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.

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: Int $\rightarrow$ Int $\rightarrow$ Int

  2 has type: Int

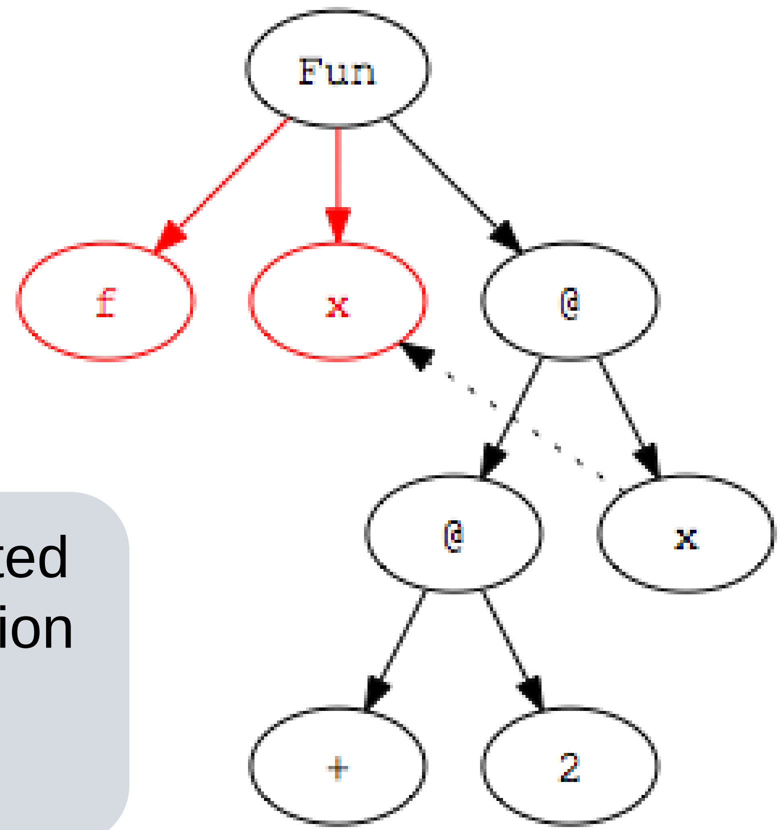
  Since we are applying + to x we need x :: Int

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

Step 1: Parse Program

- Parse program text to construct parse tree.

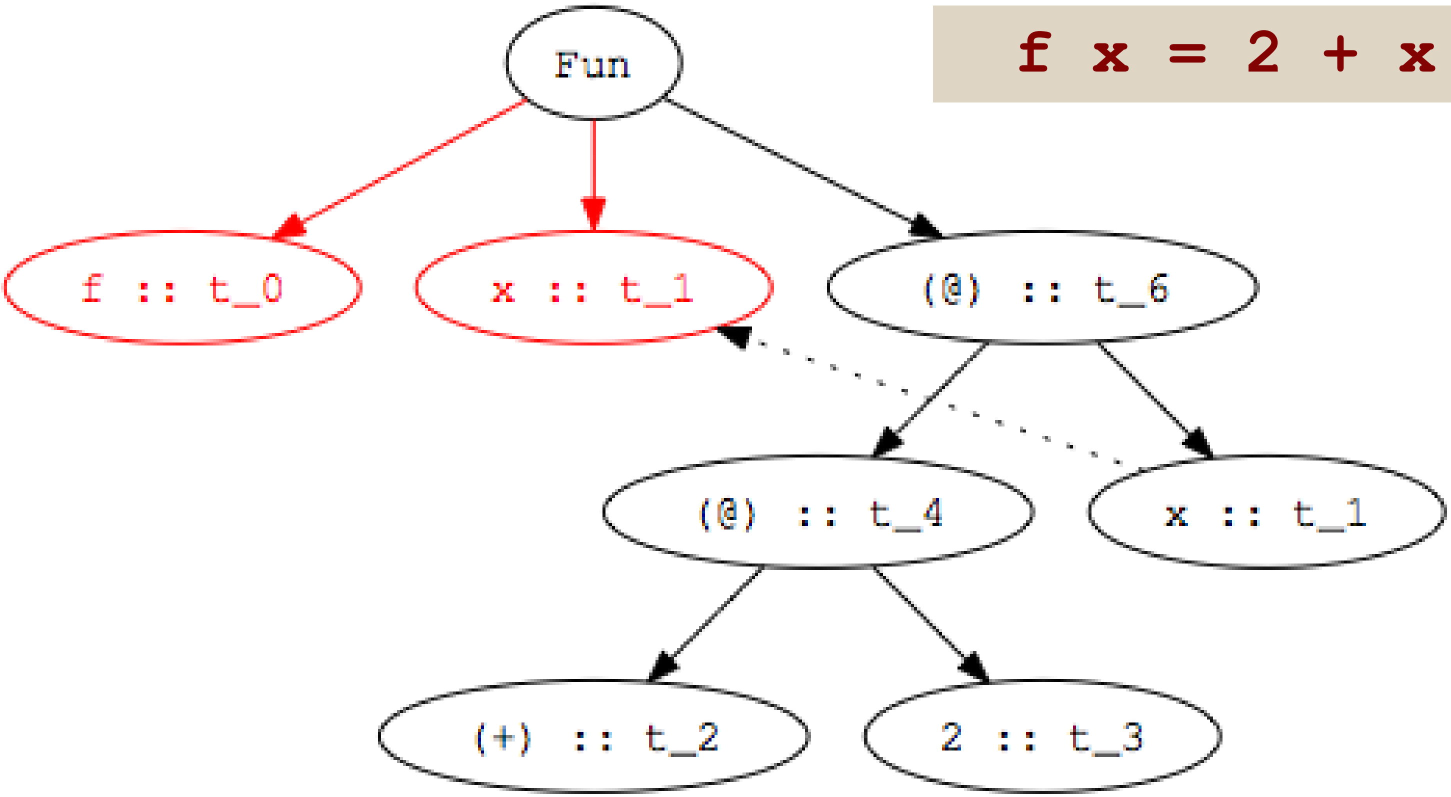
```
f x = 2 + x
```



Infix operators are converted to Curied function application during parsing:

2 + x ➔ (+) 2 x

Step 2: Assign type variables to nodes

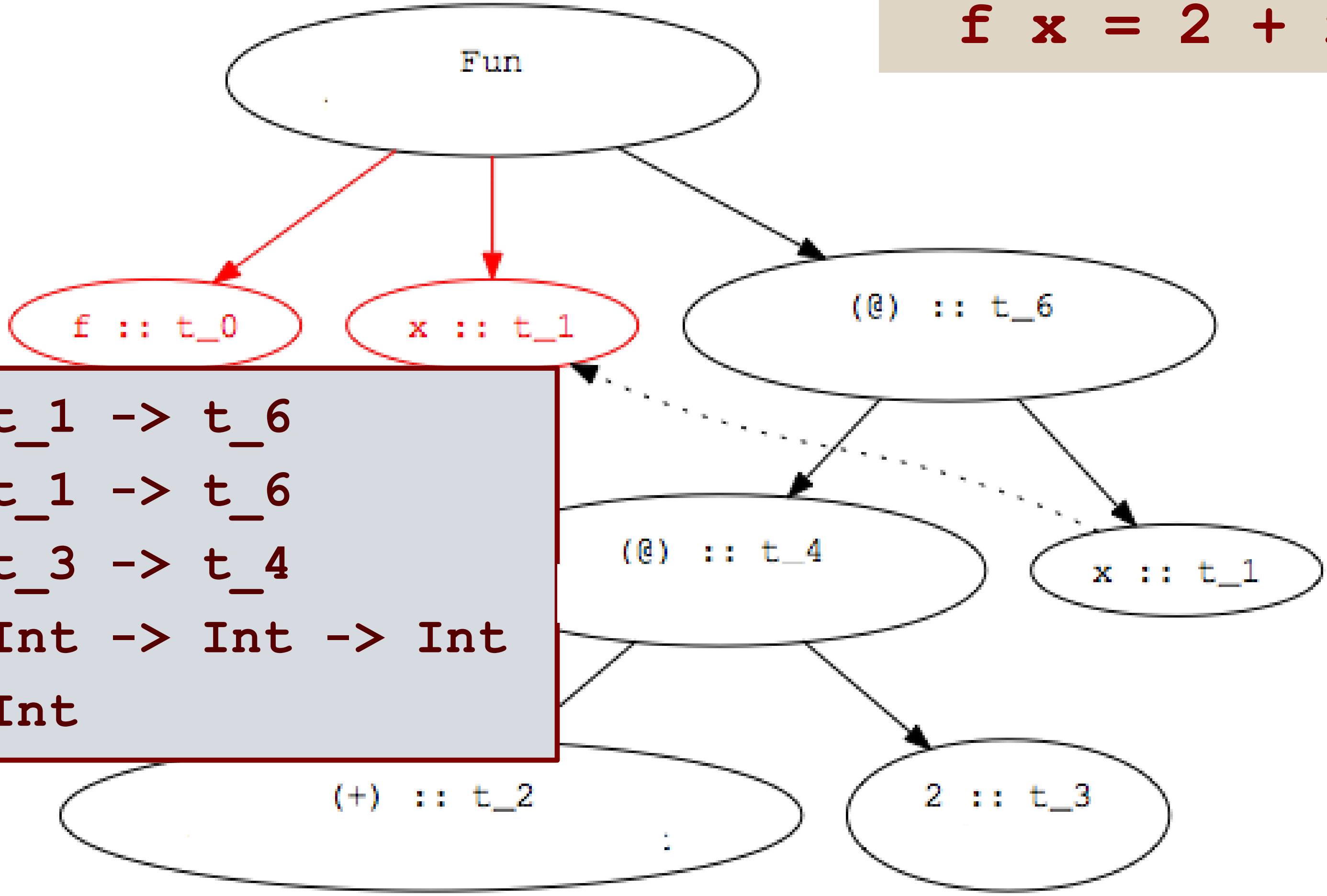


`f x = 2 + x`

Variables are given same type as binding occurrence.

Step 3: Add Constraints

```
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

```
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
```

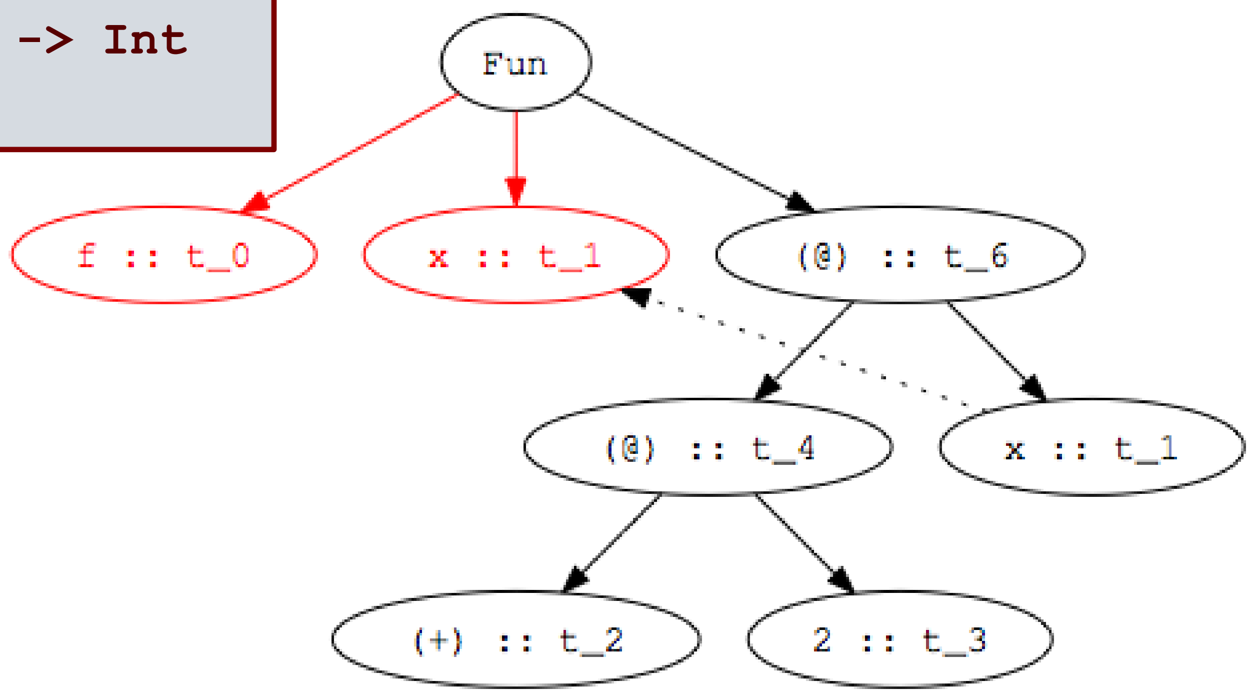
```
t_1 = Int
t_6 = Int
```

```
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
```

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

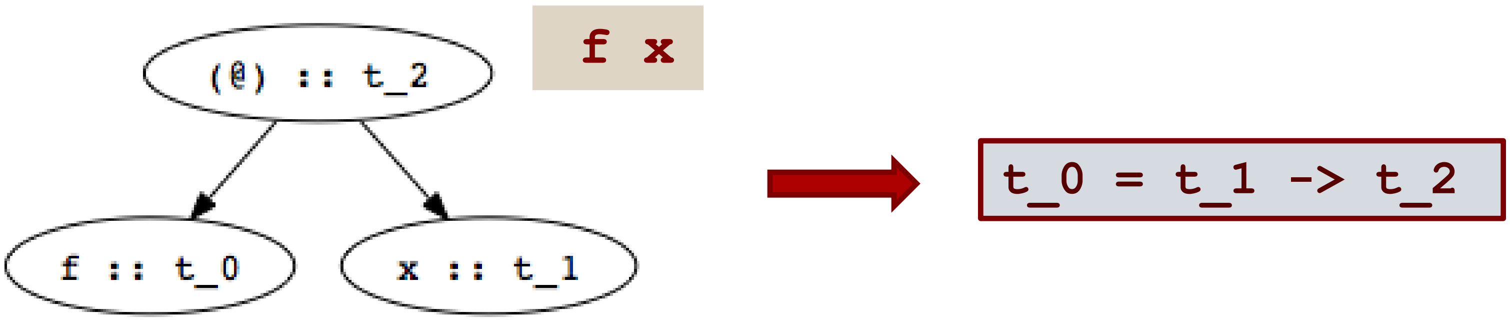


Type Inference Algorithm

- Parse program to build parse tree
- Assign type variables to nodes in tree
- Generate constraints:
  - From environment: constants (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

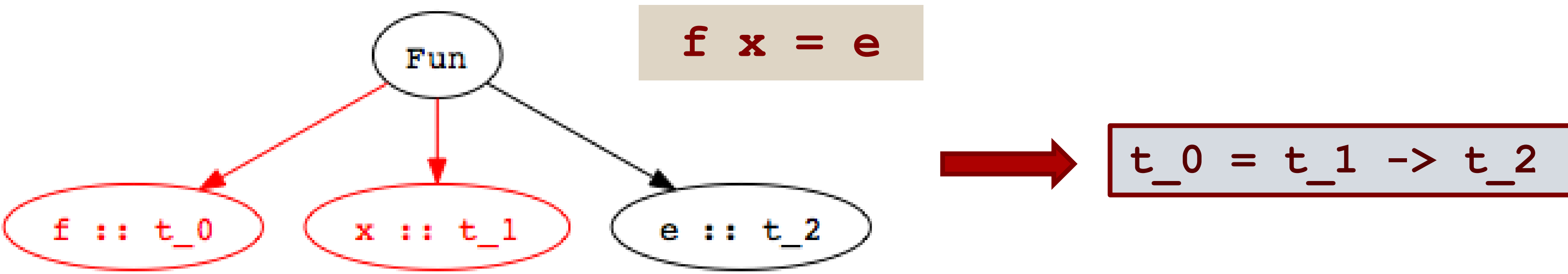
J. A. Robinson, *A Machine-oriented logic based on the resolution principle*,. J. Assoc. Comput. Mach. 12, 23–41 (1965).

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 -> t_2

Constraints from Abstractions



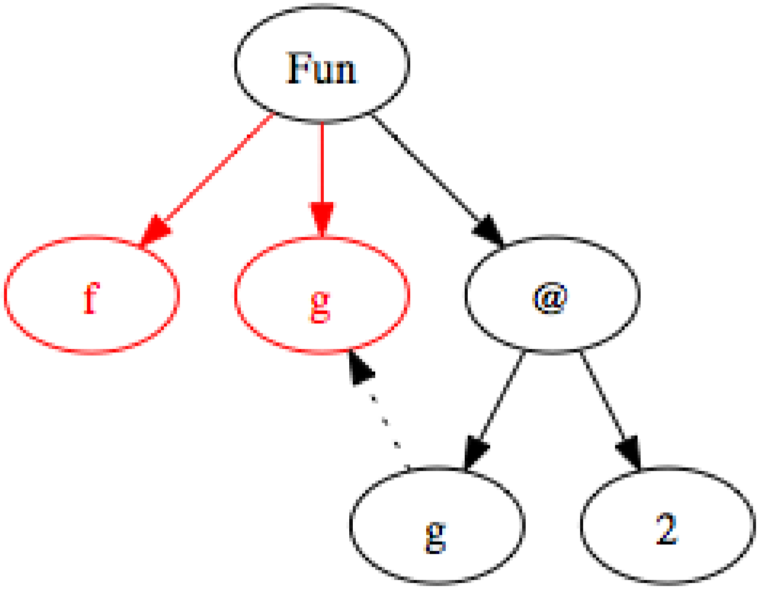
- Function declaration:
  - Type of f (t_0 in figure) must be domain $\rightarrow$ range
  - Domain is type of abstracted variable x (t_1 in fig)
  - Range is type of function body e (t_2 in fig)
  - Constraint: t_0 = t_1 -> t_2

Inferring Polymorphic Types

- Example:

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

- Step 1:
  Build Parse Tree

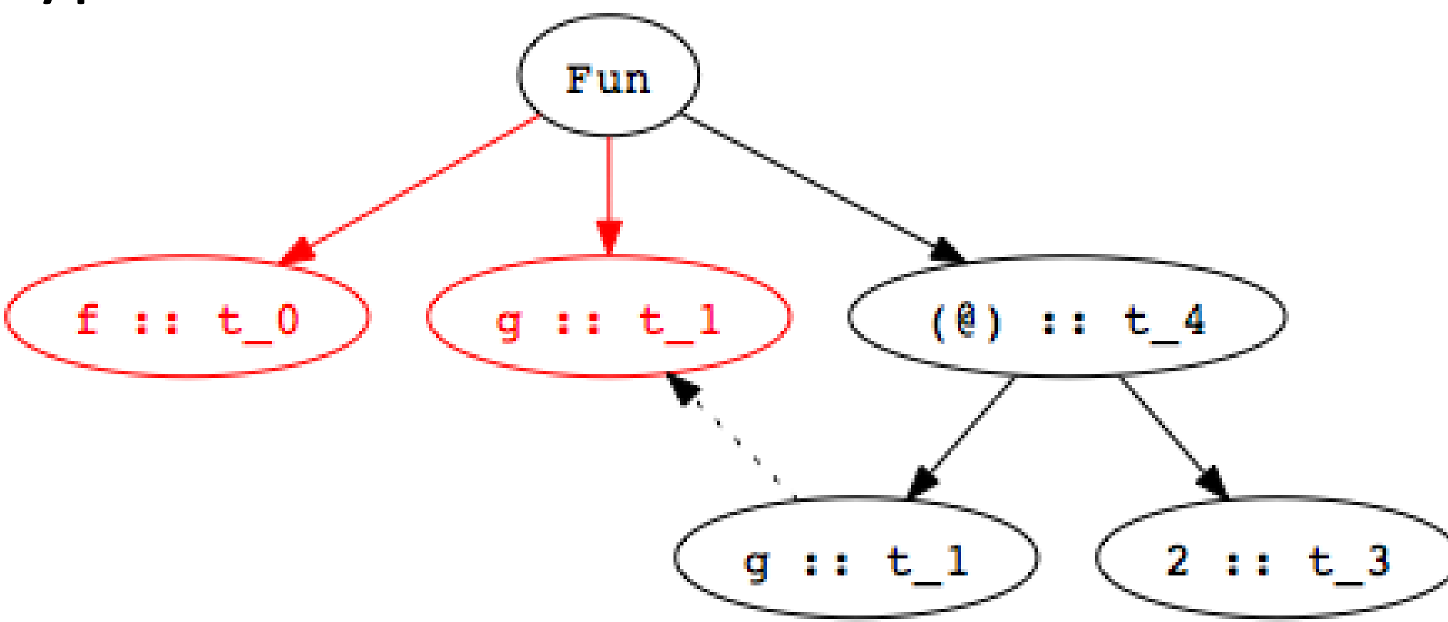


Inferring Polymorphic Types

- Example:

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

- Step 2:
  Assign type variables



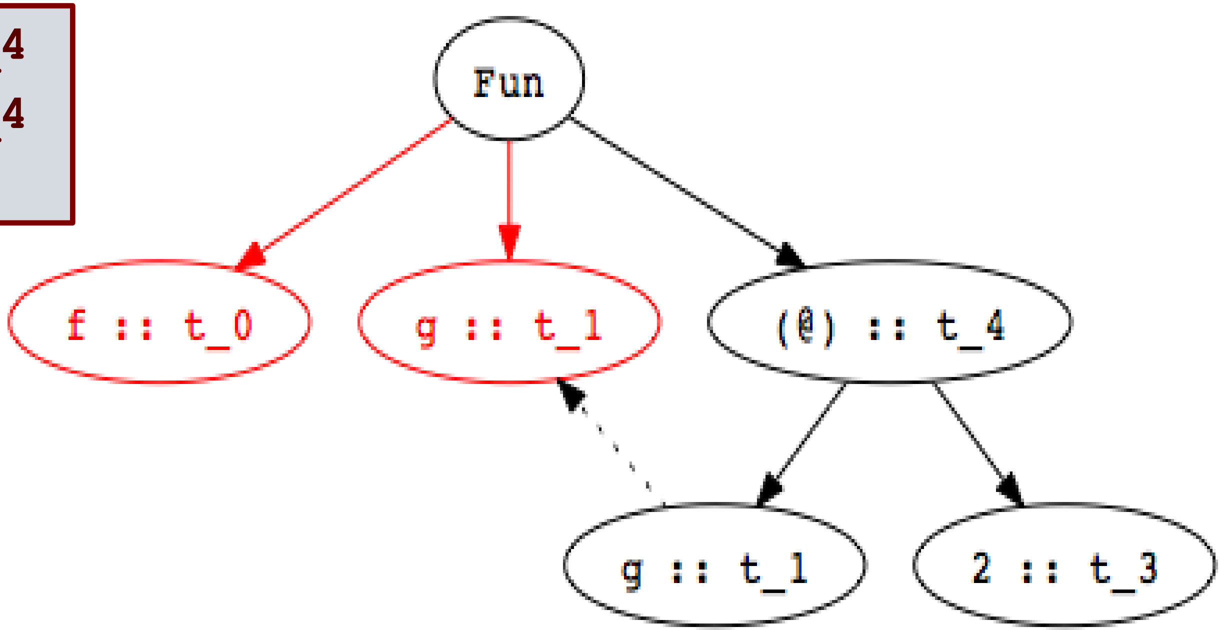
Inferring Polymorphic Types

- 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

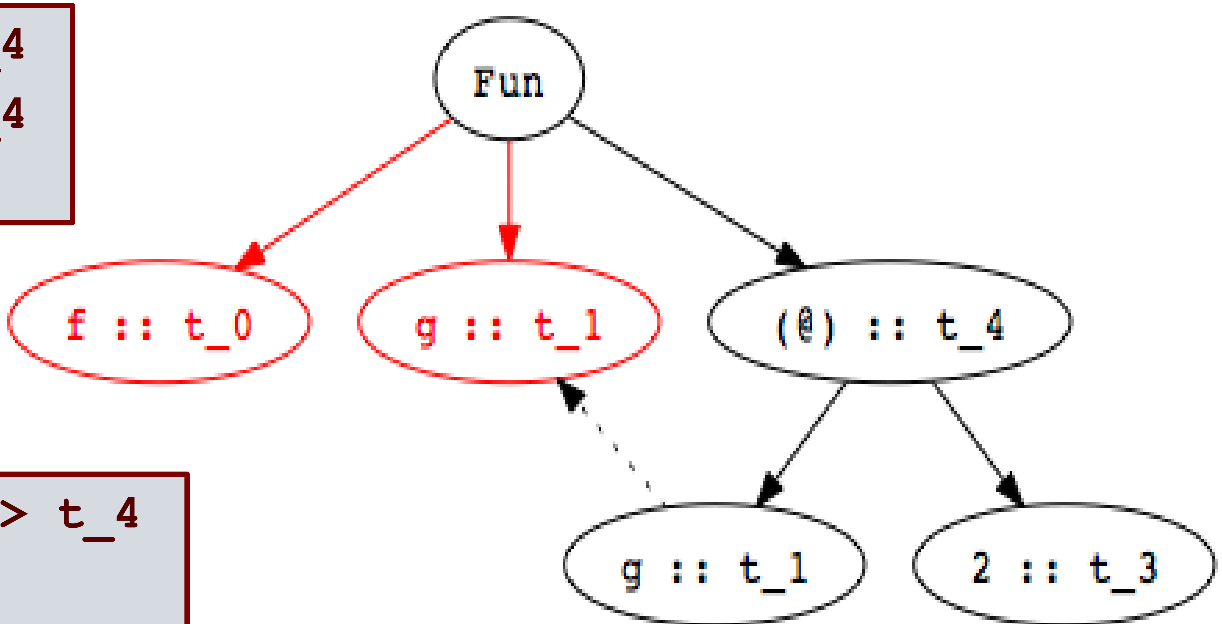
- Example:

```
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
```



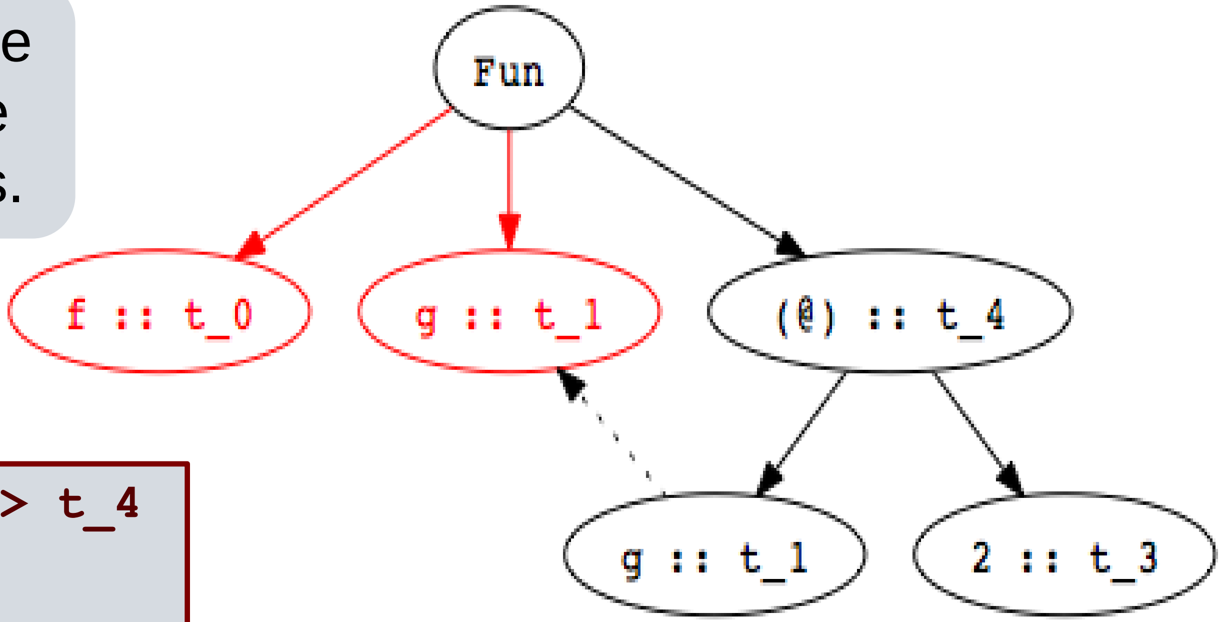
Inferring Polymorphic Types

- 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 :: Bool
```

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
```

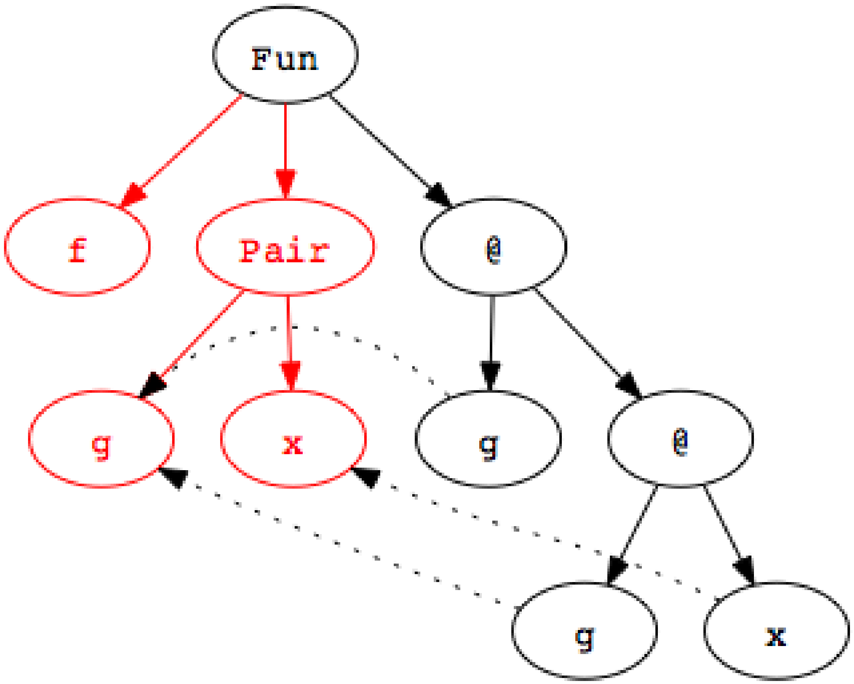
- Type error:
  cannot unify Bool $\rightarrow$ Bool and Int $\rightarrow$ t

Another Example

- Example:

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

- Step 1:
  Build Parse Tree

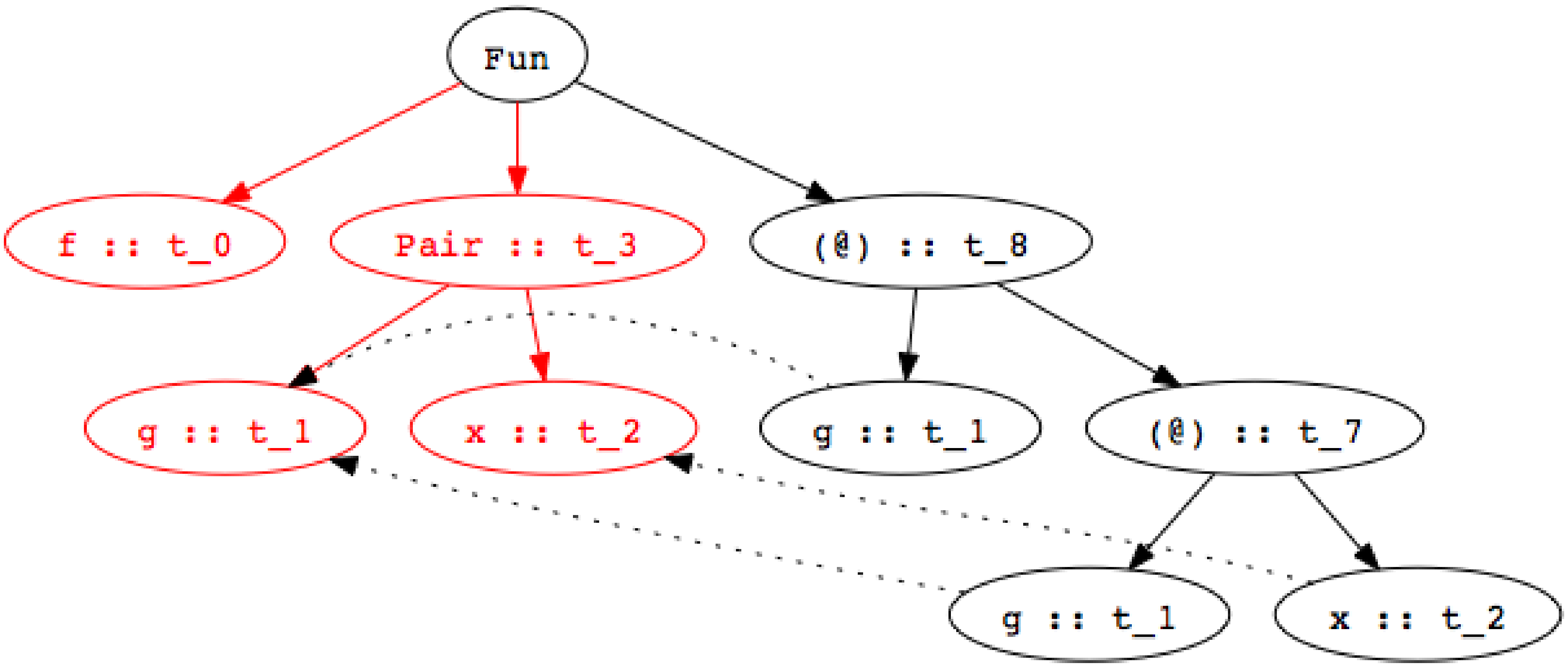


Another Example

- Example:

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

- Step 2:
  Assign type variables



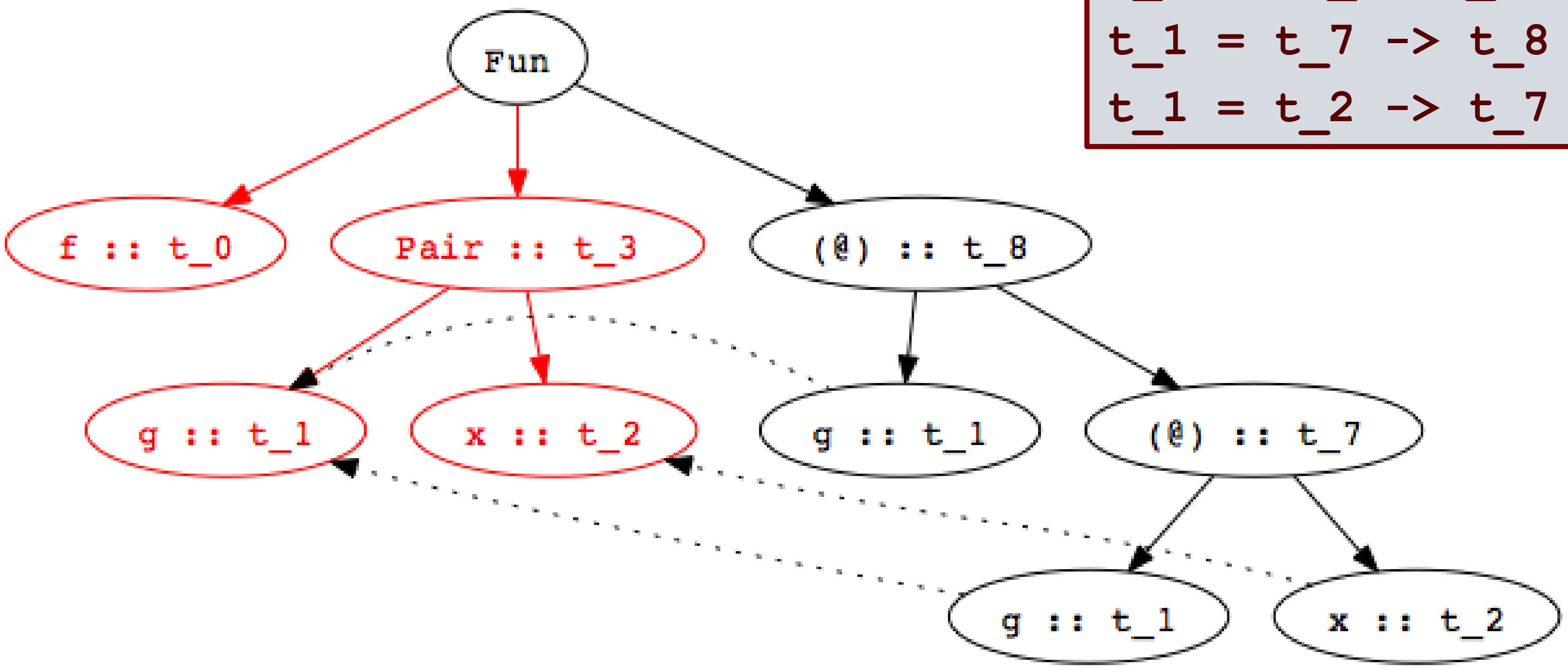
Another Example

- Example:

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

- Step 3:
  Generate constraints

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



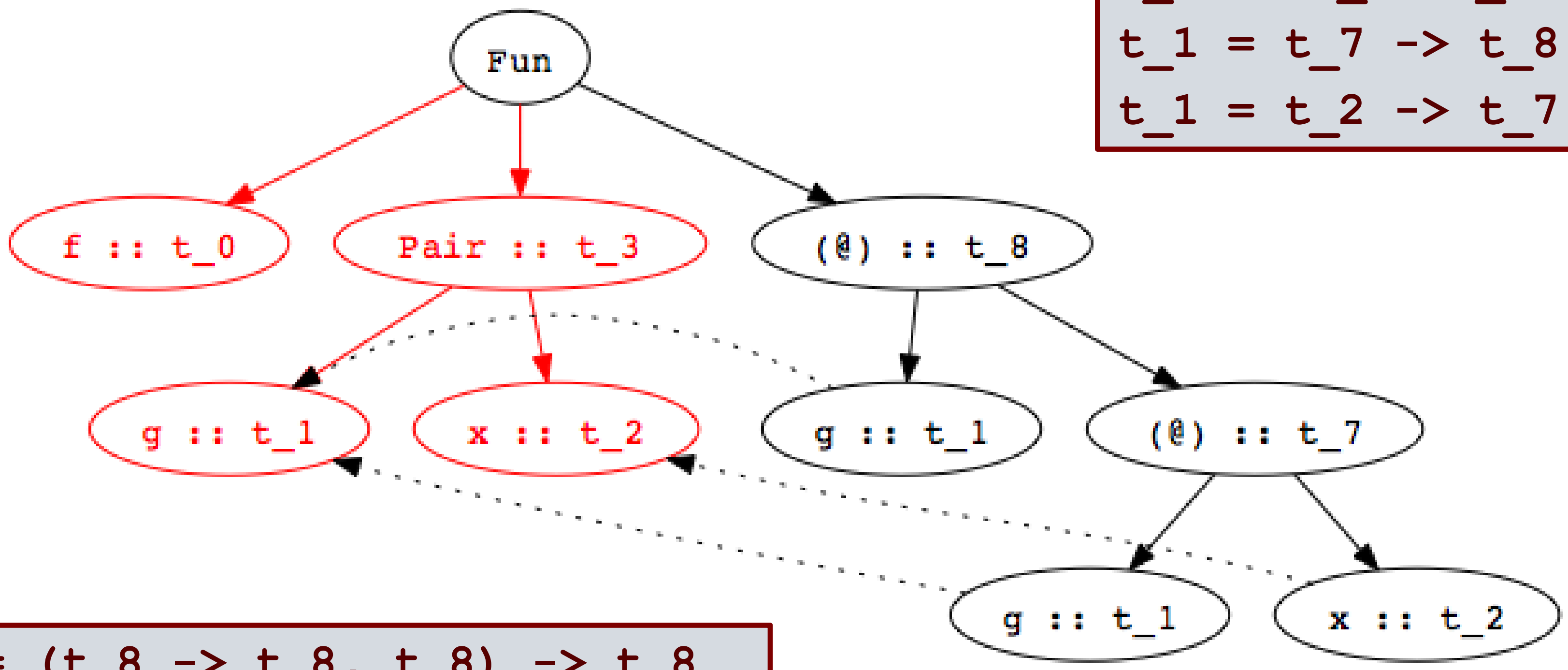
Another Example

- Example:

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

- Step 4:
  Solve constraints

```
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
```

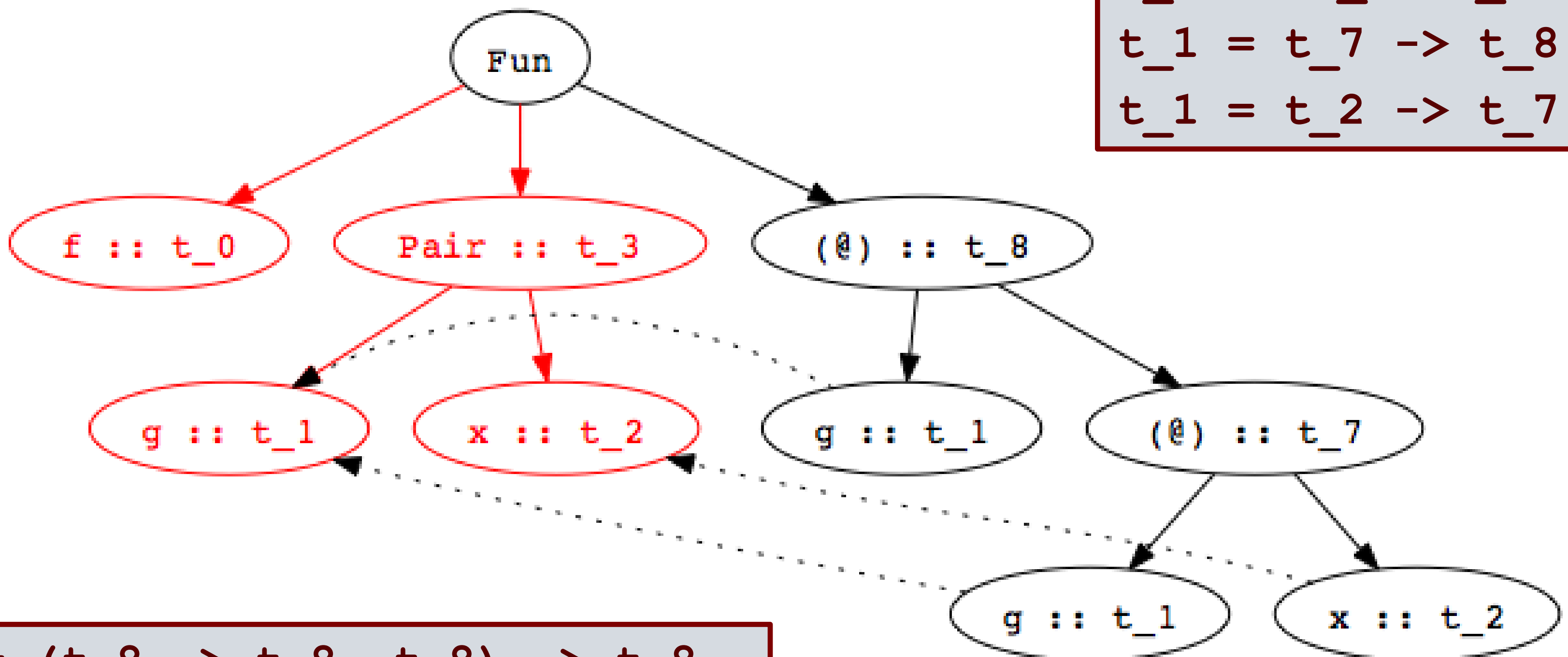
Another Example

- Example:

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

- Step 5:
  Determine type of f

```
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
```

Polymorphic Datatypes

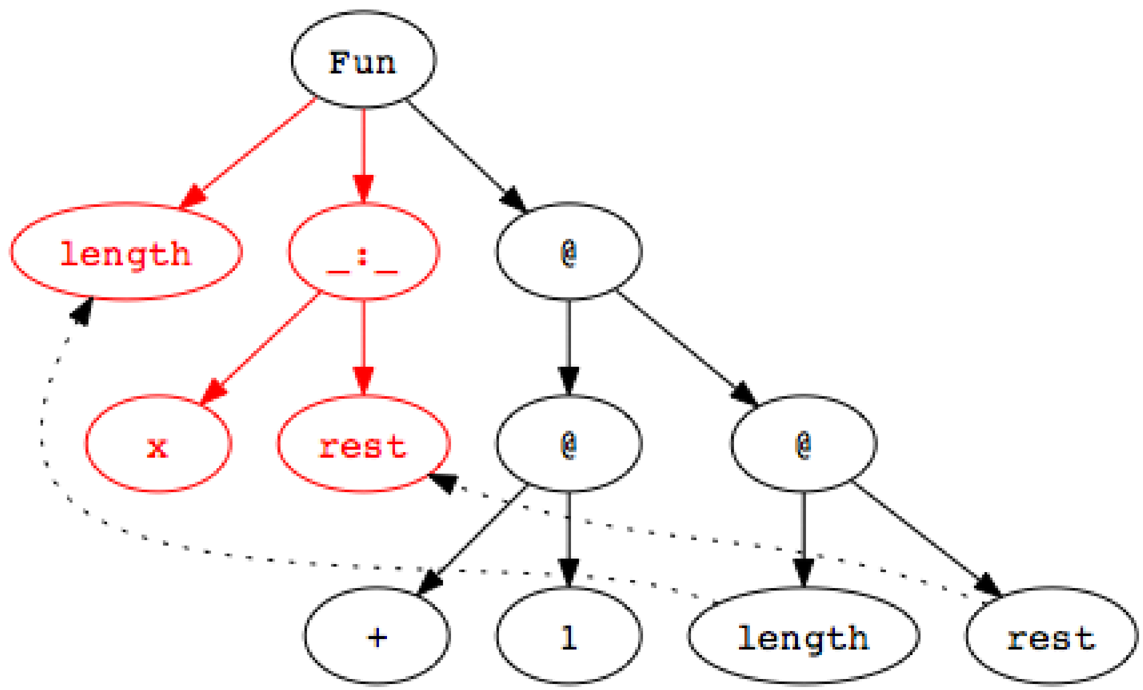
- Functions may have multiple clauses

```
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

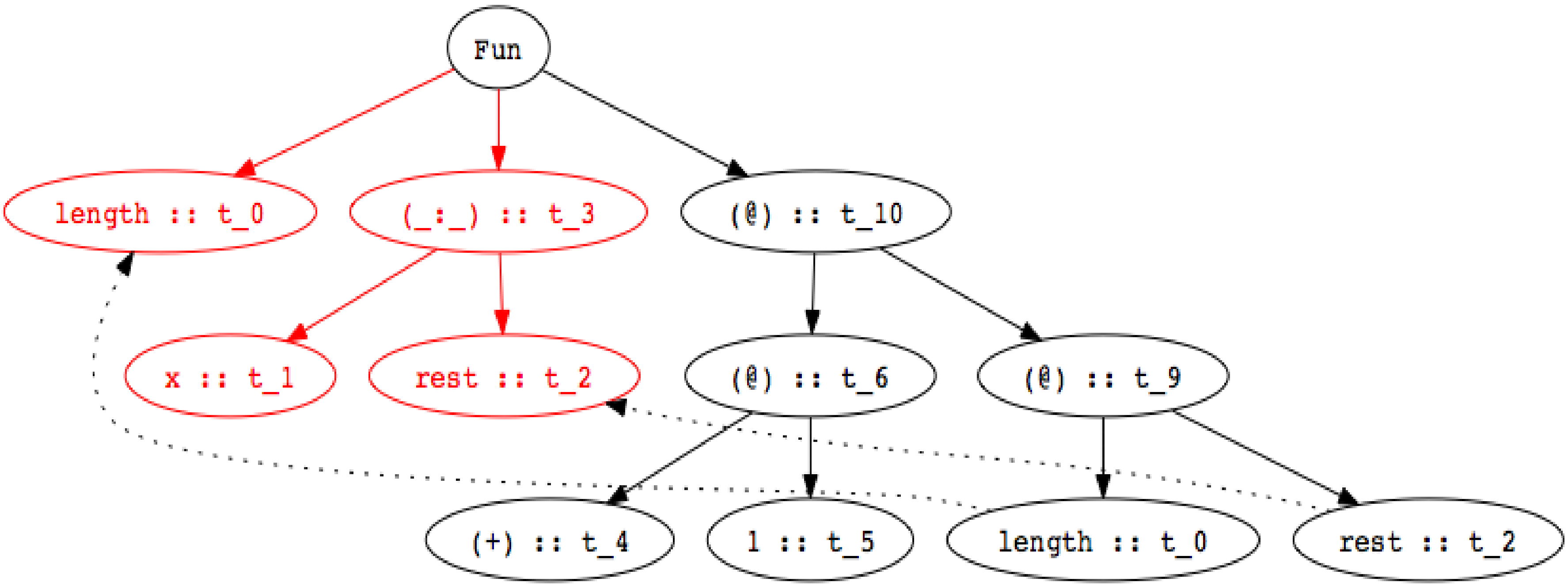
Type Inference with Datatypes

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



Type Inference with Datatypes

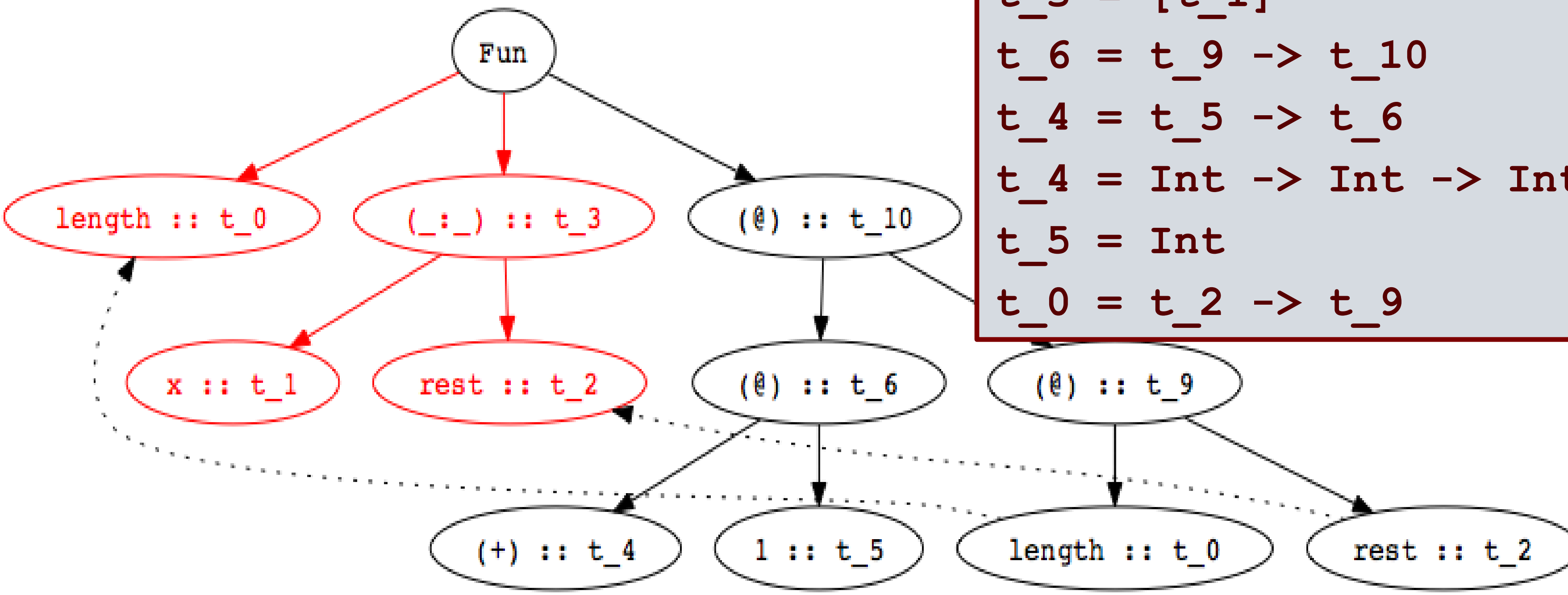
- Example: `length (x:rest) = 1 + (length rest)`
- Step 2: Assign type variables



Type Inference with Datatypes

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

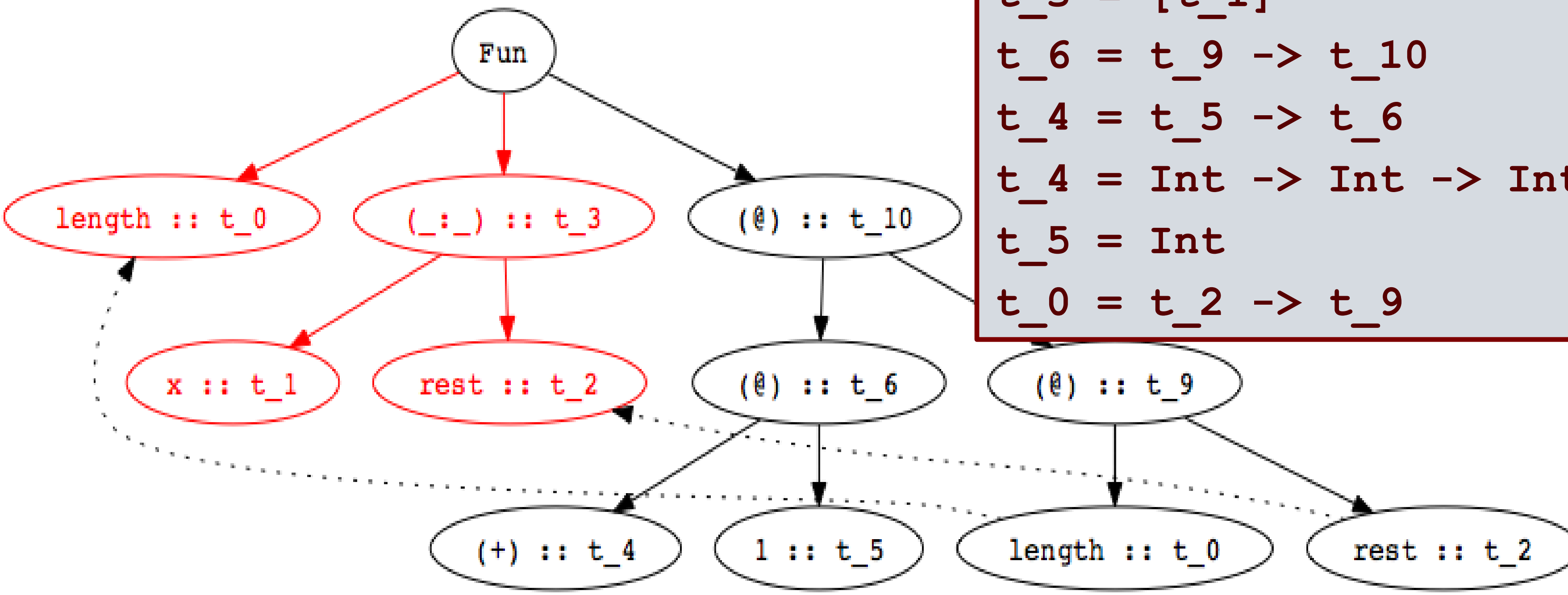
```
t_0 = t_3 -> t_10
t_3 = t_2
t_3 = [t_1]
t_6 = t_9 -> t_10
t_4 = t_5 -> t_6
t_4 = Int -> Int -> Int
t_5 = Int
t_0 = t_2 -> t_9
```



Type Inference with Datatypes

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

```
t_0 = t_3 -> t_10
t_3 = t_2
t_3 = [t_1]
t_6 = t_9 -> t_10
t_4 = t_5 -> t_6
t_4 = Int -> Int -> Int
t_5 = Int
t_0 = t_2 -> t_9
```




```
t_0 = [t_1] -> 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