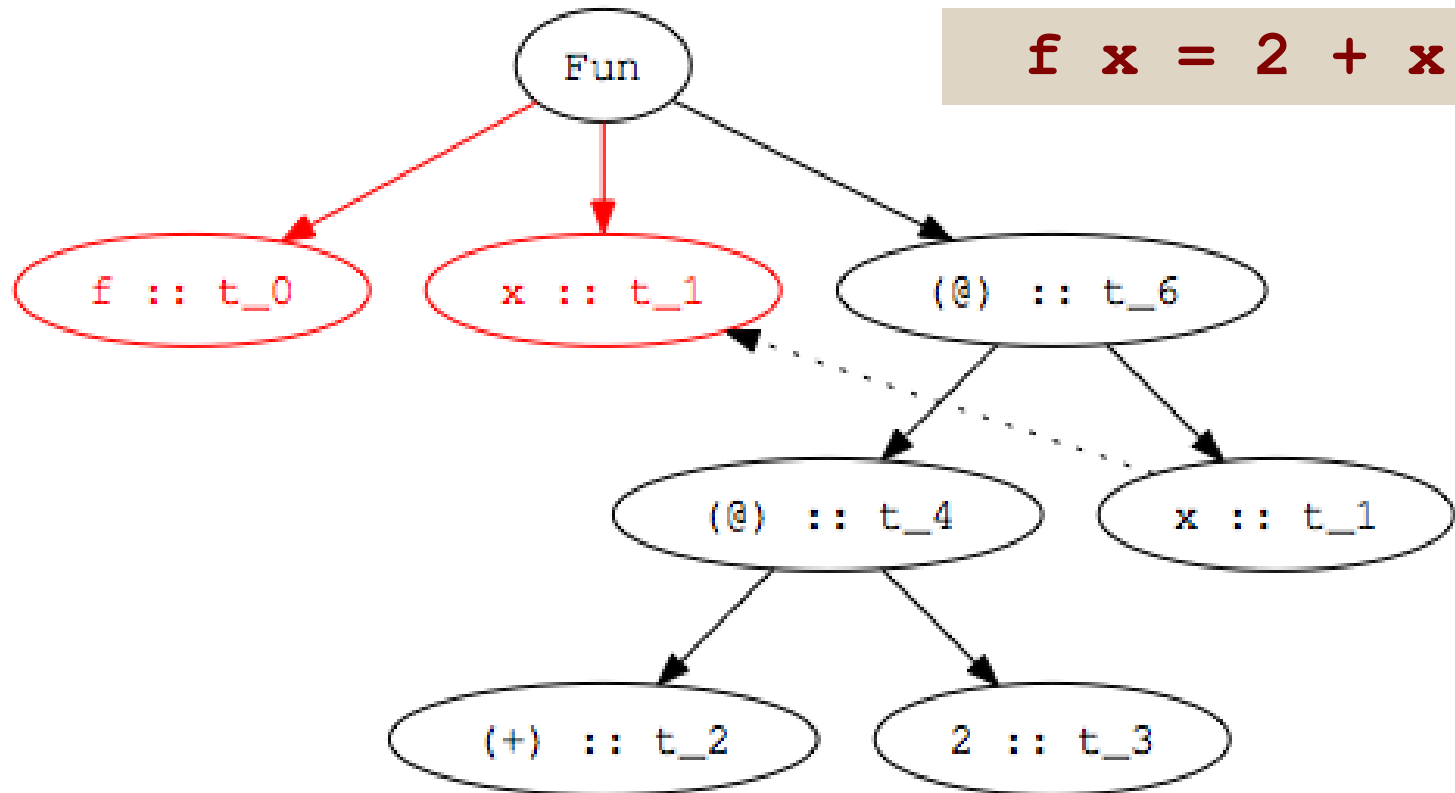
Therefore $f\ x = 2 + x$ has type $\text{Int} \rightarrow \text{Int}$

Step 1: Parse Program

- Parse program text to construct parse tree.



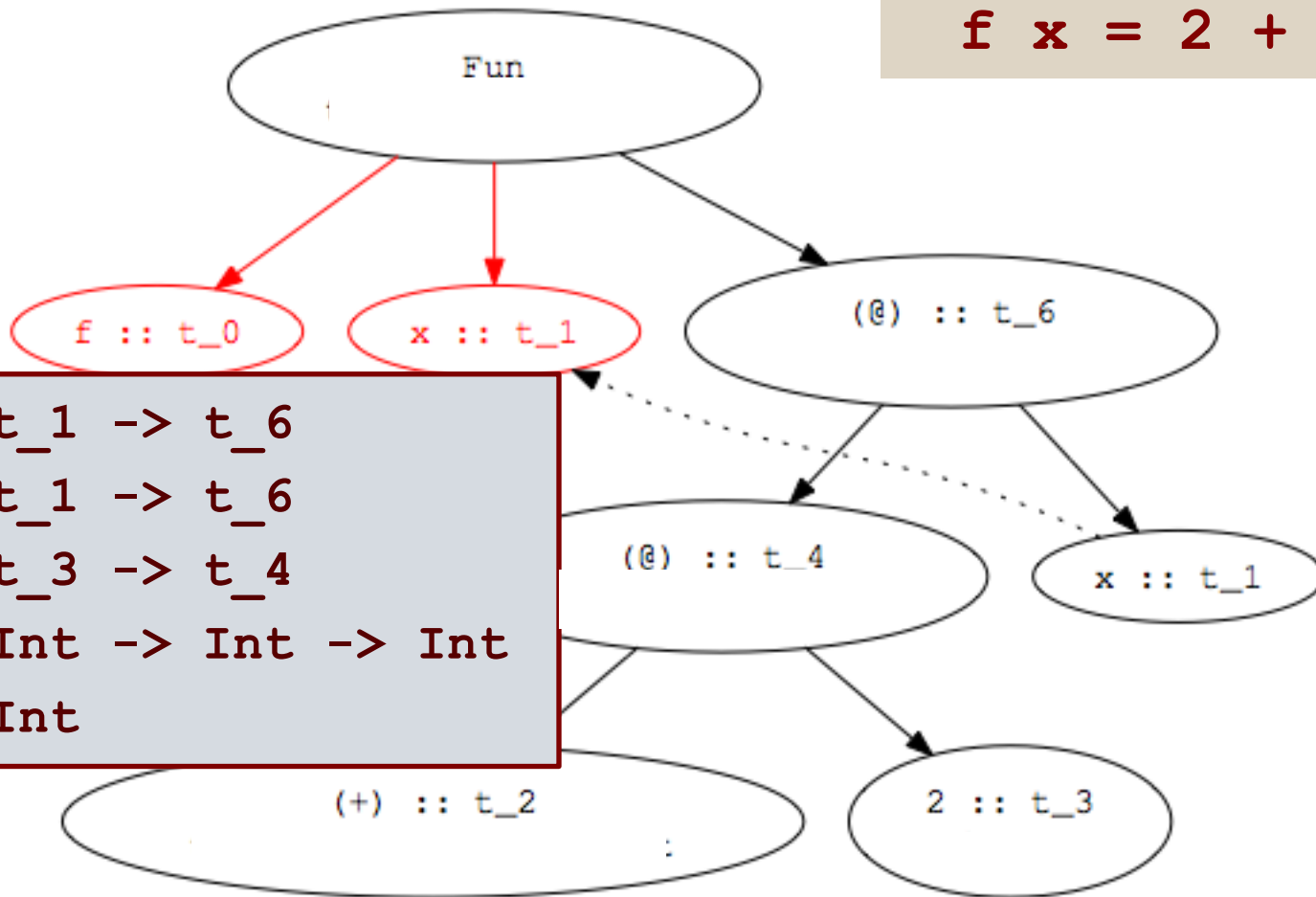
Step 2: Assign type variables to nodes



Variables are given same type as binding occurrence.

Step 3: Add Constraints

$f \ x = 2 + x$



$t_0 = t_1 \rightarrow t_6$
 $t_4 = t_1 \rightarrow t_6$
 $t_2 = t_3 \rightarrow t_4$
 $t_2 = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
 $t_3 = \text{Int}$

Step 4: Solve Constraints

```
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
```

$t_3 \rightarrow t_4 = \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$

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

$t_3 = \text{Int}$
 $t_4 = \text{Int} \rightarrow \text{Int}$

$t_1 \rightarrow t_6 = \text{Int} \rightarrow \text{Int}$

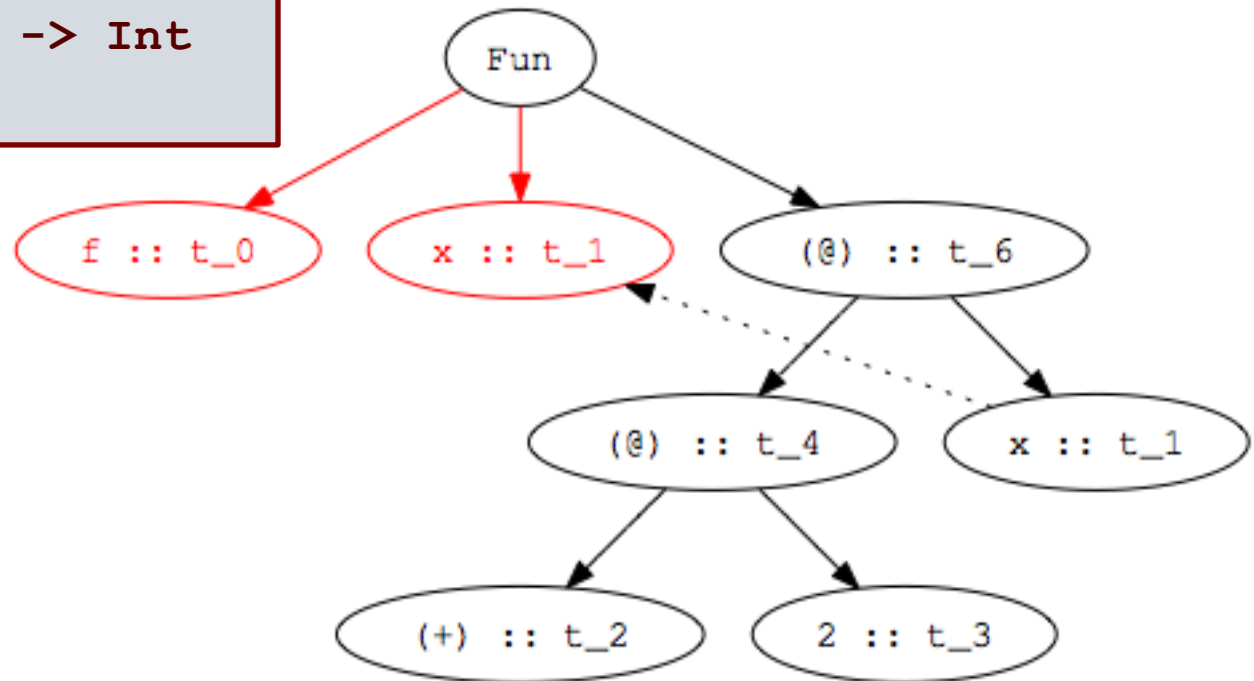
```
t_0 = Int -> Int  
t_1 = Int  
t_6 = Int  
t_4 = Int -> Int  
t_2 = Int -> Int -> Int  
t_3 = Int
```

$t_1 = \text{Int}$
 $t_6 = \text{Int}$

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
```

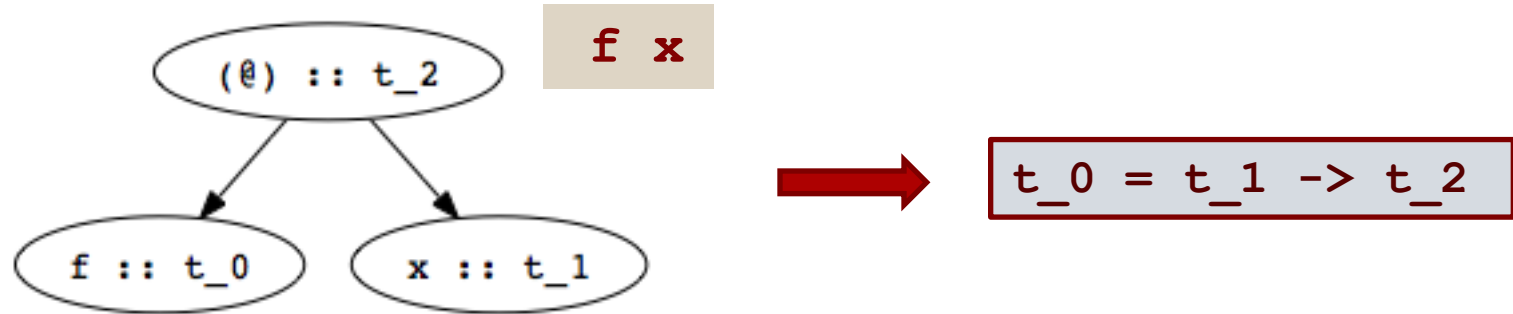
```
f x = 2 + x  
> f :: Int -> Int
```



Type Inference Algorithm

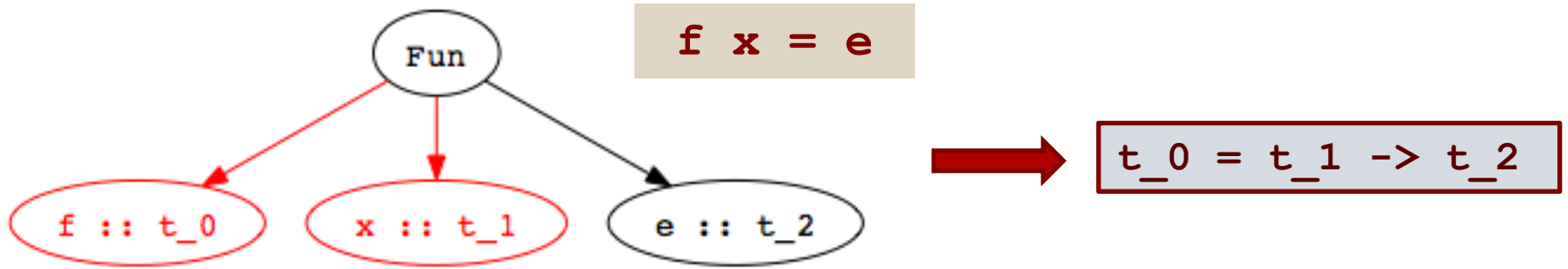
1. Parse program to build parse tree
2. Assign type variables to nodes in tree
3. Generate constraints:
 - From environment: constants (**2**), built-in operators (**+**), known functions (**tail**).
 - From form of parse tree: e.g., application and abstraction nodes.
4. Solve constraints using *unification*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

Constraints from Application Nodes



- Function application (apply f to x)
 - Type of f (t_0 in figure) must be domain \rightarrow 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: $t_0 = t_1 \rightarrow t_2$

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

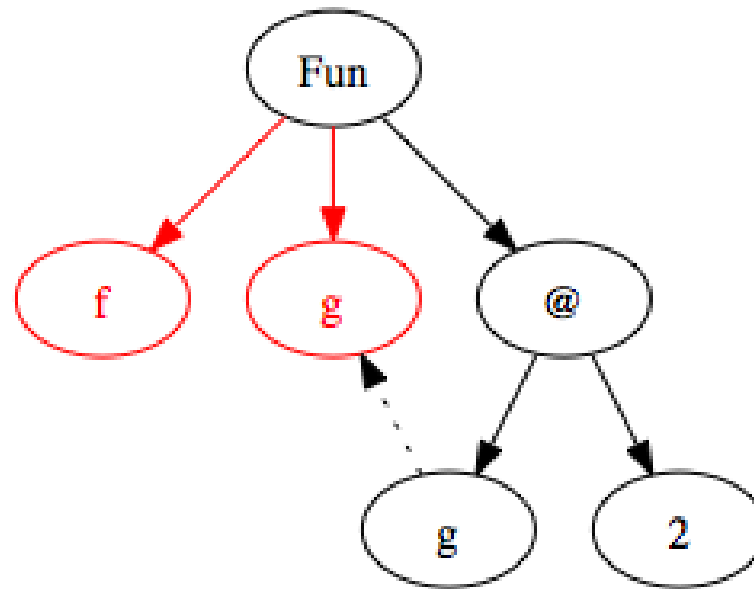
- Example:

```
f g = g 2
```

```
> f :: (Int -> t_4) -> t_4
```

- Step 1:

Build Parse Tree



Inferring Polymorphic Types

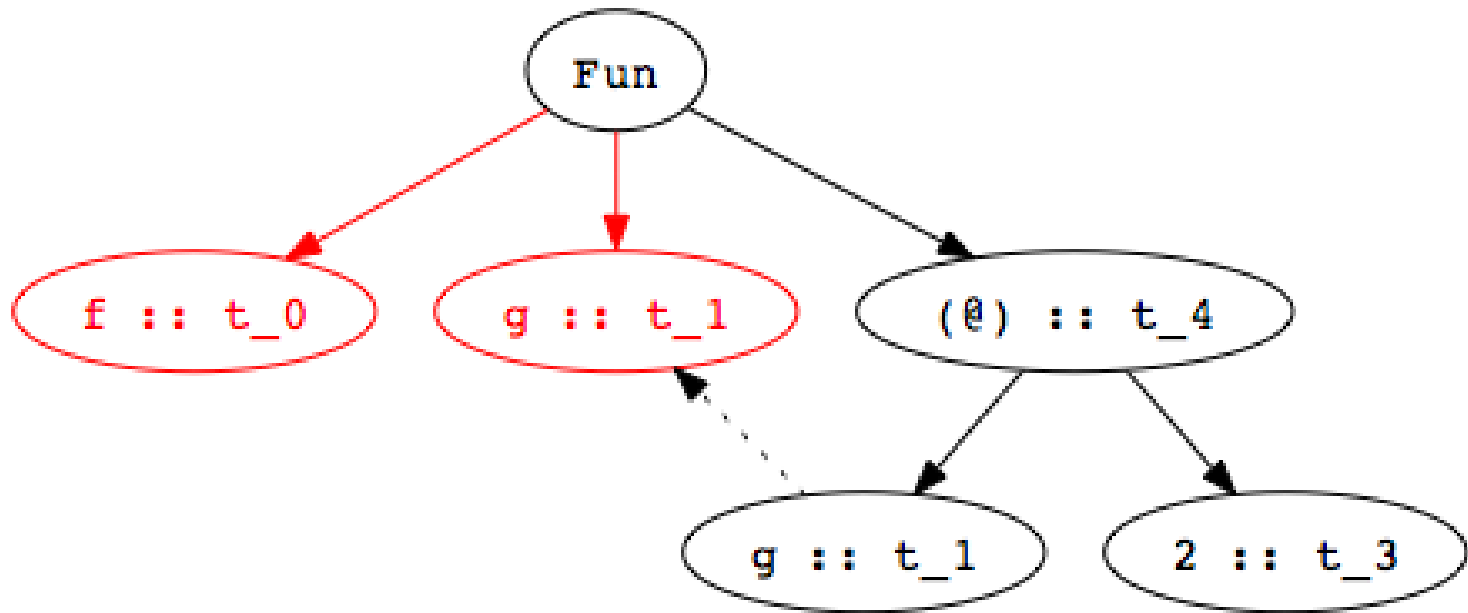
- Example:

```
f g = g 2
```

```
> f :: (Int -> t_4) -> t_4
```

- Step 2:

Assign type variables



Inferring Polymorphic Types

- Example:

```
f g = g 2
```

```
> f :: (Int -> t_4) -> t_4
```

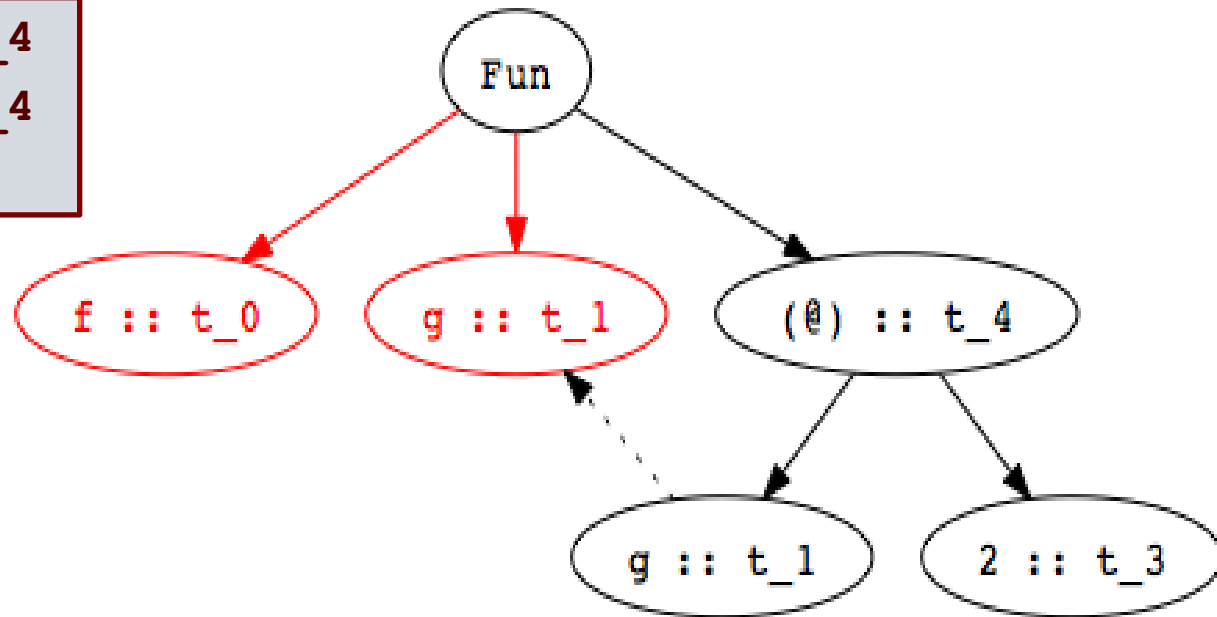
- Step 3:

Generate constraints

```
t_0 = t_1 -> t_4
```

```
t_1 = t_3 -> t_4
```

```
t_3 = Int
```



Inferring Polymorphic Types

- Example:

```
f g = g 2
```

```
> f :: (Int -> t_4) -> t_4
```

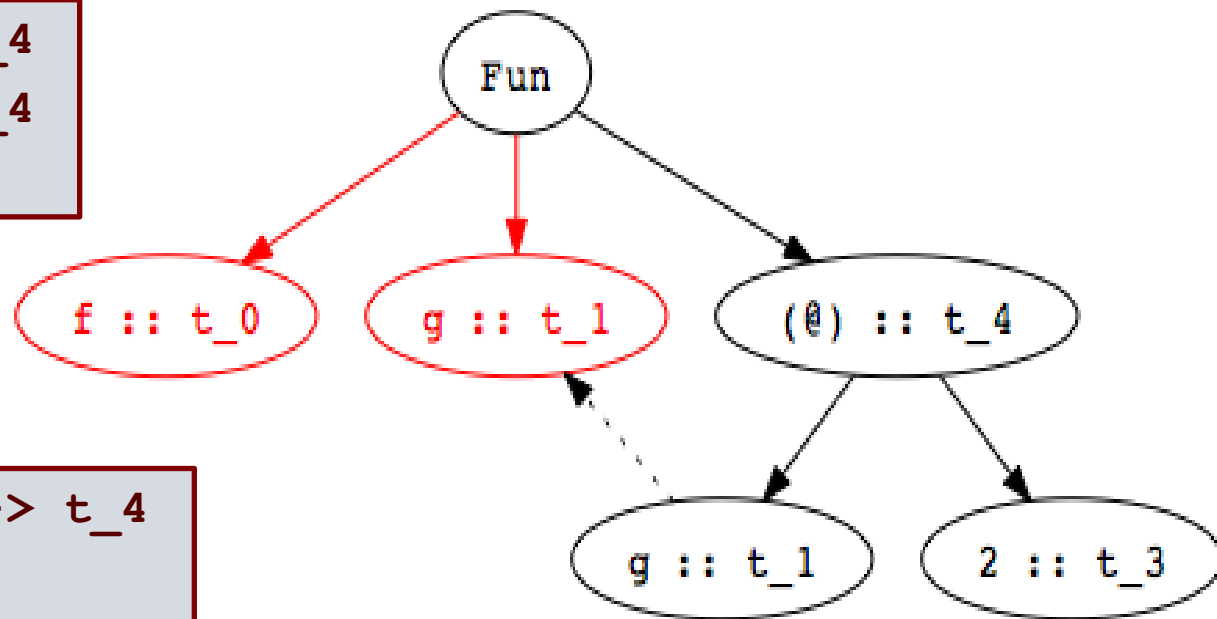
- Step 4:

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:

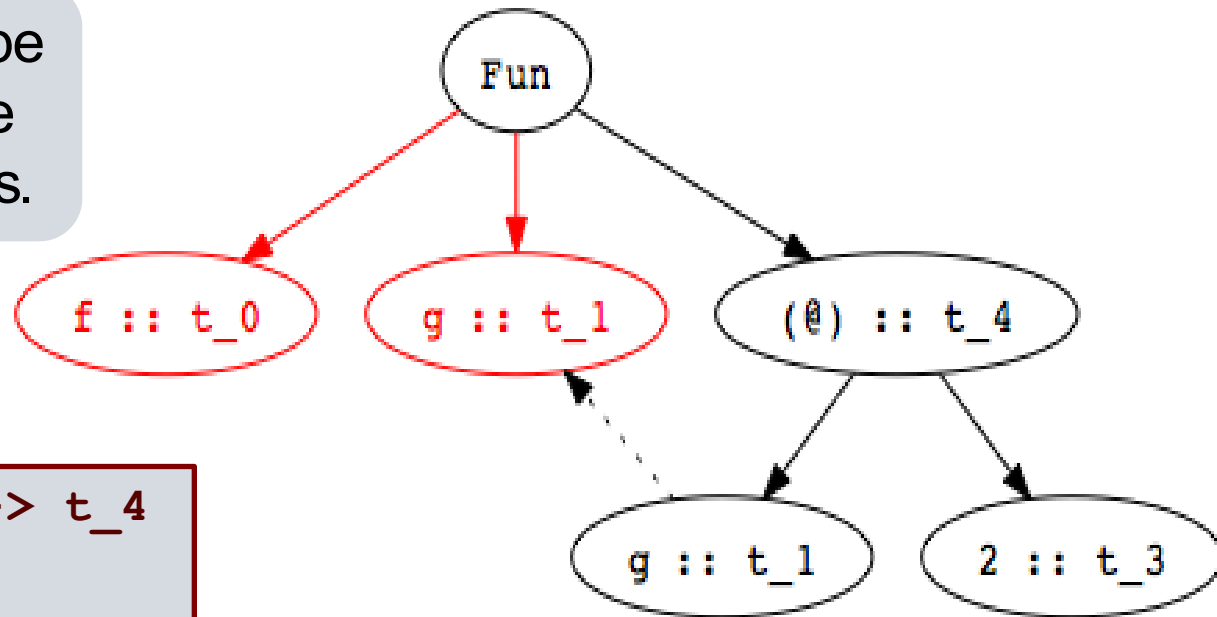
```
f g = g 2
```

```
> f :: (Int -> t_4) -> t_4
```

- 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
```

Using Polymorphic Functions

- Function:

```
f g = g 2  
> f :: (Int -> t_4) -> t_4
```

- Possible applications:

```
add x = 2 + x  
> add :: Int -> Int
```

```
f add  
> 4 :: Int
```

```
isEven x = mod (x, 2) == 0  
> isEven :: Int -> Bool
```

```
f isEven  
> True :: Int
```

Recognizing Type Errors

- Function:

```
f g = g 2  
> f :: (Int -> t_4) -> t_4
```

- Incorrect use

```
not x = if x then True else False  
> not :: Bool -> Bool  
f not  
> Error: operator and operand don't agree  
operator domain: Int -> a  
operand:          Bool -> Bool
```

- Type error:
cannot unify $\text{Bool} \rightarrow \text{Bool}$ and $\text{Int} \rightarrow t$

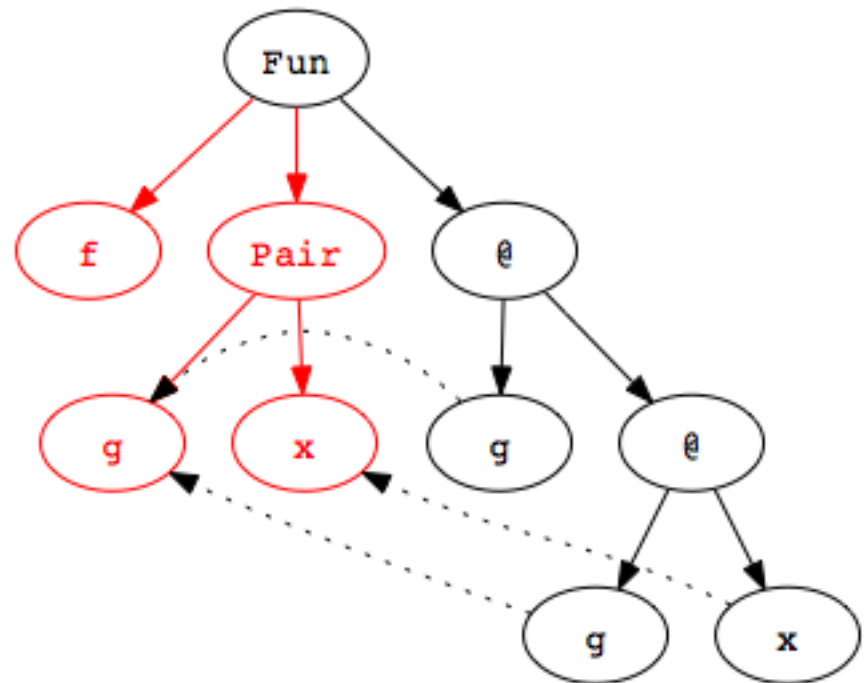
Another Example

- Example:

$f(g, x) = g(g\ x)$

$> f :: (t_8 \rightarrow t_8, t_8) \rightarrow t_8$

- Step 1:
Build Parse Tree



Another Example

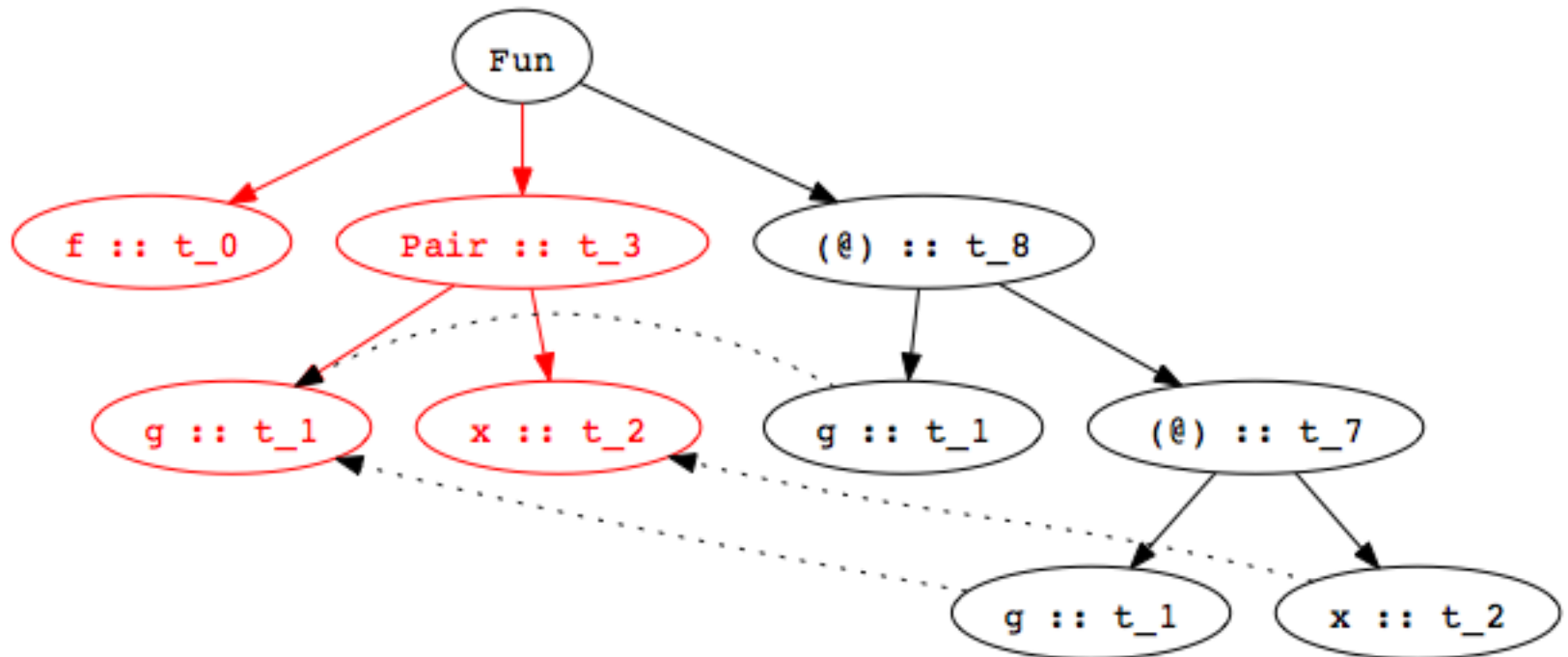
- Example:

$f \ (g, x) = g \ (g \ x)$

$> f :: (t_8 \rightarrow t_8, t_8) \rightarrow t_8$

- Step 2:

Assign type variables



Another Example

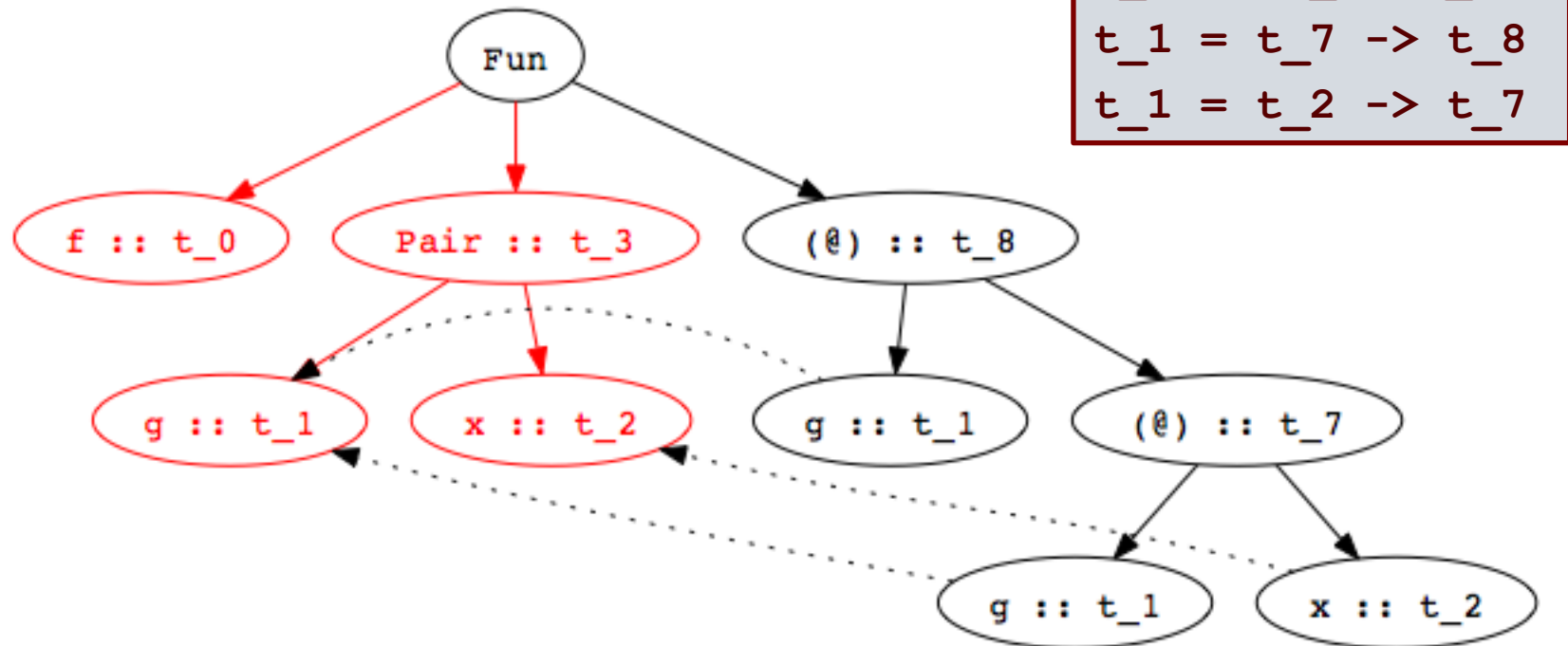
- Example:

$f \ (g, x) = g \ (g \ x)$

$> f :: (t_8 \rightarrow t_8, t_8) \rightarrow t_8$

- Step 3:

Generate constraints



Another Example

- Example:

$f \ (g, x) = g \ (g \ x)$

$> f :: (t_8 \rightarrow t_8, t_8) \rightarrow t_8$

- Step 4:

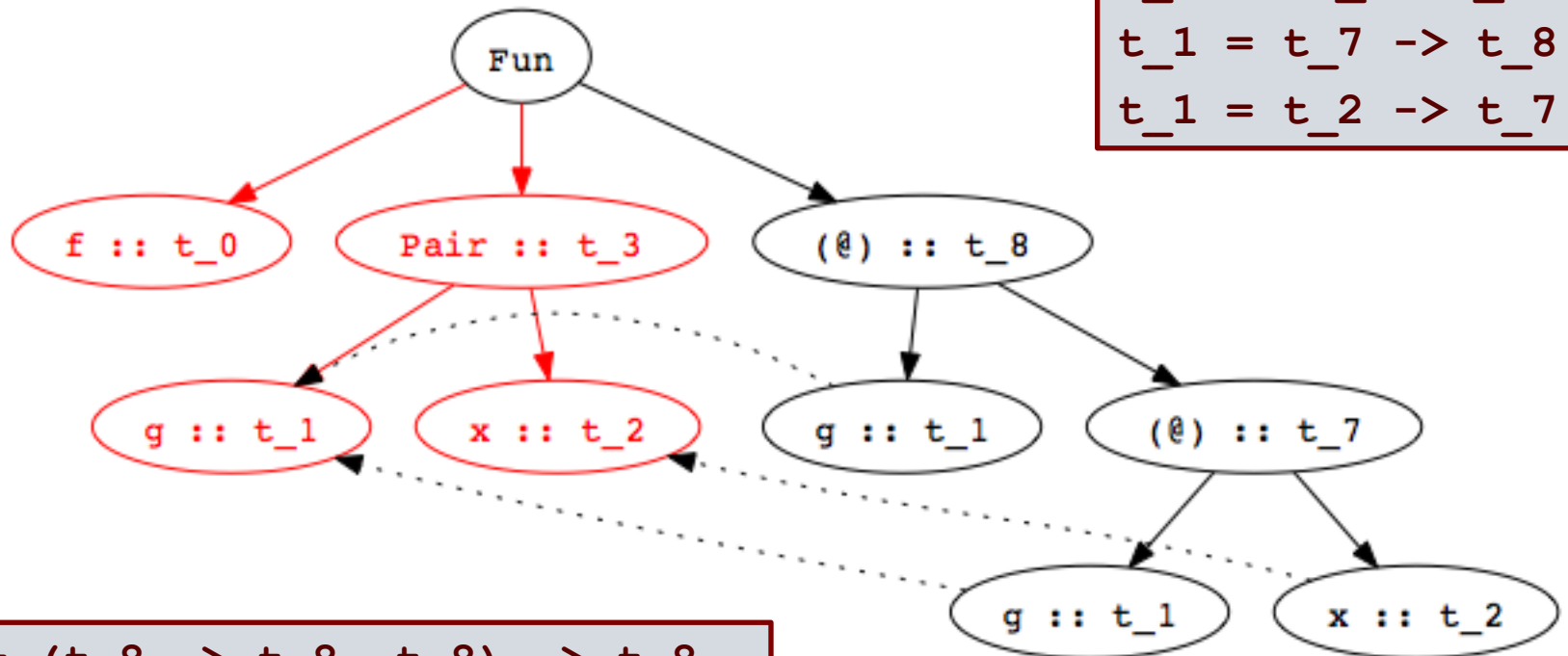
Solve constraints

$t_0 = t_3 \rightarrow t_8$

$t_3 = (t_1, t_2)$

$t_1 = t_7 \rightarrow t_8$

$t_1 = t_2 \rightarrow t_7$



$t_0 = (t_8 \rightarrow t_8, t_8) \rightarrow t_8$

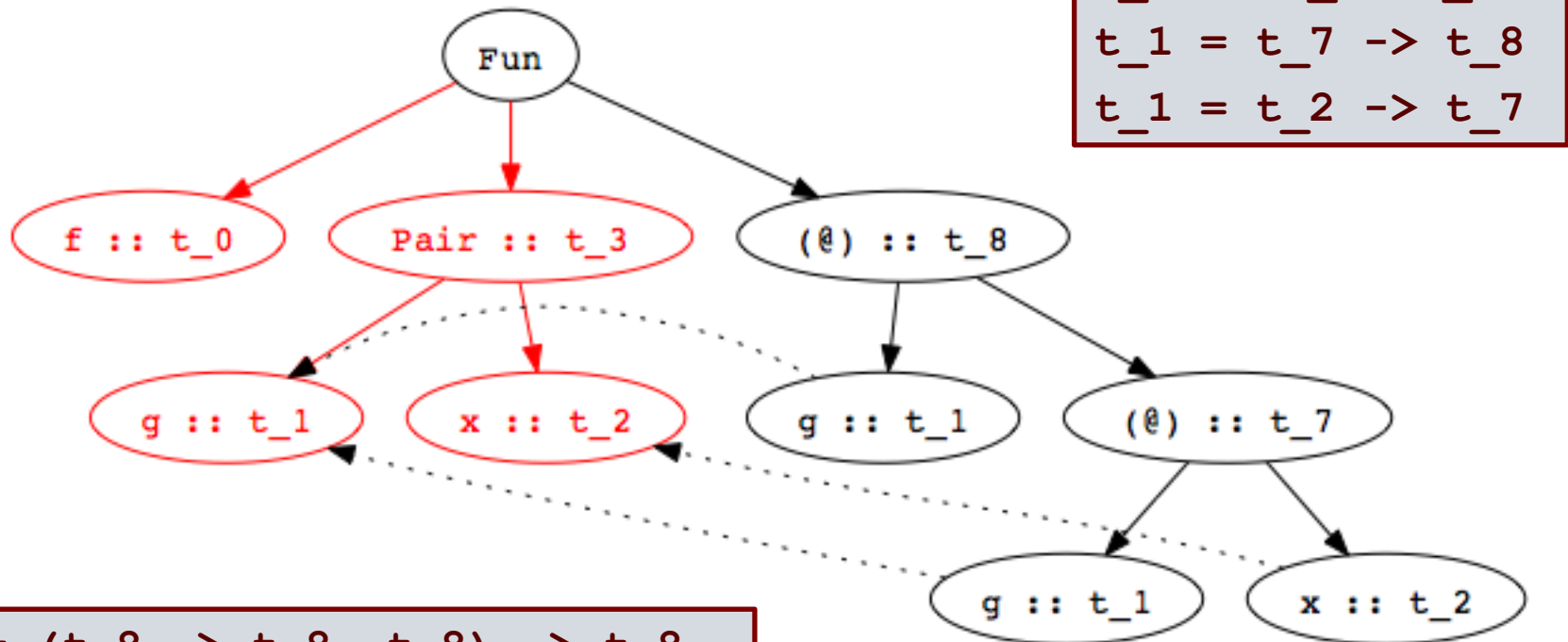
Another Example

- Example:

$$f \ (g, x) = g \ (g \ x)$$
$$> f :: (t_8 \rightarrow t_8, t_8) \rightarrow t_8$$

- Step 5:

Determine type of f



Polymorphic Datatypes

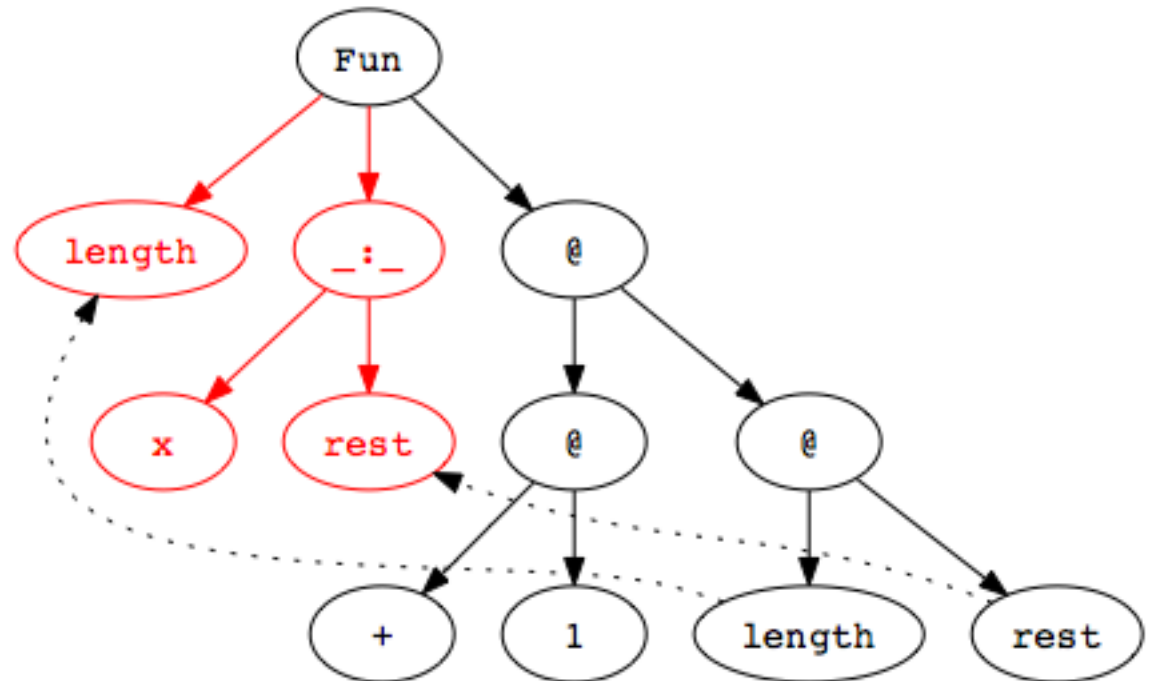
- Functions may have multiple clauses

```
length [] = 0  
length (x:rest) = 1 + (length rest)
```

- Type inference
 - Infer separate type for each clause
 - Combine by adding constraint that all clauses must have the same type
 - Recursive calls: function has same type as its definition

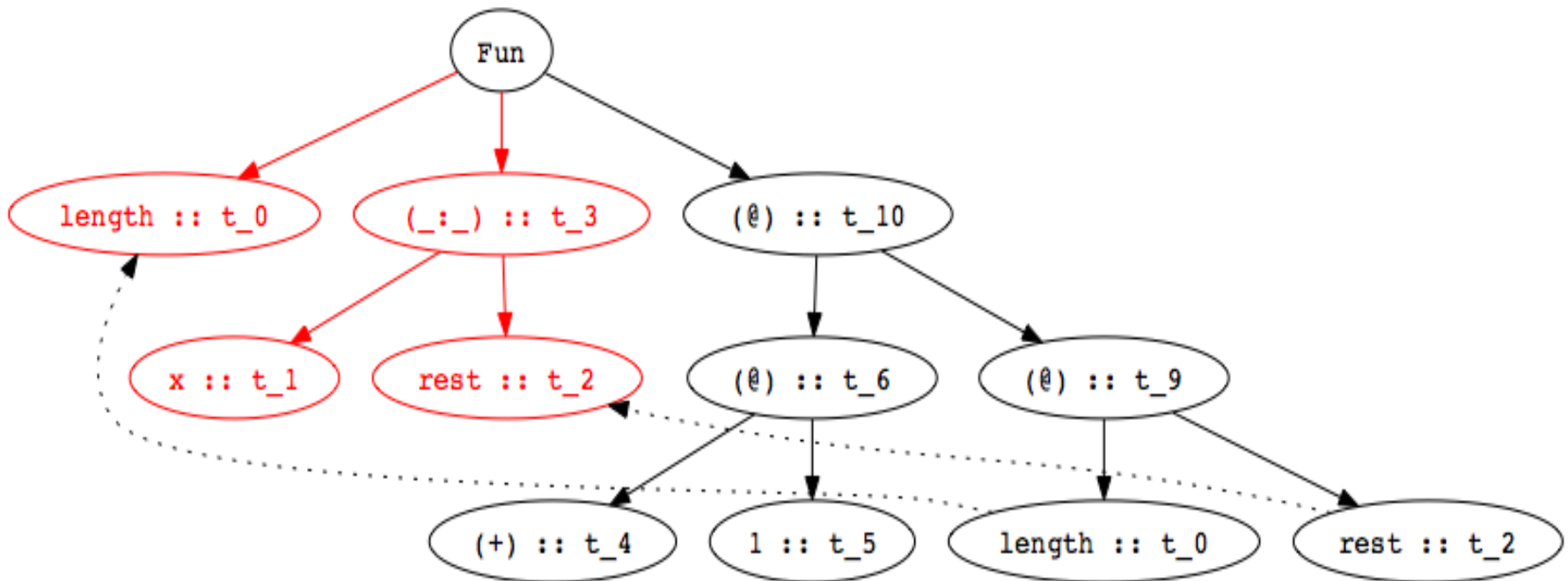
Type Inference with Datatypes

- Example: `length (x:rest) = 1 + (length rest)`
- Step 1: Build Parse Tree



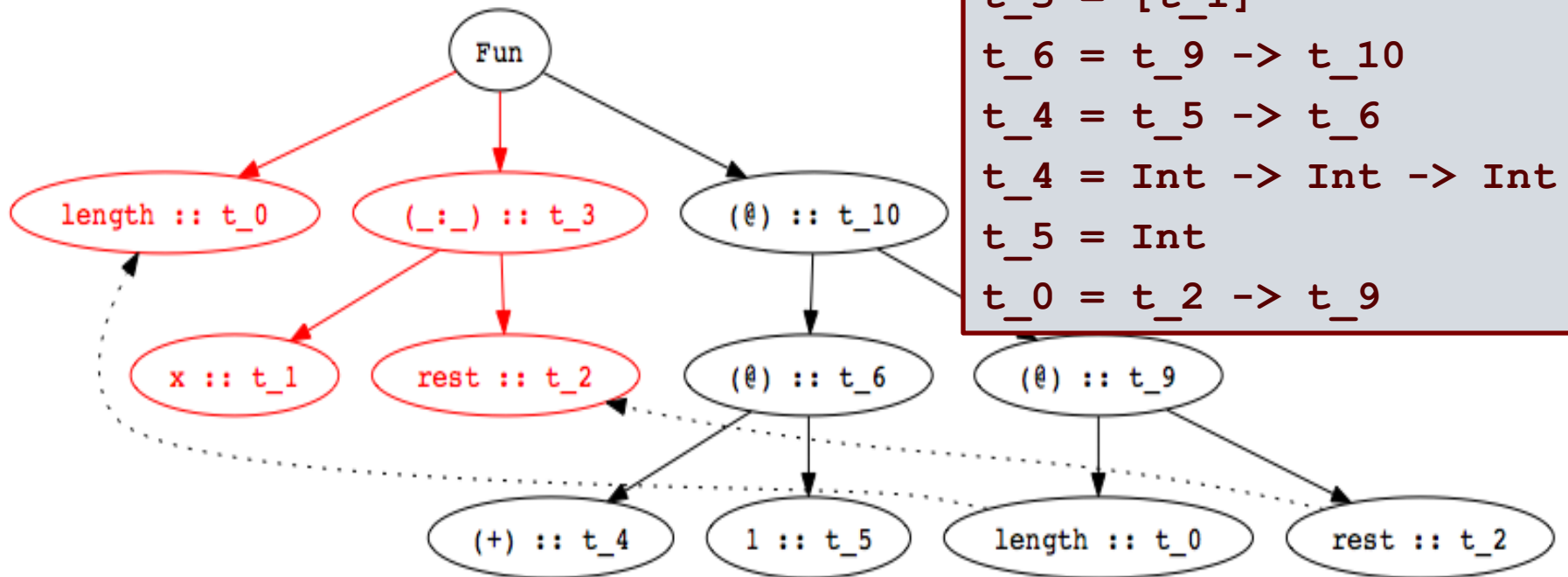
Type Inference with Datatypes

- Example: `length (x:rest) = 1 + (length rest)`
- Step 2: Assign type variables



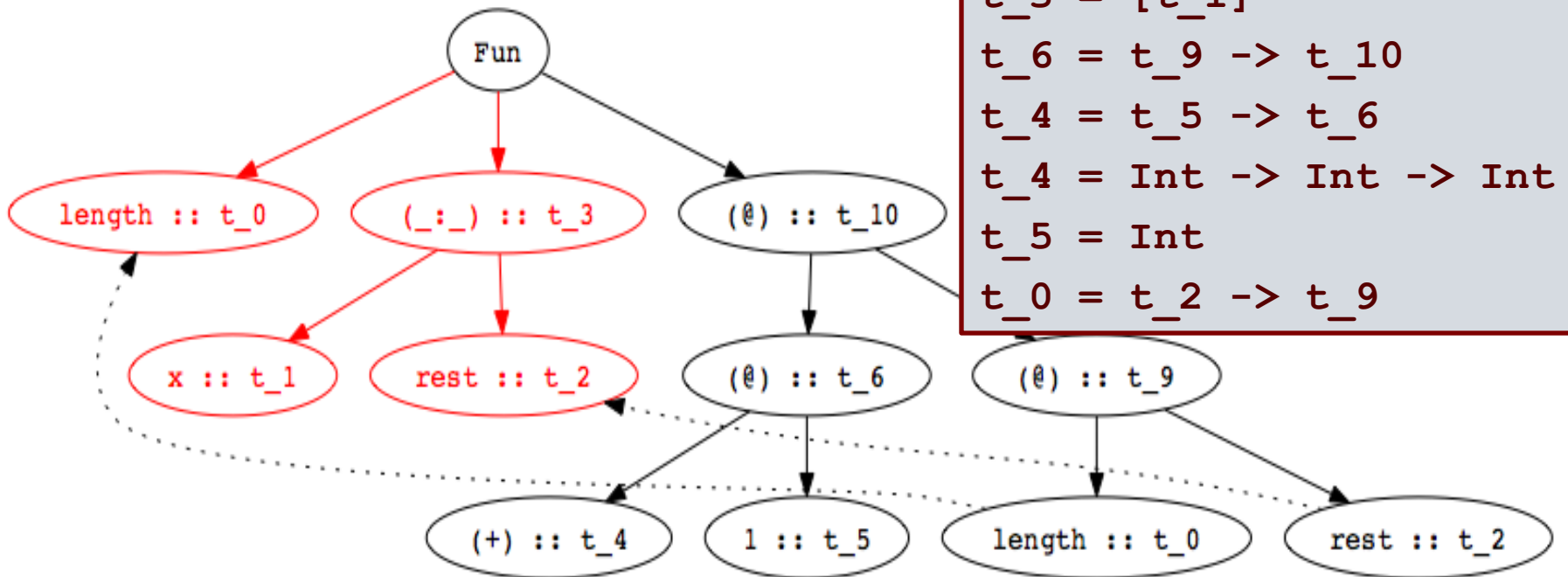
Type Inference with Datatypes

- Example: `length (x:rest) = 1 + (length rest)`
- Step 3: Generate constraints



Type Inference with Datatypes

- Example: `length (x:rest) = 1 + (length rest)`
- Step 3: Solve Constraints



`t0 = [t1] -> Int`

Multiple Clauses

- Function with multiple clauses

```
append ([], r) = r
append (x:xs, r) = x : append (xs, r)
```

- Infer type of each clause

- First clause:

```
> append :: ([t_1], t_2) -> t_2
```

- Second clause:

```
> append :: ([t_3], t_4) -> [t_3]
```

- Combine by equating types of two clauses

```
> append :: ([t_1], [t_1]) -> [t_1]
```

Most General Type

- Type inference produces the *most general type*

```
map (f, [] ) = []  
map (f, x:xs) = f x : map (f, xs)  
> map :: (t_1 -> t_2, [t_1]) -> [t_2]
```

- Functions may have many less general types

```
> map :: (t_1 -> Int, [t_1]) -> [Int]  
> map :: (Bool -> t_2, [Bool]) -> [t_2]  
> map :: (Char -> Int, [Char]) -> [Int]
```

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

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

Information from Type Inference

- Consider this function...

```
reverse [] = []  
reverse (x:xs) = reverse xs
```

... and its most general type:

```
> reverse :: [t_1] -> [t_2]
```

- What does this type mean?

Reversing a list should not change its type, so there must be an error in the definition of reverse!

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.
 - Complications regarding assignment took years to work out.
- Idea can be applied to other program properties
 - Discover properties of program using same kind of analysis

Haskell Type Inference

- Haskell uses type classes
 - supports user-defined overloading, so the inference algorithm is more complicated.
- ML restricts the language
 - to ensure that no annotations are required
- Haskell provides additional features
 - like polymorphic recursion for which types cannot be inferred and so the user must provide annotations

Parametric Polymorphism: Haskell vs C++

- Haskell polymorphic function
 - Declarations (generally) require no type information
 - Type inference uses type variables to type expressions
 - Type inference substitutes for type variables as needed to instantiate polymorphic code
- C++ function template
 - Programmer must declare the argument and result types of functions.
 - Programmers must use explicit type parameters to express polymorphism
 - Function application: type checker does instantiation

Example: Swap Two Values

- Haskell

```
swap :: (IORef a, IORef a) -> IO ()
swap (x,y) = do {
    val_x <- readIORef x; val_y <- readIORef y;
    writeIORef y val_x;   writeIORef x val_y;
    return () }
```

- 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

- Haskell
 - **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?
 - Haskell 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.

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