〈Type Inference Rules〉

1. Data constructor is a function too.

```
data Maybe a
  = Nothing
  | Just a
```

`Just :: a → Maybe a`

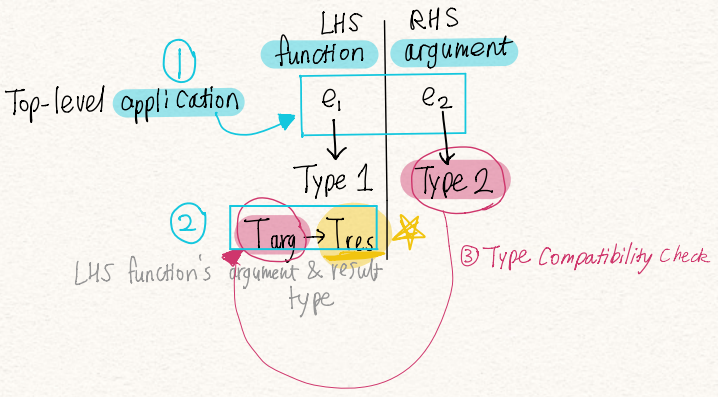
'Just' turns 'a' into 'Maybe a'

2-1. Split the top-level app. into LHS & RHS.

(function) (argument)

2-2. Split the function in LHS into Targ → Tres.

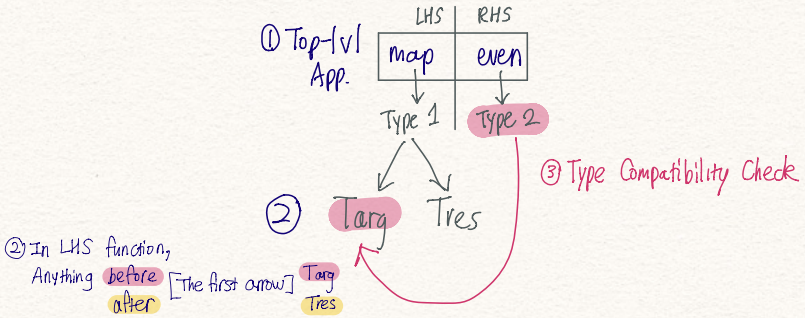
2-3. Check if the argument in RHS (Type 2) is compatible w/ the function in LHS.



2-4. Function type is determined by Tres.

The type of its result in LHS

ex> :t map even



Top Level App.

LHS = map :: (a → b) → [a] → [b] (Targ | Tres)

RHS = even :: Int → Bool (Type 2)

(Type 1)

Type Assignment Δ : {a = Int, b = Bool}

∴ The type of (map even) is Tres = [a] → [b].

By type assignment, map even :: [Int] → [Bool]

3. Function Composition

3-1. Layered

Equiv.

not . even ⟺ ((•) | not) even

Type 1 | Type 2

Targ | Tres

Compatibility Check

Step 1.

(•) :: (b → c) → (a → b) → a → c

Targ | Tres

not :: Bool → Bool

* Compatibility Check: Targ from Type 1 & The entire Type 2

Type assignment

Δ : {b = Bool, c = Bool}

Step 2.

(•) not :: (a → b) → a → c

derived from Tres

↓ by Type Assignment

(•) not :: (a → Bool) → a → Bool

T'arg | T'res

even :: Int → Bool

Type assignment : {a = Int}

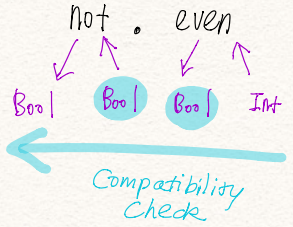
Step 3.

(•) not even :: a → Bool

derived from T'res

(•) not even :: Int → Bool

3-2. Pipeline Approach



∴ not . even :: Int → Bool

cf) even . not

Bool | Int | Bool | Bool

Type Error

3-3. Function Composition result is also a function w/ arrow (s)!