Programming Language Recap

Mooly Sagiv

Languages

- Prolog
- Javascript
- Haskel
- Lua
- ~~Scala~~
- ~~Ruby~~

Concepts

- Syntax
  - Context free grammar
  - Ambiguous grammars
  - Syntax vs. semantics
- Static semantics
  - Scope rules
- Semantics
  - Small vs. big step
- Axiomatic semantics
- ~~Static Analysis~~
- Functional programming
  - Lambda calculus
  - Recursion
  - Higher order programming
  - Lazy vs. Eager evaluation
  - Pattern matching
  - Continuation
- Types
  - Type safety
  - Static vs. dynamic
  - Type checking vs. type inference
  - Most general type
  - Polymorphism
  - Type inference algorithm

Non Ambiguous Grammars for Arithmetic Expressions

Ambiguous grammar

1 $E \rightarrow E + E$	1 $E \rightarrow E + T$	1 $E \rightarrow E * T$
2 $E \rightarrow E * E$	2 $E \rightarrow T$	2 $E \rightarrow T$
3 $E \rightarrow id$	3 $T \rightarrow T * F$	3 $T \rightarrow F + T$
4 $E \rightarrow (E)$	4 $T \rightarrow F$	4 $T \rightarrow F$
	5 $F \rightarrow id$	5 $F \rightarrow id$
	6 $F \rightarrow (E)$	6 $F \rightarrow (E)$

Formal Syntax and Semantics of Programming Languages

Mooly Sagiv

Reference: Semantics with Applications

Chapter 2

H. Nielson and F. Nielson

http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html

Natural Semantics for While

axioms

$[\text{ass}_{ns}]\ \langle x := a, s\rangle \to s[x \mapsto \mathbf{A}[\![a]\!]s]$

$[\text{skip}_{ns}]\ \langle \textbf{skip}, s\rangle \to s$

rules

$$[\text{comp}_{ns}]\ \frac{\langle S_1, s\rangle \to s', \langle S_2, s'\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''}$$

$$[\text{if}^{tt}_{ns}]\ \frac{\langle S_1, s\rangle \to s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \to s'} \quad \text{if } \mathbf{B}[\![b]\!]s = tt$$

$$[\text{if}^{ff}_{ns}]\ \frac{\langle S_2, s\rangle \to s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \to s'} \quad \text{if } \mathbf{B}[\![b]\!]s = ff$$

Natural Semantics for While (More rules)

$$[\text{while}^{ff}_{ns}] \quad \frac{}{\langle \text{while } b \text{ do } S, s\rangle \rightarrow s} \quad \text{if } \mathbf{B}[\![b]\!]s=ff$$

$$[\text{while}^{tt}_{ns}] \quad \frac{\langle S , s\rangle \rightarrow s', \langle \text{while } b \text{ do } S, s'\rangle \rightarrow s''}{\langle \text{while } b \text{ do } S, s\rangle \rightarrow s''} \quad \text{if } \mathbf{B}[\![b]\!]s=tt$$

A Derivation Tree

- A “proof” that $<S, s> \to s'$
- The root of tree is $<S, s> \to s'$
- Leaves are instances of axioms
- Internal nodes rules
  - Immediate children match rule premises
- Simple Example

$$<\text{skip}; x := x +1, s_0> \to s_0[x \mapsto 1]>$$

comp_{ns}

$$<\text{skip}, s_0> \to s_0 \qquad < x := x +1, s_0> \to s_0[x \mapsto 1]>$$

An Example Derivation Tree

$<(x := x+1;\ y := x+1)\ ;\ \ z := y),\ s0> \to s0[x \mapsto 1][y \mapsto 2][z \mapsto 2]$

comp_{ns}

$<x := x+1;\ y := x+1,\ s0> \to s0[x \mapsto 1][y \mapsto 2]$

$<z := y, s0[x \mapsto 1][y \mapsto 2]> \to s0[x \mapsto 1][y \mapsto 2][z \mapsto 2]$

comp_{ns}

$<x := x+1;\ s0> \to s0[x \mapsto 1]$

$<y := x+1,\ s0[x \mapsto 1]> \to s0[x \mapsto 1][y \mapsto 2]$

ass_{ns}

ass_{ns}

Top Down Evaluation of Derivation Trees

- Given a program S and an input state s
- Find an output state s' such that
  $<S, s> \rightarrow s'$
- Start with the root and repeatedly apply rules until the axioms are reached
- Inspect different alternatives in order
- In While s' and the derivation tree is unique

The meaning of the program

- A proof tree
  - The root is labeled by <i, com> $\rightarrow$o
    - i is the input state
    - com is the abstract syntax tree of the program
    - o is the output state
  - Leafs axioms
  - Internal nodes are rules

Semantic Equivalence

- S_1 and S_2 are semantically equivalent if
  for all s and s'
  $<S_1, s> \to s'$ if and only if $<S_2, s> \to s'$
- Simple example
  "while b do S"
  is semantically equivalent to:
  "if b then (S ; while b do S) else skip"

Properties of Natural Semantics

- Equivalence of program constructs
  - "skip ; skip" is semantically equivalent to "skip"
  - "((S_1 ; S_2) ; S_3)" is semantically equivalent to "(S_1 ;(S_2 ; S_3))"
  - "(x := 5 ; y := x * 8)" is semantically equivalent to "(x :=5; y := 40)"
- Deterministic
  - If $<S, s> \to s_1$ and $<S, s> \to s_2$ then $s_1=s_2$

Deterministic Semantics for While

- If $<S, s> \to s_1$ and $<S, s> \to s_2$ then $s_1=s_2$
- The proof uses induction on the shape of derivation trees
  - Prove that the property holds for all simple derivation trees by showing it holds for axioms
  - Prove that the property holds for all composite trees:
    - For each rule assume that the property holds for its premises (induction hypothesis) and prove it holds for the conclusion of the rule

Structural Semantics for While

axioms

$[\text{ass}_{\text{sos}}]\ \langle x := a, s\rangle \Rightarrow s[x \mapsto \mathbf{A}[\![a]\!]s]$

$[\text{skip}_{\text{sos}}]\ \langle \mathbf{skip}, s\rangle \Rightarrow s$

rules

$$[\text{comp}^1_{\text{sos}}]\ \frac{\langle S_1 , s\rangle \Rightarrow \langle S'_1, s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S'_1; S_2, s'\rangle}$$

$$[\text{comp}^2_{\text{sos}}]\ \frac{\langle S_1 , s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

Structural Semantics for While
if construct

$[\text{if}^{tt}_{sos}]$ $\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \Rightarrow \langle S_1, s\rangle$ if $\mathbf{B}[\![b]\!]s=tt$

$[\text{if}^{ff}_{os}]$ $\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \Rightarrow \langle S_2, s\rangle$ if $\mathbf{B}[\![b]\!]s=ff$

Structural Semantics for While
while construct

$[\text{while}_{sos}]$ $\langle \text{while } b \text{ do } S, s\rangle \Rightarrow$
$\langle \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}, s\rangle$

Derivation Sequences

- A finite derivation sequence starting at <S, s> $\gamma_0, \gamma_1, \gamma_2 \ldots, \gamma_k$ such that
  - γ_0=<S, s>
  - $\gamma_i \Rightarrow \gamma_{i+1}$
  - γ_k is either stuck configuration or a final state
- An infinite derivation sequence starting at <S, s>
  $\gamma_0, \gamma_1, \gamma_2 \ldots$ such that
  - γ_0=<S, s>
  - $\gamma_i \Rightarrow \gamma_{i+1}$
- $\gamma_0 \Rightarrow^i \gamma_i$ in i steps
- $\gamma_0 \Rightarrow^* \gamma_i$ in finite number of steps
- For each step there is a derivation tree

SOS vs. Natural Semantics

- Natural semantics is more intuitive
  - Simulates structural induction
- SOS allows to express more low level construct
  - Exposes implementation details
    - Program locastion
    - Storage

Untyped Lambda Calculus

t ::=	terms
x	variable
λ x. t	abstraction
t t	application

Terms can be represented as abstract syntax trees

Syntactic Conventions

- Applications associates to left

$$e_1\ e_2\ e_3 \equiv (e_1\ e_2)\ e_3$$

- The body of abstraction extends as far as possible
  - $\lambda x.\ \lambda y.\ x\ y\ x \equiv \lambda x.\ (\lambda y.\ (x\ y)\ x)$

Free vs. Bound Variables

- An occurrence of x is free in a term t if it is not in the body on an abstraction λx. t
  - otherwise it is bound
  - λx is a binder
- Examples
  - λz. λx. λy. x (y z)
  - (λx. x) x
- Terms w/o free variables are combinators
  - Identify function: id = λ x. x

Operational Semantics

$(\lambda\ x.\ t_{12})\ t_2 \to [x \mapsto t_2]\ t_{12}$ (β-reduction)

FV: $t \to P(Var)$ is the set free variables of t

$FV(x) = \{x\}$

$FV(\lambda\ x.\ t) = FV(t) - \{x\}$

$FV\ (t_1\ t_2) = FV(t_1) \cup FV(t_2)$

$[x \mapsto s]\ x = s$

$[x \mapsto s]\ y = y$ if $y \neq x$

$[x \mapsto s]\ (\lambda y.\ t_1) = \lambda y.\ [x \mapsto s]\ t_1$ if $y \neq x$ and $y \notin FV(s)$

$[x \mapsto s]\ (t_1\ t_2) = ([x \mapsto s]\ t_1)\ ([x \mapsto s]\ t_2)$

Operational Semantics

$(\lambda\ x.\ t_{12})\ t_2 \rightarrow [x \mapsto t_2]\ t_{12}$ (β-reduction)

redex

$(\lambda\ x.\ x)\ y \rightarrow\ y$

$(\lambda\ x.\ x\ (\lambda\ x.\ x)\ )\ (u\ r) \rightarrow\ u\ r\ (\lambda\ x.x)$

$(\lambda\ x\ (\lambda w.\ x\ w))\ (y\ z) \rightarrow\ \lambda w.\ y\ z\ w$

Lambda Calculus vs. JavaScript

$(\lambda\ x.\ x)\ y$

```
(function (x) {return x;}) y
```

Spring 2014

Introduction to Haskell

Shachar Itzhaky & Mooly Sagiv

(original slides by Kathleen Fisher & John Mitchell)

Example: Differentiate

- The differential operator

$$f'(x) = \lim_{h \to 0} \big(f(x+h)-f(x)\big)/h$$

- In Haskell:

```
diff f = f_prime
     where
       f_prime x = (f (x + h) - f x) / h
       h = 0.0001
```

- `diff :: (float -> float) -> (float -> float)`
- `(diff square) 0 = 0.0001`
- `(diff square) 0.0001 = 0.0003`
- `(diff (diff square)) 0 = 2`

Pattern Matching

- Patterns can be used in place of variable names

  <pat> ::= <var> | <tuple> | <cons> | <record> …
- Value declarations
  - General form: <pat> = <exp>
  - In global declarations

```
myTuple = ("Flitwick", "Snape")
(x,y)   = myTuple
myList  = [1, 2, 3, 4]
z:zs    = myList
```

  - In local declarations

```
let (x,y) = (2, "Snape") in x * 4
```

Pattern Matching

- Explicit case expression

```
myTuple = ("Flitwick", "Snape")
v = case myTuple of
      (x, "Snape")     ->  x ++ "?"
      ("Flitwick", y)  ->  y ++ "!"
      _                ->  "?!"
```

Map Function on Lists

- Apply function to every element of list

```
map f [] = []
map f (x:xs) = f x : map f xs
```

```
map (\x -> x+1) [1,2,3]          [2,3,4]
```

- Compare to Lisp

```
(define map
    (lambda (f  xs)
      (if   (eq? xs ())   ()
         (cons (f   (car xs))   (map f   (cdr xs)))
   )))
```

More Functions on Lists

- Append lists

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

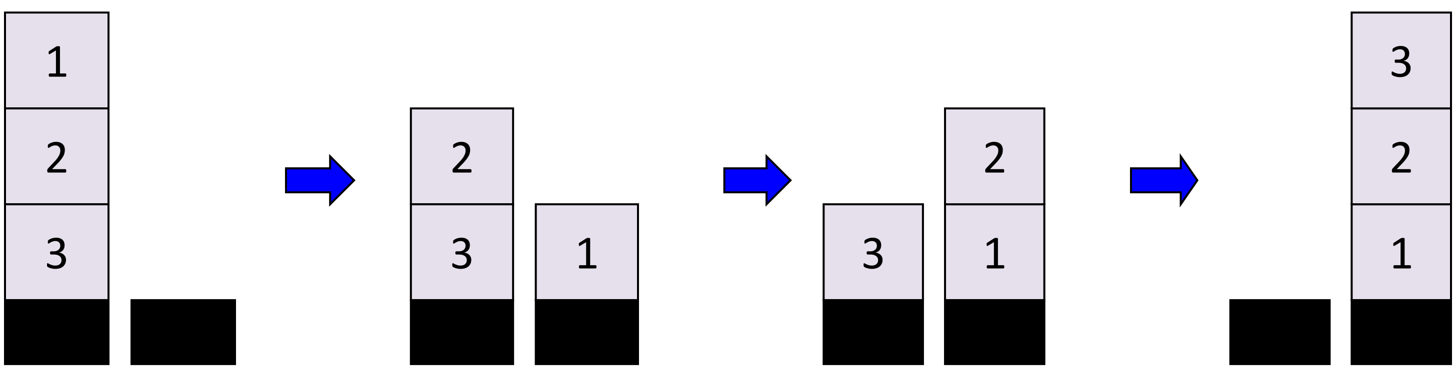
- Reverse a list

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

- Questions
  - How efficient is reverse?
  - Can it be done with only one pass through list?

More Efficient Reverse

```
reverse xs =
    let rev ( [], accum ) = accum
         rev ( y:ys, accum ) = rev ( ys, y:accum )
    in rev ( xs, [] )
```



Datatype Declarations

- Examples

```
data Color = Red | Yellow | Blue
```

elements are Red, Yellow, Blue

```
data Atom = Atom String | Number Int
```

elements are Atom "A", Atom "B", ..., Number 0, ...

```
data AtomList = Nil | Cons Atom AtomList
```

elements are Nil, Cons (Atom "A") Nil, ...

Cons (Number 2) (Cons (Atom "Bill")) Nil, ...

- General form

```
data <name> = <clause> | … | <clause>
<clause> ::= <constructor> | <constructor> <type>
```

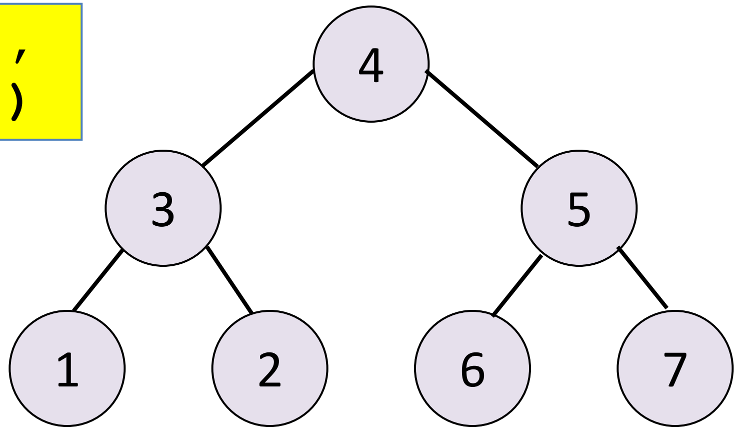
– Type name and constructors must be Capitalized

Datatypes and Pattern Matching

- Recursively defined data structure

```
data Tree = Leaf Int | Node (Int, Tree, Tree)
```

```
Node(4, Node(3, Leaf 1, Leaf 2),
        Node(5, Leaf 6, Leaf 7))
```



- Recursive function

```
sum (Leaf n) = n
sum (Node(n,t1,t2)) = n + sum(t1) + sum(t2)
```

Example: Evaluating Expressions

- Define datatype of expressions

```
data Exp = Var Int | Const Int | Plus (Exp, Exp)
```

write (x+3)+ y as Plus(Plus(Var 1, Const 3), Var 2)

- Evaluation function

```
ev(Var n)  = Var n
ev(Const n)  = Const n
ev(Plus(e1,e2))  =  …
```

- Examples

```
ev(Plus(Const 3, Const 2))        ⇒        Const 5
```

```
ev(Plus(Var 1, Plus(Const 2, Const 3)))   ⇒
                            Plus(Var 1, Const 5)
```

Use the Case Expression

- Datatype

```
data Exp = Var Int | Const Int | Plus (Exp, Exp)
```

- Case expression

```
case e of
     Var n ->  …
     Const n -> …
     Plus(e1,e2) -> …
```

Indentation matters in case statements in Haskell

Laziness

- Haskell is a **lazy** language
- Functions and data constructors don't evaluate their arguments until they need them

```
cond :: Bool -> a -> a -> a
cond True  t e = t
cond False t e = e
```

- Programmers can write control-flow operators that have to be built-in in eager languages

Short-circuiting "or"

```
(||) :: Bool -> Bool -> Bool
True  || x = True
False || x = x
```

Using Laziness

```
isSubString :: String -> String -> Bool
x `isSubString` s = or [ x `isPrefixOf` t
                       | t <- suffixes s ]
```

```
suffixes:: String -> [String]
-- All suffixes of s
suffixes[]      = [[]]
suffixes(x:xs) = (x:xs) : suffixes xs
```

type String = [Char]

```
or :: [Bool] -> Bool
-- (or bs) returns True if any of the bs is True
or []       = False
or (b:bs) = b || or bs
```

A Lazy Paradigm

- Generate all solutions (an enormous tree)
- Walk the tree to find the solution you want

```
nextMove :: Board -> Move
nextMove b = selectMove allMoves
  where
    allMoves = allMovesFrom b
```

A gigantic (perhaps infinite) tree of possible moves

Benefits of Lazy Evaluation

- Define streams:
  `main = take 100 [1 .. ]`

```
deriv f x = lim [(f (x + h) - f x) / h | h <- [1/2^n | n <- [1..]]]
            where lim (a:b:lst) = if abs(a/b-1) < eps then b
                                  else lim (b: lst)
                  eps = 1.0 e-6
```

- Lower asymptotic complexity
- Language extensibility
  - Domain specific languages
- But some costs

Core Haskell

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

Functional Programming Languages

PL	types	evaluation	Side-effect
Scheme Racket	Dynamically typed	Eager	yes
ML OCAML F#	Polymorphic strongly typed	Eager	References
Haskell	Polymorphic strongly typed	Lazy	None

Types and Type Inference

Mooly Sagiv

Slides by Kathleen Fisher and John Mitchell

Reading: “Concepts in Programming Languages”,
Revised Chapter 6 - handout on the course homepage

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);
```

Type Safety

- Type safe programming languages protect its own abstractions
- Type safe programs cannot go wrong
- No run-time errors
- But exceptions are fine
- The small step semantics cannot get stuck
- Type safety is proven at language design time

Relative Type-Safety of Languages

- Not safe: BCPL family, including C and C++
  - Casts, unions, 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
    - Hard to make languages with explicit deallocation of memory 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

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; };
```

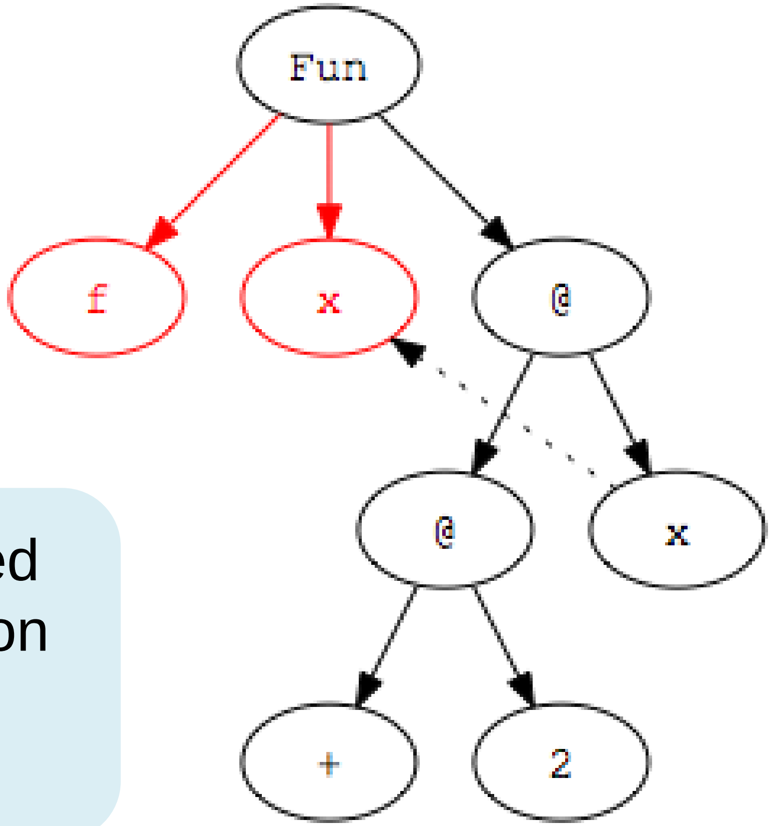
  - Examine code without type information
  - Infer the most general types that could have been declared

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

Step 1: Parse Program

- Parse program text to construct parse tree

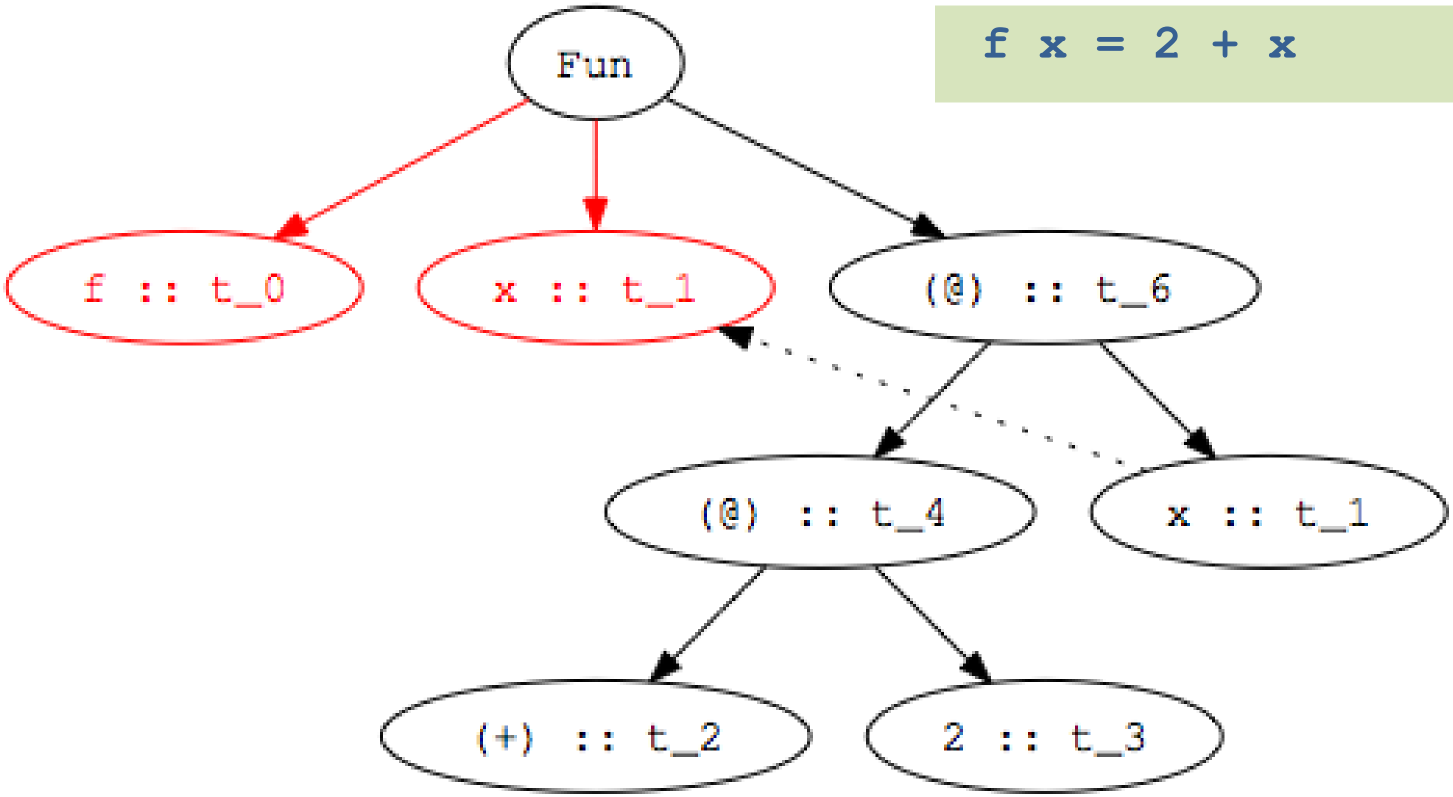
```
f x = 2 + x
```



Infix operators are converted to Curied function application during parsing:

2 + x ➔ (+) 2 x

Step 2: Assign type variables to nodes

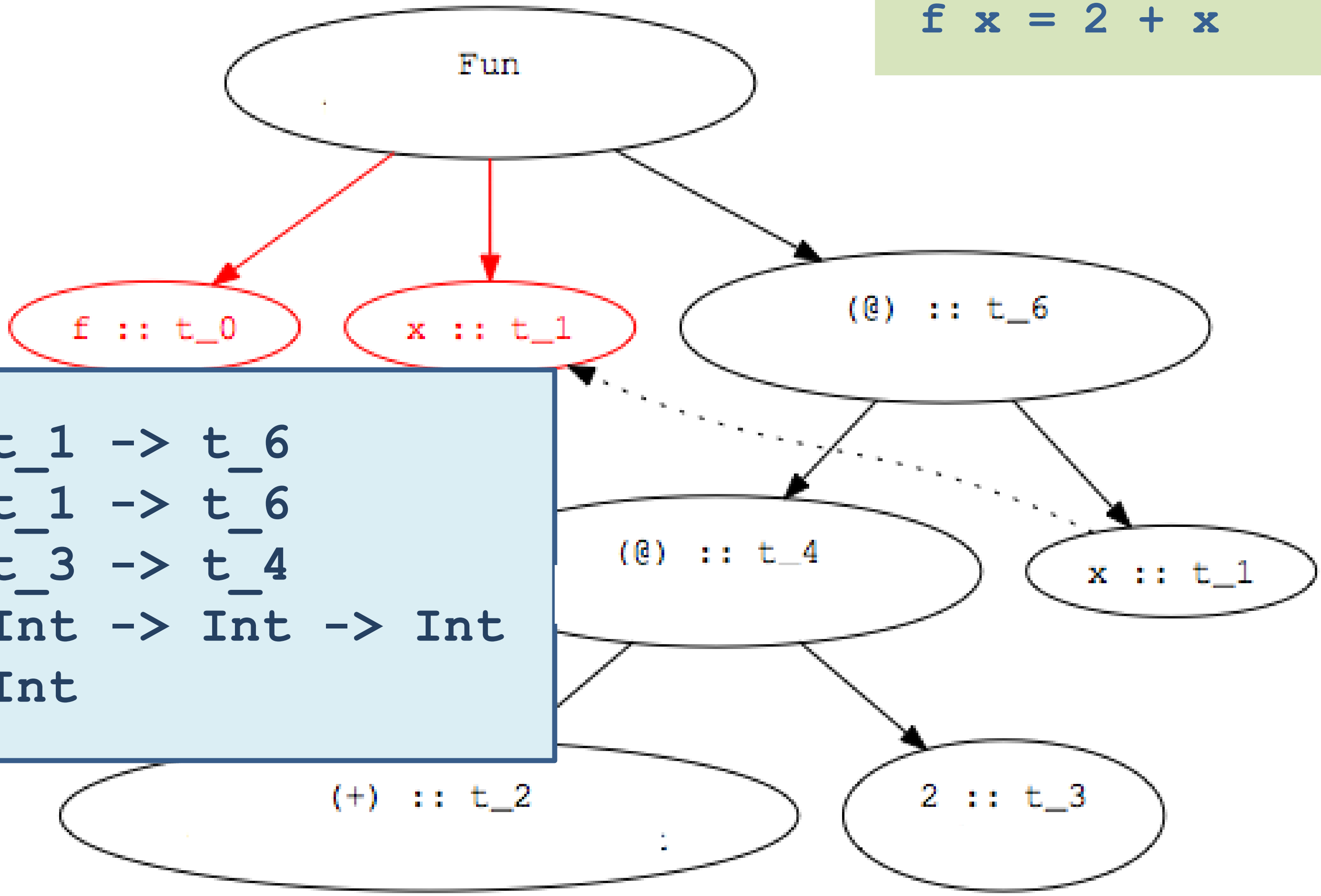



```
f x = 2 + x
```

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
```

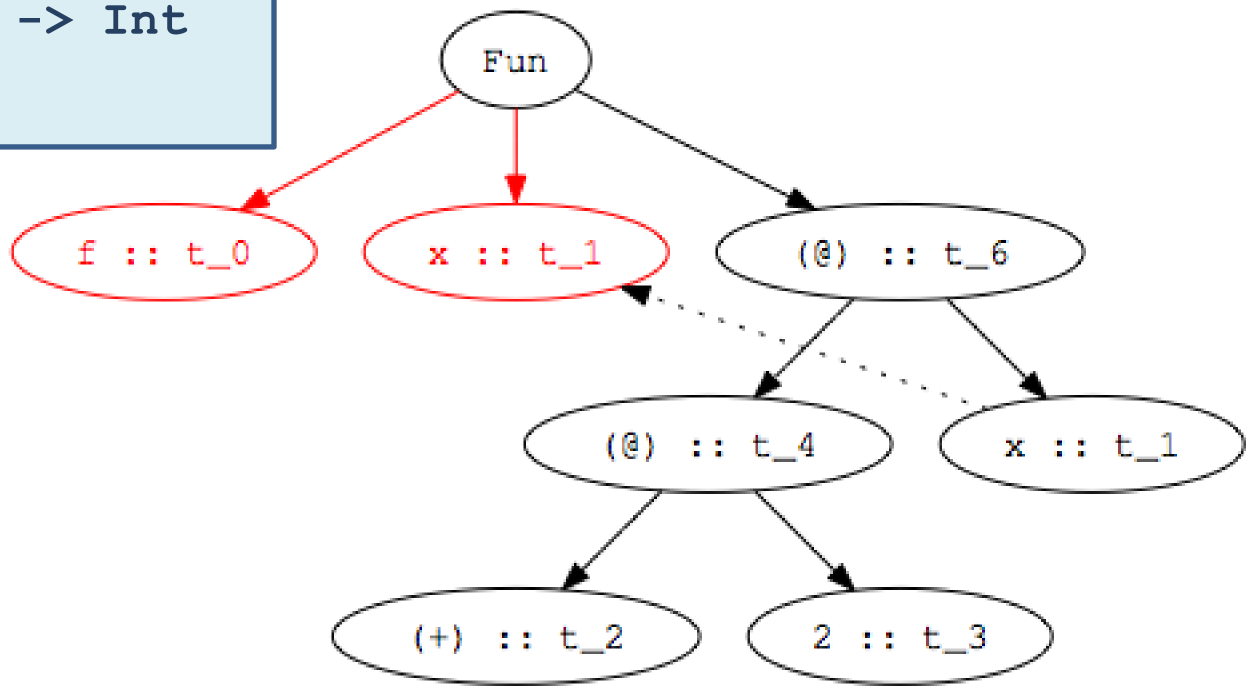
```
t_1 = Int
t_6 = Int
```

```
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
```



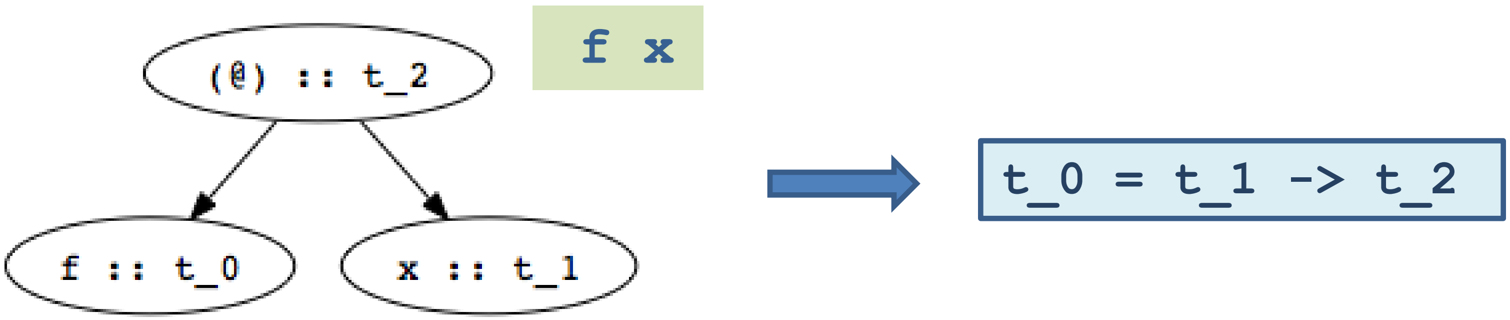
Unification

- Given two type terms t_1, t_2
- Compute the most general unifier of t_1 and t_2
  - A mapping m from type variables to typed terms such that
    - t_1 {m } == t_2 {m}
    - Every other unifier is a refinement of m
- Example
  mgu(`t_3 -> t_4, Int -> (Int -> Int)=`
  `[t_3 ↦ Int, t_4 ↦Int -> Int]=`

Type Inference Algorithm

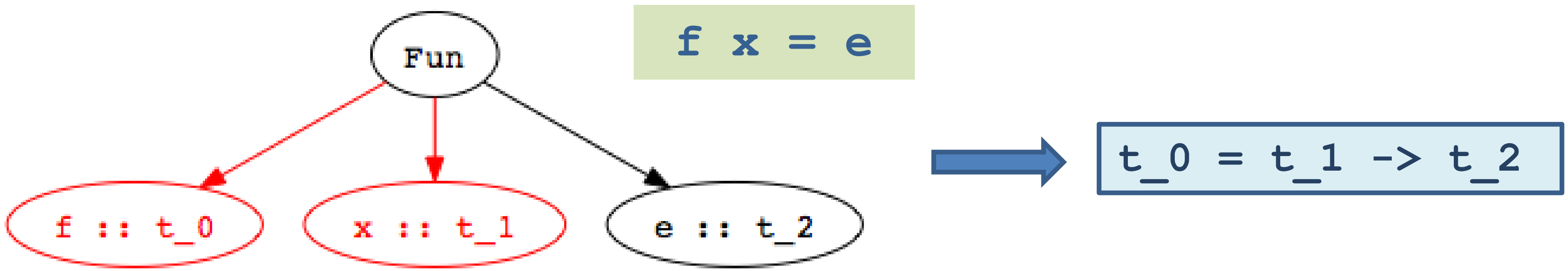
- Parse program to build parse tree
- Assign type variables to nodes in tree
- Generate constraints:
  - From environment: literals (2), built-in operators (+), known functions (tail)
  - From form of parse tree: e.g., application and abstraction nodes
- Solve constraints using *unification*
- Determine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

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



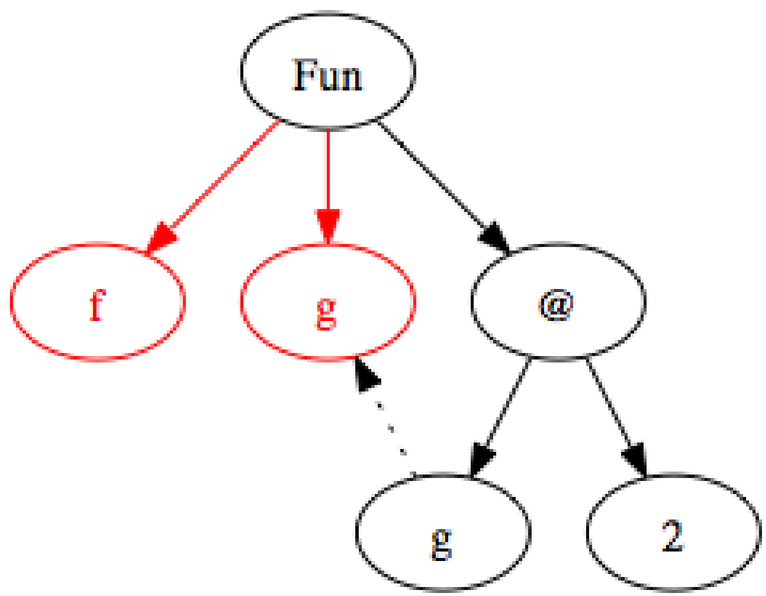
- Function declaration:
  - Type of f (t_0 in figure) must domain $\rightarrow$ range
  - Domain is type of abstracted variable x (t_1 in fig)
  - Range is type of function body e (t_2 in fig)
  - Constraint: t_0 = t_1 -> t_2

Inferring Polymorphic Types

- Example:

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

- Step 1:
  Build Parse Tree

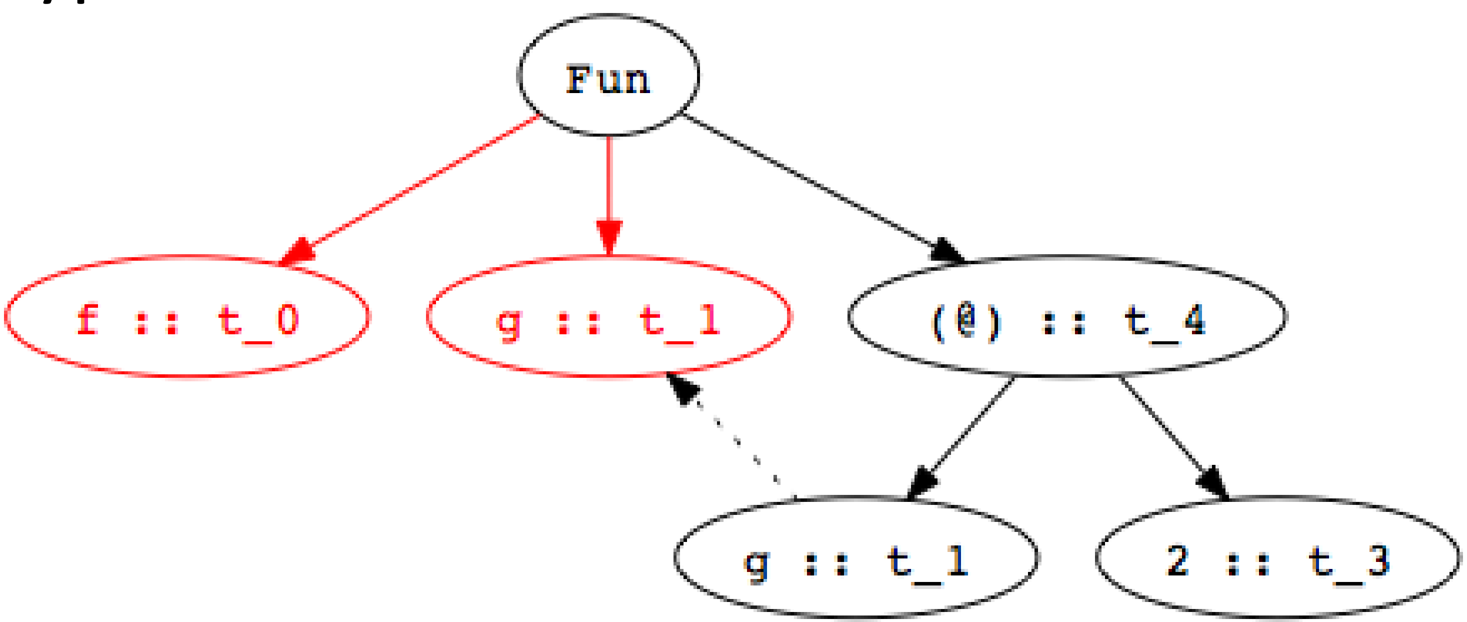


Inferring Polymorphic Types

- Example:

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

- Step 2:
  Assign type variables



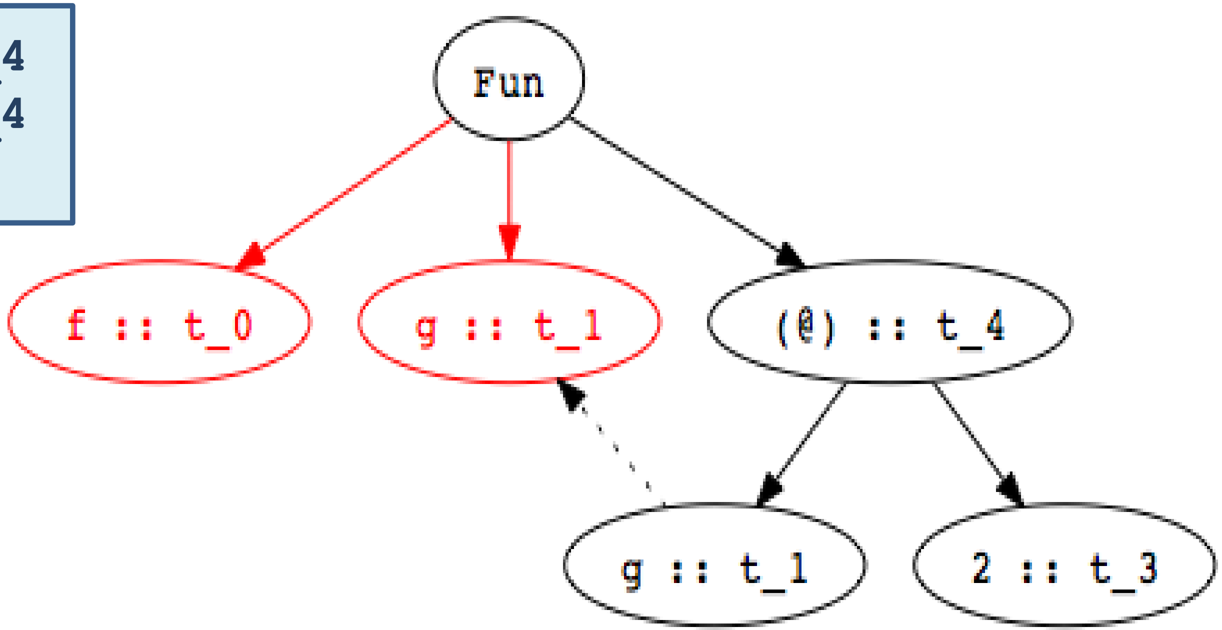
Inferring Polymorphic Types

- 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

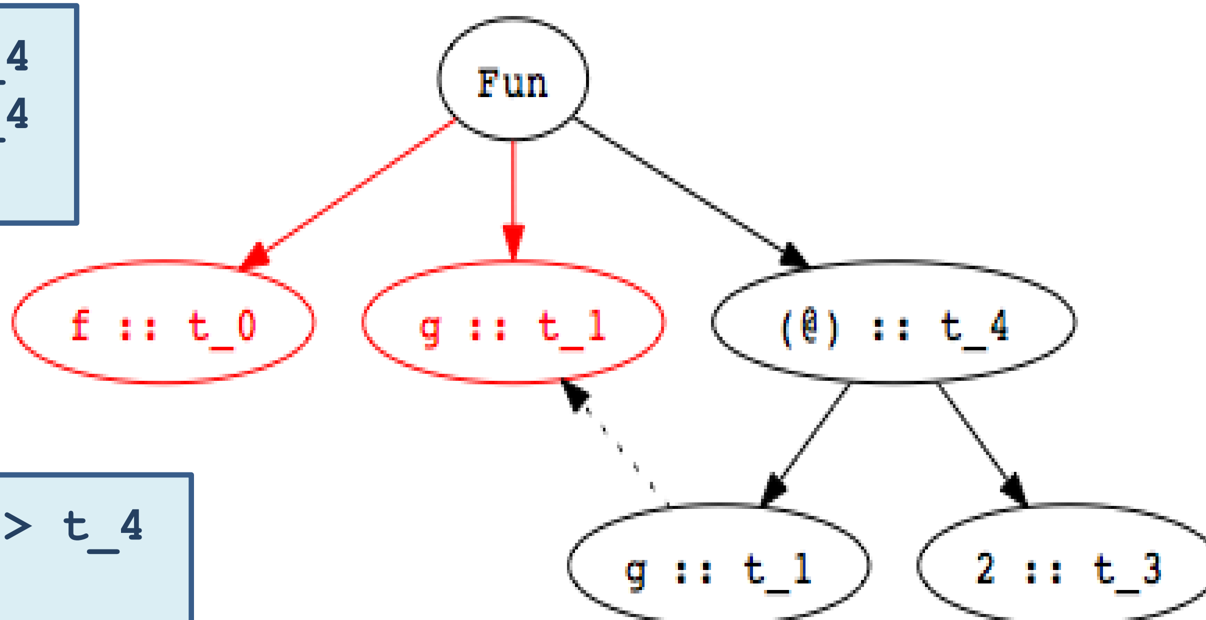
- Example:

```
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
```



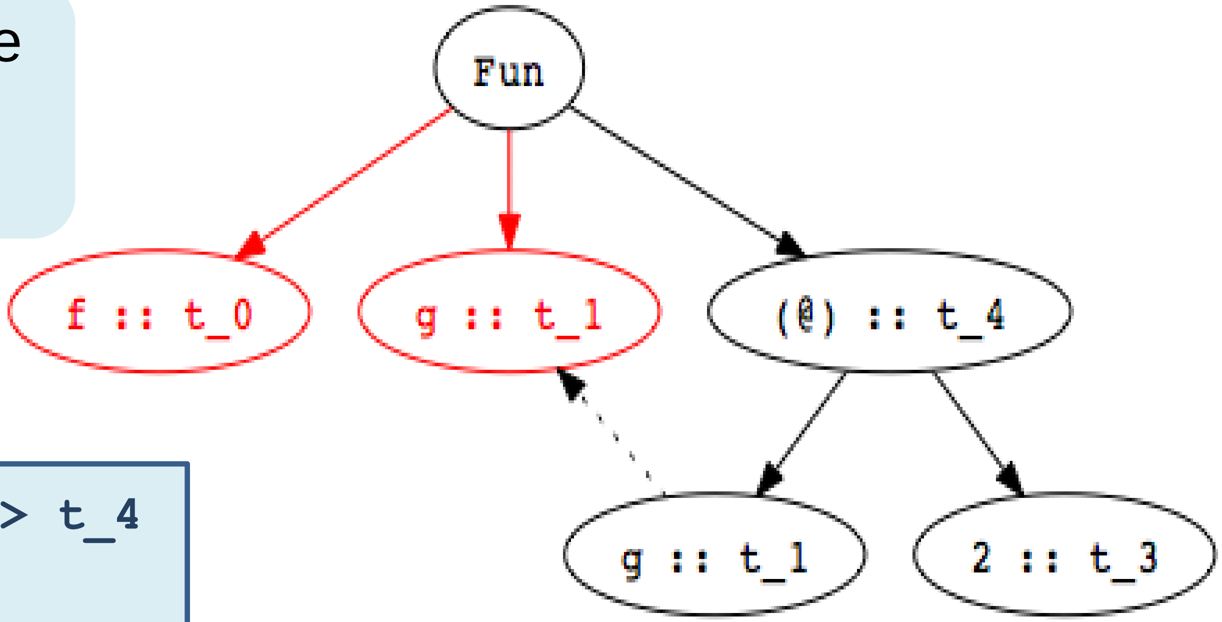
Inferring Polymorphic Types

- 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

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

JavaScript

John Mitchell

Adapted by Mooly Sagiv

Closures

- Return a function from function call

```
function f(x) {
    var y = x;
    return function (z){y += z; return y;}
 }
 var h = f(5);
 h(3);
```

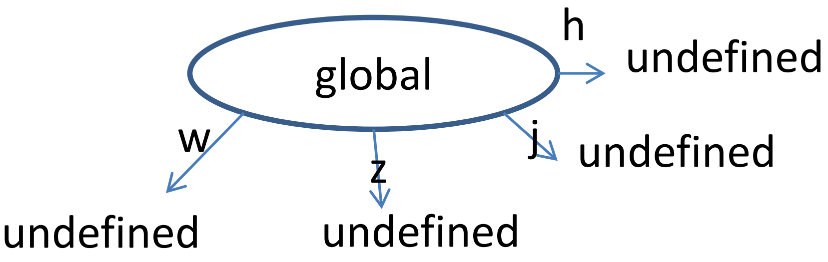
- Can use this idea to define objects with “private” fields

```
uniqueId function () {
  if (!argument.calle.id) arguments.calee.id=0;
  return arguments.callee.id++;
};
```

– Can implement breakpoints

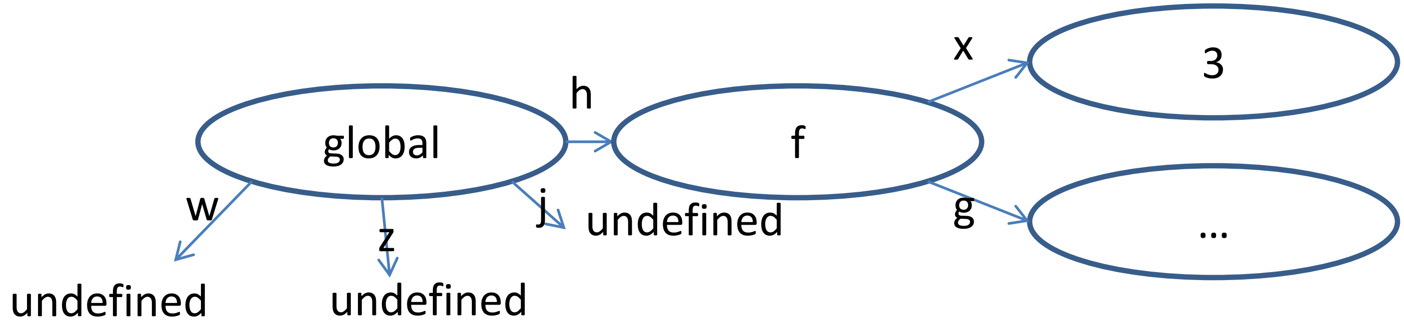
Implementing Closures

```
function f(x) {
  function g(y) { return x + y; };
  return g ;
 }
var h = f(3);
var j  = f(4);
var z = h(5);
var w = j(7);
```



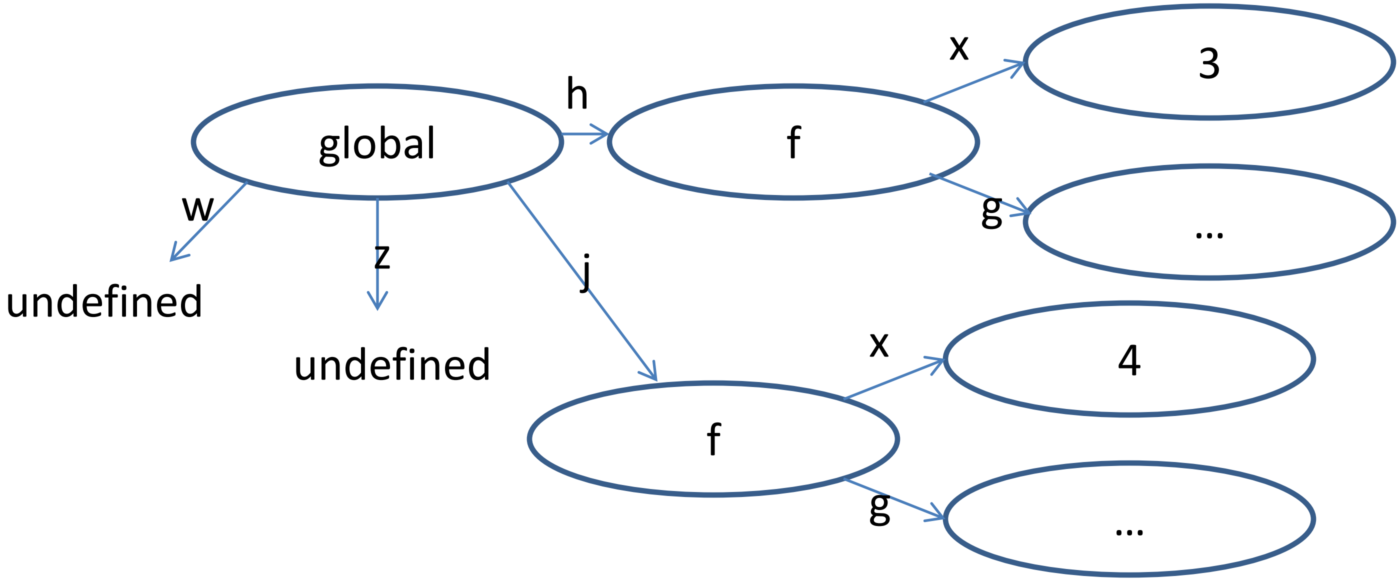
Implementing Closures(1)

```
function f(x) {
  function g(y) { return x + y; };
  return g ;
 }
var h = f(3);
var j  = f(4);
var z = h(5);
var w = j(7);
```



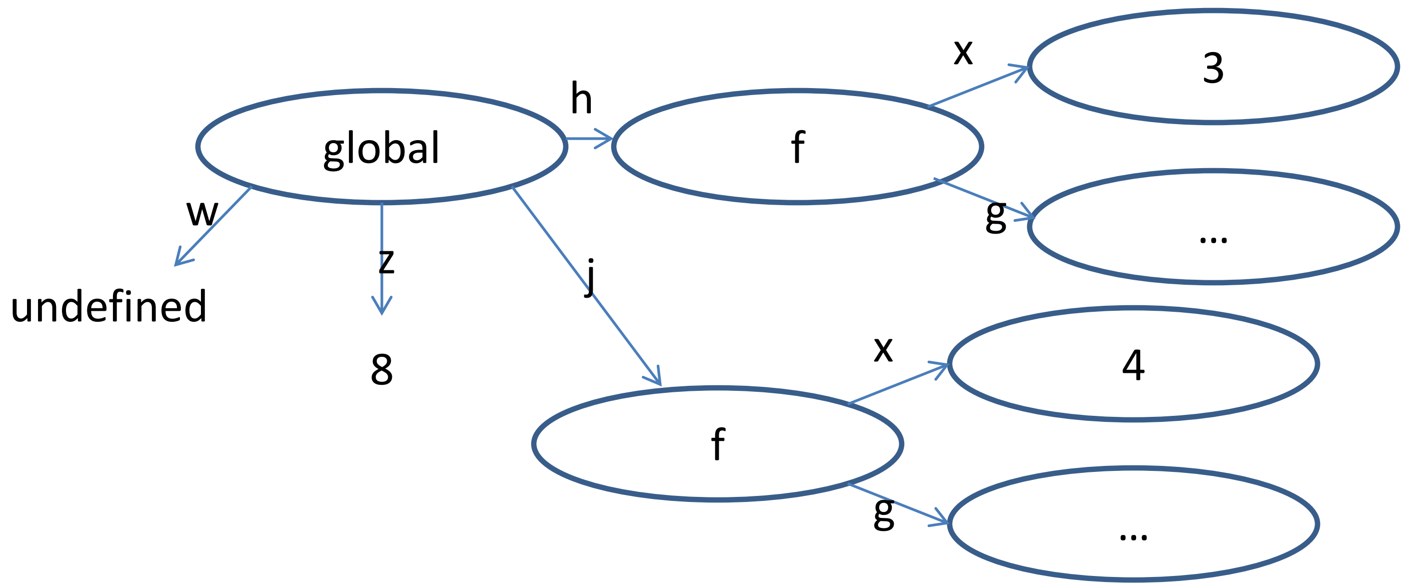
Implementing Closures(2)

```
function f(x) {
  function g(y) { return x + y; };
  return g ;
 }
var h = f(3);
var j  = f(4);
var z = h(5);
var w = j(7);
```



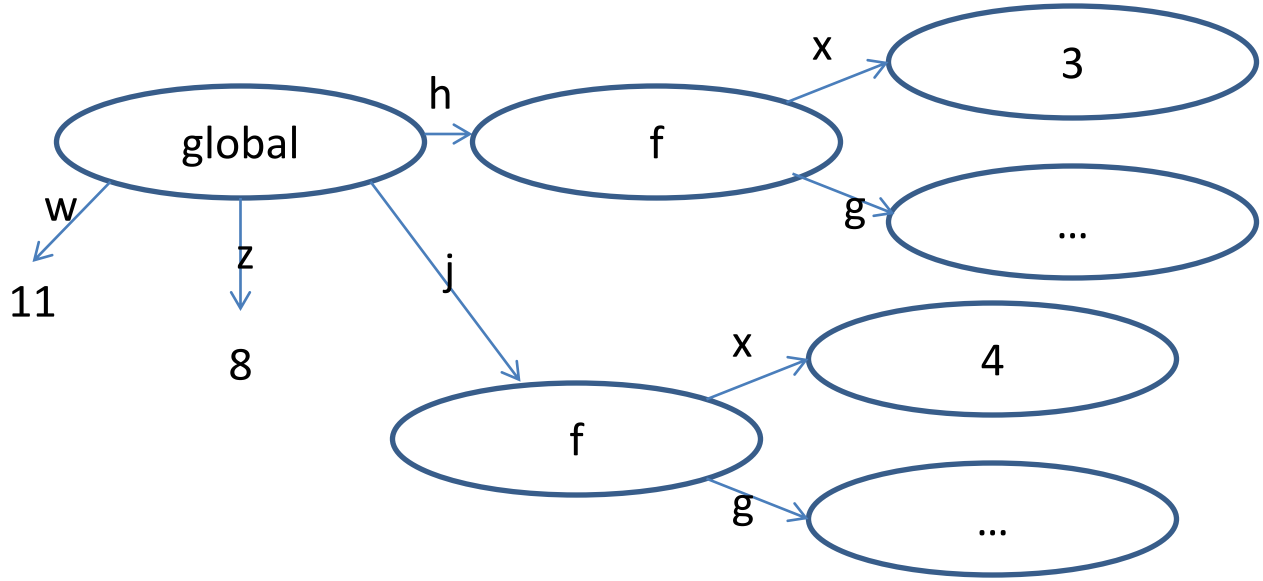
Implementing Closures(3)

```
function f(x) {
  function g(y) { return x + y; };
  return g ;
}
var h = f(3);
var j  = f(4);
var z = h(5);
var w = j(7);
```



Implementing Closures(4)

```
function f(x) {
  function g(y) { return x + y; };
  return g ;
}
var h = f(3);
var j  = f(4);
var z = h(5);
var w = j(7);
h= null;
```



Garbage collection

- Automatic reclamation of unused memory
  - Navigator 2: per page memory management
    - Reclaim memory when browser changes page
  - Navigator 3: reference counting
    - Each memory region has associated count
    - Count modified when pointers are changed
    - Reclaim memory when count reaches zero
  - Navigator 4: mark-and-sweep, or equivalent
    - Garbage collector marks reachable memory
    - Sweep and reclaim unreachable memory

Reference http://www.unix.org.ua/orelly/web/jscript/ch11_07.html

Discuss garbage collection in connection with memory management