Types

Week 4

- General discussion of types
- Type inference
- Type polymorphism

What is a type?

- Examples of types:
  - Integer
  - [Char]
  - Either (Either Char Int) Bool
- Working, informal definition: set of values
  - Where does this definition break down?

A type is: a way to prevent errors

- E.g.,

```
const y = 1;
y + “w00t”;
```

- E.g.,

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

A type is: a way to prevent errors

- E.g.,

```haskell
-- | Function must be applied to 2 Ints
plus :: Int -> Int -> Int
plus a b = ...
```

- E.g.,

```haskell
-- | Must be applied to a function and
-- argument that that function can be applied to
apply :: (a -> b) -> a -> b
apply f x = f x
```

A type is: a way to prevent errors

- The world's most lightweight* and widely-used formal method!
  - Prevent meaningless computations from being expressed or executed

A type is: a method of organization & documentation

- E.g., consider abstract data type for sets

```
data Set k = …
empty  :: Set k
insert :: k -> Set k -> Set k
delete :: k -> Set k -> Set k
member :: k -> Set k -> Bool
```

- `E.g., consider type for reading a file`

```
readFile :: FilePath -> IO String
```

A type is: a hint to the compiler

- E.g., what should `obj.prop1` be compiled down to?

Who enforces types?

- Consider, for example: `arr[200]`
  - What happens in JavaScript if `arr` is `null`?
  - What happens in C/C++ if `arr` is of size `10`?
  - What happens in Haskell if `arr` is not an array?

Who enforces types?

- This is language dependent…
  - ➤ The compiler at compile time
  - ➤ The runtime system at run-time
  - ➤ The hardware at run-time

What are the tradeoffs of each?

	Compile-time	Run-time checks	Hardware
Pro	No runtime overhead	Permissive	Super fast
Con	Over approximates	Runtime overhead	Catch bugs late

Compile-time is the best! (Is it?)

The cost of compile-time checking

- Sometimes you give up expressivity

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

  - More advanced type systems can “type” this function (dependent types); at what cost?

- Why is this fundamental? A: static analysis approximates — it has to work for every run of the program

Why do we check types? Safety!

- <u>Def:</u> A language is **type safe** if no program is allowed to violate its type distinctions
  - ➤ Is Haskell type safe? A: yes, B: no
  - ➤ Is JavaScript type safe? A: yes, B: no
  - ➤ Is C/C++ type safe? A: yes, B: no
- What language features make it hard to guarantee type safety? A: raw pointer/memory access, casts, etc.

Why do we check types? Safety!

- Def: A language is **type safe** if no program is allowed to violate its type distinctions
  - ➤ Is Haskell type safe? A: yes, B: no
  - ➤ Is JavaScript type safe? A: yes, B: no
  - ➤ Is C/C++ type safe? A: yes, B: no
- What language features make it hard to guarantee type safety? A: raw pointer/memory access, casts, etc.

Why do we check types? Safety!

- Def: A language is **type safe** if no program is allowed to violate its type distinctions
  - Is Haskell type safe? A: yes, B: no
  - Is JavaScript type safe? A: yes, B: no
  - Is C/C++ type safe? A: yes, B: no
- What language features make it hard to guarantee type safety? A: raw pointer/memory access, casts, etc.

Why do we check types? Safety!

- <u>Def:</u> A language is **type safe** if no program is allowed to violate its type distinctions
  - Is Haskell type safe? A: yes, B: no
  - Is JavaScript type safe? A: yes, B: no
  - Is C/C++ type safe? A: yes, B: no
- What language features make it hard to guarantee type safety? A: raw pointer/memory access, casts, etc.

Week 4

- General discussion of types ✓
- Type inference
- Type polymorphism

Type inference

- What's the difference between type checking and type inference?
  - E.g.,

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

  - Type checking: checks that x is actually used as an int
  - Type inference: based usage infers that x is an int

Why study type inference?

- Reduces syntactic overhead of expressive types
- Guaranteed to produce the most general type
- One of the most important language innovations
    - Even C++ has type inference now!
- Good example of a flow-insensitive static analysis alg

What we’re going to look at

Hindley-Milner type inference for uHaskell!

Hindley-Milner type inference

- [1958] Curry and Feys invented type inference algorithm for the simply typed λ calculus

- [1969] Hindley extended algorithm to richer language and proved it always produced most general type

- [1978] Milner developed Algorithm W

- [1982] Damas prove the algorithm was compete

Hindley-Milner type inference

1. Parse the program
2. Assign type variables to all nodes
3. Generate constraints between type variables
4. Solve constraints (via unification)
5. Read out types of top-level declarations

uHaskell

- Declarations: `d ::= name p = e`
- Patterns: `p ::= id | (p, p) | p:p | []`
- Expressions
```
e ::= n | True | False | [] | id
    | (e) | e ⊕ e | e e | (e,e)
    | if e then e else e
```
- Types:
```
τ ::= τ -> τ | [τ] | (τ,τ)
    | Bool | Int
```

Type inference by example

1. Basic idea
2. Polymorphism
3. Data types
4. Type error: cannot unify
5. Type error: occurs check

Ex1. Basic idea

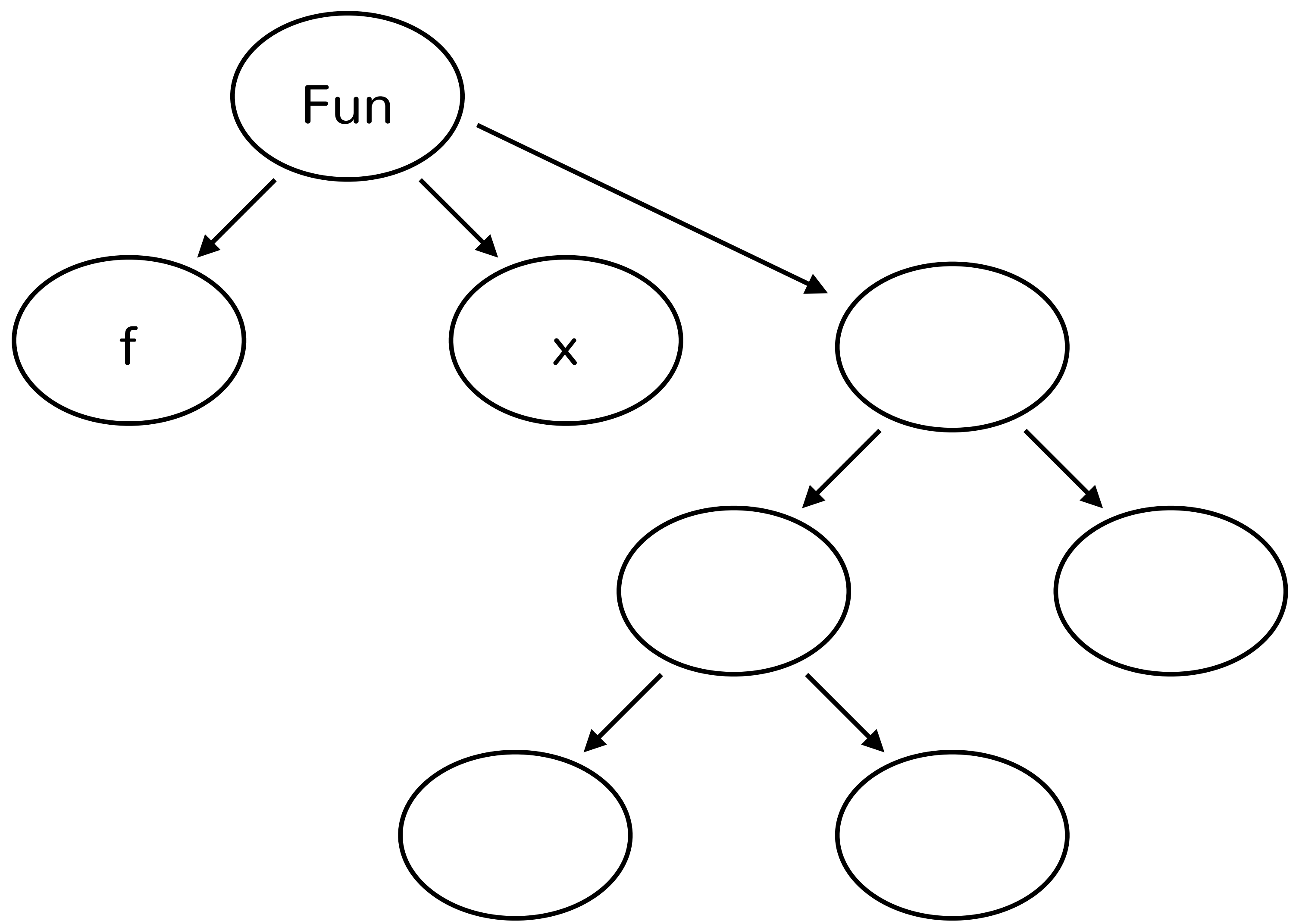
- Example: `f x = 2 + x`
- Goal: What is the type of `f`? Let's do it informally:
  - `2 :: Int`
  - `(+) :: Int -> Int -> Int`
  - We are applying `(+)` to `x`, we need `x :: Int`
  - Thus: `f x = 2 + x :: Int -> Int -> Int`

Ex1. Basic idea

- Step 1: parse program to construct parse tree
  - f x = 2 + x

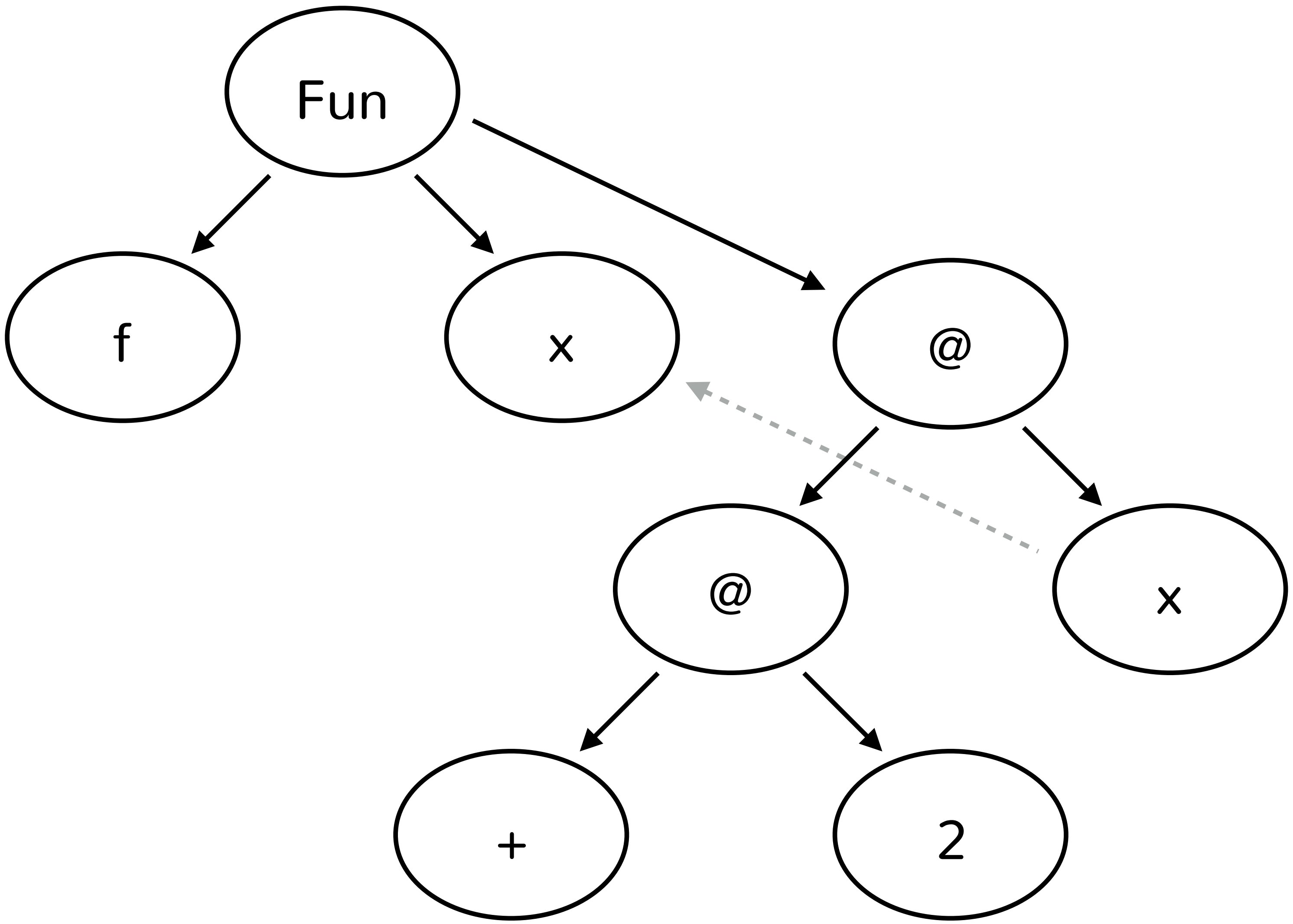
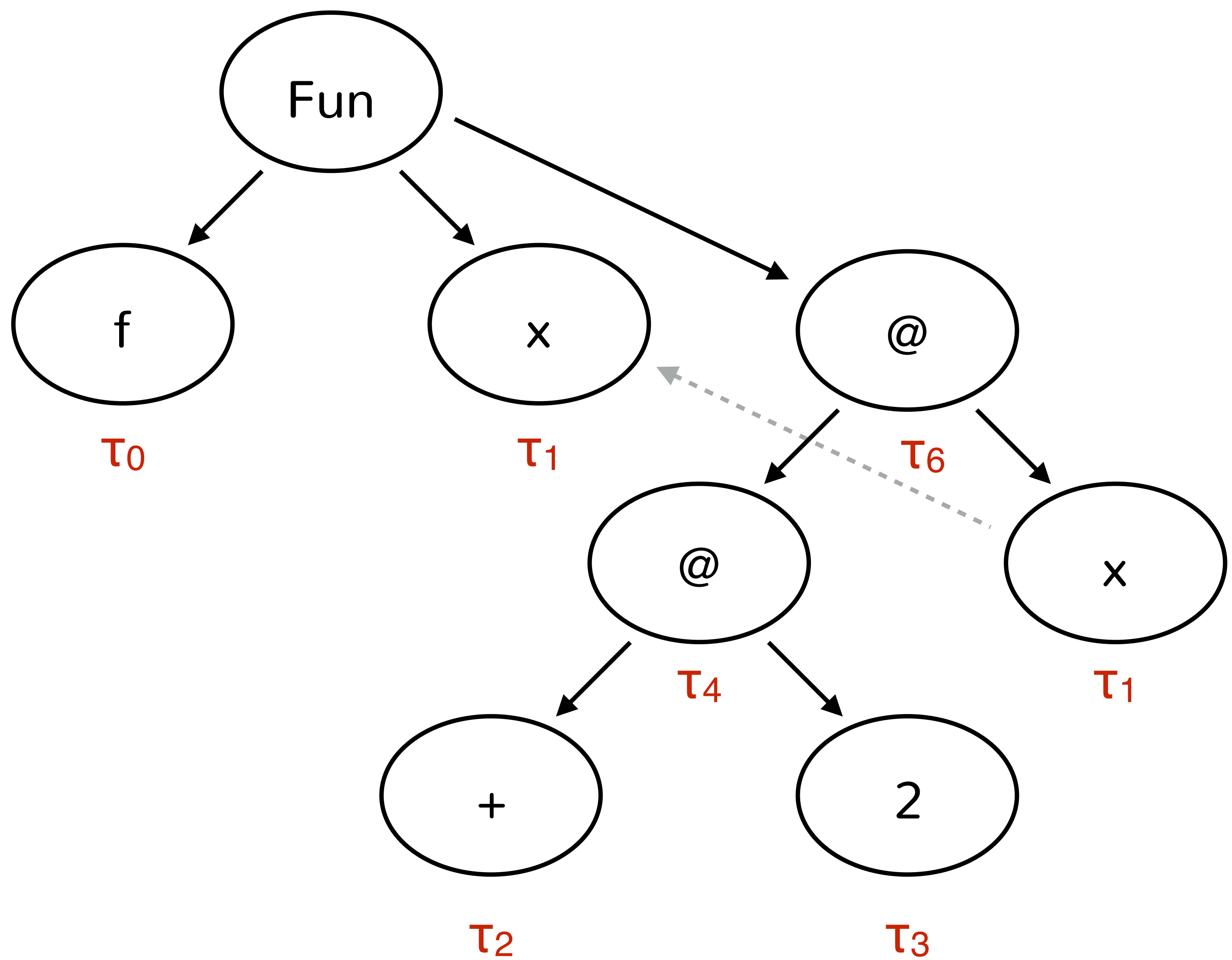
    =

    =



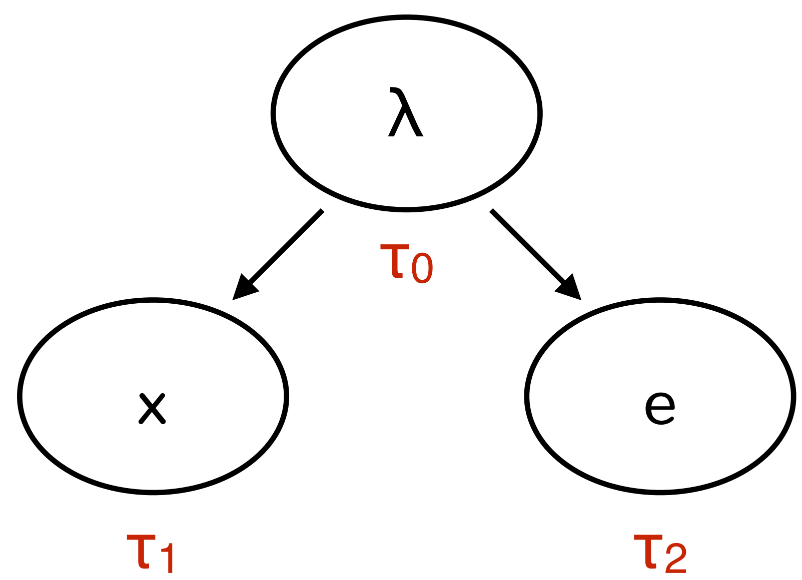
Ex1. Basic idea

- Step 2: assign type variables to nodes
  - ```
    f x = 2 + x
    = (+) 2 x
    = ((+) 2) x
    ```


    ```

Ex1. Basic idea

- Step 3: add constraints
  - ```
    f x = 2 + x
    = (+) 2 x
    = ((+) 2) x
    ```


    ```

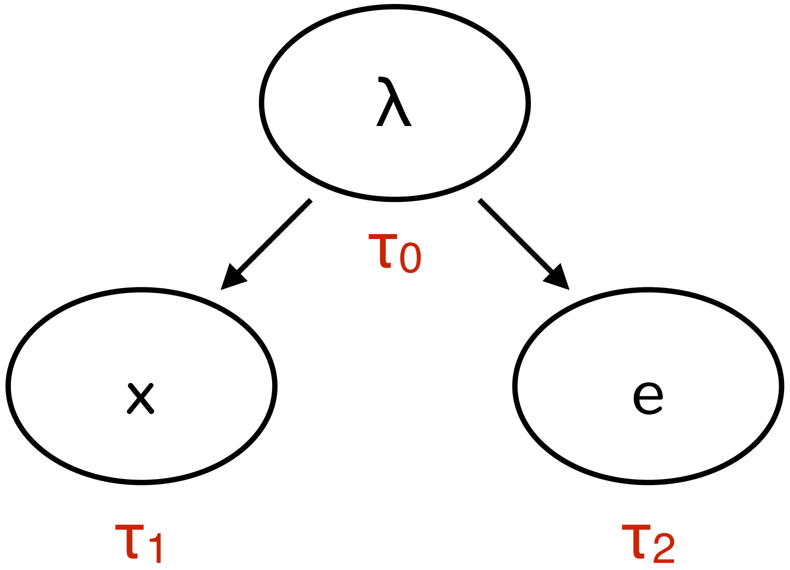
Generating constraints

- Lambda abstraction (λx.e)
  - ➤ τ_0 =



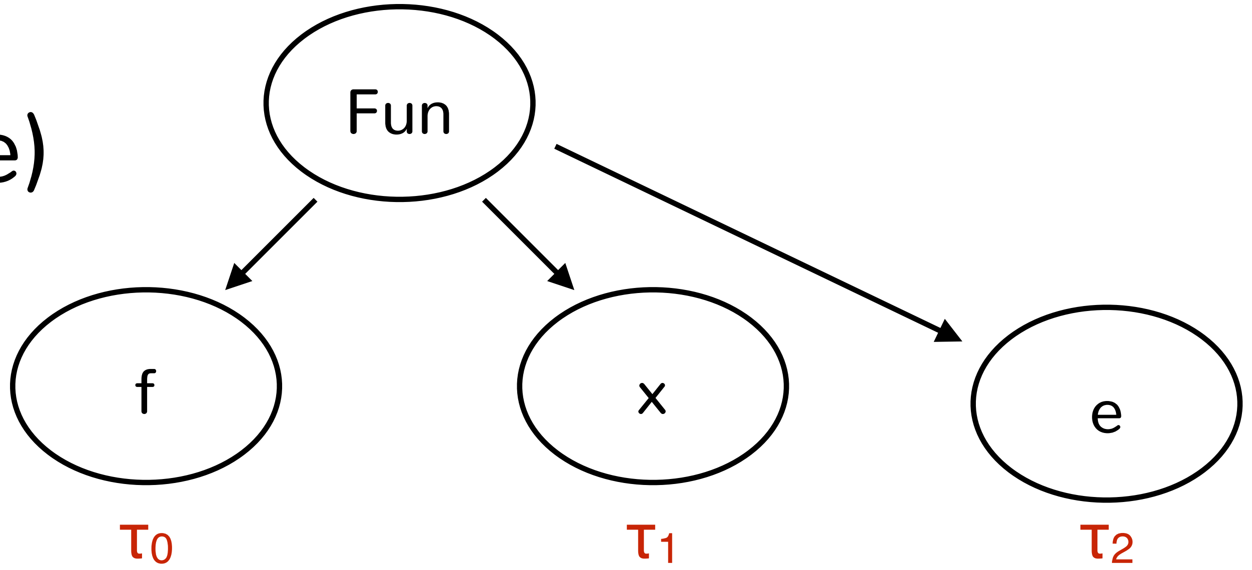
Generating constraints

- Lambda abstraction (λx.e)
    - $\tau_0 = \tau_1 \rightarrow \tau_2$



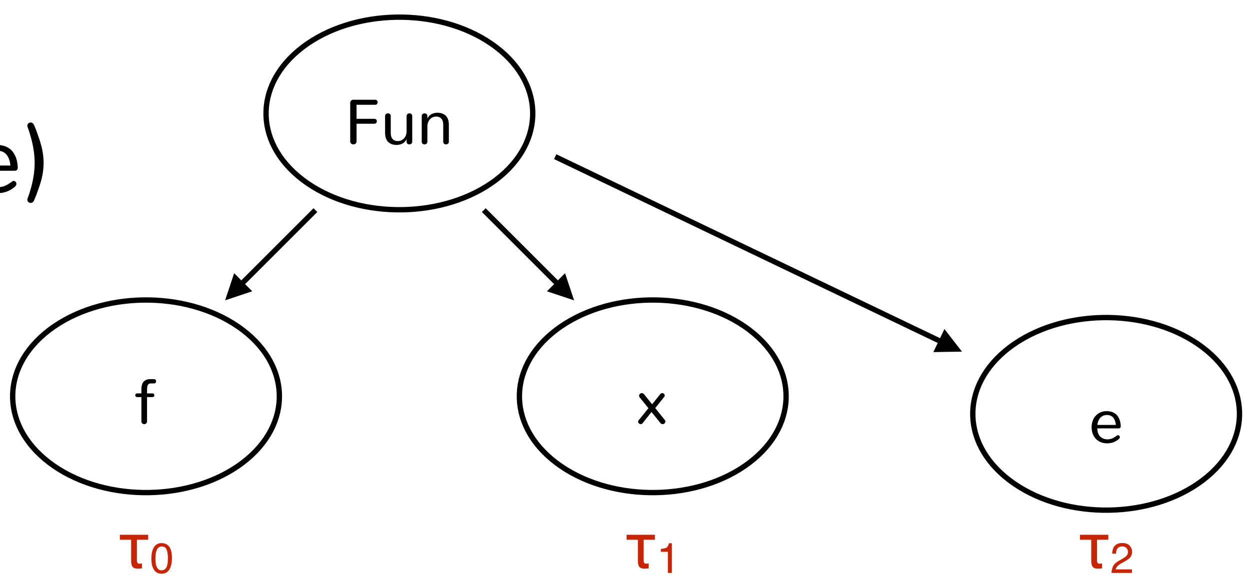
Generating constraints

- Function declaration (f x = e)
  - $\tau_0 =$



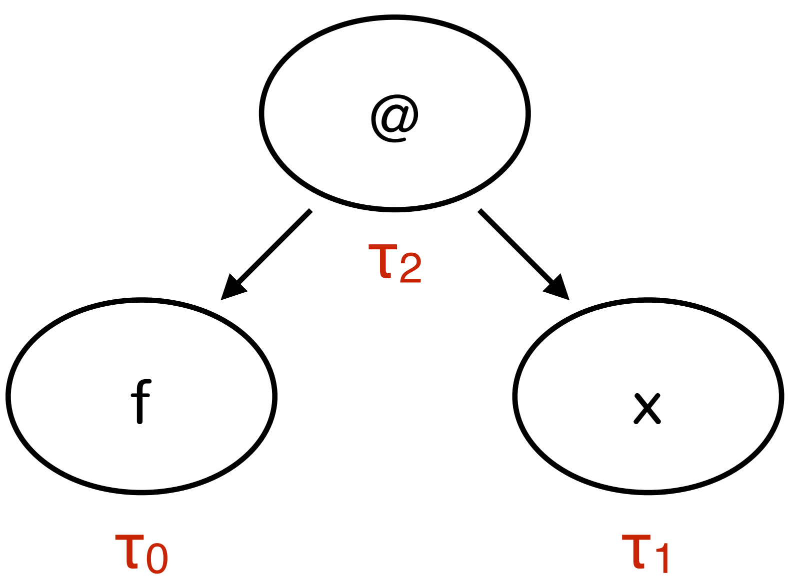
Generating constraints

- Function declaration (f x = e)
    - ➤ $\tau_0 = \tau_1 \rightarrow \tau_2$



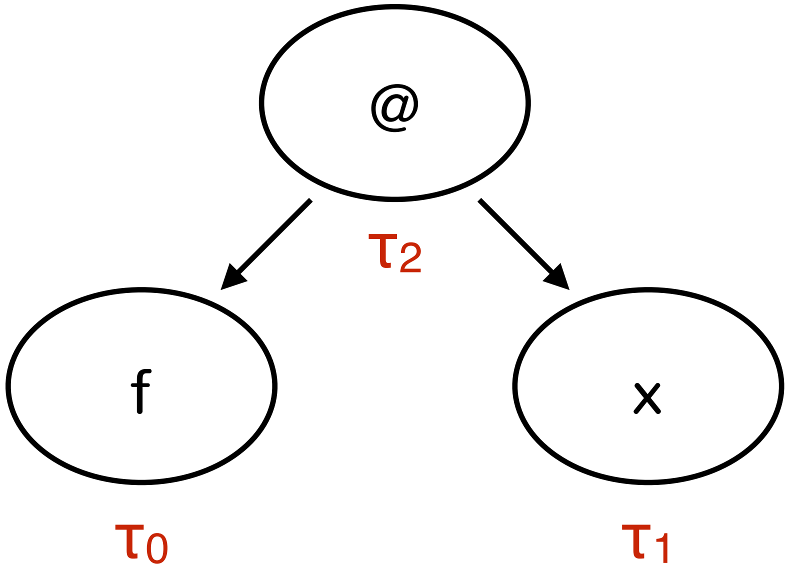
Generating constraints

- Function application (f x)
  - $\tau_0 =$



Generating constraints

- Function application (f x)
  - $\tau_0 = \tau_1 \rightarrow \tau_2$



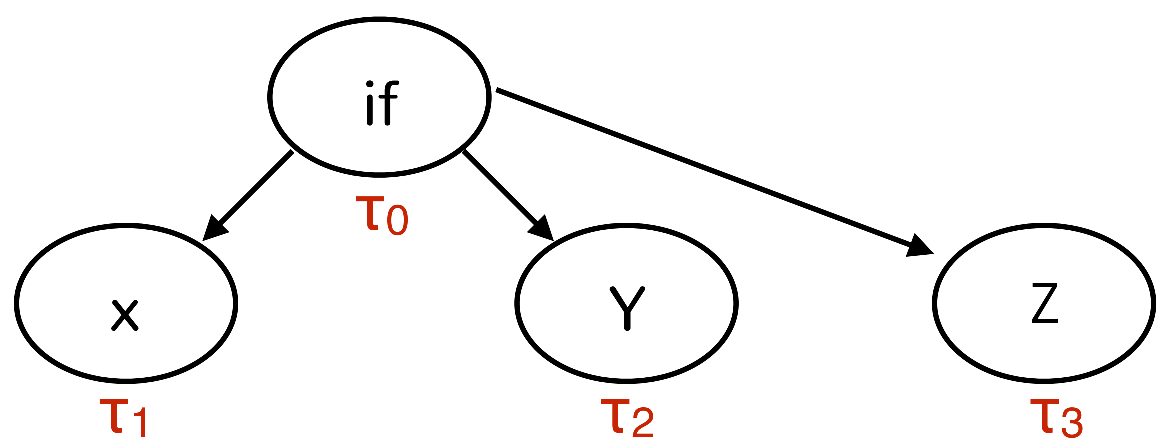
Generating constraints

- Conditionals `if x then y else y`

    ➤

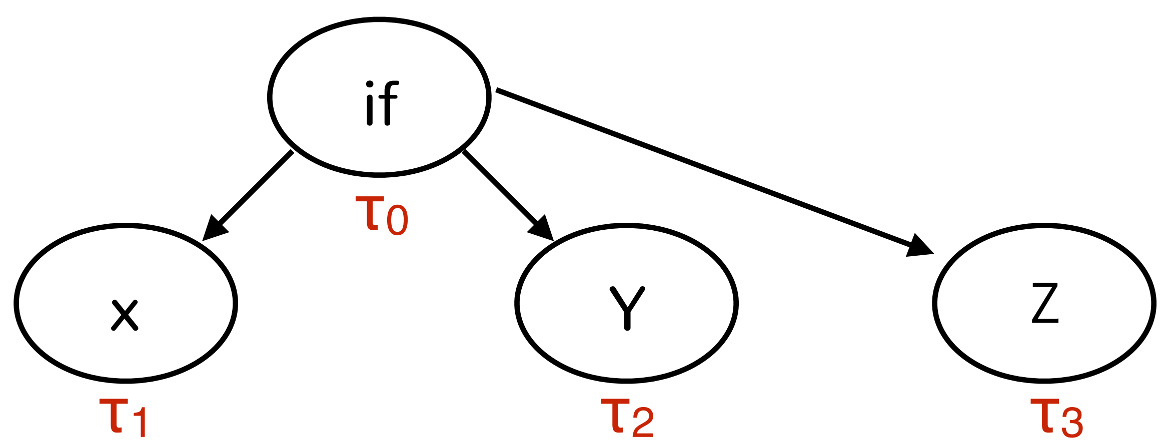
    ➤

    ➤



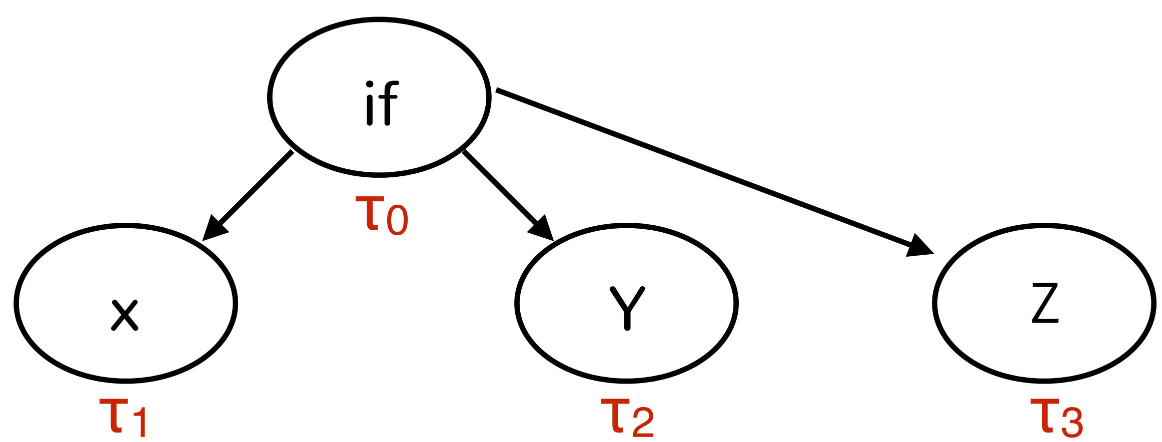
Generating constraints

- Conditionals `if x then y else y`
  - $\tau_1 = \texttt{Bool}$
  -
  -



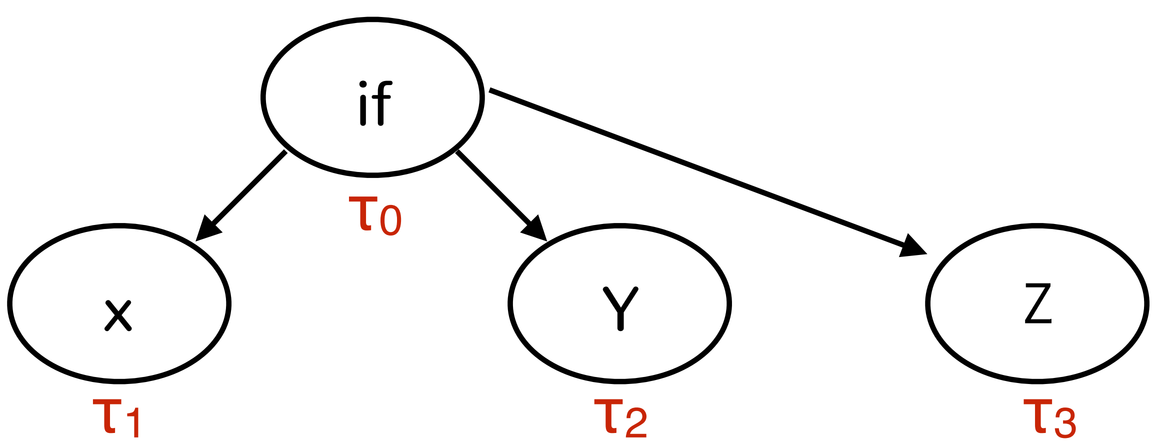
Generating constraints

- Conditionals `if x then y else y`
  - $\tau_1 = \texttt{Bool}$
  - $\tau_0 = \tau_2$
  -



Generating constraints

- Conditionals `if x then y else y`
  - τ_1 = `Bool`
  - $\tau_0 = \tau_2$
  - $\tau_2 = \tau_3$



Ex1. Basic idea

- Step 4: solve constraints via unification

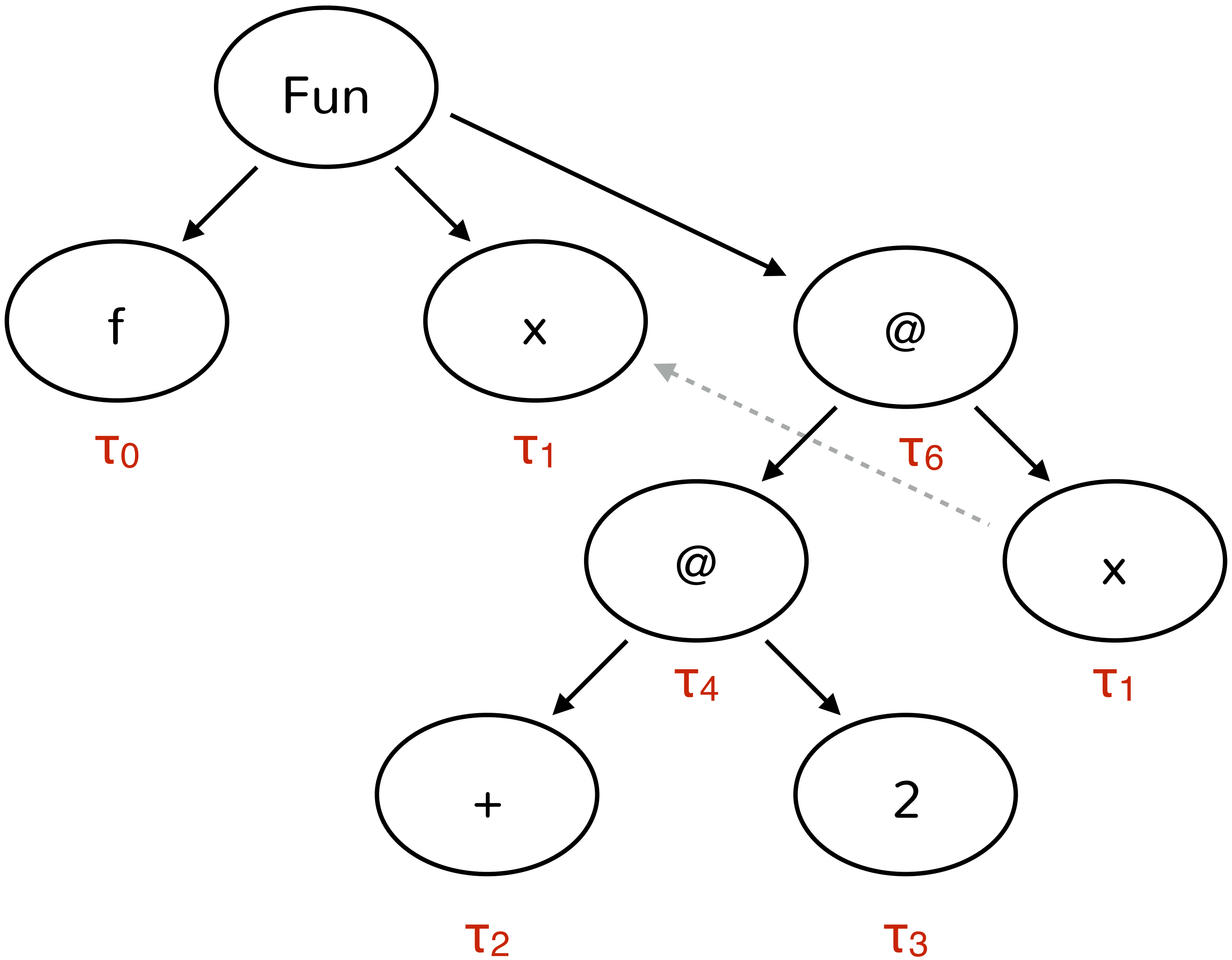
$\tau_0 = \tau_1 \text{ -> } \tau_6$

$\tau_2 = \tau_3 \text{ -> } \tau_4$

$\tau_4 = \tau_1 \text{ -> } \tau_6$

$\tau_2 = \texttt{Int->Int->Int}$

$\tau_3 = \texttt{Int}$



Ex1. Basic idea

- Step 5: read out type

τ_0 = `Int->Int`
τ_1 = `Int`
τ_2 = `Int->Int->Int`
τ_3 = `Int`
τ_4 = `Int->Int`
τ_6 = `Int`

`f ::` τ_0
`f ::Int->Int->Int`

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 types of top-level declarations