Types and Type Inference

Edward Z. Yang

What is a type?

Hindley-Milner type inference
and polymorphism

What is a type?

What is a type?

`True :: Bool`

What is a type?

expr :: type

What is a type?

expr :: $\boxed{\text{Bool}}$ $\boxed{\text{Int} \to \text{Bool}}$ $\boxed{\text{Int}}$

What is a type?

$$
\begin{aligned}
\tau \; ::= \;& \mathrm{Int} \\
\mid \;& \mathrm{Bool} \\
\mid \;& \tau_1 \rightarrow \tau_2 \\
\mid \;& \ldots
\end{aligned}
$$

What is a type ... really?

It's a specification!

A TYPE IS: A Way to Prevent Errors

```
print(100 + "bob")
```

A TYPE IS: A Way to Prevent Errors

```
function apply(f, x) {
    return f(x);
}
```

A TYPE IS: A Way to Prevent Errors

The world's MOST POPULAR lightweight formal method!

A TYPE IS: A method of program organization

2 degrees Farenheit

2 degrees Celsius

A TYPE IS: A method of program organization

```
-- This function takes two integers
-- and returns their sum.
plus :: Int -> Int -> Int
plus a b = a+b
```

A TYPE IS: A method of program organization

```
-- This function takes a function
-- in its first argument and a value
-- in its second argument.
apply :: (a -> b) -> a -> b
apply f x = f x
```

A TYPE IS: A method of program organization

```
data Set k

empty :: Set k
insert :: k → Set k → Set k
delete :: k → Set k → Set k
member :: k → Set k → Bool
```

A TYPE IS: A Hint to the Compiler

x = record["key"]

Compilers use types for their memory layout. The way they see types, from C++, is it's just a memory layout. The optimization, on the hash table, is that you get to acces data given memory layout for the data structure.

A TYPE IS: A Hint to the Compiler

$x = $ hashTableLookup(record, "key")

A TYPE IS: A Hint to the Compiler

$$x = *(record + keyOffset)$$

A TYPE IS:

The central organizing principle of the theory of programming languages.

—Bob Harper

The transition here is not great: the idea is to talk about the different choices languages can make: dynamically/statically typed and strongly/weakly typed.

Types $\rightarrow$ Type Errors

Type Errors are language dependent

size 10 array

arr[200]

Type Errors are language dependent

size 10 array

arr[200]

Segfault

C/C++

Type Errors are language dependent

size 10 array

arr[200]

Out of bounds!

Haskell / Java

Type Errors are language dependent

null pointer

arr[200]

Segfault

C/C++

Type Errors are language dependent

null pointer

arr[200]

Null pointer derefence

Java

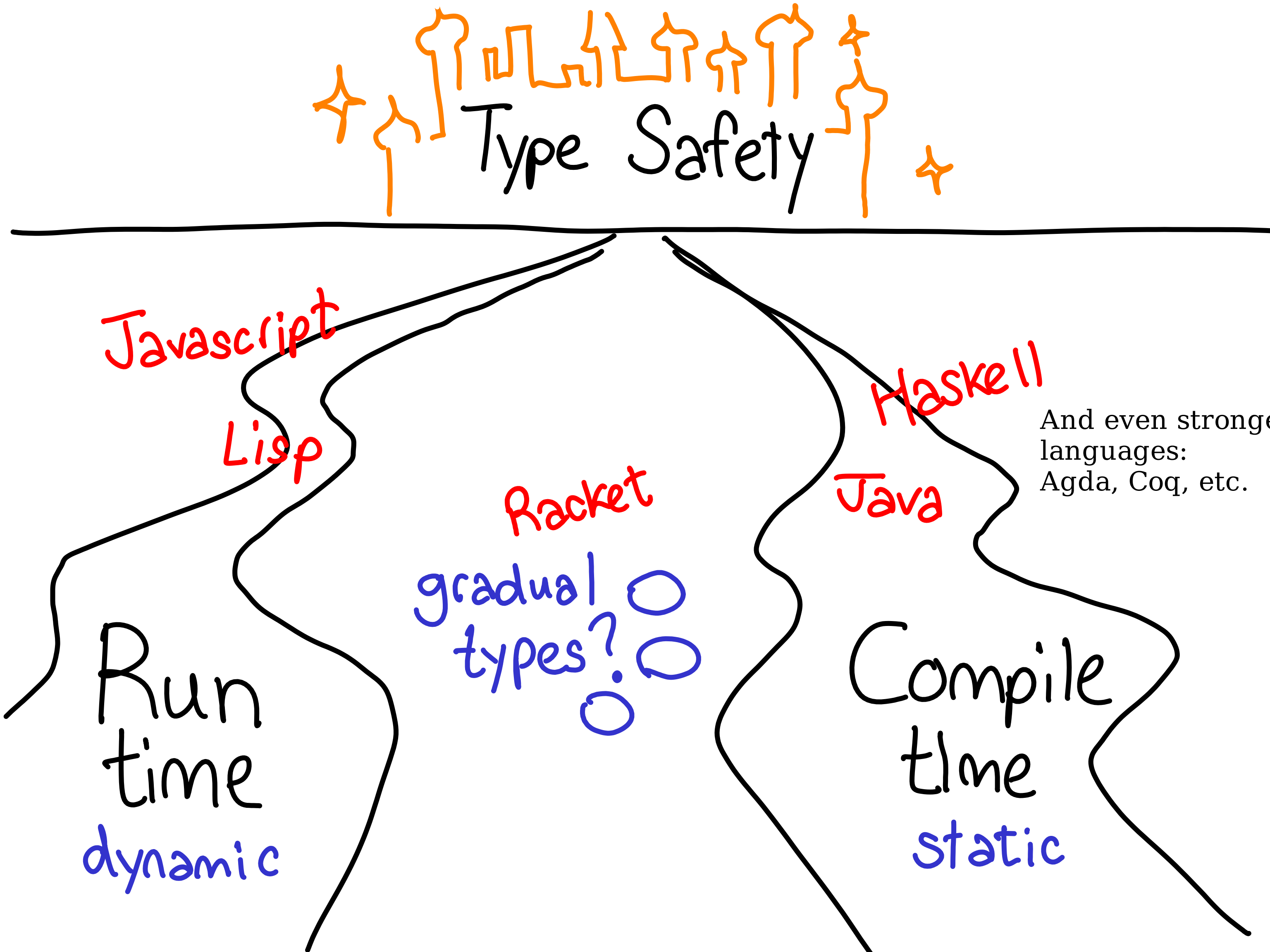
Type Errors are language dependent

maybe type

~~arr ! 200~~

Cannot unify Maybe Array
with expected Array

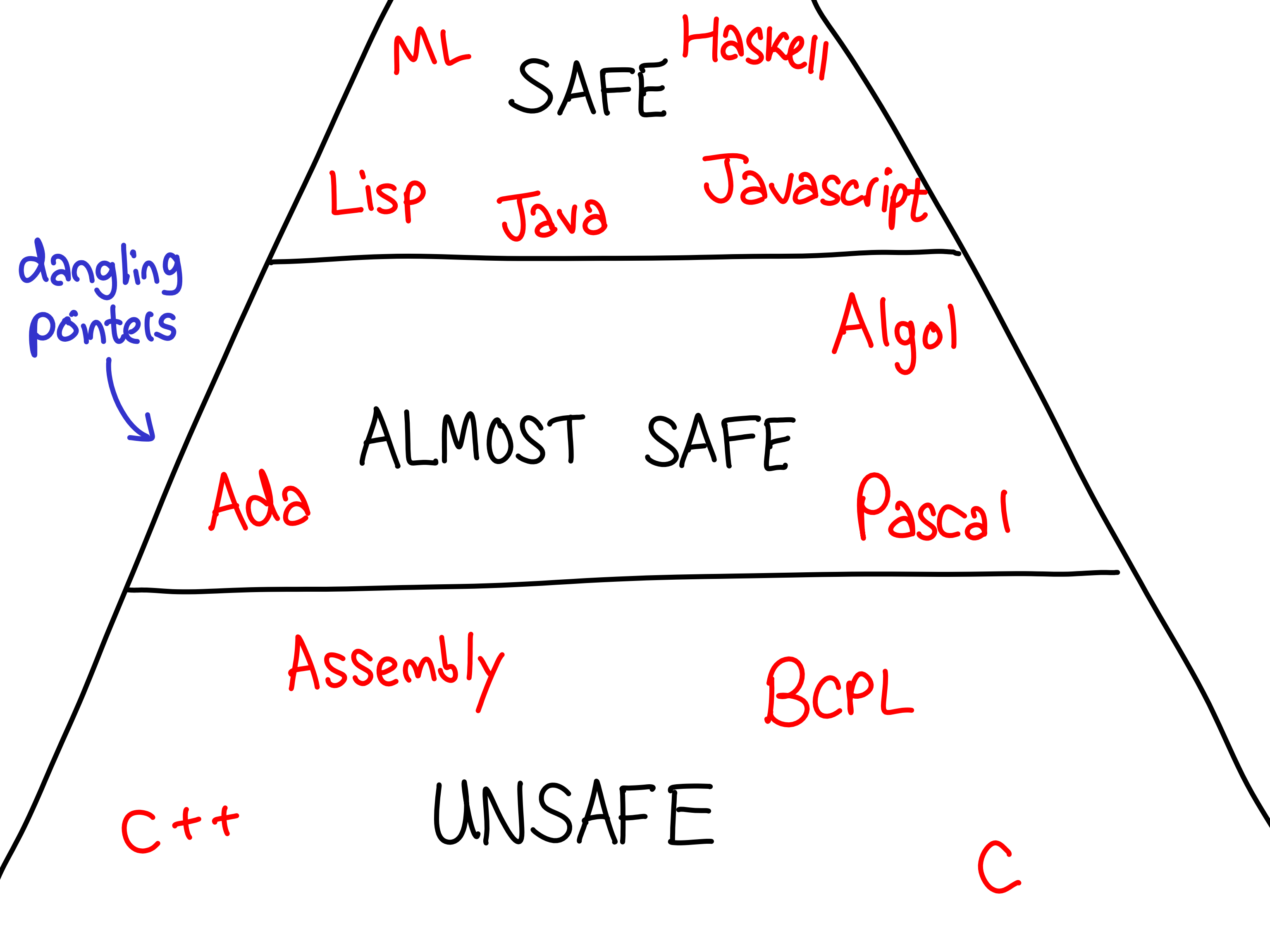
Haskell


Type Safety
Javascript
Lisp
Racket
gradual types?
Haskell
Java
And even stronge
languages:
Agda, Coq, etc.
Run time
dynamic
Compile time
static

expressivity versus information

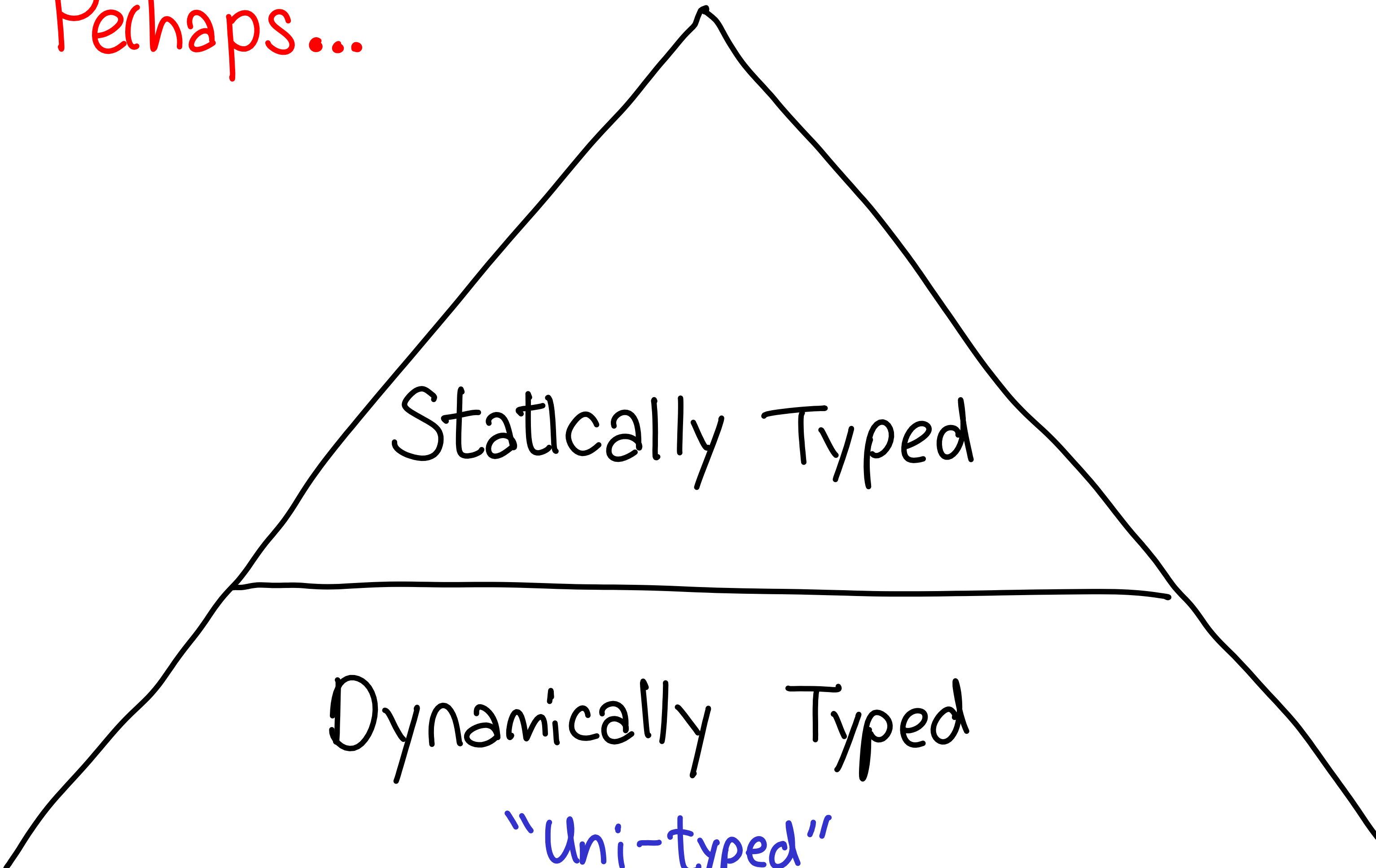
```
function f(x) {
    return x<10 ? x : x();
}
```

(dependent types?)



ML
SAFE
Haskell
Lisp
Java
Javascript
dangling pointers
Algol
ALMOST SAFE
Ada
Pascal
Assembly
BCPL
UNSAFE
C++
C

Perhaps...



(pause)

Hindley-Milner type inference

The Hindley-Milner type inference algorithm is one of the most elegant algorithms for type checking in the PL literature (too elegant, some would say: HM hits a sweet spot for various design constraints, and as soon as one tries to tinker, things fall apart.) It is also algorithmically quite simple, which is why you are going to learn about it today.

What is type inference?

```
int f(int x) { return x+1 :}
```

Imagine you are writing a function in C or C++ or Java. Ordinarily, you have to explicitly write types for the inputs and outputs.

What is type inference?

```
f(    x) { return x + 1 ;}
```

With type inference, such annotations are unnecessary; instead, the typechecker can *infer* what these types should be, by looking at the body of your function.

I don't have to annotate all my types? Sweet!

If you were writing Python because you loathe writing types in your programs, Hindley-M
type inference might be the thing for you! (Although Haskellers like writing the types of
their programs for documentation reasons anyway.)

To keep the subsequent algorithmic discussion simpler, we are going to tal about a subset of Haskell when discussing type inference. This langua is essentially Haskell without advance features, but we also make some other simplifications, such as getting rid of type classes (to be discussed later in this class!)

uHaskell

Haskell Subset

```
decl ::= name pat = exp

pat  ::= id | (pat, pat) | pat : pat | []

exp  ::= n | True | False | [] | id | (exp)
       | exp op exp | exp exp | (exp, exp)
       | if exp then exp else exp

type ::= type -> type | [type] | (type, type) | Bool | Int
```

Lists, Booleans, Pairs, Integers

Type Inference by Example

Ex 1 The Basics

the important one!

Ex 2 Polymorphism

Ex 3 Data Types

Ex 4 Type Error: Cannot Unify

Ex 5 Type Error: Occurs Check

The overall pattern is that we are going to work the type inference algorithm for five examples, which will go through the major points of Hindley-Milner unification.

Our first example will perform type inference for the following function. Intuitively, we can tell that the fact that x is used for an addition means that it is an integer, and that it returns an integer. The question will be how, algorithmically, a compiler can determine these types.

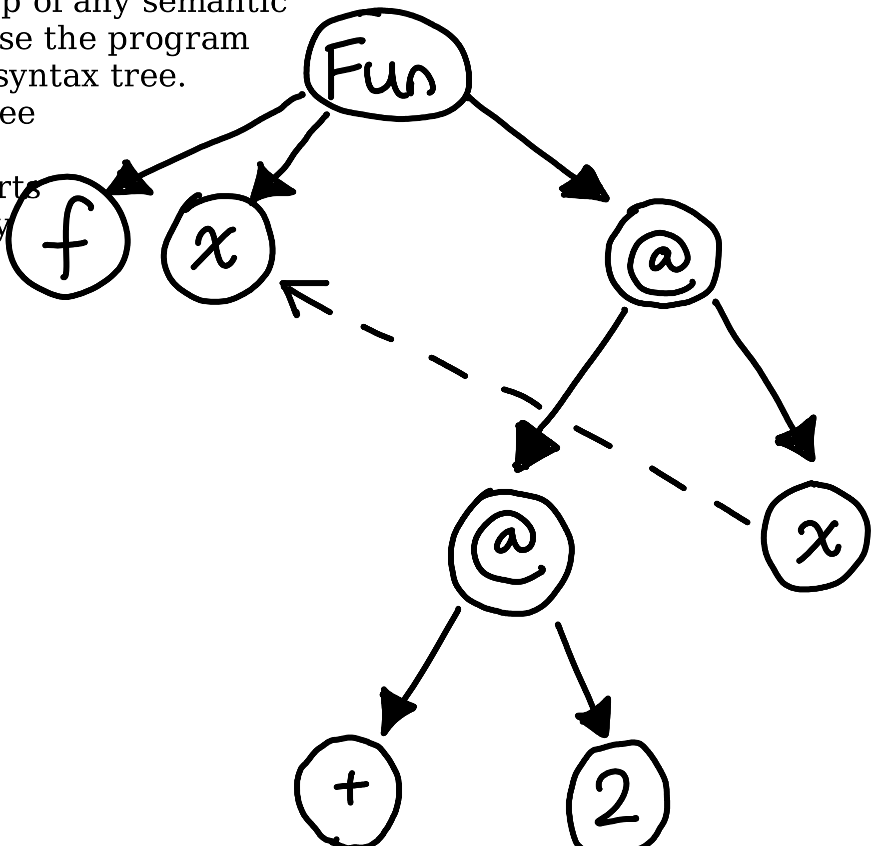
$$f\ x = 2 + x$$

Ex 1

1. Parsing

$$f\ x = 2 + x$$

The very first step of any semantic analysis is to parse the program into an abstract syntax tree. I've drawn the tree here; let's look at each of the parts of the tree one by one

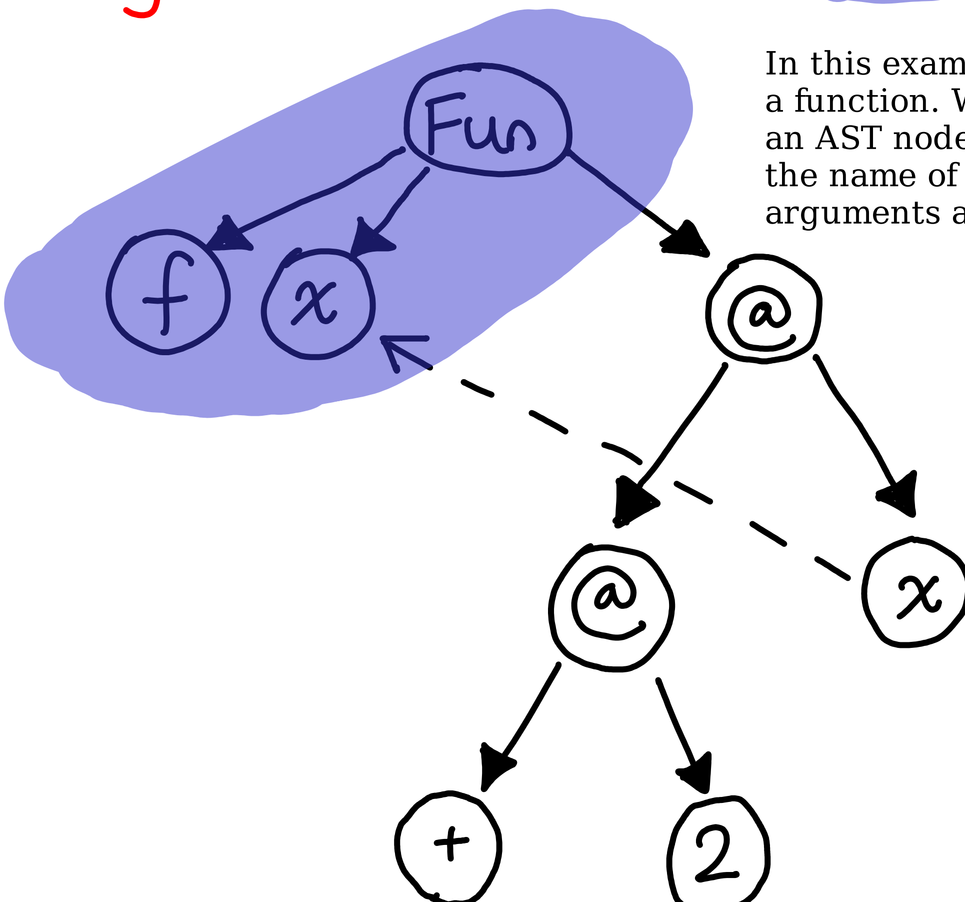


Ex 1

1. Parsing

$f\ x = 2 + x$

In this example, we are defining a function. We represent this as an AST node "Fun" which defines the name of the function and its arguments as subnodes.

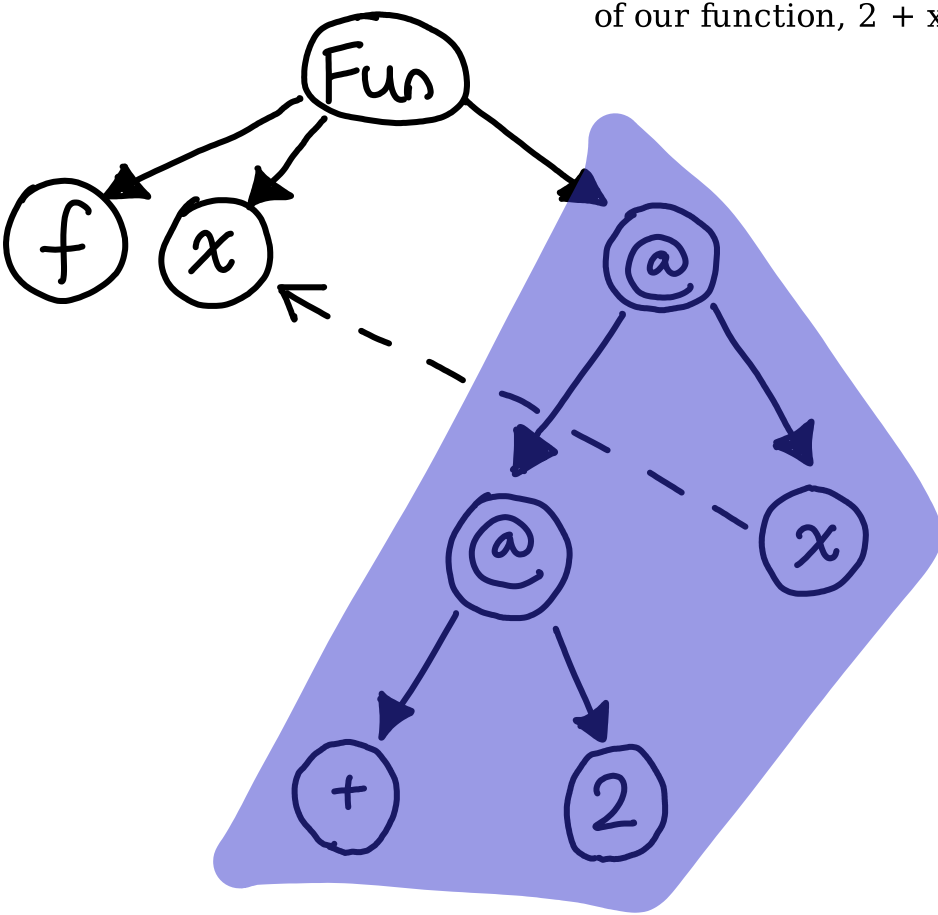


Ex 1

1. Parsing

$f\ x = 2 + x$

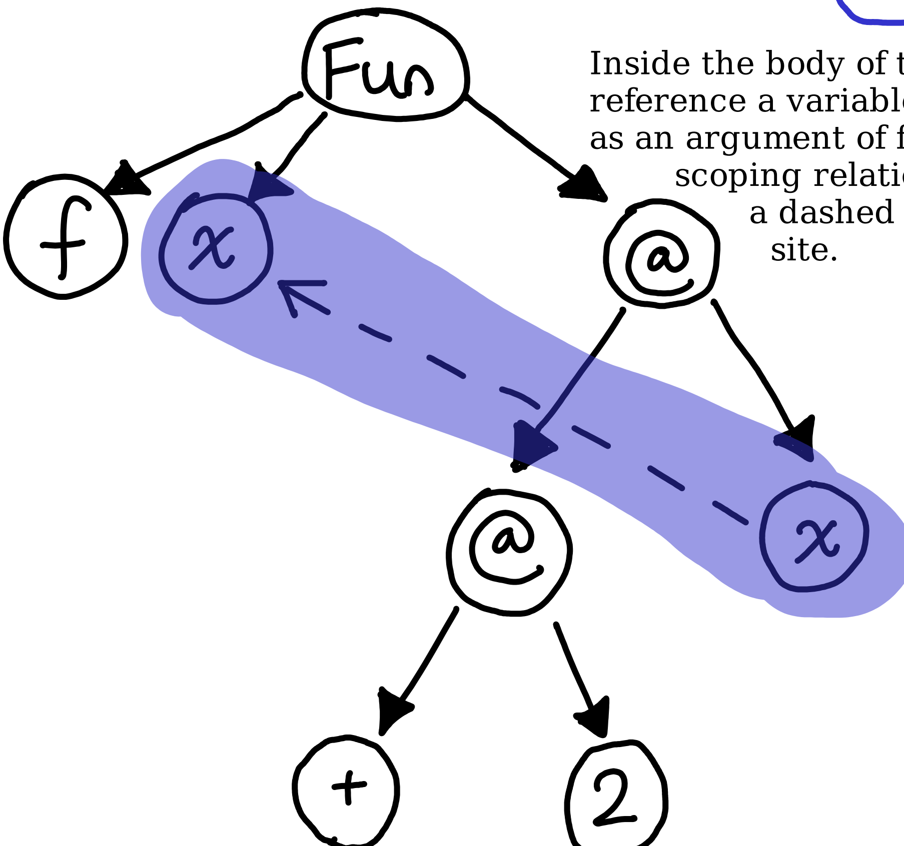
The rightmost child of Fun is the bod
of our function, 2 + x.



Ex 1

1. Parsing

$$f\ x = 2 + x$$



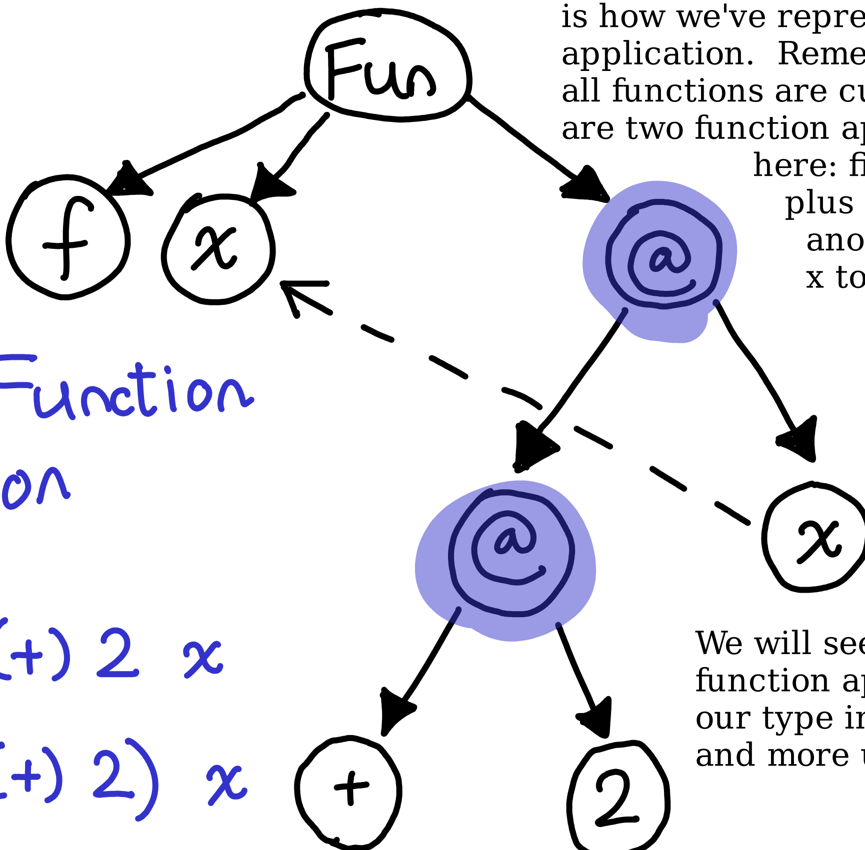
Inside the body of the function, we reference a variable that was declare as an argument of f. To make the scoping relationship clear, we dr a dashed arrow to the bindi site.

Ex 1

1. Parsing

$$f\ x = 2 + x$$

Another peculiarity of the syntax tree is how we've represented function application. Remember that in Haskell, all functions are curried. So there really are two function applications going on here: first to partially apply plus with two, and then another to finally apply x to this.



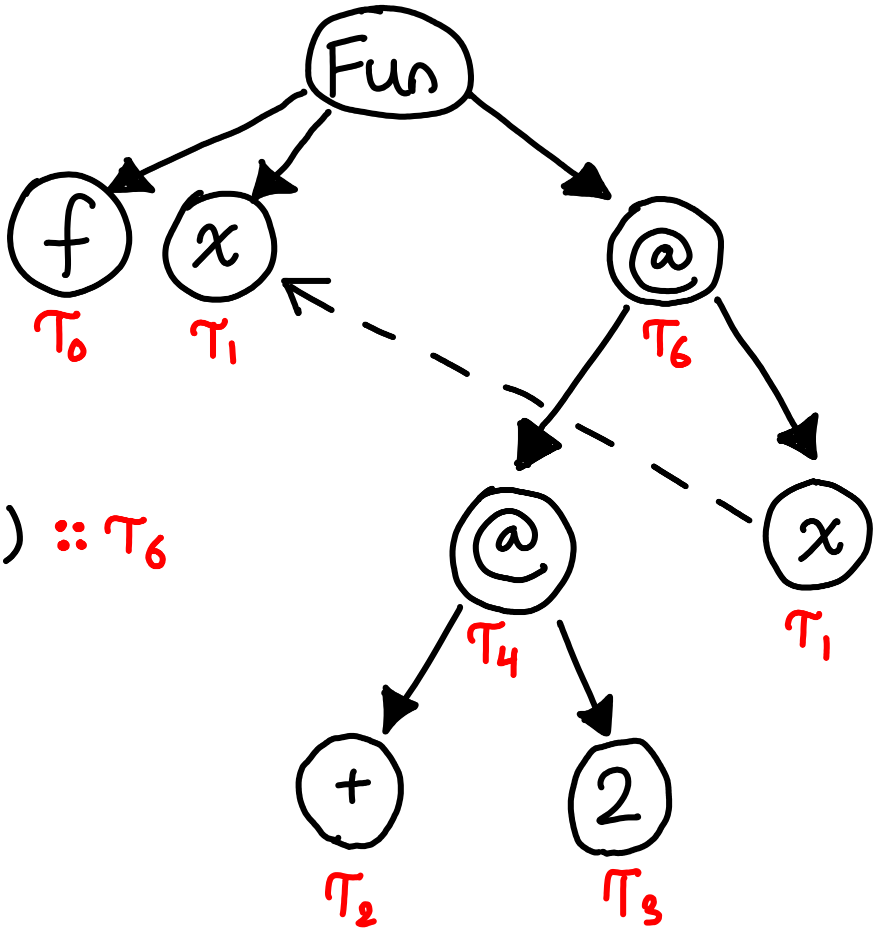
Curried Function Application

$$2 + x \cong (+)\ 2\ x$$
$$\cong ((+)\ 2)\ x$$

We will see how curried function application makes our type inference rules simpler and more uniform.

$\boxed{\text{Ex 1}}$

2. Assign Type Variables $f\ x = 2 + x$

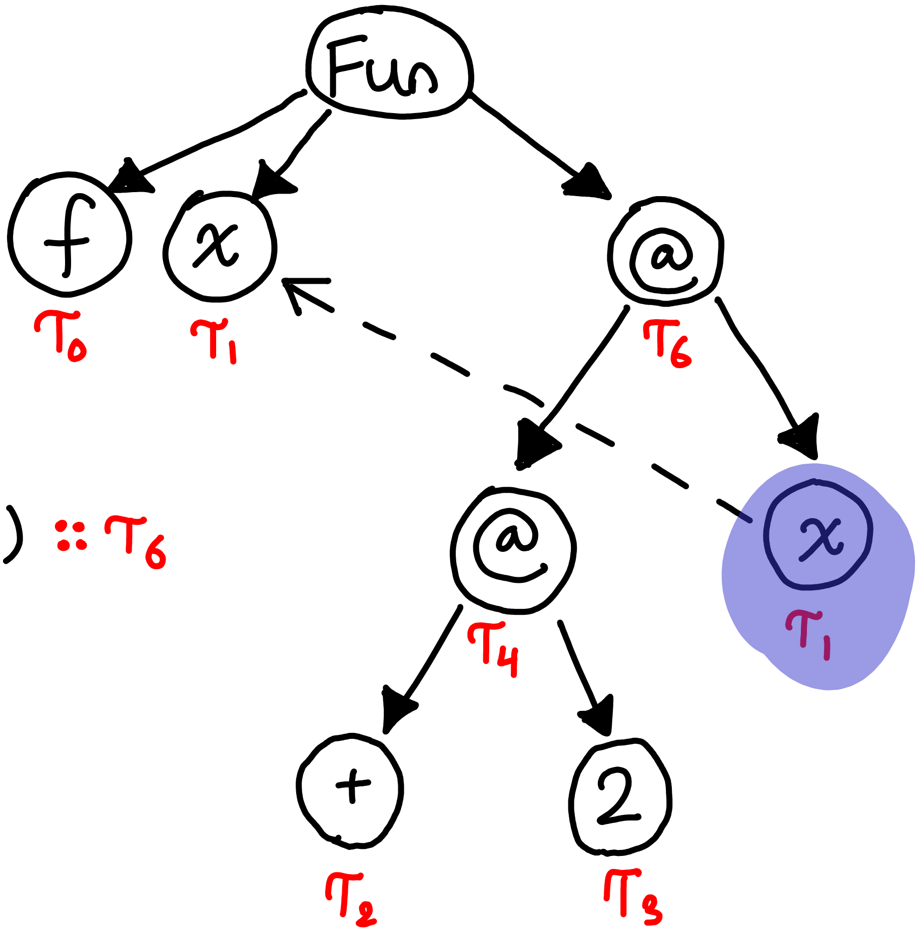


$f :: T_0$

$(((+)\ 2)\ x) :: T_6$

Ex 1

2. Assign Type Variables $f\ x = 2 + x$



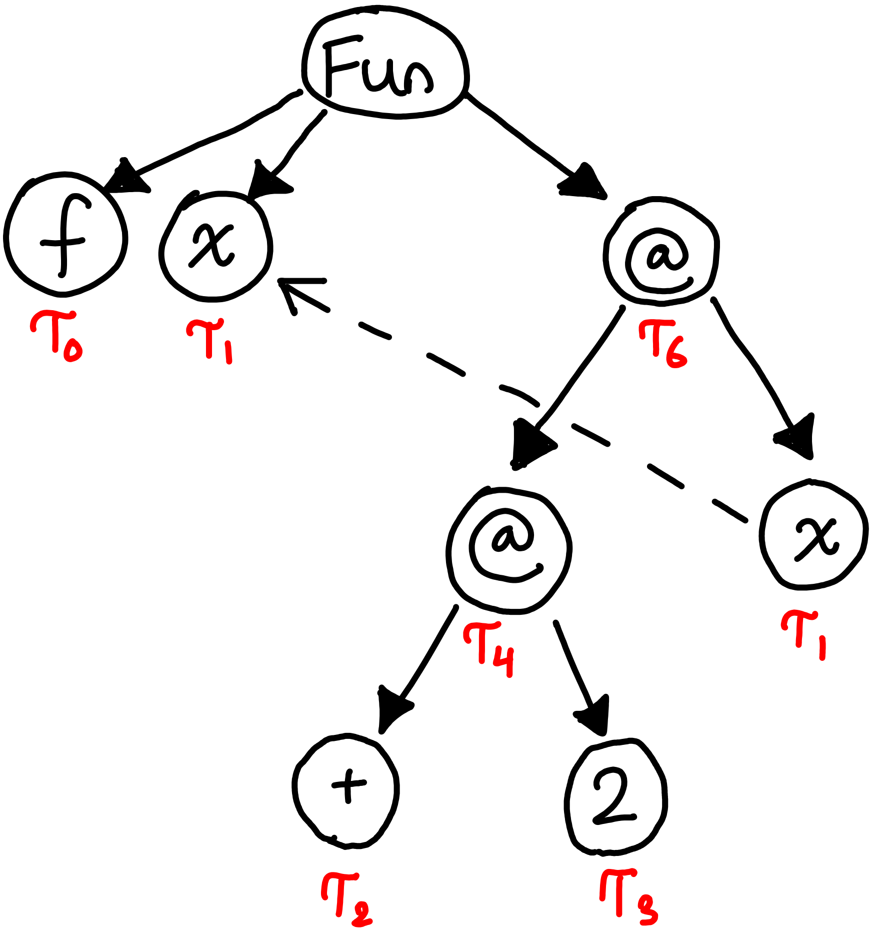
$f :: \tau_0$

$(((+)\ 2)\ x) :: \tau_6$

Ex 1

3. Add Constraints

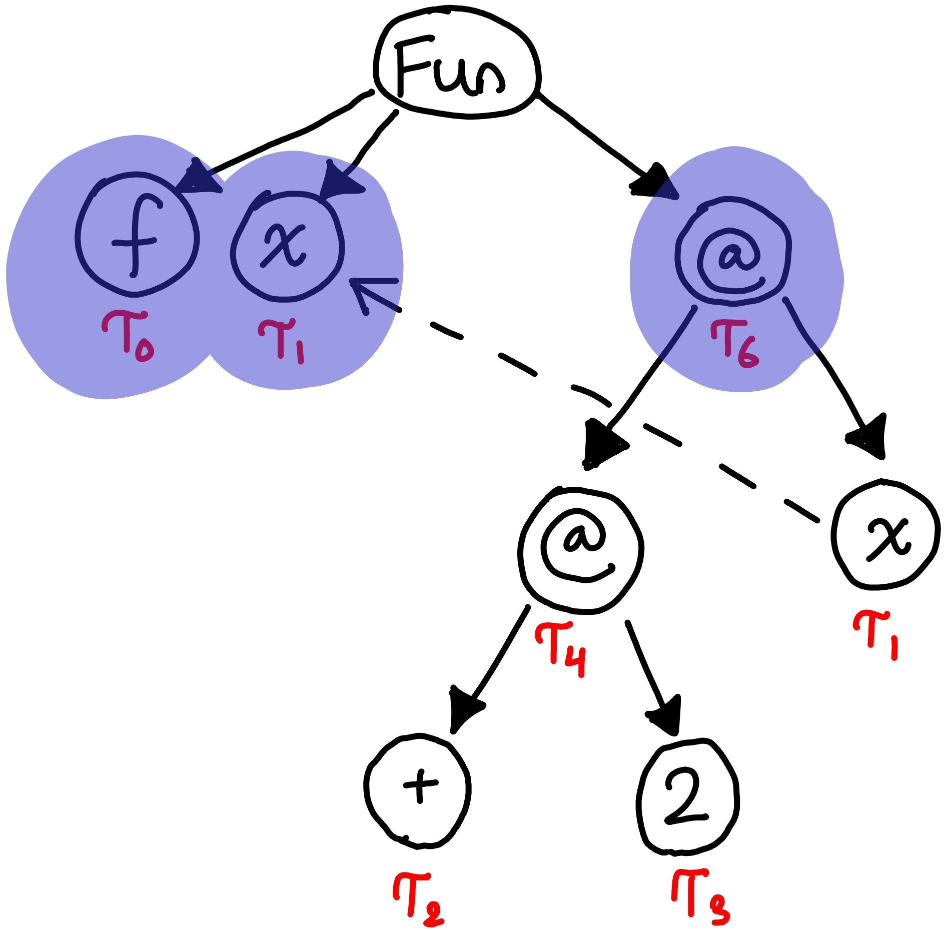
$$f\ x = 2 + x$$



Ex 1

3. Add Constraints

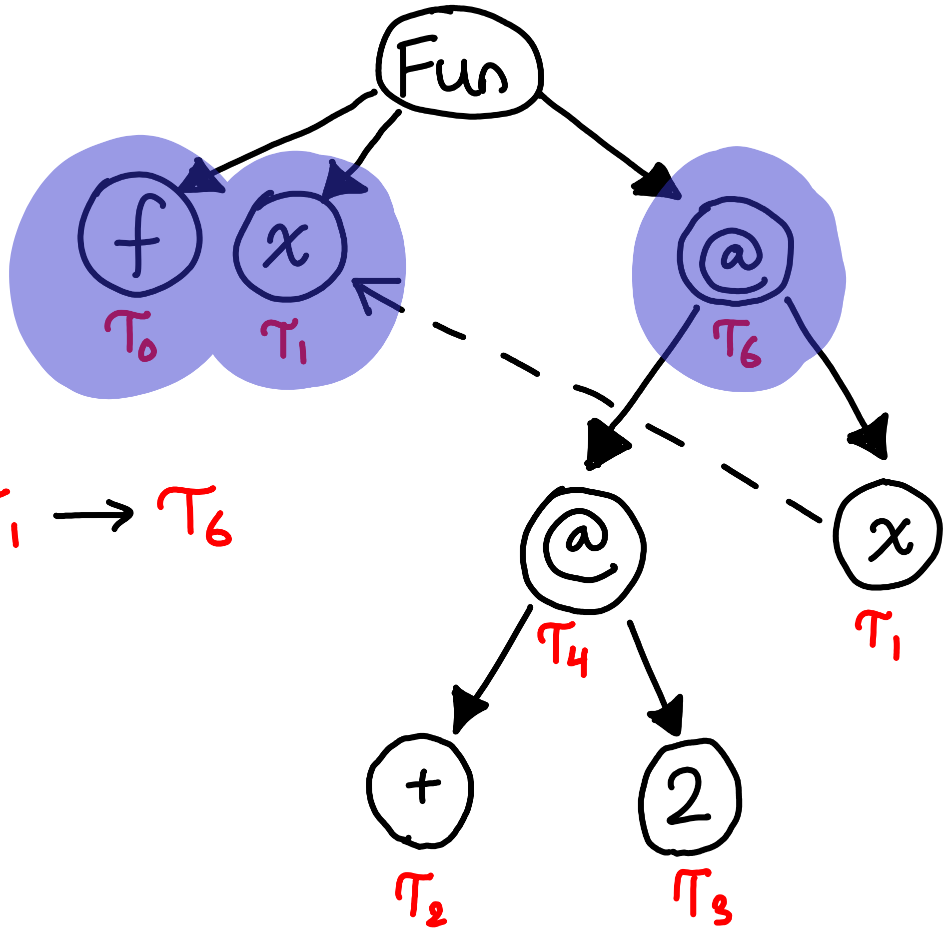
$f\ x = 2 + x$



Ex 1

3. Add Constraints

$f\ x = 2 + x$

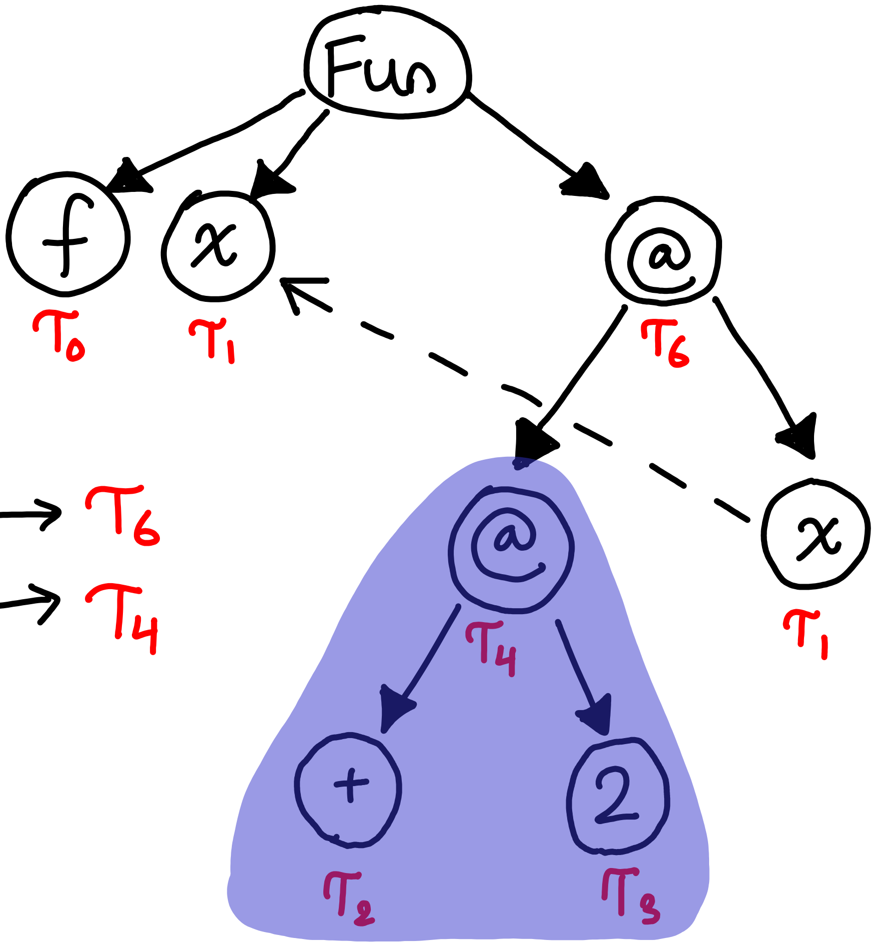


$\tau_0 = \tau_1 \rightarrow \tau_6$

Ex 1

3. Add Constraints

$f\ x = 2 + x$



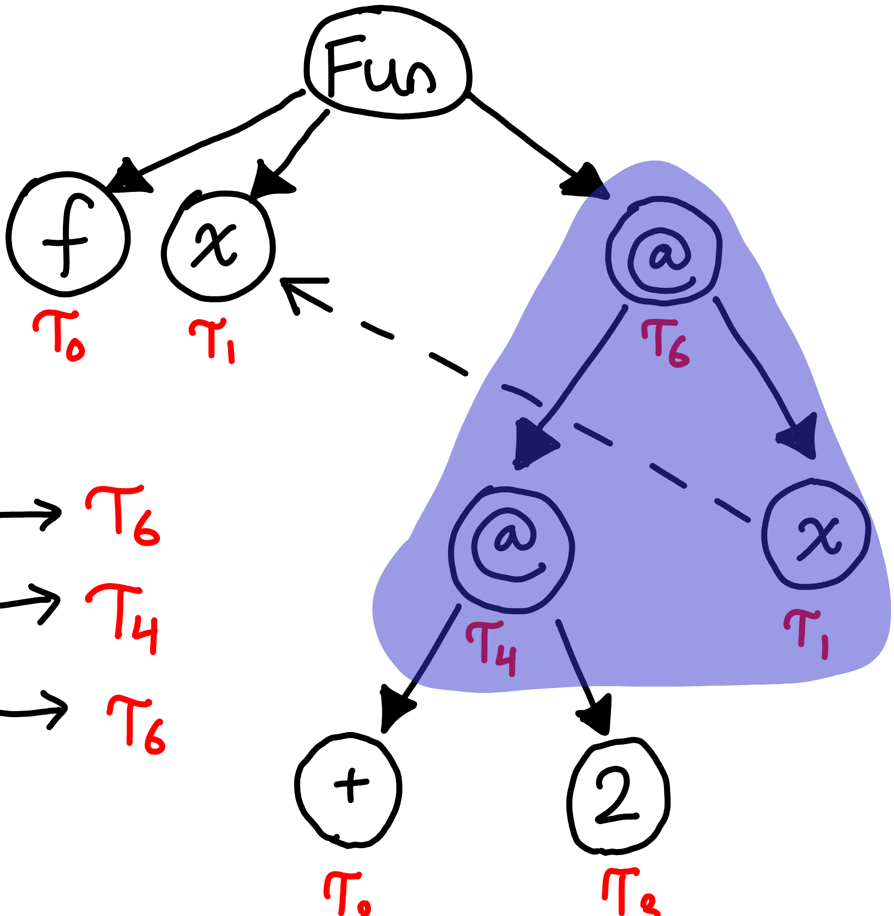
$\tau_0 = \tau_1 \rightarrow \tau_6$

$\tau_2 = \tau_3 \rightarrow \tau_4$

Ex 1

3. Add Constraints

$f\ x = 2 + x$



$\tau_0 = \tau_1 \to \tau_6$

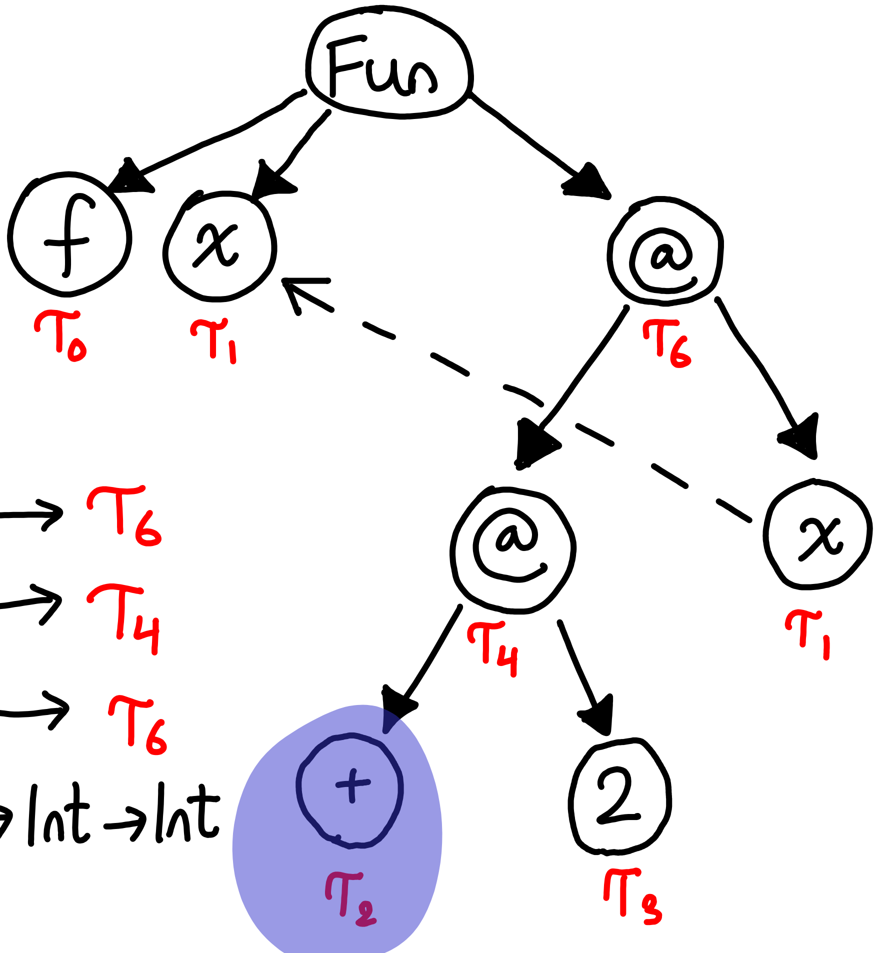
$\tau_2 = \tau_3 \to \tau_4$

$\tau_4 = \tau_1 \to \tau_6$

Ex 1

3. Add Constraints

$f\ x = 2 + x$



$\tau_0 = \tau_1 \to \tau_6$

$\tau_2 = \tau_3 \to \tau_4$

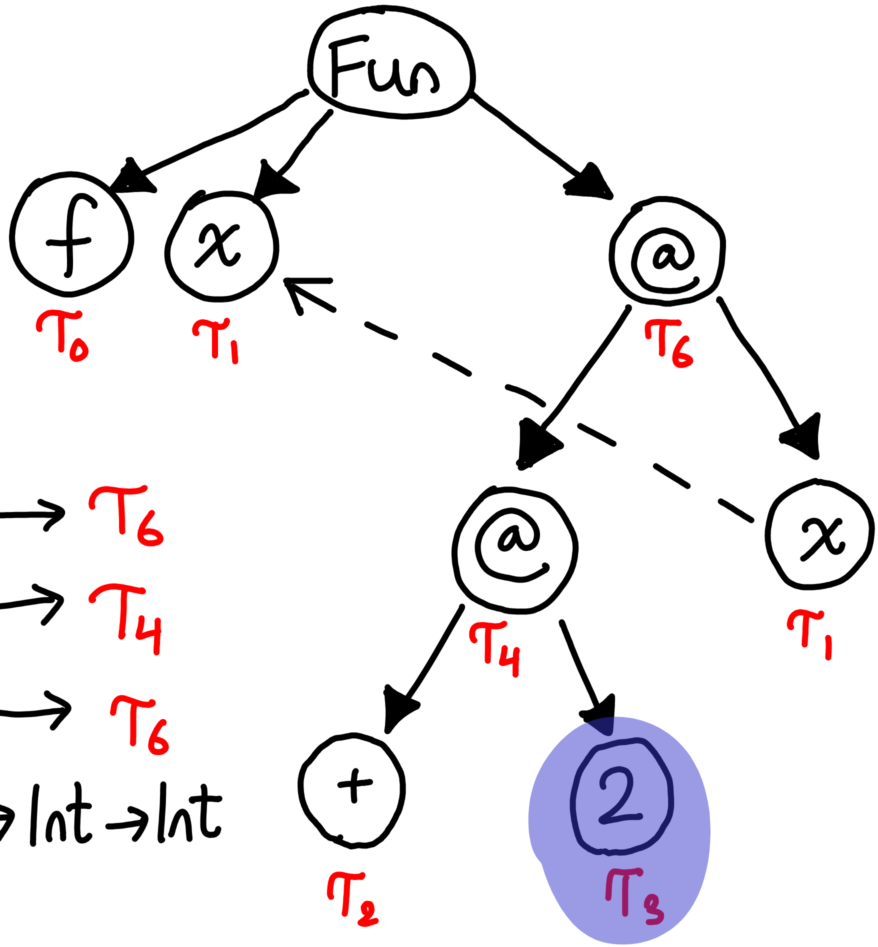
$\tau_4 = \tau_1 \to \tau_6$

$\tau_2 = Int \to Int \to Int$

$\boxed{\text{Ex 1}}$

3. Add Constraints

$f\ x = 2 + x$



$\tau_0 = \tau_1 \to \tau_6$

$\tau_2 = \tau_3 \to \tau_4$

$\tau_4 = \tau_1 \to \tau_6$

$\tau_2 = \mathrm{Int} \to \mathrm{Int} \to \mathrm{Int}$

$\tau_3 = \mathrm{Int}$

Ex 1

4. Solve constraints

$f\ x = 2 + x$

$T_0 = T_1 \to T_6$

$T_2 = T_3 \to T_4$

$T_4 = T_1 \to T_6$

$T_2 = Int \to Int \to Int$

$T_3 = Int$

Ex 1

4. Solve constraints

$f\ x = 2 + x$

process a constraint

$T_0 = T_1 \rightarrow T_6$

$T_2 = T_3 \rightarrow T_4$

$T_4 = T_1 \rightarrow T_6$

$T_2 = Int \rightarrow Int \rightarrow Int$

$T_3 = Int$

$\boxed{\text{Ex 1}}$

4. Solve constraints

$$f\ x = 2 + x$$

"finished" constraints

$T_0 = T_1 \rightarrow T_6$

$T_2 = T_3 \rightarrow T_4$

$T_4 = T_1 \rightarrow T_6$

$T_2 = Int \rightarrow Int \rightarrow Int$

$T_3 = Int$

Ex 1

4. Solve constraints

$$f\ x = 2 + x$$

$$\tau_0 = \tau_1 \to \tau_6$$

$$\tau_2 = \tau_3 \to \tau_4$$

$$\tau_4 = \tau_1 \to \tau_6$$

$$\tau_3 \to \tau_4 = Int \to Int \to Int$$

$$\tau_3 = Int$$

substitute

Ex 1

4. Solve constraints

$f\ x = 2 + x$

$$\tau_0 = \tau_1 \to \tau_6$$

$$\tau_2 = \tau_3 \to \tau_4$$

$$\tau_4 = \tau_1 \to \tau_6$$

$$\tau_3 \to \tau_4 = Int \to Int \to Int$$

$$\tau_3 = Int$$

Ex 1

4. Solve constraints

$f\ x = 2 + x$

$$T_0 = T_1 \to T_6$$

$$T_2 = T_3 \to (T_1 \to T_6)$$

$$T_4 = T_1 \to T_6$$

$$T_3 \to (T_1 \to T_6) = Int \to Int \to Int$$

$$T_3 = Int$$

$\boxed{\text{Ex 1}}$

4. Solve constraints

$f\ x = 2 + x$

$T_0 = T_1 \to T_6$

$T_2 = T_3 \to T_1 \to T_6$

$T_4 = T_1 \to T_6$

$T_3 \to T_1 \to T_6 = Int \to Int \to Int$

$T_3 = Int$

Ex 1

4. Solve constraints

$f\ x = 2 + x$

$T_0 = T_1 \to T_6$

$T_2 = T_3 \to T_1 \to T_6$

$T_4 = T_1 \to T_6$

$T_3 \to T_1 \to T_6 = Int \to Int \to Int$

$T_3 = Int$

no variable to substitute

$\boxed{\text{Ex 1}}$

4. Solve constraints

$f\ x = 2 + x$

$$\tau_0 = \tau_1 \to \tau_6$$

$$\tau_2 = \tau_3 \to \tau_1 \to \tau_6$$

$$\tau_4 = \tau_1 \to \tau_6$$

$$\tau_3 \to \tau_1 \to \tau_6 = \mathrm{Int} \to \mathrm{Int} \to \mathrm{Int}$$

$$\tau_3 = \mathrm{Int}$$

unification...

$\boxed{\text{Ex 1}}$

4. Solve constraints

$f\ x = 2 + x$

$$T_0 = T_1 \to T_6$$
$$T_2 = T_3 \to T_1 \to T_6$$
$$T_4 = T_1 \to T_6$$
$$T_3 = \mathrm{Int}$$
$$T_1 \to T_6 = \mathrm{Int} \to \mathrm{Int}$$
$$T_3 = \mathrm{Int}$$

... splitting an equality up!

$\boxed{\text{Ex 1}}$

4. Solve constraints

$$f\ x = 2 + x$$

$$T_0 = T_1 \to T_6$$

$$T_2 = T_3 \to T_1 \to T_6$$

$$T_4 = T_1 \to T_6$$

$$T_3 = Int$$

$$T_1 \to T_6 = Int \to Int$$

$$T_3 = Int$$

$$\boxed{\text{Ex 1}}$$

4. Solve constraints

$f\ x = 2 + x$

$$T_0 = T_1 \to T_6$$

$$T_2 = Int \to T_1 \to T_6$$

$$T_4 = T_1 \to T_6$$

$$T_3 = Int$$

$$T_1 \to T_6 = Int \to Int$$

$$\cancel{Int = Int}$$

pointless constraint

$\boxed{\text{Ex 1}}$

4. Solve constraints

$$f\ x = 2 + x$$

$$T_0 = T_1 \to T_6$$

$$T_2 = \text{Int} \to T_1 \to T_6$$

$$T_4 = T_1 \to T_6$$

$$T_3 = \text{Int}$$

$$T_1 \to T_6 = \text{Int} \to \text{Int}$$

$\boxed{\text{Ex 1}}$

4. Solve constraints

$f\ x = 2 + x$

$$\tau_0 = \tau_1 \to \tau_6$$
$$\tau_2 = Int \to \tau_1 \to \tau_6$$
$$\tau_4 = \tau_1 \to \tau_6$$
$$\tau_3 = Int$$
$$\tau_1 = Int$$
$$\tau_6 = Int$$

$\boxed{\text{Ex 1}}$

4. Solve constraints

$$f\ x = 2 + x$$

$$\tau_0 = \tau_1 \to \tau_6$$

$$\tau_2 = Int \to \tau_1 \to \tau_6$$

$$\tau_4 = \tau_1 \to \tau_6$$

$$\tau_3 = Int$$

$$\tau_1 = Int$$

$$\tau_6 = Int$$

Ex 1

4. Solve constraints

$f\ x = 2 + x$

$T_0 = \text{Int} \to T_6$

$T_2 = \text{Int} \to \text{Int} \to T_6$

$T_4 = \text{Int} \to T_6$

$T_3 = \text{Int}$

$T_1 = \text{Int}$

$T_6 = \text{Int}$

$\boxed{\text{Ex 1}}$

4. Solve constraints

$f\ x = 2 + x$

$$\tau_0 = \text{Int} \to \tau_6$$
$$\tau_2 = \text{Int} \to \text{Int} \to \tau_6$$
$$\tau_4 = \text{Int} \to \tau_6$$
$$\tau_3 = \text{Int}$$
$$\tau_1 = \text{Int}$$
$$\tau_6 = \text{Int}$$

$\boxed{\text{Ex } 1}$

4. Solve constraints

$$f\ x = 2 + x$$

$$T_0 = Int \to Int$$
$$T_2 = Int \to Int \to Int$$
$$T_4 = Int \to Int$$
$$T_3 = Int$$
$$T_1 = Int$$
$$T_6 = Int$$

$\boxed{\text{Ex 1}}$

5. Read out type

$$f\ x = 2 + x$$

$$T_0 = Int \to Int$$
$$T_1 = Int$$
$$T_2 = Int \to Int \to Int$$
$$T_3 = Int$$
$$T_4 = Int \to Int$$
$$T_6 = Int$$

$$f :: T_0$$
$$\Downarrow$$
$$f :: Int \to Int$$

$\boxed{\text{Ex 1}}$

Hindley-Milner type inference

(1. Parse the program)

2. Assign type variables to all nodes

3. Generate constraints

4. Solve constraints (via Unification)

(5. Read out top-level types)

Hindley-Milner type inference

(1. Parse the program)

2. Assign type variables to all nodes

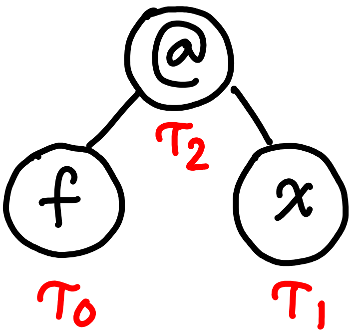
3. Generate constraints

4. Solve constraints (via Unification)

(5. Read out top-level types)

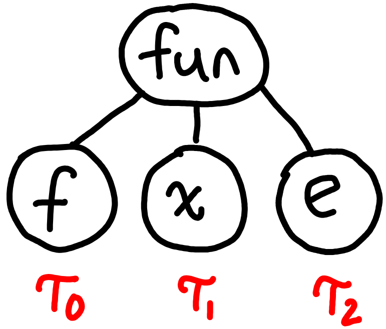
Generating constraints

application



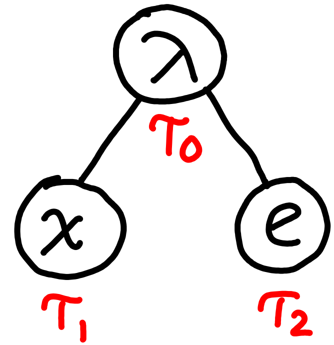
$$\tau_0 = \tau_1 \to \tau_2$$

fn declaration



$$\tau_0 = \tau_1 \to \tau_2$$

lambda



$$\tau_0 = \tau_1 \to \tau_2$$

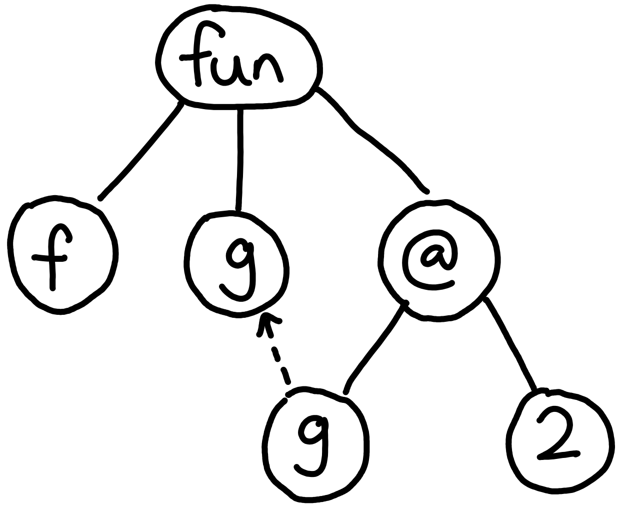
Polymorphism

$$f\,g = g\,2$$

$\boxed{\text{Ex 2}}$

Polymorphism

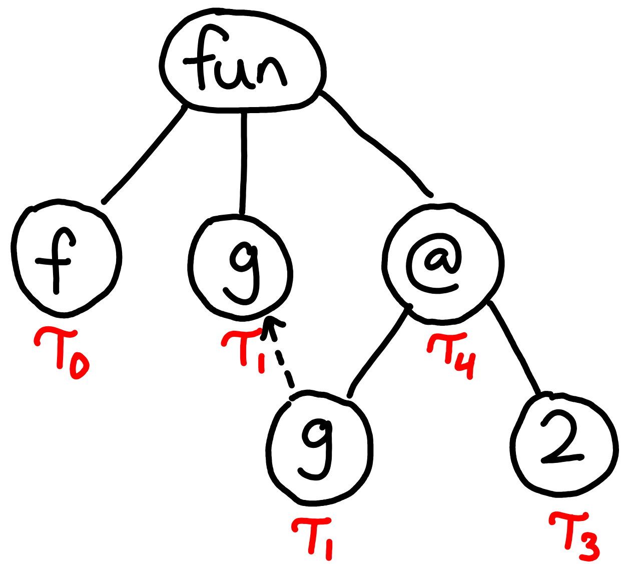
$$f\ g = g\ 2$$



Ex 2

Polymorphism

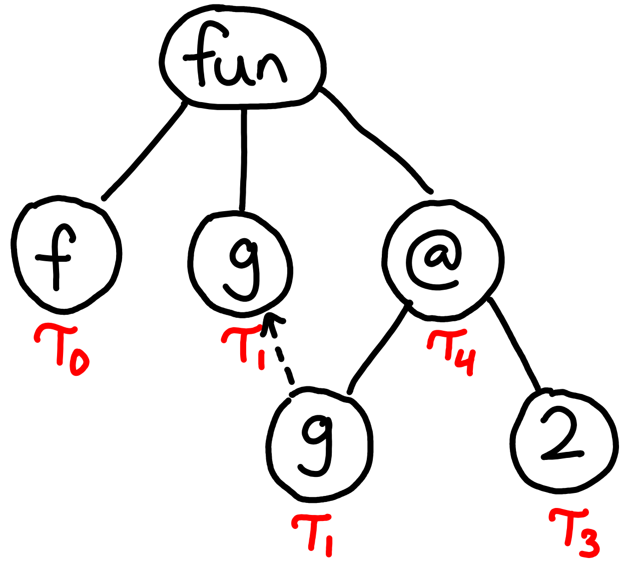
$$f\ g = g\ 2$$



Ex 2

Polymorphism

$f\ g = g\ 2$



$\tau_0 = \tau_1 \rightarrow \tau_4$

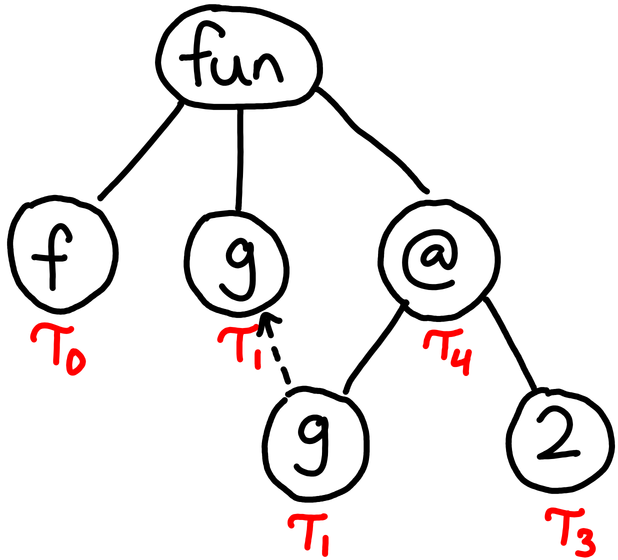
$\tau_1 = \tau_3 \rightarrow \tau_4$

$\tau_3 = \text{Int}$

$\boxed{\text{Ex } 2}$

Polymorphism

f g = g 2



$$\tau_0 = \tau_1 \rightarrow \tau_4$$

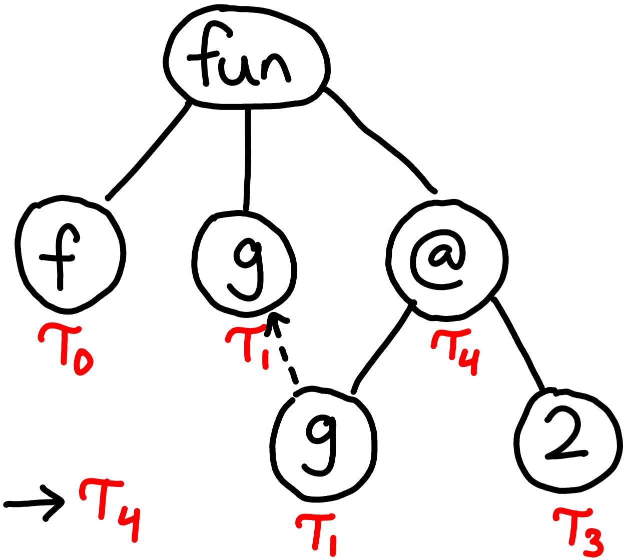
$$\tau_1 = \text{Int} \rightarrow \tau_4$$

$$\tau_3 = \text{Int}$$

Ex 2

Polymorphism

$f\ g = g\ 2$



$\tau_0 = (\mathrm{Int} \to \tau_4) \to \tau_4$

$\tau_1 = \mathrm{Int} \to \tau_4$

$\tau_3 = \mathrm{Int}$

Ex 2

Polymorphism

$$f\ g = g\ 2$$

$$f :: (Int \to \tau_4) \to \tau_4$$

polymorphic types

$$\tau_0 = (Int \to \tau_4) \to \tau_4$$

$$\tau_1 = Int \to \tau_4$$

$$\tau_3 = Int$$

$\boxed{\text{Ex 2}}$

Polymorphism

$f\ g = g\ 2$

$f_{\text{Int}} :: (\text{Int} \to \text{Int}) \to \text{Int}$

$f_{\text{Int}}\ (+2)$

$\boxed{\text{Ex 2}}$

Polymorphism

$$f\ g = g\ 2$$

$$f_{Int} :: (Int \to Int) \to Int \qquad f_{Int}\ (+2)$$

$$f_{Bool} :: (Int \to Bool) \to Bool \qquad f_{Bool}\ (==2)$$

$\boxed{\text{Ex 2}}$

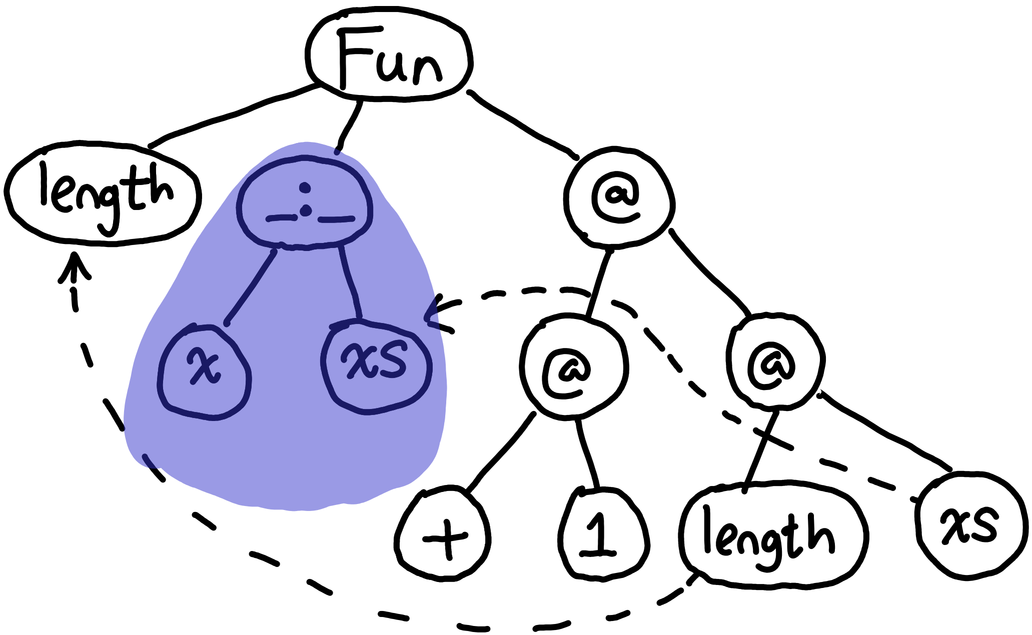
Data Types

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

Ex 3

Data Types

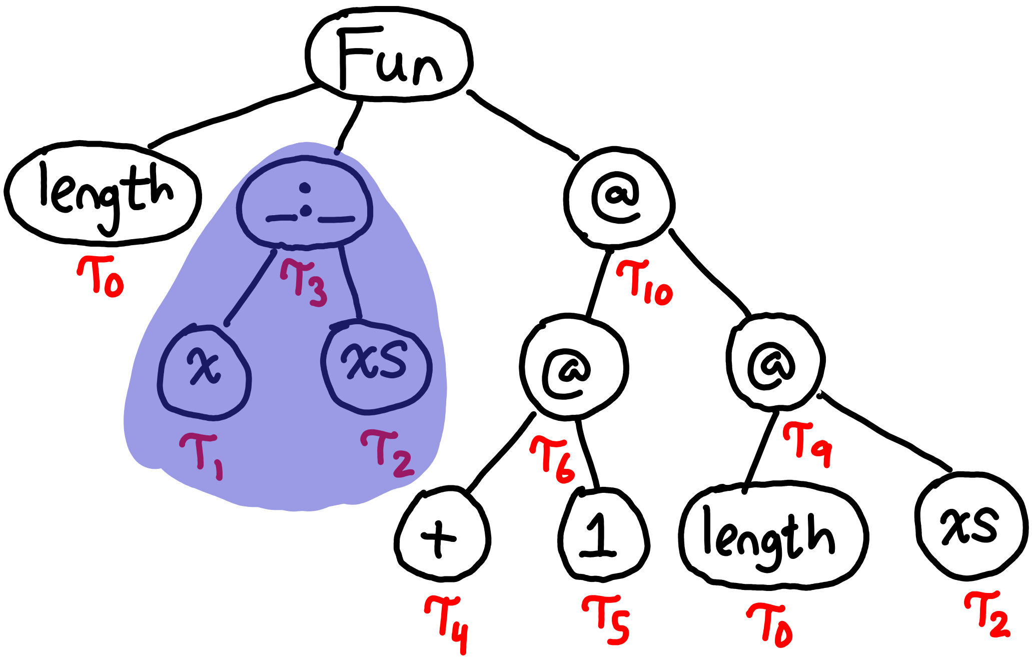
length (x:xs) = 1 + length xs



Ex 3

Data Types

$$\text{length } (x : xs) = 1 + \text{length } xs$$



Ex 3

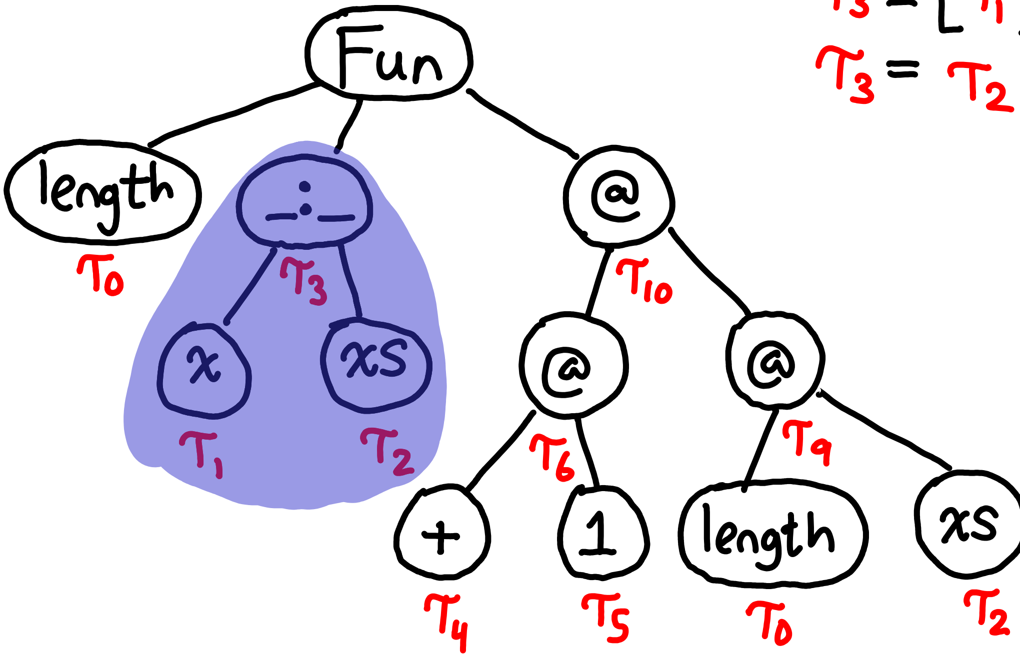
Data Types

length (x:xs) = 1 + length xs

$\tau_0 = \tau_3 \to \tau_{10}$

$\tau_3 = [\tau_1]$

$\tau_3 = \tau_2$



Ex 3

Data Types

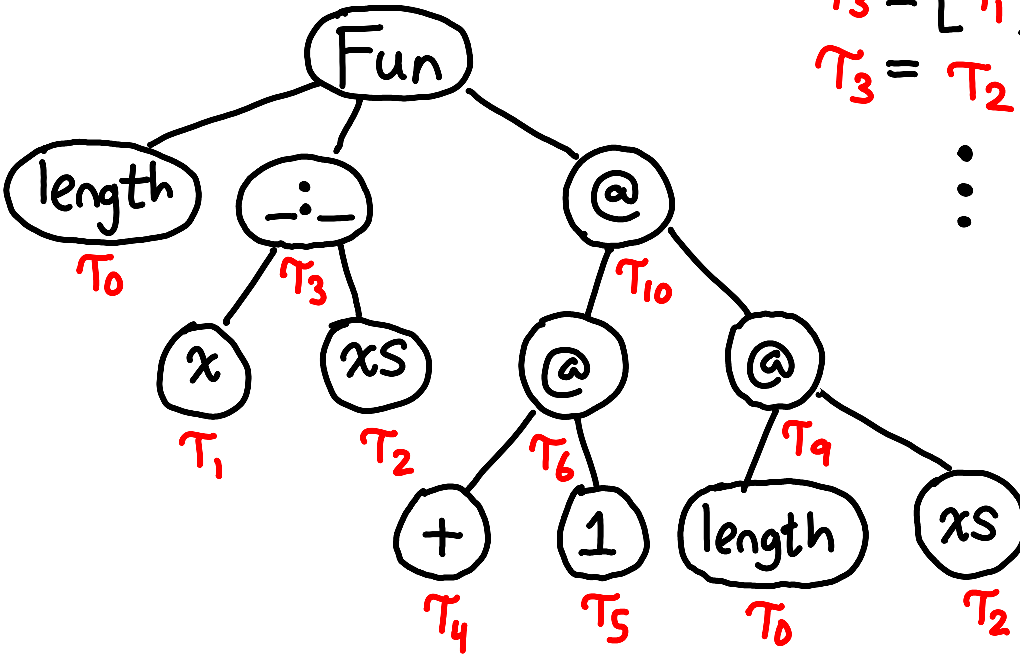
length (x:xs) = 1 + length xs

$\tau_0 = \tau_3 \rightarrow \tau_{10}$

$\tau_3 = [\tau_1]$

$\tau_3 = \tau_2$

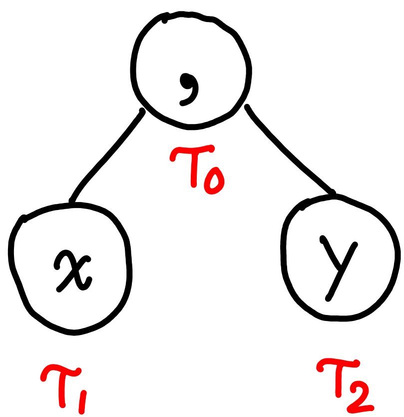
$\vdots$



length :: $[\tau_1] \rightarrow$ Int

$\boxed{\text{Ex 3}}$

Exercise: What are the constraints generated by products?

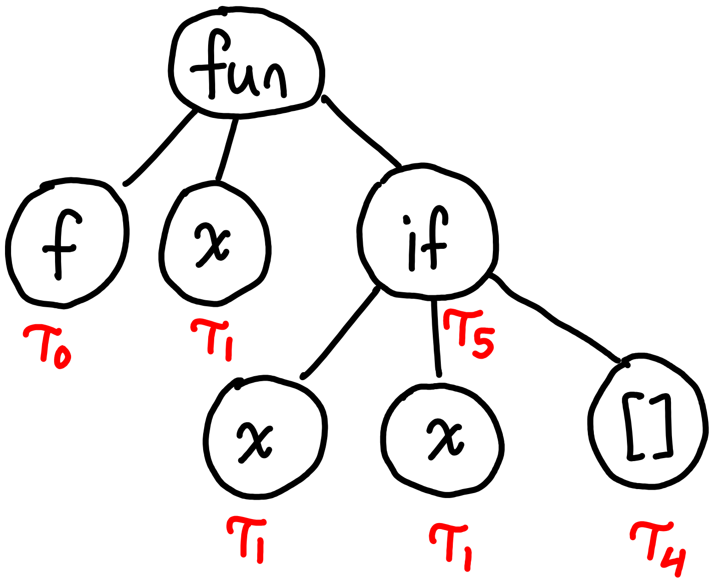


Type errors: Cannot unify □ and □

$$f\ x = \text{if } x \text{ then } x \text{ else } []$$

$\boxed{\text{Ex } 4}$

Type errors: Cannot unify □ and □



$\tau_0 = \tau_1 \to \tau_5$

$\tau_5 = \tau_1$

$\tau_1 = \tau_4$

$\tau_1 = \text{Bool}$

$\tau_4 = [\tau_6]$

Ex 4

Type errors: Cannot unify ☐ and ☐

$$\tau_1 = \text{Bool} \neq [\tau_6] = \tau_4$$

$\tau_0 = \tau_1 \to \tau_5$
$\tau_5 = \tau_1$
$\tau_1 = \tau_4$
$\tau_1 = \text{Bool}$
$\tau_4 = [\tau_6]$

$\boxed{\text{Ex 4}}$

Type errors: Occurs check

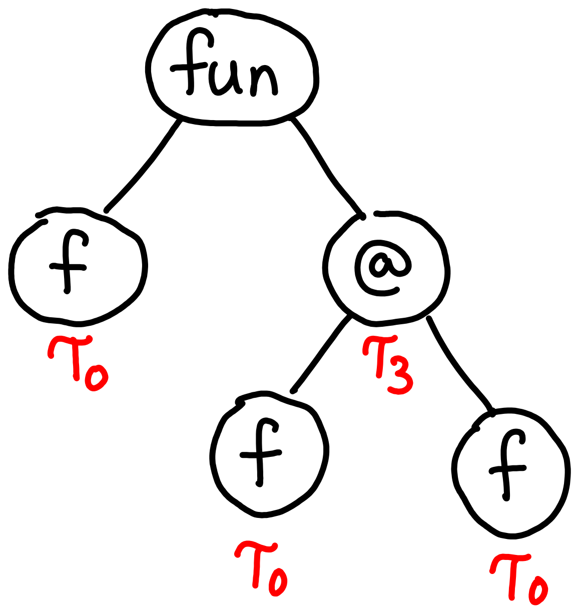
$$f = f\ f$$

remember Ω?

$\boxed{\text{Ex 5}}$

Type errors: Occurs check

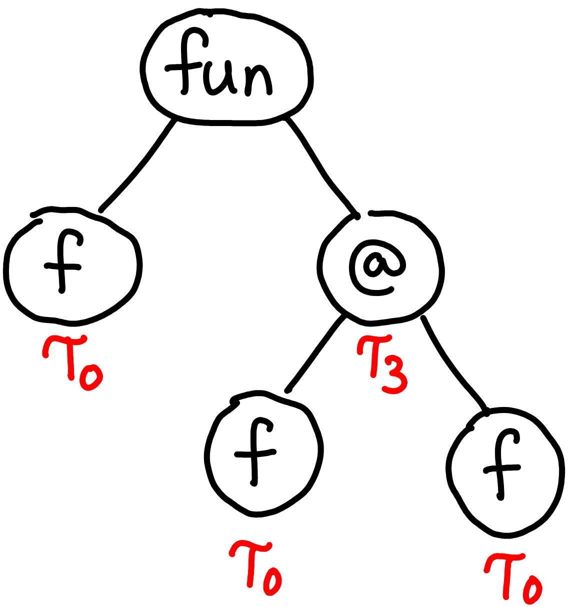
$$f = f\ f$$



Ex 5

Type errors: Occurs check

$$f = f\ f$$



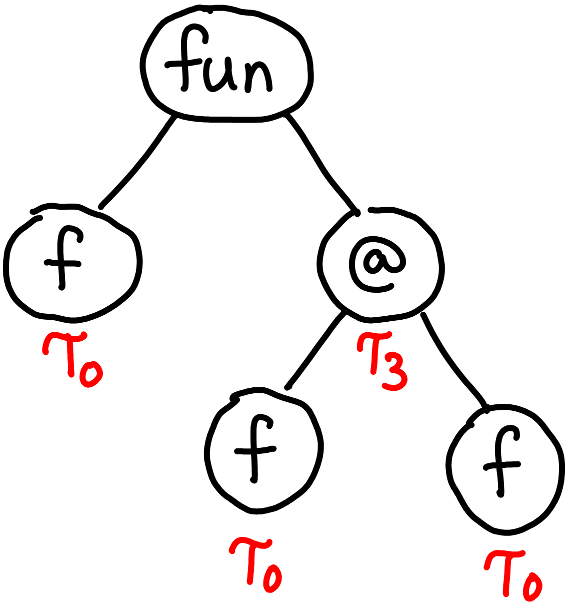
$$\tau_0 = \tau_3$$

$$\tau_0 = \tau_0 \to \tau_3$$

Ex 5

Type errors: Occurs check

$$f = f\ f$$



$$\tau_0 = \tau_3$$

$$\tau_0 = \tau_0 \to \tau_3$$

Ex 5

$$T_0 = T_0 \rightarrow T_3$$

$$T_0 = (T_0 \rightarrow T_3) \rightarrow T_3$$

$$T_0 = ((T_0 \rightarrow T_3) \rightarrow T_3) \rightarrow T_3$$

$$\vdots$$

Type errors: Occurs check

if e contains x and $e \neq x$
then $\text{unify}(x, e)$ fails

e.g. $\text{unify}(\tau_0, \tau_0 \to \tau_3)$ fails

Left out:

- let-bindings

  let f x = x
  in (f 2, f True)

  these need distinct type variables

- the "deductive system"

$$\frac{\Gamma, x:\tau \vdash e : \tau'}{\Gamma \vdash \lambda x . e : \tau \to \tau'}$$

more inference rules!

Fun fact: Hindley-Milner type inference is DEXPTIME-complete

[Kanellakis, Mairson, Mitchell '89]

$$
\begin{aligned}
\mathrm{pair}\ x\ f &= f\ x\ x \\
f_1\ x &= \mathrm{pair}\ x \\
f_2\ x &= f_1\ (f_1\ x) \\
f_3\ x &= f_2\ (f_2\ x) \\
g\ z &= f_3\ (\lambda x.\ x)\ z
\end{aligned}
$$

Fun fact: Hindley-Milner type inference infers a **unique most general type** for all expressions (**principal typing**)

$$\begin{tikzcd}
 & a \to b \to a & \\
 \mathrm{Int} \to b \to \mathrm{Int} \arrow[ur] & & a \to \mathrm{Int} \to a \arrow[ul] \\
 & \mathrm{Int} \to \mathrm{Int} \to \mathrm{Int} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Comparison: C++ templates

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

Comparison: C++ templates

```
void swap(Dog &x, Dog &y) {
    Dog tmp = x;
    x = y;
    y = tmp;
}
```

```
void swap(Cat &x, Cat &y) {
    Cat tmp = x;
    x = y;
    y = tmp;
}
```

Comparison: Go "type inference"

```
var int y;
x := 2+y;
```

int

no polymorphism & annotations

Hindley-Milner Type Inference

+ No more annotations
+ Polymorphism
+ Technique generalizes
- Non-local errors
- Mutable assignment
- Implementation requires boxing
- Not what Haskell or ML uses