301AA - Advanced Programming

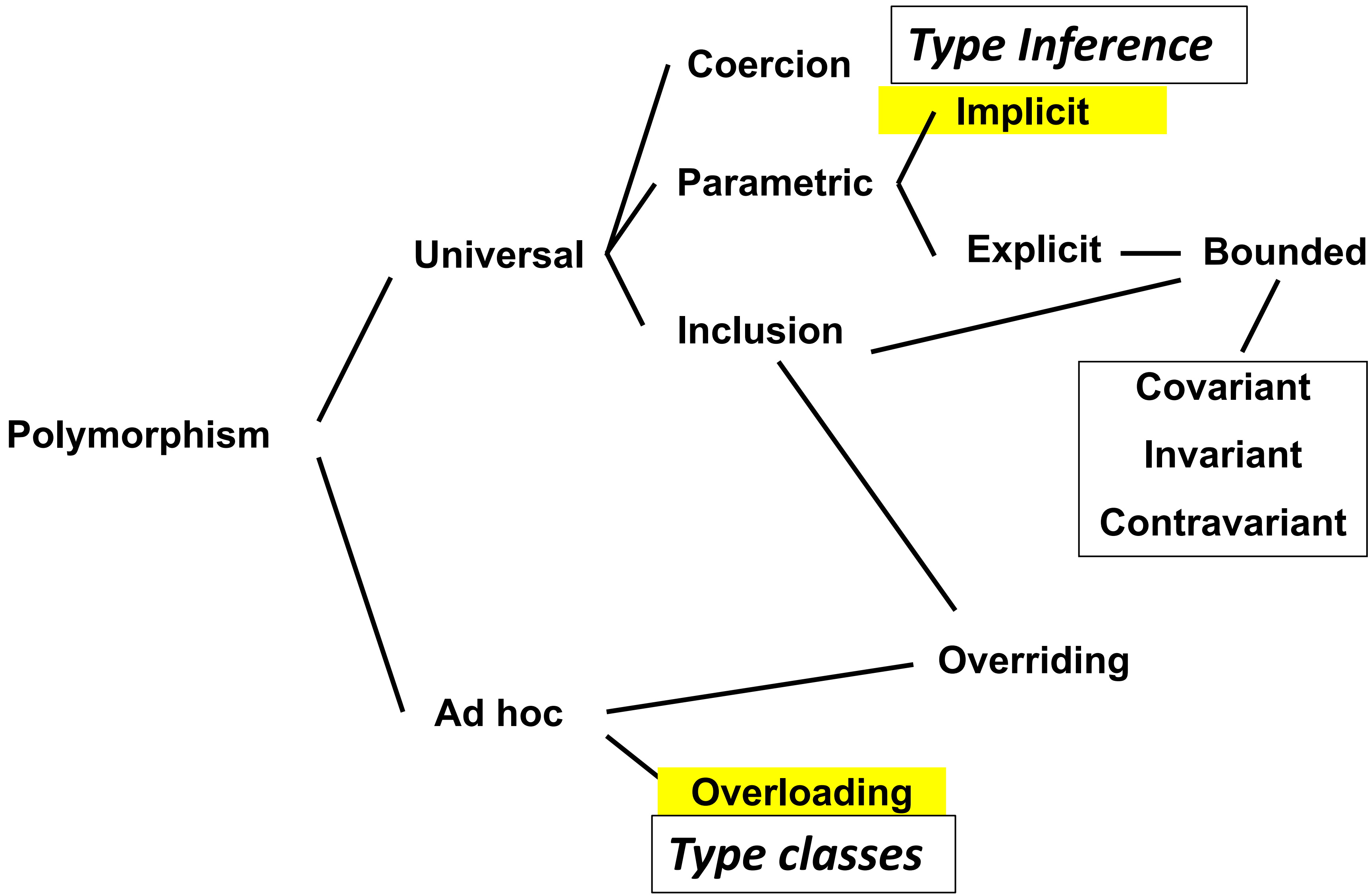
Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

AP-16*: Type Classes & Type Inference in Haskell*

Core Haskell

- Basic Types
  - Unit
  - Booleans
  - Integers
  - Strings
  - Reals
  - Tuples
  - Lists
  - Records
- Patterns
- Declarations
- Functions
- Polymorphism
- Type declarations
- Type Classes
- Monads
- Exceptions

Polymorphism in Haskell



Ad hoc polymorphism: **overloading**

- Present in all languages, at least for built-in arithmetic operators: +, *, -, …
- Sometimes supported for user defined functions (Java, C++, …)
- C++, Haskell allow overloading of primitive operators
- The code to execute is determined by the type of the arguments, thus
  - **early binding** in statically typed languages
  - **late binding** in dynamically typed languages

Overloading: an example

- Function for squaring a number:

```
sqr(x) { return x * x; }
```

- Typed version (like in C) :

```
int sqr(int x) { return x * x; }
```

- Multiple versions for different types:

```
int sqrInt(int x) { return x * x; }
double sqrDouble(double x) { return x * x; }
```

- Overloading (Java, C++):

```
int sqr(int x) { return x * x; }
double sqr(double x) { return x * x; }
```

- But which type can be inferred by ML/Haskell?

```
> sqr x = x * x
```

Overloading besides arithmetic

- Some functions are "fully polymorphic"

```
length :: [w] -> Int
```

- Many useful functions are less polymorphic

```
member :: [w] -> w -> Bool
```

- Membership only works for types that support equality.

```
sort :: [w] -> [w]
```

- List sorting only works for types that support ordering.

Overloading Arithmetic, Take 1

- Allow functions containing overloaded symbols to define multiple functions:

```
square x = x * x          -- legal
-- Defines two versions:
-- Int -> Int and Float -> Float
```

- But consider:

```
squares (x,y,z) =
   (square x, square y, square z)
-- There are 8 possible versions!
```

- Approach not widely used because of exponential growth in number of versions.

Overloading Arithmetic, Take 2

- Basic operations such as + and * can be overloaded, but not functions defined from them

```
3 * 3                 -- legal
3.14 * 3.14           -- legal
square x = x * x      -- Int -> Int
square 3              -- legal
square 3.14           -- illegal
```

- **Standard ML** uses this approach.
- Not satisfactory: Programmers cannot define functions that implementation might support

Overloading Equality, Take 1

- Equality defined only for types that admit equality: types not containing **function types** or **abstract types**.

```
3 * 3 == 9                -- legal
'a' == 'b'                -- legal
\x->x == \y->y+1          -- illegal
```

- Overload equality like arithmetic ops + and * in SML.
- But then we can't define functions using '==':

```
member [] y      = False
member (x:xs) y = (x==y) || member xs y

member [1,2,3] 3          -- ok if default is Int
member "Haskell" 'k'      -- illegal
```

- Approach adopted in first version of SML.

Overloading Equality, Take 2

- Make type of equality fully polymorphic

```
(==) :: a -> a -> Bool
```

- Type of list-membership function

```
member :: [a] -> a -> Bool
```

- **Miranda** used this approach. But…
  - equality applied to a **function** yields a runtime error
  - equality applied to an **abstract type** compares the underlying representation, which violates abstraction principles

Overloading Equality, Take 3

- Make equality polymorphic **in a limited way**:

```
(==) :: a(==) -> a(==) -> Bool
```

  where a(==) is type variable restricted to **types with equality**

- Now we can type the member function:

```
member :: a(==) -> [a(==)] -> Bool
member  4           [2,3] :: Bool
member 'c'          ['a', 'b', 'c'] :: Bool
member (\y -> y*2)   [\x -> x, \x -> x+2]  -- type error
```

- Approach used in SML today, where the type a(==) is called an **eqtype variable** and is written **''a** (while normal type variables are written **'a**)

Type Classes

- Type classes solve these problems
  - Idea: Generalize ML's eqtypes to arbitrary types
  - Provide concise types to describe overloaded functions, so no exponential blow-up
  - Allow users to define functions using overloaded operations, eg, square, squares, and member
  - Allow users to declare new collections of overloaded functions: equality and arithmetic operators are not privileged built-ins
  - Fit within type inference framework

Behind type classes: Intuition

- A function to sort lists can be passed a comparison operator as an argument:

```
qsort:: (a -> a -> Bool) -> [a] -> [a]
qsort cmp [] = []
qsort cmp (x:xs) = qsort cmp (filter (cmp x) xs)
                 ++ [x] ++
                    qsort cmp (filter (not.cmp x) xs)
```

  - This allows the function to be parametric
- We can build on this idea …

Intuition (continued)

- Consider the "overloaded" parabola function

```
parabola x = (x * x) + x
```

- We can rewrite the function to take the operators it contains as an argument

```
parabola' (plus, times) x = plus (times x x) x
```

  - The extra parameter is a "dictionary" that provides implementations for the overloaded ops.
  - We have to rewrite all calls to pass appropriate implementations for plus and times:

```
y = parabola'(intPlus,intTimes) 10
z = parabola'(floatPlus, floatTimes) 3.14
```

Systematic programming style

```
-- Dictionary type
data MathDict a = MkMathDict (a->a->a) (a->a->a)

-- Accessor functions
get_plus :: MathDict a -> (a->a->a)
get_plus (MkMathDict p t) = p

get_times :: MathDict a -> (a->a->a)
get_times (MkMathDict p t) = t

-- "Dictionary-passing style"
parabola :: MathDict a -> a -> a
parabola dict x = let plus  = get_plus  dict
                      times = get_times dict
                  in plus (times x x) x
```

Type class declarations will generate Dictionary type and selector functions

Systematic programming style

Type class **instance declarations** produce instances of the Dictionary

```
-- Dictionary type
data MathDict a = MkMathDict (a->a->a) (a->a->a)

-- Dictionary construction
intDict   = MkMathDict intPlus   intTimes
floatDict = MkMathDict floatPlus floatTimes

-- Passing dictionaries
y = parabola intDict   10
z = parabola floatDict 3.14
```

Compiler will add a dictionary parameter and rewrite the body as necessary

Type Class Design Overview

- **Type class declarations**
  - Define a set of operations, give the set a name
  - Example: `Eq a` type class
    - operations == and \= with type `a -> a -> Bool`
- **Type class instance declarations**
  - Specify the implementations for a particular type
  - For `Int` instance, == is defined to be integer equality
- **Qualified types (or Type Constraints)**
  - Concisely express the operations required on otherwise polymorphic type

```
member:: Eq w => w -> [w] -> Bool
```

Qualified Types

"for all types w that support the Eq operations"

```
Member :: Eq w => w -> [w] -> Bool
```

If a function works for every type with particular properties, the type of the function says just that:

```
sort      :: Ord a  => [a] -> [a]
serialise :: Show a => a -> String
square    :: Num n  => n -> n
squares   ::(Num t, Num t1, Num t2) =>
                 (t, t1, t2) -> (t, t1, t2)
```

Otherwise, it must work for any type

```
reverse :: [a] -> [a]
filter  :: (a -> Bool) -> [a] -> [a]
```

Type Classes

Works for any type 'n' that supports the Num operations

```
square :: Num n => n -> n
square x = x*x
```

The class declaration says what the Num operations are

```
class Num a where
  (+)    :: a -> a -> a
  (*)    :: a -> a -> a
  negate :: a -> a
  ...etc...
```

An instance declaration for a type T says how the Num operations are implemented on T's

```
instance Num Int where
  a + b    = intPlus  a b
  a * b    = intTimes a b
  negate a = intNeg a
  ...etc...
```

```
intPlus  :: Int -> Int -> Int
intTimes :: Int -> Int -> Int
etc, defined as primitives
```

Compiling Overloaded Functions

When you write this...

```
square :: Num n => n -> n
square x = x*x
```

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

The "Num n =>" turns into an extra value argument to the function. It is a value of data type Num n and it represents a dictionary of the required operations.

A value of type (Num n) is a dictionary of the Num operations for type n

Compiling Type Classes

When you write this...

```
square :: Num n => n -> n
square x = x*x
```

```
class Num n where
  (+)    :: n -> n -> n
  (*)    :: n -> n -> n
  negate :: n -> n
  ...etc...
```

The class decl translates to:
A data type decl for Num
A selector function for each class operation

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

```
data Num n
  = MkNum (n -> n -> n)
          (n -> n -> n)
          (n -> n)
          ...etc...
...
(*) :: Num n -> n -> n -> n
(*) (MkNum _ m _ ...) = m
```

A value of type (Num n) is a dictionary of the Num operations for type n

Compiling Instance Declarations

When you write this...

```
square :: Num n => n -> n
square x = x*x
```

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

```
instance Num Int where
  a + b    = intPlus  a b
  a * b    = intTimes a b
  negate a = intNeg a
  ...etc...
```

```
dNumInt :: Num Int
dNumInt = MkNum intPlus
                intTimes
                intNeg
                ...
```

An instance decl for type T translates to a value declaration for the Num dictionary for T

A value of type (Num n) is a dictionary of the Num operations for type n

Implementation Summary

- Each overloaded symbol has to be introduced in at least one type class.
- The compiler translates each function that uses an overloaded symbol into a function with an extra parameter: the dictionary.
- References to overloaded symbols are rewritten by the compiler to lookup the symbol in the dictionary.
- The compiler converts each type class declaration into a dictionary type declaration and a set of selector functions.
- The compiler converts each instance declaration into a dictionary of the appropriate type.
- The compiler rewrites calls to overloaded functions to pass a dictionary. It uses the static, qualified type of the function to select the dictionary.

Functions with Multiple Dictionaries

```
squares :: (Num a, Num b, Num c) => (a, b, c) -> (a, b, c)
squares(x,y,z) = (square x, square y, square z)
```

Note the concise type for the squares function!

```
squares :: (Num a, Num b, Num c) -> (a, b, c) -> (a, b, c)
squares (da,db,dc) (x, y, z) =
                     (square da x, square db y, square dc z)
```

Pass appropriate dictionary on to each square function.

Compositionality

Overloaded functions can be defined from other overloaded functions:

```
sumSq :: Num n => n -> n -> n
sumSq x y = square x + square y
```

```
sumSq :: Num n -> n -> n -> n
sumSq d x y = (+) d (square d x)
                    (square d y)
```

Extract addition operation from d

Pass on d to square

Compositionality

Build compound instances from simpler ones:

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Int where
  (==) = intEq      -- intEq primitive equality

instance (Eq a, Eq b) => Eq(a,b)
  (u,v) == (x,y)      = (u == x) && (v == y)

instance Eq a => Eq [a] where
  (==) []     []      = True
  (==) (x:xs) (y:ys) = x==y && xs == ys
  (==) _      _       = False
```

Compound Translation

Build compound instances from simpler ones.

```
class Eq a where
  (==) :: a -> a -> Bool
instance Eq a => Eq [a] where
  (==) []     []     = True
  (==) (x:xs) (y:ys) = x==y && xs == ys
  (==) _      _      = False
```

```
data Eq = MkEq (a->a->Bool)     -- Dictionary type
(==) (MkEq eq) = eq             -- Selector
dEqList :: Eq a -> Eq [a]       -- List Dictionary
dEqList d = MkEq eql
  where
    eql []     []     = True
    eql (x:xs) (y:ys) = (==) d x y && eql xs ys
    eql _      _      = False
```

Many Type Classes

- **Eq**: equality
- **Ord**: comparison
- **Num**: numerical operations
- **Show**: convert to string
- **Read**: convert from string
- **Testable**, **Arbitrary**: testing.
- **Enum**: ops on sequentially ordered types
- **Bounded**: upper and lower values of a type
- Generic programming, reflection, monads, …
- And many more.

Subclasses

- We could treat the Eq and Num type classes separately

```
memsq :: (Eq a, Num a) => a -> [a] -> Bool
memsq x xs = member (square x) xs
```

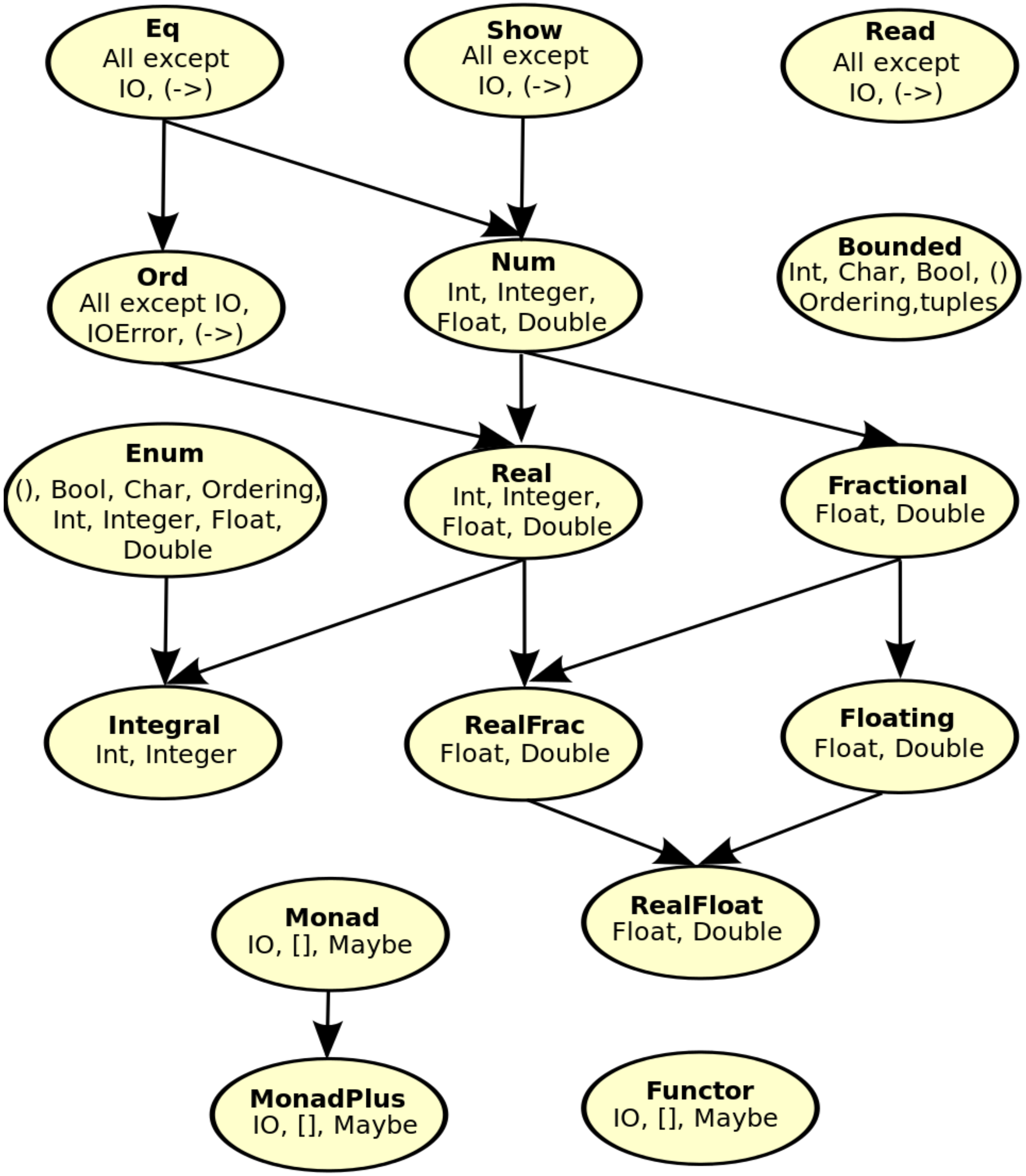
  – But we expect any type supporting Num to also support Eq

- A subclass declaration expresses this relationship:

```
class Eq a => Num a where
   (+) :: a -> a -> a
   (*) :: a -> a -> a
```

- With that declaration, we can simplify the type of the function

```
memsq :: Num a => a -> [a] -> Bool
memsq x xs = member (square x) xs
```


Eq
All except
IO, (->)
Show
All except
IO, (->)
Read
All except
IO, (->)
Ord
All except IO,
IOError, (->)
Num
Int, Integer,
Float, Double
Bounded
Int, Char, Bool, ()
Ordering,tuples
Enum
(), Bool, Char, Ordering,
Int, Integer, Float,
Double
Real
Int, Integer,
Float, Double
Fractional
Float, Double
Integral
Int, Integer
RealFrac
Float, Double
Floating
Float, Double
Monad
IO, [], Maybe
RealFloat
Float, Double
MonadPlus
IO, [], Maybe
Functor
IO, [], Maybe

Default Methods

- Type classes can define "default methods"

```
-- Minimal complete definition:
--      (==) or (/=)
class Eq a where
     (==) :: a -> a -> Bool
     x == y     =  not (x /= y)
     (/=) :: a -> a -> Bool
     x /= y     =  not (x == y)
```

- Instance declarations can override default by providing a more specific definition.

Deriving

- For **Read**, **Show**, **Bounded**, **Enum**, **Eq**, and **Ord**, the compiler can generate instance declarations automatically

```
data Color = Red | Green | Blue
     deriving (Show, Read, Eq, Ord)
```

```
Main>:t show
show :: Show a => a -> String
Main> show Red
"Red"
Main> Red < Green
True
Main>:t read
read :: Read a => String -> a
Main> let c :: Color = read "Red"
Main> c
Red
```

- *Ad hoc* : derivations apply only to types where derivation code works

Numeric Literals

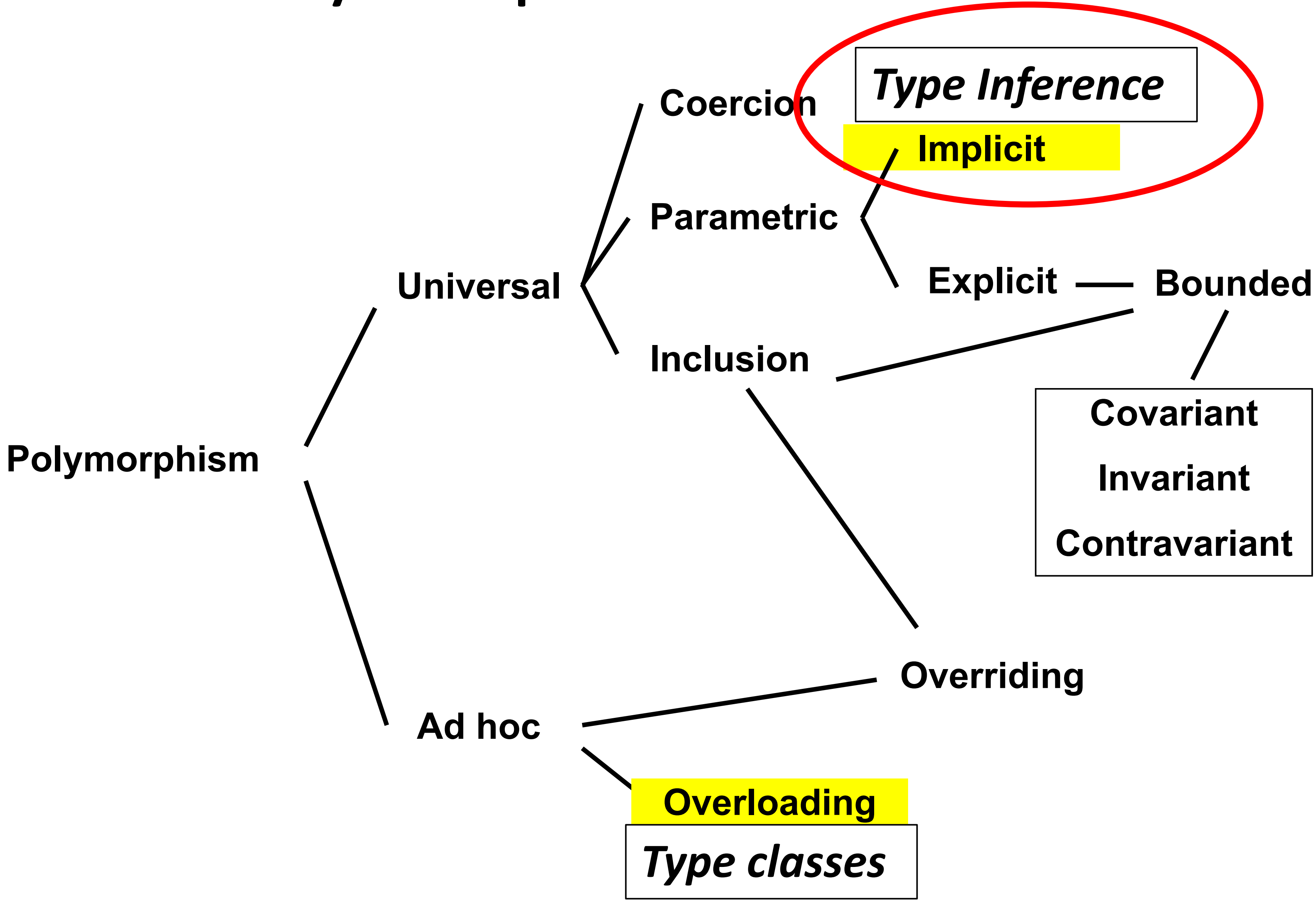
```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  fromInteger :: Integer -> a
  ...

inc :: Num a => a -> a
inc x = x + 1
```

Even literals are overloaded.
1 :: (Num a) => a

"1" means "fromInteger 1"

Advantages:
- Numeric literals can be interpreted as values of any appropriate numeric type
- Example: 1 can be an Integer or a Float or a user-defined numeric type.

Polymorphism in Haskell



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?

- Reduces syntactic overhead of expressive types, still allowing for static type checking
- Guaranteed to produce most general type
- Originally developed for functional languages, now used more and more in any kind of languages
- Illustrative example of a flow-insensitive static analysis algorithm

History & Complexity

- Original type inference algorithm
  - Invented by **Haskell Curry** and **Robert Feys** for the simply typed lambda calculus in 1958
- In 1969, **J. Roger Hindley**
  - extended the algorithm to a richer language and proved it always produced the most general type
- In 1978, **Robin Milner**
  - independently developed andequivalent algorithm, called algorithm W, during his work designing ML.
- In 1982, **Luis Damas** proved the algorithm was complete.
- 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

uHaskell

- Subset of Haskell to explain type inference.
  - Haskell and ML both have **overloading**
  - Will do not consider overloading now

```
<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      -- a simple declaration
```

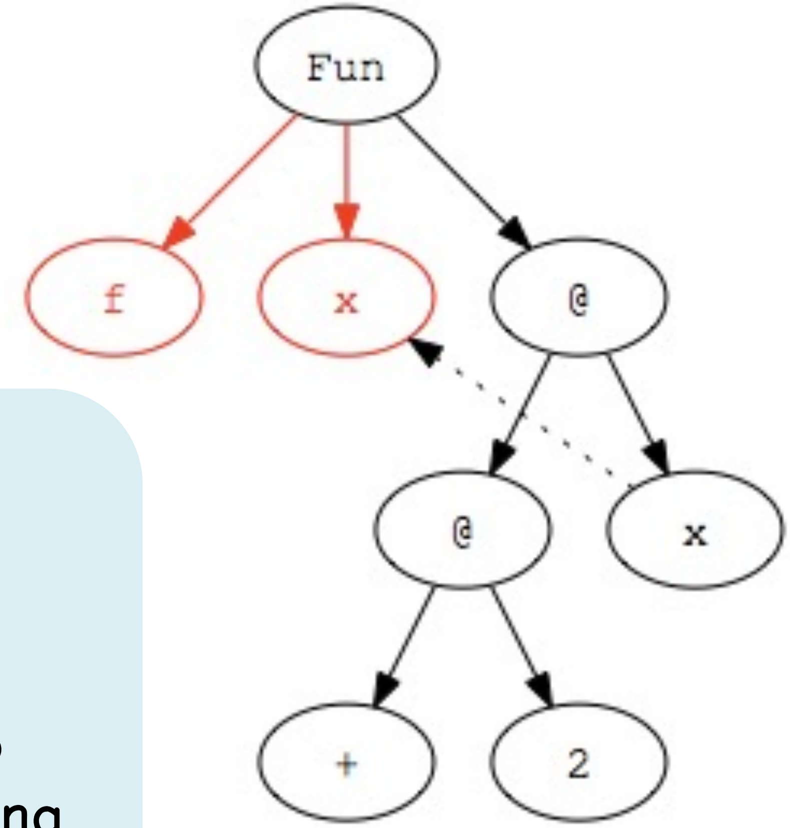
- What is the type of **`f`**?
  - $+$ has type: Int $\rightarrow$ Int $\rightarrow$ Int
    - *(with overloading would be Num a => a $\rightarrow$ a $\rightarrow$ a)*
  - 2 has type: Int
  - Since we are applying + to x we need x :: Int
  - Therefore f x = 2 + x has type Int $\rightarrow$ Int

```
f x = 2 + x      -- a simple declaration
> f :: Int -> Int
```

Step 1: Parse Program

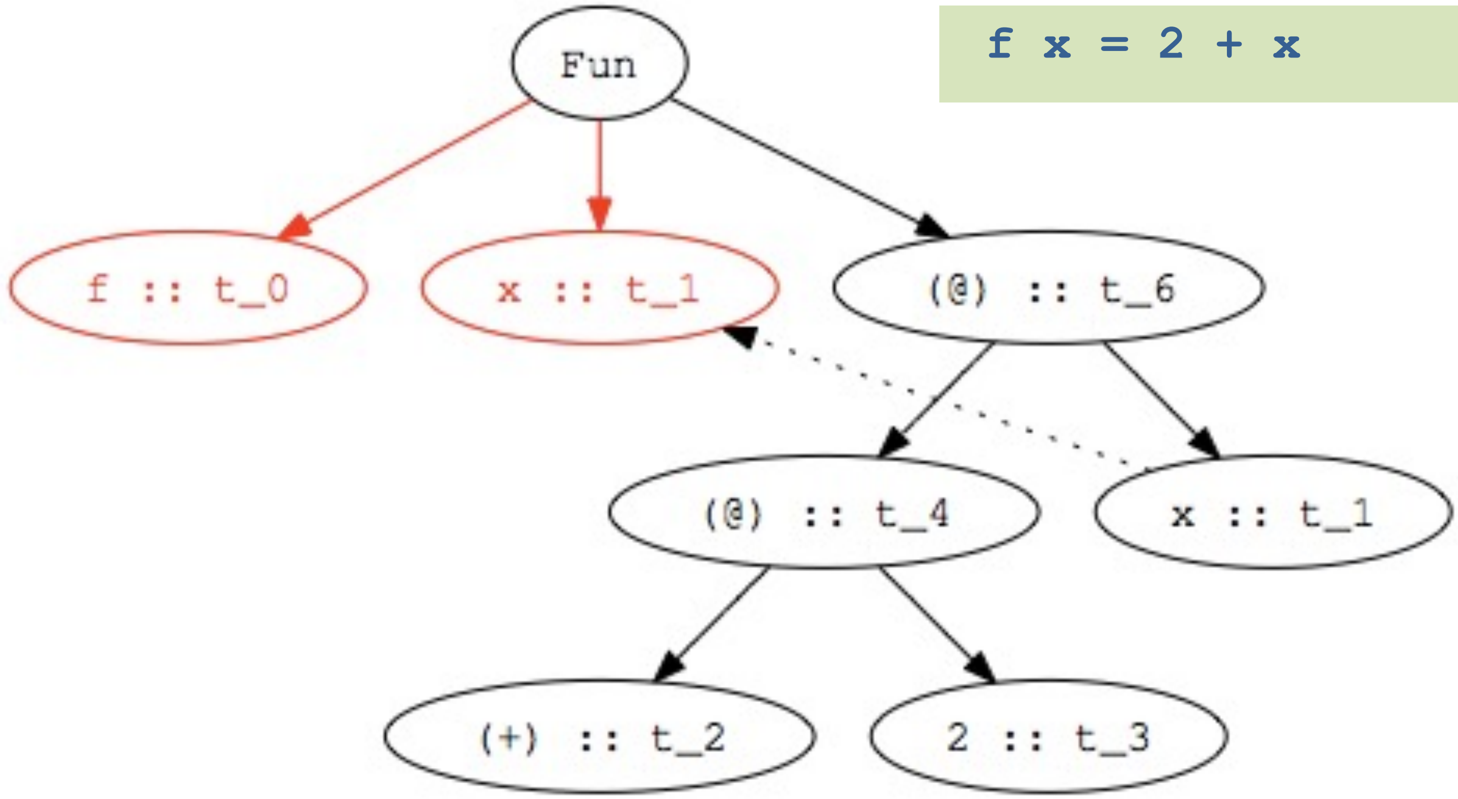
- Parse program text to construct parse tree.

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



- Binary @-nodes to represent application
- Ternary **Fun**-node for function definitions
- Infix operators are converted to Curried function application during parsing: 2 + x ➔ (+) 2 x

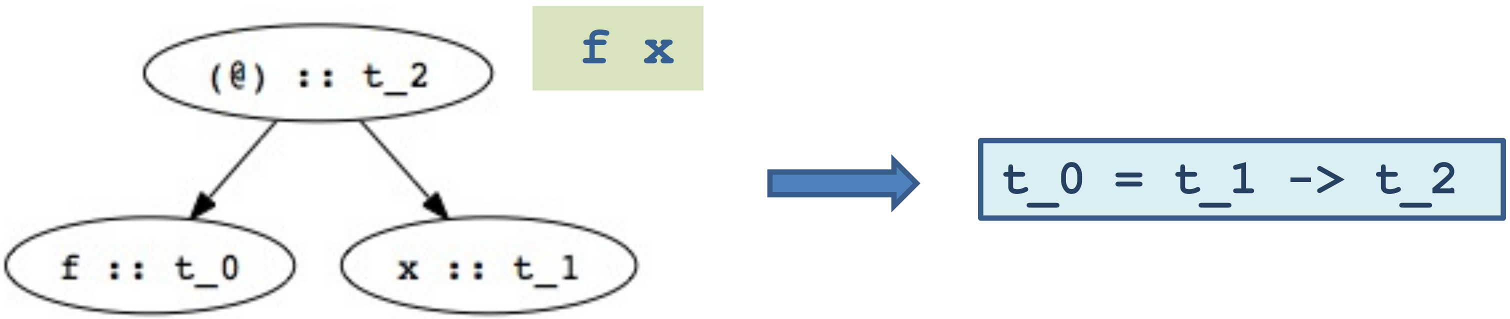
Step 2: Assign type variables to nodes




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

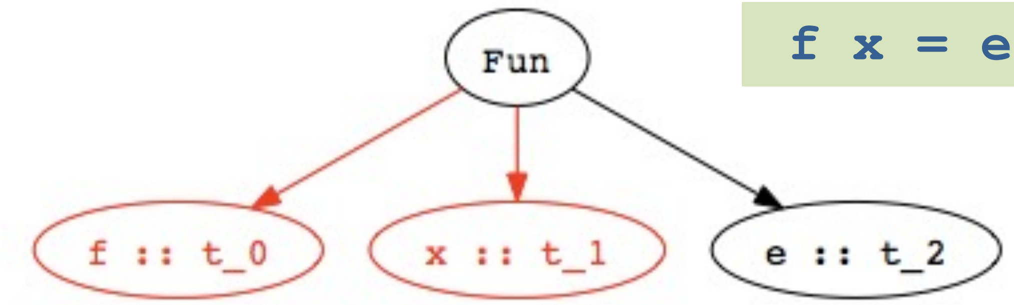
Variables are given same type as binding occurrence.

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)
  - **Range** of **`f`** must be result of application (t_2)
  - **Constraint**: t_0 = t_1 -> t_2

Constraints from Abstractions



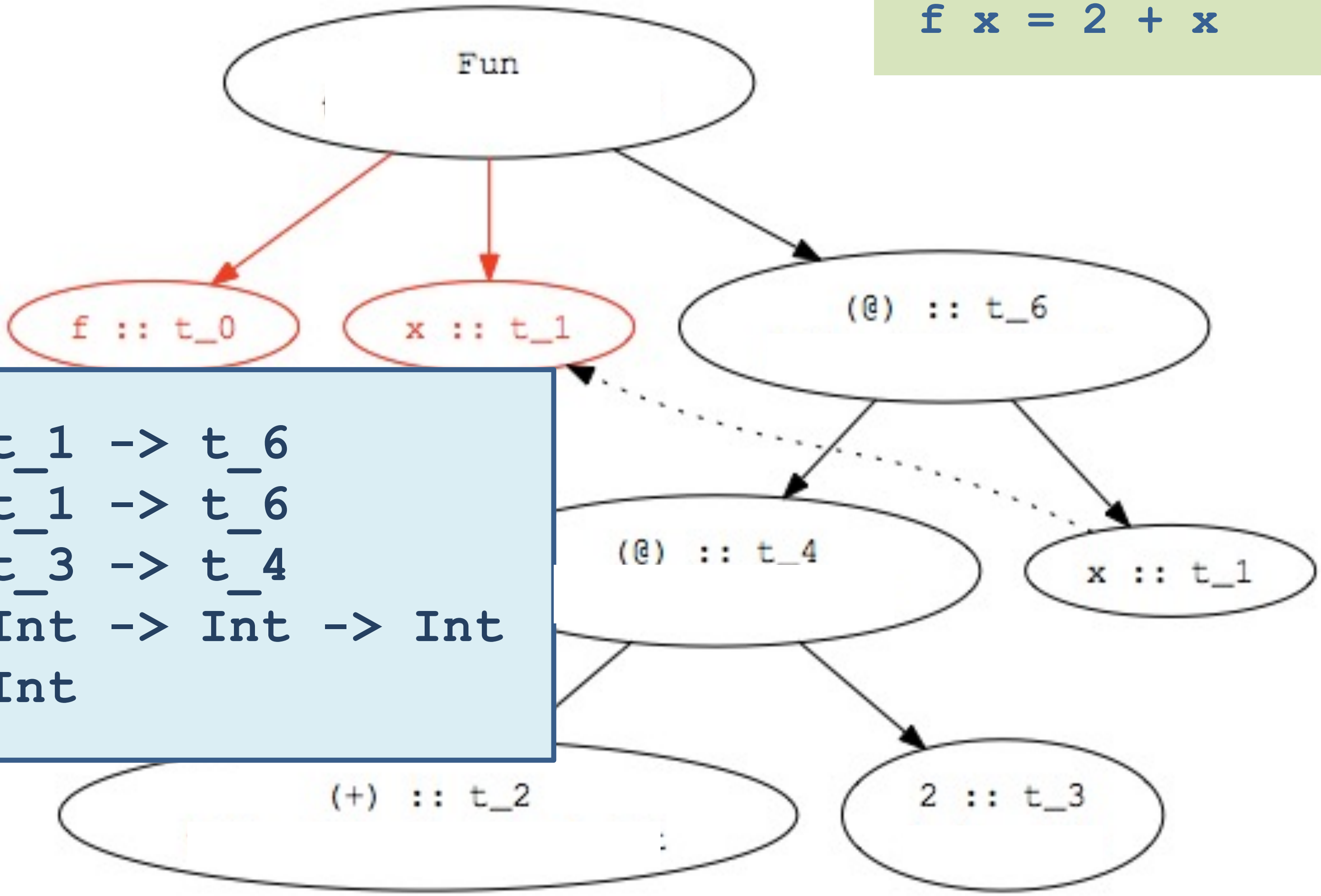
`f x = e`

`t_0 = t_1 -> t_2`

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

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
```

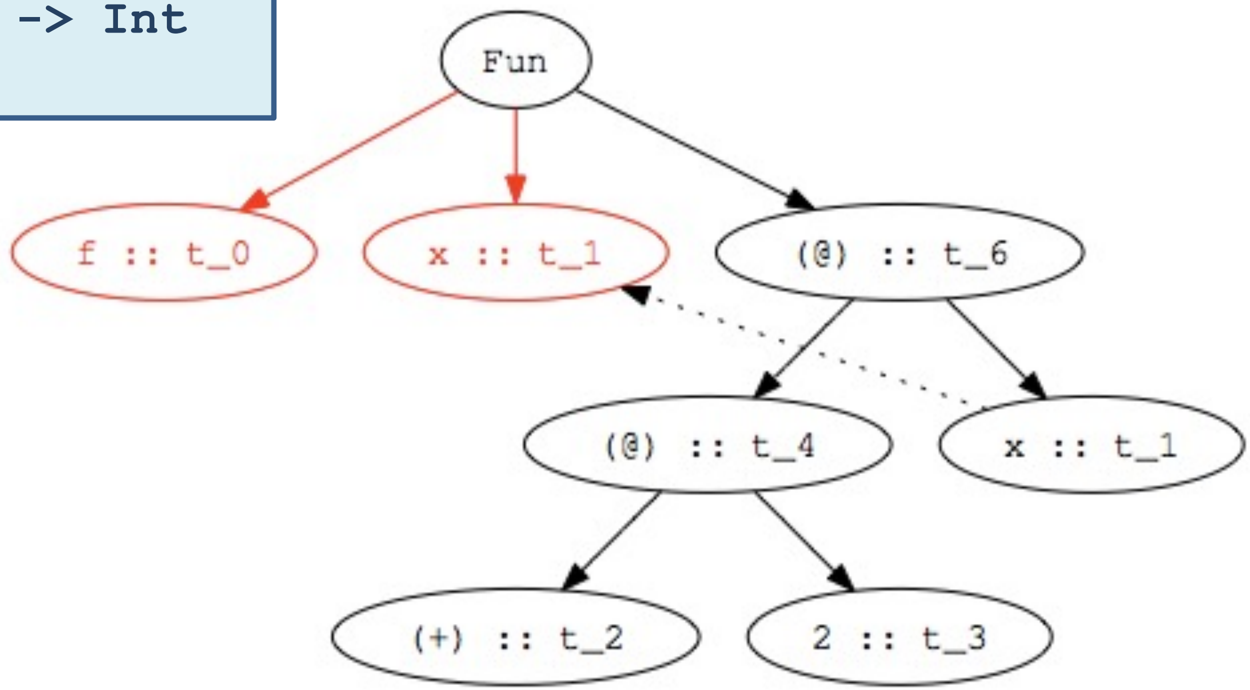
```
t_1 = Int
t_6 = Int
```

```
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
t_4 = Int -> Int
t_2 = Int -> Int -> Int
t_3 = Int
```

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



Type Inference Algorithm

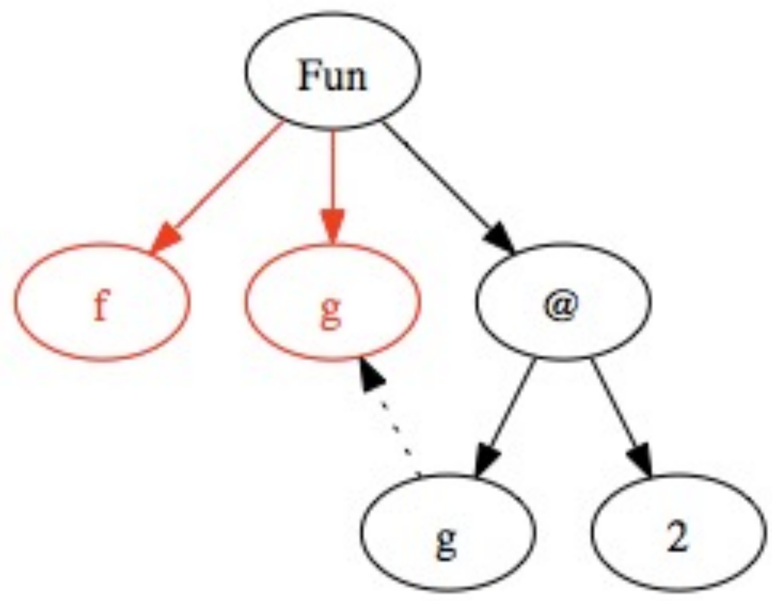
- Parse program to build parse tree
- Assign type variables to nodes in tree
- Generate constraints:
  - From environment: constants (2), built-in operators (+), known functions (tail).
  - From shape of parse tree: e.g., application and abstraction nodes.
- Solve constraints using *unification*
- Determine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

Inferring Polymorphic Types

- Example:

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

- Step 1:
  Build Parse Tree

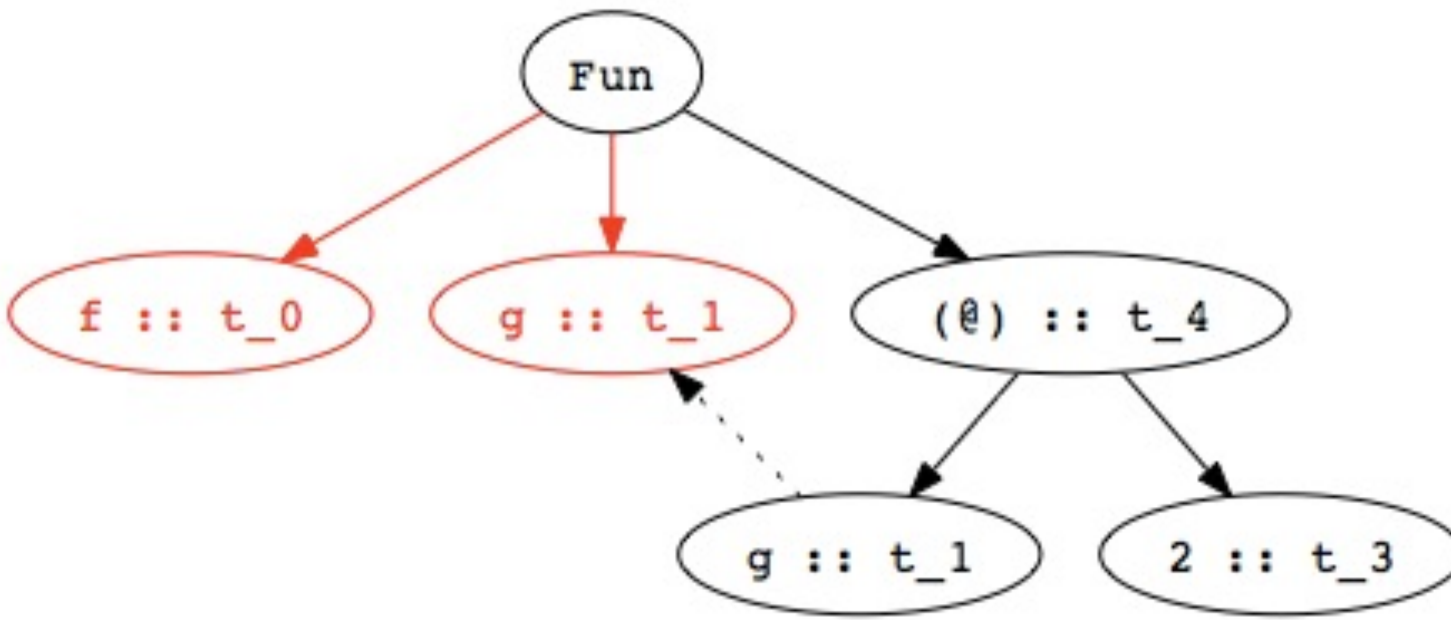


Inferring Polymorphic Types

- Example:

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

- Step 2:
  Assign type variables



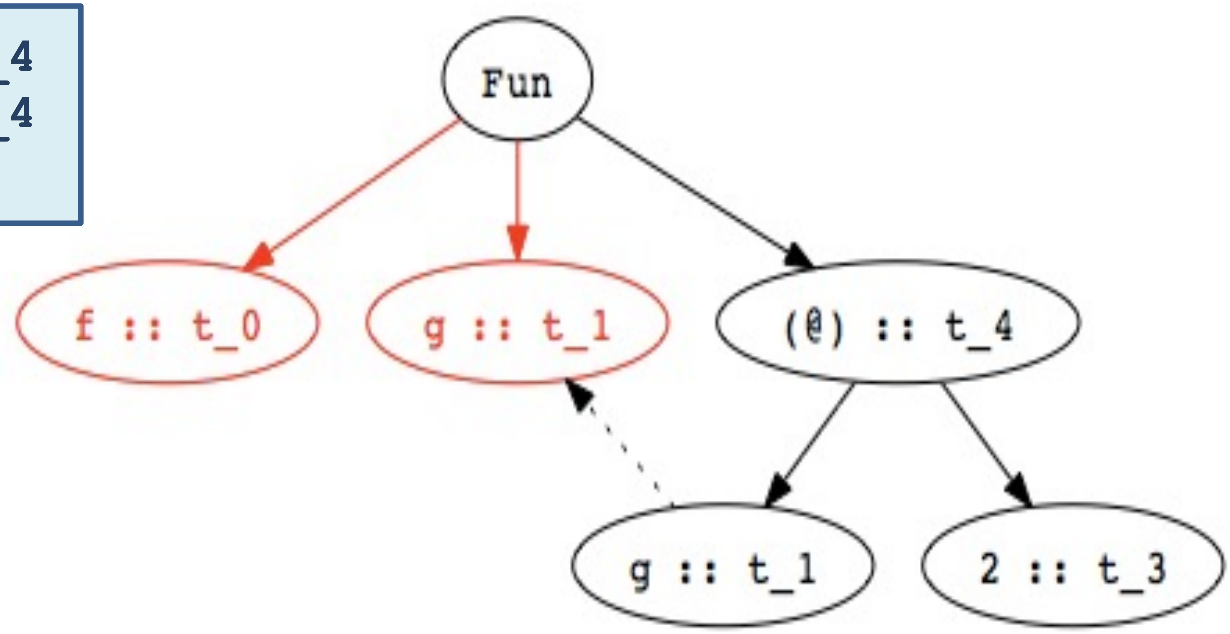
Inferring Polymorphic Types

- 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

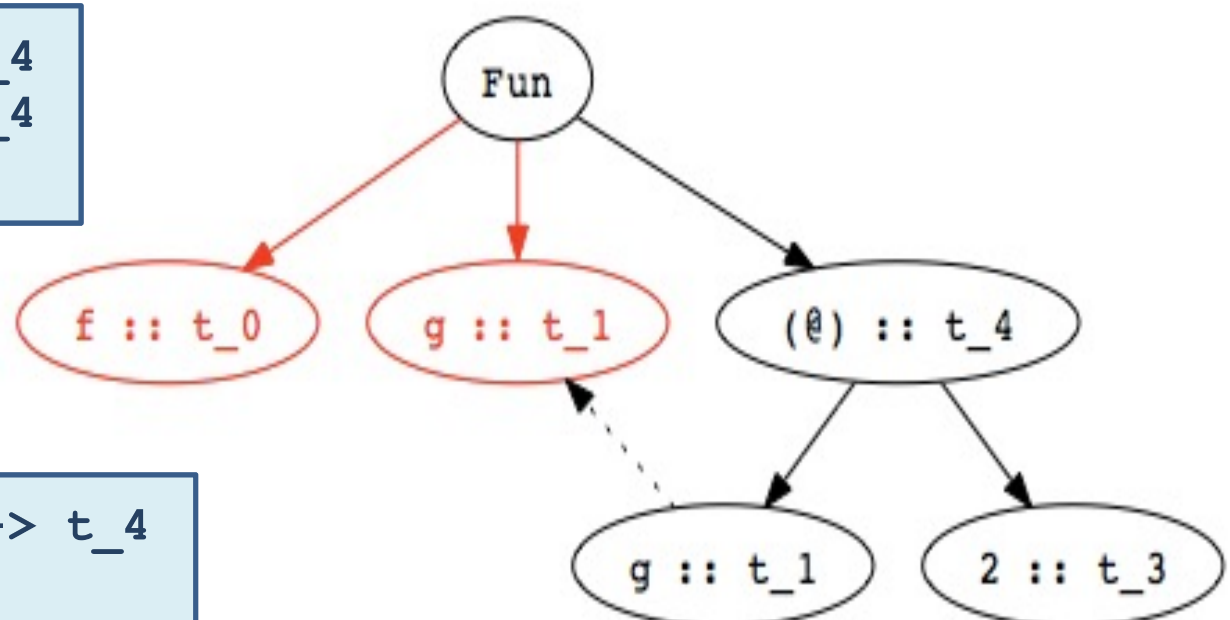
- Example:

```
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
```



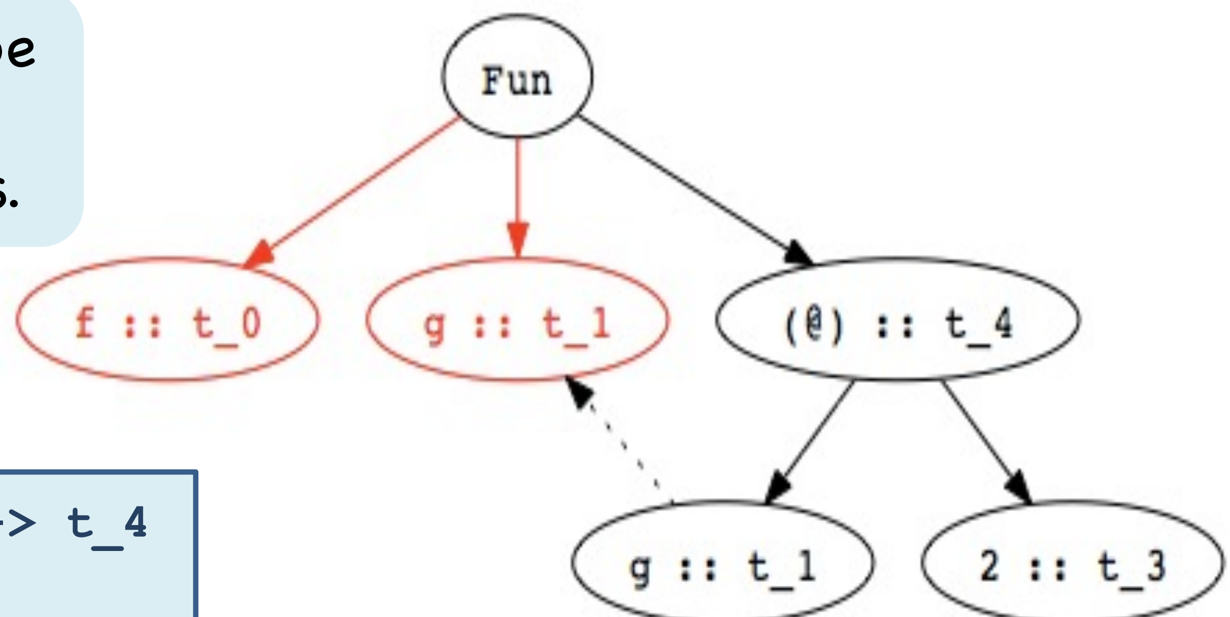
Inferring Polymorphic Types

- 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
```

Polymorphic Datatypes

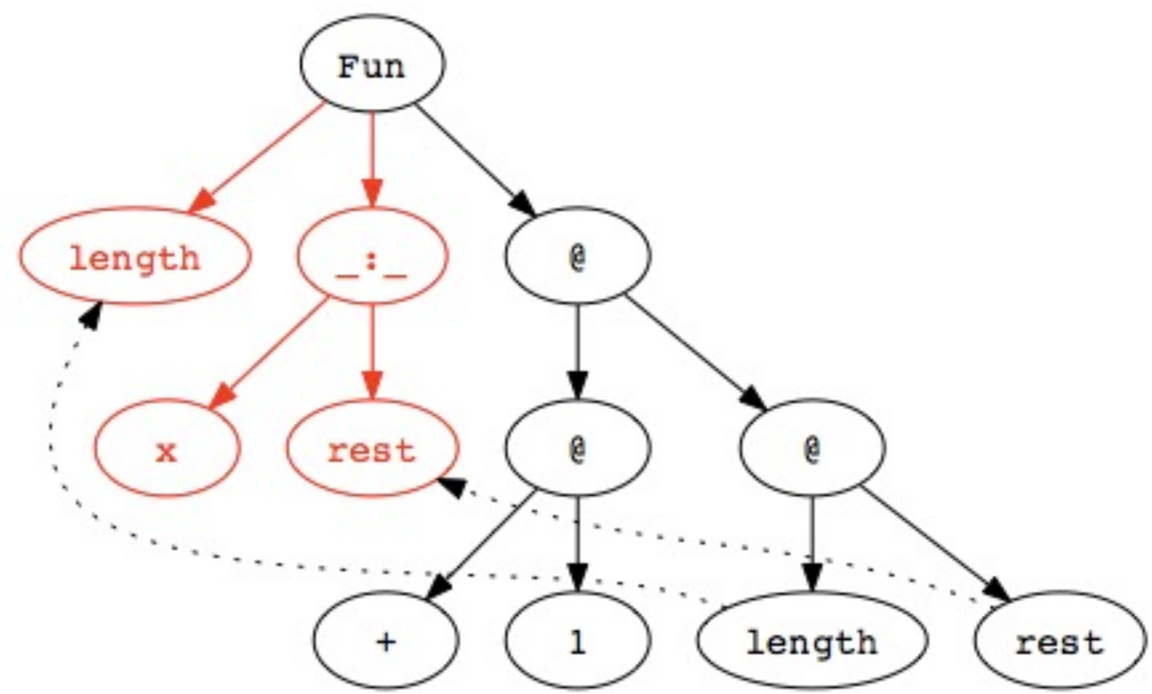
- Functions may have multiple clauses

```
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

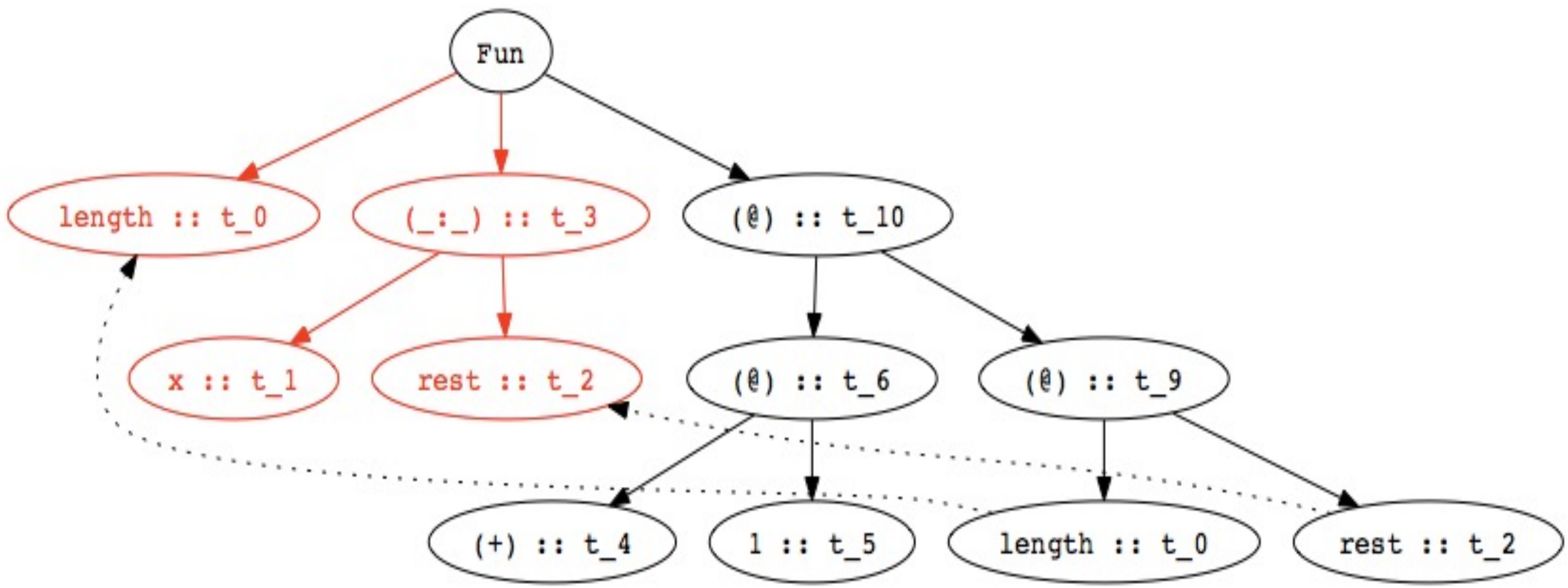
Type Inference with Datatypes

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



Type Inference with Datatypes

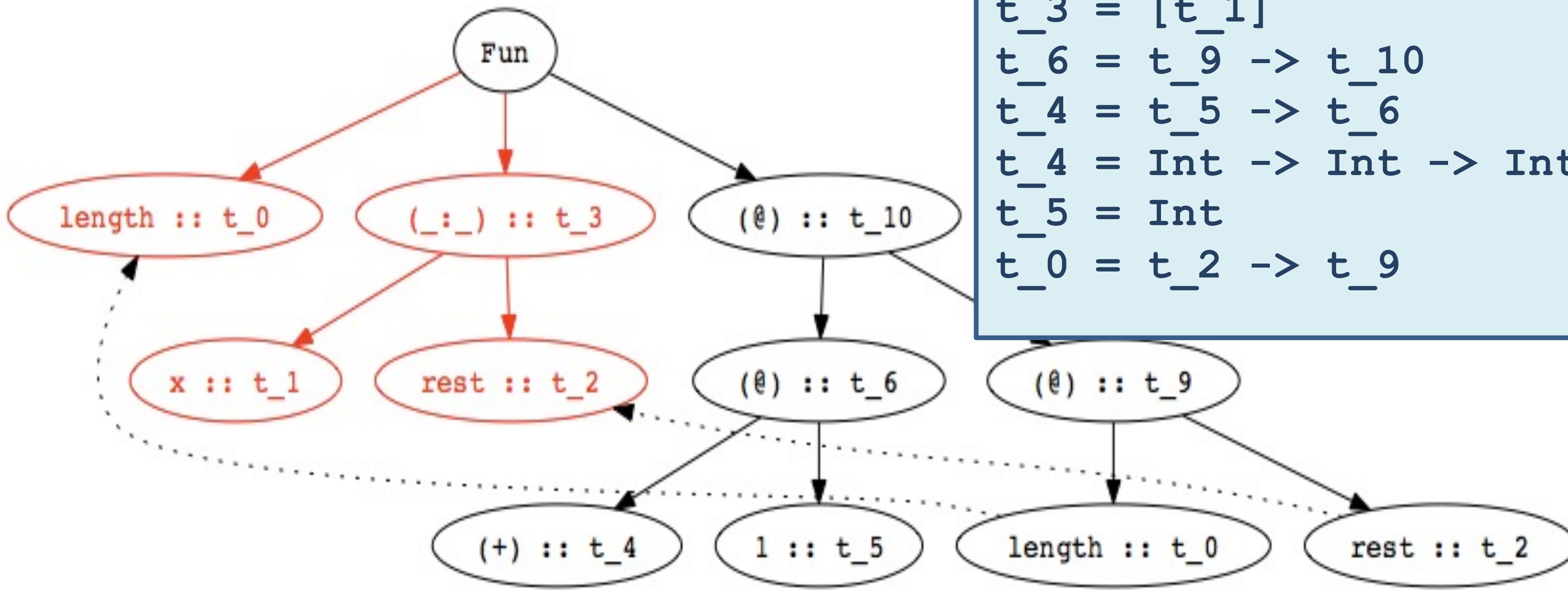
- Example: `length (x:rest) = 1 + (length rest)`
- Step 2: Assign type variables



Type Inference with Datatypes

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

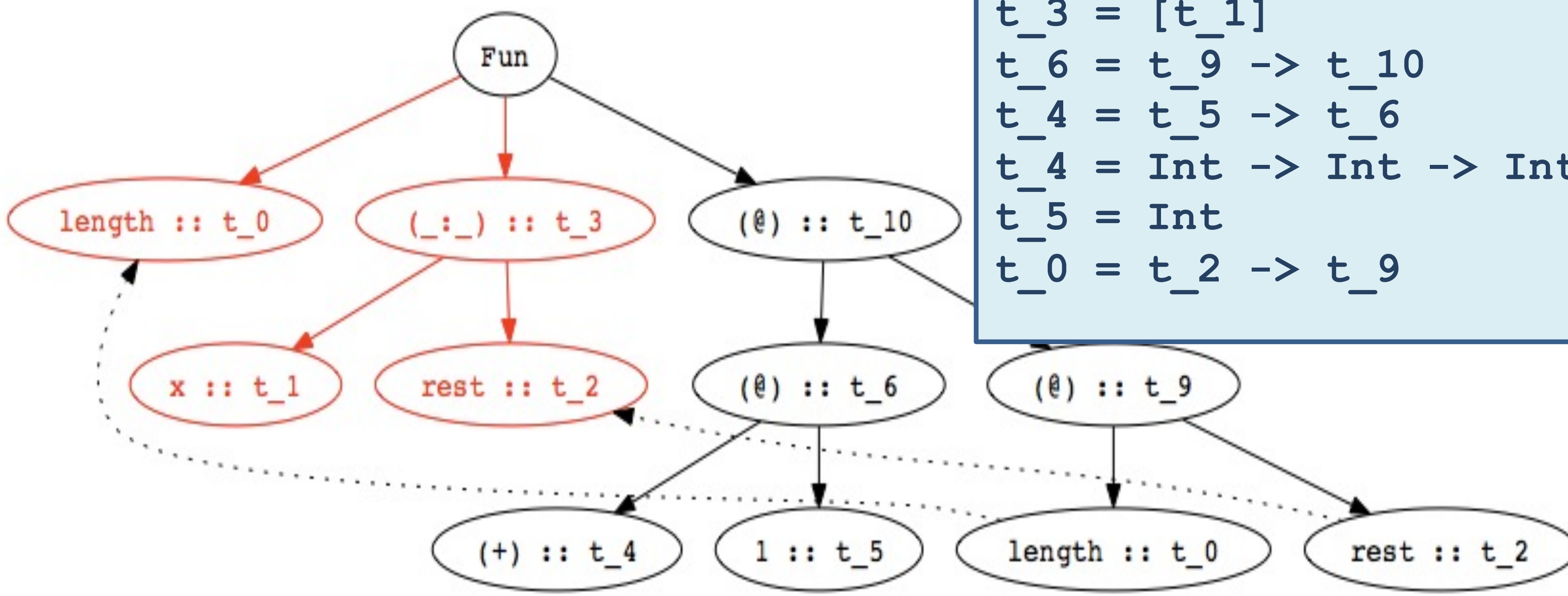
```
t_0 = t_3 -> t_10
t_3 = t_2
t_3 = [t_1]
t_6 = t_9 -> t_10
t_4 = t_5 -> t_6
t_4 = Int -> Int -> Int
t_5 = Int
t_0 = t_2 -> t_9
```



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 with overloading

- In presence of overloading (Type Classes), type inference infers a ***qualified type* Q => T**
  - **T** is a Hindley Milner type, inferred as seen before
  - **Q** is set of type class predicates, called a **constraint**
- Consider the example function:

```
example z xs =
   case xs of
      []      -> False
      (y:ys) -> y > z || (y==z && ys == [z])
```

  - Type **T** is a -> [a] -> Bool
  - Constraint **Q** is { Ord a, Eq a, Eq [a]}

Ord a	because	y>z
Eq a	because	y==z
Eq [a]	because	ys == [z]

Simplifying Type Constraints

- Constraint sets Q can be simplified:
  - Eliminate duplicates
    - **(Eq a, Eq a)** simplifies to **Eq a**
  - Use an **instance declaration**
    - If we have instance **Eq a => Eq [a]**,
      then **(Eq a, Eq [a])** simplifies to **Eq a**
  - Use a **class declaration**
    - If we have class **Eq a => Ord** a where ...,
      then **(Ord a, Eq a)** simplifies to **Ord a**
- Applying these rules,
  - **(Ord a, Eq a, Eq[a])** simplifies to **Ord a**

Type Inference with overloading

- Putting it all together:

```
example z xs =
   case xs of
     []      -> False
     (y:ys) -> y > z || (y==z && ys ==[z])
```

  – T = a -> [a] -> Bool
  – Q = (Ord a, Eq a, Eq [a])
  – Q simplifies to Ord a
  – `example :: Ord a => a -> [a] -> Bool`

Detecting Errors

- Errors are detected when predicates are known not to hold:

```
Prelude> 'a' + 1
 <interactive>:33:1: error:
    • No instance for (Num Char) arising from a use of ‘+’
    • In the expression: 1 + 'a'
      In an equation for ‘it’: it = 1 + 'a'
```

```
Prelude> (\x -> x)
 <interactive>:34:1: error:
    • No instance for (Show (p0 -> p0)) arising from a use of ‘print’
        (maybe you haven't applied a function to enough arguments?)
    • In a stmt of an interactive GHCi command: print it
```