## Type Inference

Figuring out an expression's type without help from the programmer

Type Checking VS Type Synthesis VS Type Inference The difference between Checking and Synthesis:

## The inputs/outputs

Checking: program and type -> Boolean Synthesis : program -> type

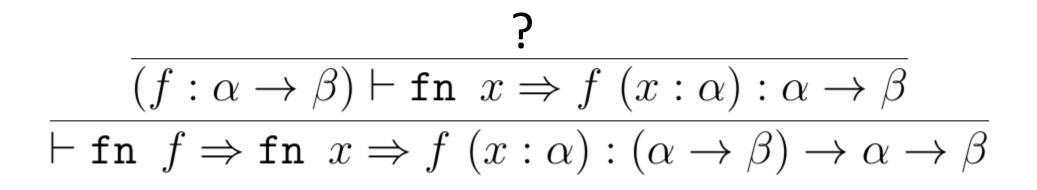
#### An example

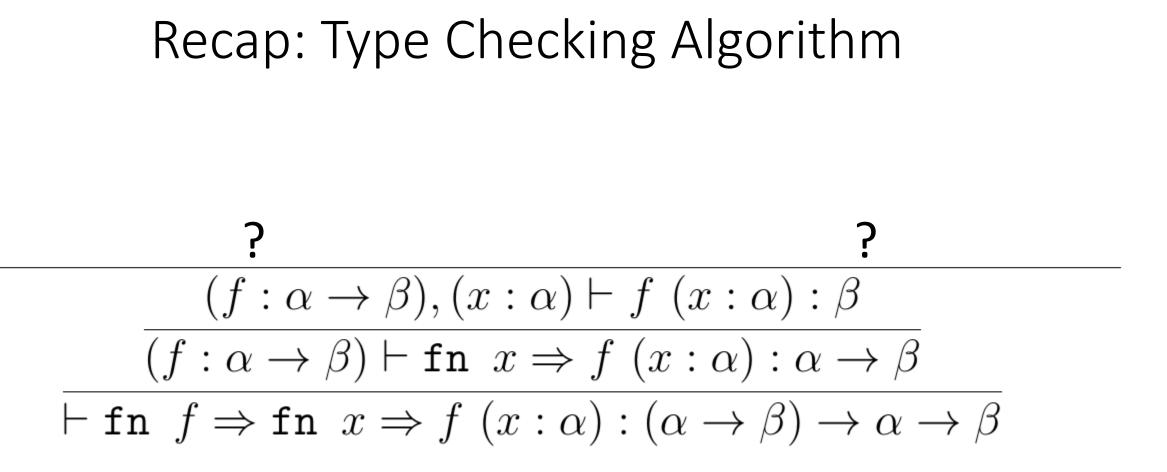
#### fn $f \Rightarrow$ fn $x \Rightarrow f x$ : ?

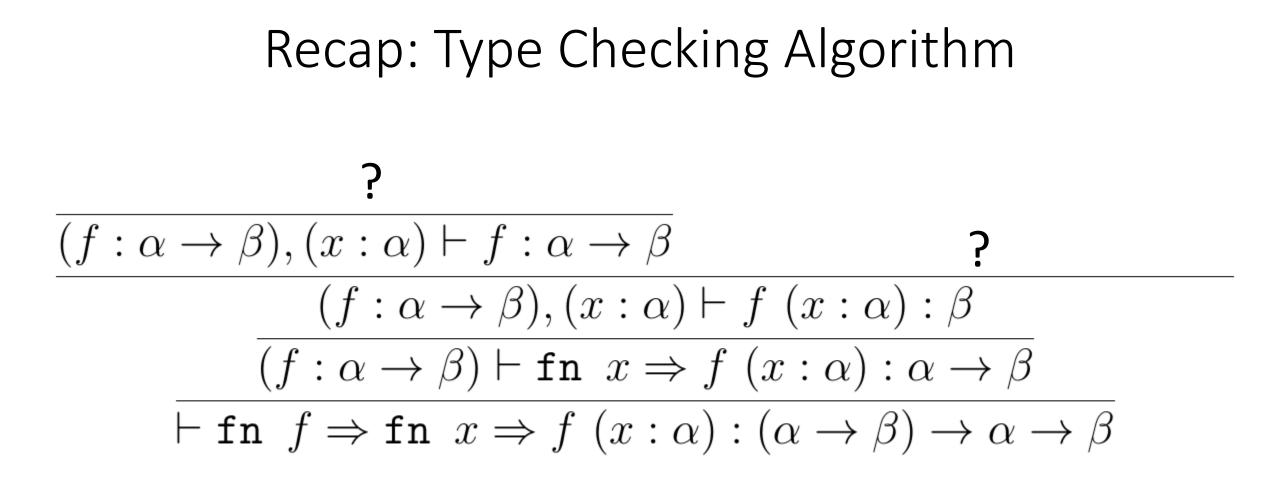
$$\vdash \underbrace{\operatorname{fn} \ f \Rightarrow \operatorname{fn} \ x \Rightarrow f \ (x:\alpha)}_{\text{input}} : \underbrace{(\alpha \to \beta) \to \alpha \to \beta}_{\text{input}}$$

?  

$$\vdash \texttt{fn} \ f \Rightarrow \texttt{fn} \ x \Rightarrow f \ (x : \alpha) : (\alpha \to \beta) \to \alpha \to \beta$$







$$\begin{array}{c} \mbox{Recap: Type Checking Algorithm} \\ \hline \hline $I$ \\ \hline $(f:\alpha \to \beta), (x:\alpha) \vdash f:\alpha \to \beta$ \\ \hline $(f:\alpha \to \beta), (x:\alpha) \vdash f(x:\alpha):\beta$ \\ \hline $(f:\alpha \to \beta) \vdash fn \ x \Rightarrow f(x:\alpha):\alpha \to \beta$ \\ \hline $\vdash fn \ f \Rightarrow fn \ x \Rightarrow f(x:\alpha): (\alpha \to \beta) \to \alpha \to \beta$ \end{array}$$

$$\begin{array}{c} (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f:\alpha \rightarrow \beta & ?\\ \hline & (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f \ (x:\alpha):\beta \\ \hline & (f:\alpha \rightarrow \beta) \vdash \texttt{fn} \ x \Rightarrow f \ (x:\alpha):\alpha \rightarrow \beta \\ \hline & \vdash \texttt{fn} \ f \Rightarrow \texttt{fn} \ x \Rightarrow f \ (x:\alpha): (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \end{array}$$

$$\begin{array}{l} \mbox{Recap: Type Checking Algorithm} \\ \hline \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f:\alpha \rightarrow \beta & \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash x:\alpha \\ \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f \ (x:\alpha):\beta \\ \hline \hline (f:\alpha \rightarrow \beta) \vdash \mbox{fn} \ x \Rightarrow f \ (x:\alpha):\alpha \rightarrow \beta \\ \hline \vdash \mbox{fn} \ f \Rightarrow \mbox{fn} \ x \Rightarrow f \ (x:\alpha): (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \end{array}$$

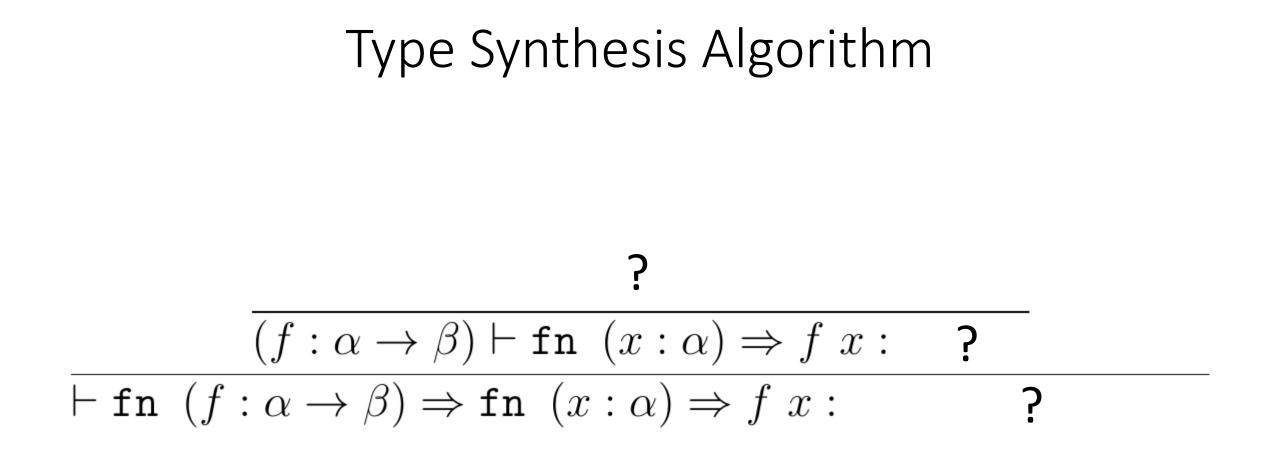
$$\begin{array}{l} \mbox{Recap: Type Checking Algorithm} \\ \hline \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f:\alpha \rightarrow \beta & \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash x:\alpha \\ \hline \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f \ (x:\alpha):\beta \\ \hline \hline (f:\alpha \rightarrow \beta) \vdash \mbox{fn} \ x \Rightarrow f \ (x:\alpha):\alpha \rightarrow \beta \\ \hline \vdash \mbox{fn} \ f \Rightarrow \mbox{fn} \ x \Rightarrow f \ (x:\alpha): (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \end{array}$$

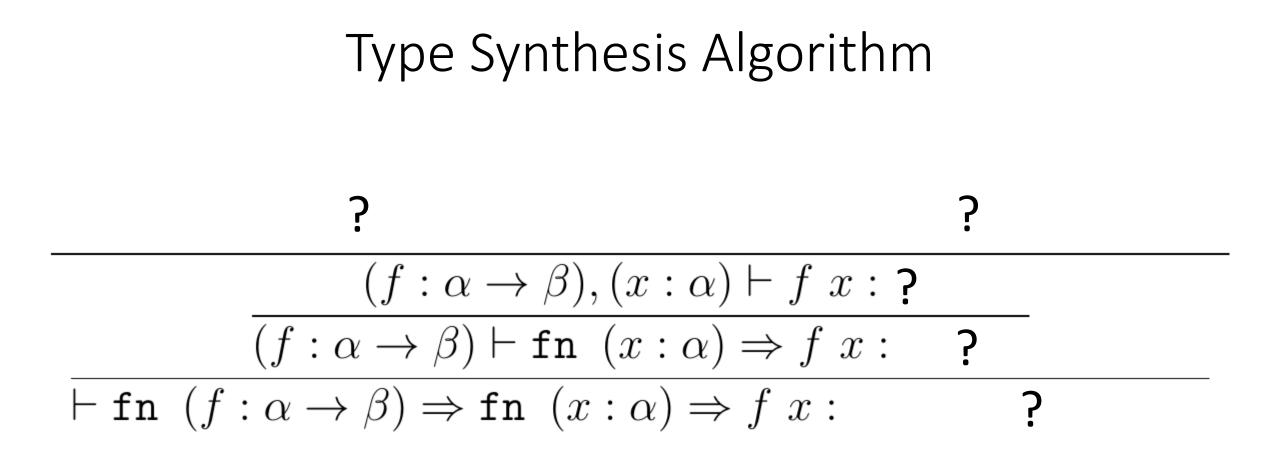
$$\begin{array}{c} (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f:\alpha \rightarrow \beta & \hline{(f:\alpha \rightarrow \beta), (x:\alpha) \vdash x:\alpha} \\ \\ \hline{(f:\alpha \rightarrow \beta), (x:\alpha) \vdash f \ (x:\alpha):\beta} \\ \hline{(f:\alpha \rightarrow \beta) \vdash \texttt{fn} \ x \Rightarrow f \ (x:\alpha):\alpha \rightarrow \beta} \\ \hline{\vdash \texttt{fn} \ f \Rightarrow \texttt{fn} \ x \Rightarrow f \ (x:\alpha): (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta} \end{array}$$

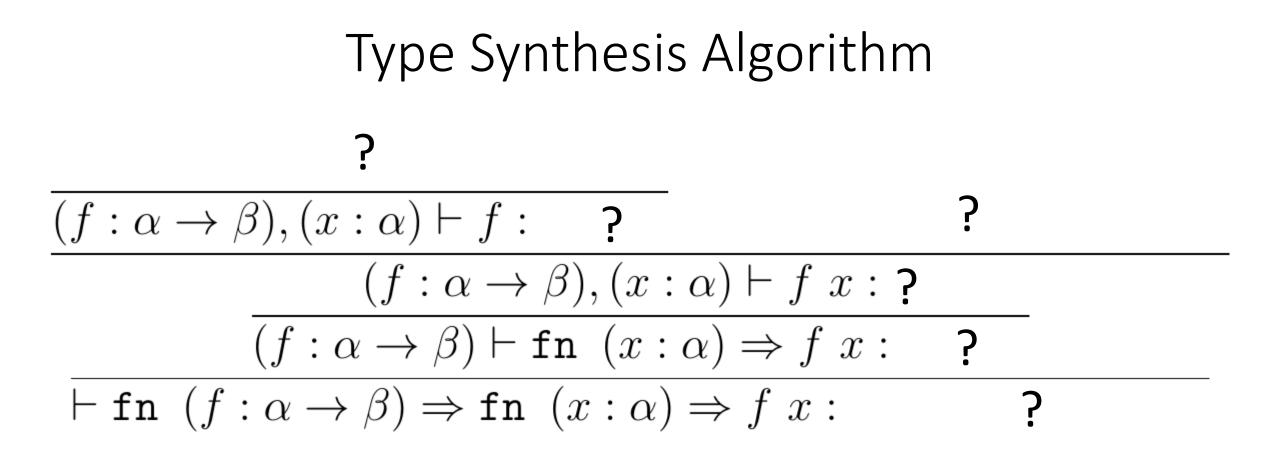
$$\vdash \underbrace{\operatorname{fn} \ (f: \alpha \to \beta) \Rightarrow \operatorname{fn} \ (x: \alpha) \Rightarrow f \ x:}_{\operatorname{INPUT}}$$

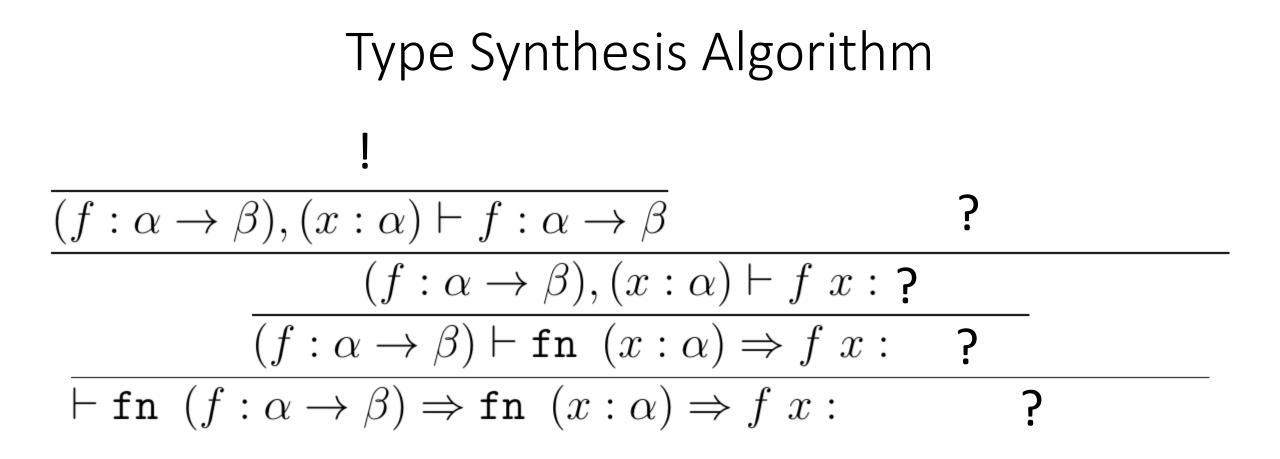
#### ?

$$\vdash \texttt{fn} \ (f: \alpha \to \beta) \Rightarrow \texttt{fn} \ (x: \alpha) \Rightarrow f \ x:$$









$$\begin{array}{c} \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f:\alpha \rightarrow \beta & ? \\ \hline & (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f \; x:? \\ \hline & \hline (f:\alpha \rightarrow \beta) \vdash \texttt{fn} \; (x:\alpha) \Rightarrow f \; x: & ? \\ \hline & \vdash \texttt{fn} \; (f:\alpha \rightarrow \beta) \Rightarrow \texttt{fn} \; (x:\alpha) \Rightarrow f \; x: & ? \end{array}$$

### Type Synthesis Algorithm $(f: \alpha \to \beta), (x: \alpha) \vdash f: \alpha \to \beta \quad (f: \alpha \to \beta), (x: \alpha) \vdash x: \mathbf{P}$ $(f: \alpha \to \beta), (x: \alpha) \vdash f x: \mathbf{?}$ $(f: \alpha \to \beta) \vdash \texttt{fn} \ (x: \alpha) \Rightarrow f \ x:$ ? $\vdash$ fn $(f: \alpha \to \beta) \Rightarrow$ fn $(x: \alpha) \Rightarrow f x:$ 2

$$\begin{array}{c} \mbox{Type Synthesis Algorithm} \\ \hline \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f:\alpha \rightarrow \beta & \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash x:\alpha \\ \hline \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f x:? \\ \hline \hline (f:\alpha \rightarrow \beta) \vdash \mbox{fn} \ (x:\alpha) \Rightarrow f x:? \\ \hline \vdash \mbox{fn} \ (f:\alpha \rightarrow \beta) \Rightarrow \mbox{fn} \ (x:\alpha) \Rightarrow f x:? \end{array}$$

$$\begin{array}{c} \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f:\alpha \rightarrow \beta & \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash x:\alpha \\ \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f x: \ref{algebra} \\ \hline (f:\alpha \rightarrow \beta) \vdash \texttt{fn} \ (x:\alpha) \Rightarrow f x: \ref{algebra} \\ \hline \texttt{fn} \ (f:\alpha \rightarrow \beta) \Rightarrow \texttt{fn} \ (x:\alpha) \Rightarrow f x: \ref{algebra} \end{array}$$

$$\begin{array}{c} \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f:\alpha \rightarrow \beta & \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash x:\alpha \\ \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f \; x:\beta \\ \hline (f:\alpha \rightarrow \beta) \vdash \texttt{fn} \; (x:\alpha) \Rightarrow f \; x: & \ref{algebra} \\ \hline \vdash \texttt{fn} \; (f:\alpha \rightarrow \beta) \Rightarrow \texttt{fn} \; (x:\alpha) \Rightarrow f \; x: & \ref{algebra} \end{array}$$

$$\begin{array}{c} \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f:\alpha \rightarrow \beta & \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash x:\alpha \\ \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f \; x:\beta \\ \hline (f:\alpha \rightarrow \beta) \vdash \texttt{fn} \; (x:\alpha) \Rightarrow f \; x:\alpha \rightarrow \beta \\ \hline \vdash \texttt{fn} \; (f:\alpha \rightarrow \beta) \Rightarrow \texttt{fn} \; (x:\alpha) \Rightarrow f \; x: & \ree{} \end{array}$$

$$\begin{array}{c} \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f:\alpha \rightarrow \beta & \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash x:\alpha \\ \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f \; x:\beta \\ \hline (f:\alpha \rightarrow \beta) \vdash \texttt{fn} \; (x:\alpha) \Rightarrow f \; x:\alpha \rightarrow \beta \\ \hline \vdash \texttt{fn} \; (f:\alpha \rightarrow \beta) \Rightarrow \texttt{fn} \; (x:\alpha) \Rightarrow f \; x:(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \end{array}$$

$$\begin{array}{c} (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f:\alpha \rightarrow \beta & \overline{(f:\alpha \rightarrow \beta)}, (x:\alpha) \vdash x:\alpha \\ \hline (f:\alpha \rightarrow \beta), (x:\alpha) \vdash f \; x:\beta \\ \hline (f:\alpha \rightarrow \beta) \vdash \texttt{fn} \; (x:\alpha) \Rightarrow f \; x:\alpha \rightarrow \beta \\ \hline \vdash \texttt{fn} \; (f:\alpha \rightarrow \beta) \Rightarrow \texttt{fn} \; (x:\alpha) \Rightarrow f \; x:(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \end{array}$$

**Output:** 
$$(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

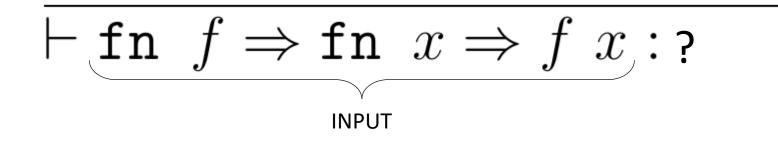
The difference between Synthesis and Inference

## Type annotations in the programming language

If we can do type inference, then the programmer doesn't need to annotate types

# Let's try to feel out an algorithm for inference

Starting point: it'll be a lot like synthesis

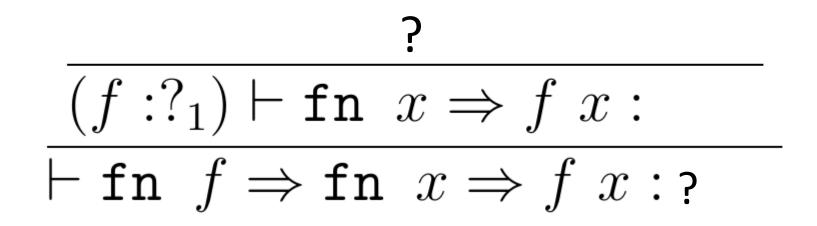


#### What type do we put in the context for f?

#### UHHH IDK..... LETS GUESS!

 $\vdash fn \quad f \Rightarrow fn \quad x \Rightarrow f \quad x : ?$ 

?  $\vdash \texttt{fn} \ f \Rightarrow \texttt{fn} \ x \Rightarrow f \ x : ?$ 



 $\begin{array}{c} ?\\ \hline (f:?_1), (x:?_2) \vdash f \ x:\\ \hline (f:?_1) \vdash \texttt{fn} \ x \Rightarrow f \ x:\\ \hline \vdash \texttt{fn} \ f \Rightarrow \texttt{fn} \ x \Rightarrow f \ x:? \end{array}$ 

### Trying to infer types Now what? We need ?<sub>1</sub> to be an arrow type $(f:?_1), (x:?_2) \vdash f:?_1$ $(f:?_1), (x:?_2) \vdash f x:$ $(f:?_1) \vdash \texttt{fn} \ x \Rightarrow f \ x:$ $\vdash fn f \Rightarrow fn x \Rightarrow f x : ?$

Trying to infer types Make new guess variables  $?_3$  and  $?_4$ "Force" the equality  $?_1 = ?_3 \rightarrow ?_4$  $(f:?_1), (x:?_2) \vdash f:?_1$  $(f:?_1), (x:?_2) \vdash f x:$  $(f:?_1) \vdash \texttt{fn} \ x \Rightarrow f \ x:$  $\vdash fn f \Rightarrow fn x \Rightarrow f x : ?$ 

Trying to infer types  
Make new guess variables ?<sub>3</sub> and ?<sub>4</sub>  
"Force" the equality ?<sub>1</sub> = ?<sub>3</sub> -> ?<sub>4</sub>  

$$\frac{(f:?_3 \rightarrow ?_4), (x:?_2) \vdash f:?_3 \rightarrow ?_4}{(f:?_3 \rightarrow ?_4), (x:?_2) \vdash f x:?}$$

$$\frac{(f:?_3 \rightarrow ?_4) \vdash \text{fn } x \Rightarrow f x:?}{\vdash \text{fn } f \Rightarrow \text{fn } x \Rightarrow f x:?}$$

$$\begin{array}{c} \hline \hline (f:?_3 \rightarrow ?_4), (x:?_2) \vdash f:?_3 \rightarrow ?_4 & \hline (f:?_3 \rightarrow ?_4), (x:?_2) \vdash x:?_2 \\ \\ \hline \hline (f:?_3 \rightarrow ?_4), (x:?_2) \vdash f \; x: ? \\ \hline \hline \hline (f:?_3 \rightarrow ?_4) \vdash \texttt{fn} \; x \Rightarrow f \; x: ? \\ \hline \vdash \texttt{fn} \; f \Rightarrow \texttt{fn} \; x \Rightarrow f \; x: ? \end{array}$$

# Trying to infer types "Force" the equality $?_2 = ?_3$

$$\frac{(f:?_3 \rightarrow ?_4), (x:?_2) \vdash f:?_3 \rightarrow ?_4 \quad (f:?_3 \rightarrow ?_4), (x:?_2) \vdash x:?_2}{(f:?_3 \rightarrow ?_4), (x:?_2) \vdash f x:?}$$

$$\frac{(f:?_3 \rightarrow ?_4) \vdash \operatorname{fn} x \Rightarrow f x:?}{\vdash \operatorname{fn} f \Rightarrow \operatorname{fn} x \Rightarrow f x:?}$$

$$\begin{array}{c} \hline (f:?_3 \rightarrow ?_4), (x:?_3) \vdash f:?_3 \rightarrow ?_4 & \hline (f:?_3 \rightarrow ?_4), (x:?_3) \vdash x:?_3 \\ \\ \hline (f:?_3 \rightarrow ?_4), (x:?_3) \vdash f \ x:?_3 \\ \hline (f:?_3 \rightarrow ?_4) \vdash \texttt{fn} \ x \Rightarrow f \ x: & ? \\ \hline \vdash \texttt{fn} \ f \Rightarrow \texttt{fn} \ x \Rightarrow f \ x: & ? \end{array}$$

$$\begin{array}{c} \hline (f:?_3 \rightarrow ?_4), (x:?_3) \vdash f:?_3 \rightarrow ?_4 \\ \hline (f:?_3 \rightarrow ?_4), (x:?_3) \vdash f x:?_4 \\ \hline (f:?_3 \rightarrow ?_4), (x:?_3) \vdash f x:?_4 \\ \hline (f:?_3 \rightarrow ?_4) \vdash \operatorname{fn} x \Rightarrow f x: & ? \\ \hline \vdash \operatorname{fn} f \Rightarrow \operatorname{fn} x \Rightarrow f x: & ? \end{array}$$

$$\begin{array}{c} \hline (f:?_3 \rightarrow ?_4), (x:?_3) \vdash f:?_3 \rightarrow ?_4 \\ \hline (f:?_3 \rightarrow ?_4), (x:?_3) \vdash f x:?_4 \\ \hline (f:?_3 \rightarrow ?_4) \vdash \texttt{fn} \ x \Rightarrow f \ x:?_3 \rightarrow ?_4 \\ \hline \vdash \texttt{fn} \ f \Rightarrow \texttt{fn} \ x \Rightarrow f \ x: (?_3 \rightarrow ?_4) \rightarrow ?_3 \rightarrow ?_4 \end{array}$$

# I think we need to talk about this "forcing equality" stuff...

It seems like it could be complicated, so let's be more formal about it.

Unification: A problem in math/CS

- Input: a pair of terms
- •Output: a list of mappings from term variables to terms, such that applying the mappings to the terms results in the same term

# Example of unification:

•Input:  $\alpha \doteq \beta \times \gamma$ 

## Example of unification:

• Input: 
$$\alpha \doteq \beta \times \gamma$$

# • Output: $[\alpha \mapsto \beta \times \gamma]$

# Example of unification:

• Input: 
$$\alpha \doteq \beta \times \gamma$$

• Output: 
$$[\alpha \mapsto \beta \times \gamma]$$

• Result of applying the mappings:

$$\beta \times \gamma \doteq \beta \times \gamma$$

# Is unification always possible?

#### Is unification always possible?

• Input: 
$$lpha imes eta \doteq \gamma + \delta$$

#### Is unification always possible?

• Input: 
$$\alpha imes eta \doteq \gamma + \delta$$

Cannot unify; terms have different head symbols.

• No, unification is not always possible.

# How else can unification fail?

# How else can unification fail?

• Input: 
$$\alpha \doteq \alpha \times \beta$$

# How else can unification fail?

• Input: 
$$\alpha \doteq \alpha \times \beta$$

Cannot unify; circularity

• There are two ways for unification to fail:

head symbol conflict and circularity.

• Input: 
$$\alpha \doteq \beta$$

• Input: 
$$\alpha \doteq \beta$$

• Many valid outputs:

$$[\alpha \mapsto \beta],$$
$$\beta \doteq \beta$$

• Input: 
$$\alpha \doteq \beta$$

• Many valid outputs:

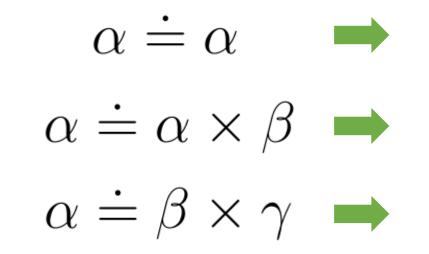
$$[\alpha \mapsto \beta], [\beta \mapsto \alpha], \\ \downarrow \\ \beta \doteq \beta \quad \alpha \doteq \alpha$$

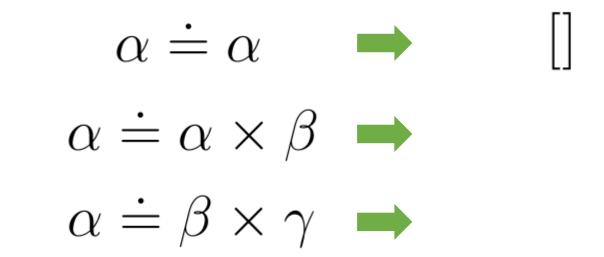
• Input: 
$$\alpha \doteq \beta$$

• Many valid outputs:

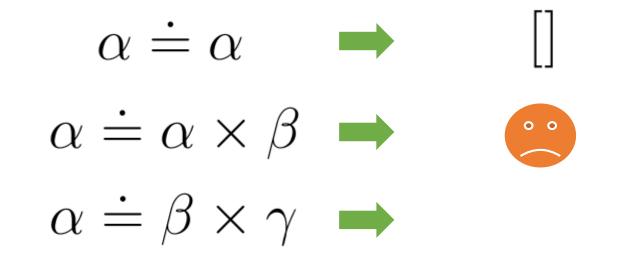
$$[\alpha \mapsto \beta], [\beta \mapsto \alpha], [\alpha \mapsto \gamma, \beta \mapsto \gamma]$$
  
$$\beta \doteq \beta \quad \alpha \doteq \alpha \qquad \gamma \doteq \gamma$$

# Coming up with an algorithm for unification



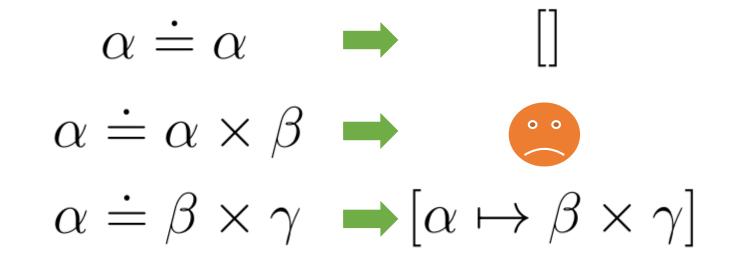


If the other term is the same variable, output []



If the other term is the same variable, output []

Otherwise, if the other term contains the variable, circularity error.



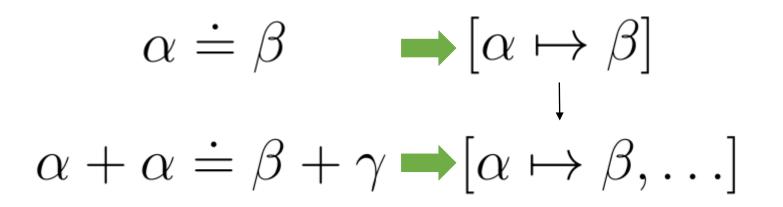
If the other term is the same variable, output []

Otherwise, if the other term contains the variable, circularity error. Otherwise, just map the variable to the other term.

 $\alpha + \alpha \doteq \beta + \gamma \blacksquare$ 

$$\begin{array}{c} \alpha \doteq \beta \\ \uparrow \\ \alpha + \alpha \doteq \beta + \gamma \end{array}$$

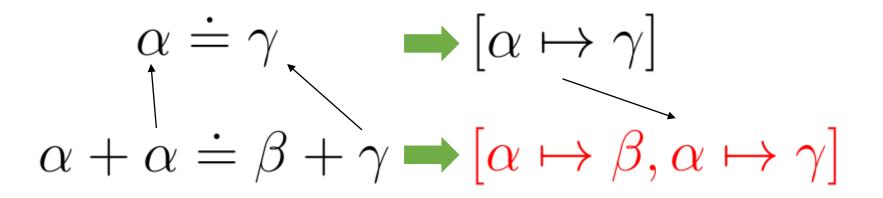
$$\alpha \doteq \beta \qquad \Longrightarrow [\alpha \mapsto \beta]$$
$$\alpha + \alpha \doteq \beta + \gamma \Longrightarrow$$



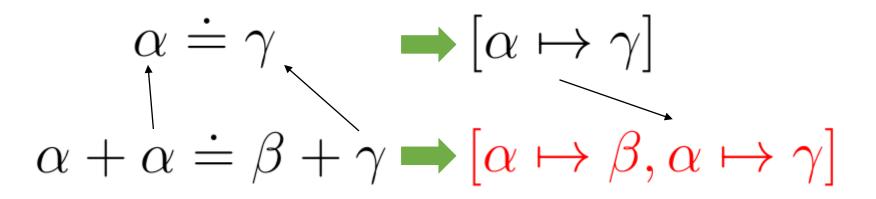
# $\alpha + \alpha \doteq \beta + \gamma \Longrightarrow [\alpha \mapsto \beta, \ldots]$

## $\alpha + \alpha \doteq \beta + \gamma \Longrightarrow [\alpha \mapsto \beta, \ldots]$

- 1. Unify the first subterms.
- 2. Unify the second subterms?

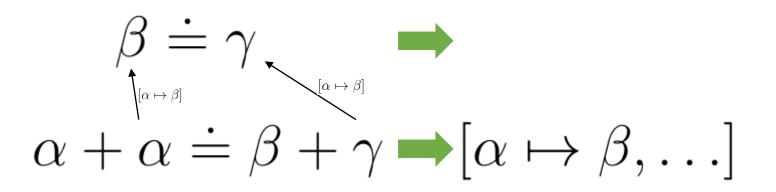


- 1. Unify the first subterms.
- 2. Unify the second subterms? Wait... that doesn't make sense

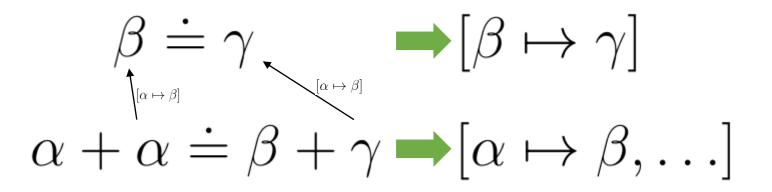


- 1. Unify the first subterms.
- 2. Unify the second subterms? Wait... that doesn't make sense

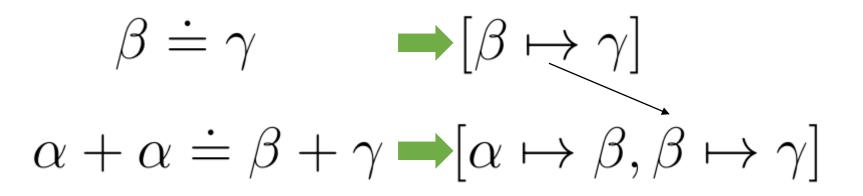
Solution: Apply the mappings from unifying the first subterms to the second subterms before unifying them.



- 1. Unify the first subterms.
- 2. Apply the mappings from unifying the first subterms to the second subterms, then unify the second subterms.



- 1. Unify the first subterms.
- 2. Apply the mappings from unifying the first subterms to the second subterms, then unify the second subterms.



- 1. Unify the first subterms.
- 2. Apply the mappings from unifying the first subterms to the second subterms, then unify the second subterms.

## $\alpha + \alpha \doteq \beta + \gamma \Longrightarrow [\alpha \mapsto \beta, \beta \mapsto \gamma]$

- 1. Unify the first subterms.
- 2. Apply the mappings from unifying the first subterms to the second subterms, then unify the second subterms.

#### Case 3: Terms have different head symbols

 $\alpha \times \beta \doteq \gamma + \delta \Longrightarrow$ 

#### Case 3: Terms have different head symbols

$$\alpha \times \beta \doteq \gamma + \delta \Longrightarrow \bigcirc$$

Head symbol conflict.

- DON'T CARE ABOUT EFFICIENCY
  - Type inference <u>can be</u> done in linear time (with respect to program size) but that's **not fun**
- Focus on maintaining invariants about which type variables can appear where
- Keep the difference between expression terms and type terms straight
  - Unification happens to type terms and maps type variables to other types
  - Expression terms (including expression variables) have types, which are type terms which contain type variables.

- At any point during the execution of the algorithm, every type variable will fall into one of two categories:
  - Still allowed to appear in our guess types
  - Has been "unified away" and should no longer show up in any types in our context
- A variable moves from the first category to the second category when the unification algorithm maps it to something
- Take care to apply the variable mappings returned by unification, so that variables that have been "unified away" don't reappear or get mapped twice

- Consider using a dictionary to store the type that each "unified away" variable is currently mapped to
- Your inference algorithm will then be passing two dictionaries around: one representing the context which maps expression variables to types, and one representing the mapping from "unified away" type variables to types
- You'll want to maintain an invariant like: if a variable has been "unified away", it cannot appear in any type mapped to by either dictionary
- Every time you call the unification algorithm, apply the mappings (in left-to-right order) to the type variable dictionary

- Whenever you want to force a type to be of a certain form generate new (not used anywhere else) type variables and unify
  - e.g. say we want to force a type  $\tau$  to be an arrow type.
  - Create fresh type variables  $\alpha$  and  $\beta$ , then unify:

$$\tau \doteq \alpha \to \beta$$

# Questions?