We have learned

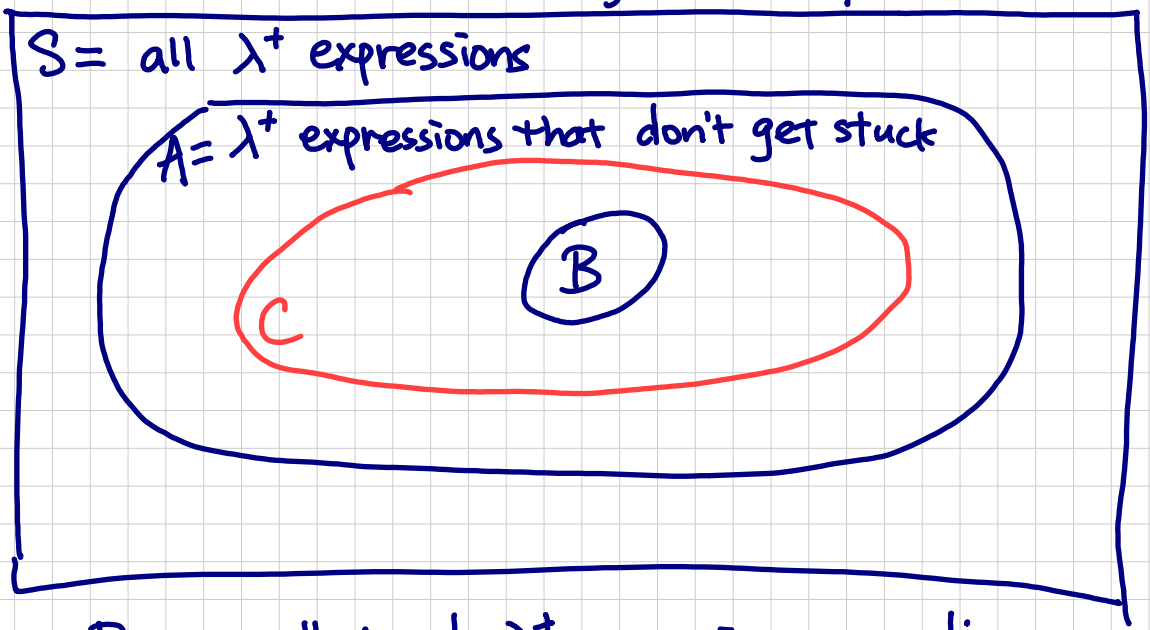
$e \Downarrow v$ and $\Gamma \vdash e : T$

"How to execute a program" Type checking

Question Why do we need type checking?

↳ To predict runtime behaviors w/o running the program.

↳ In particular, we make sure our predictions are sound (but maybe incomplete):



B = well-typed λ^+ expressions according to type checking rules.

Problem with B

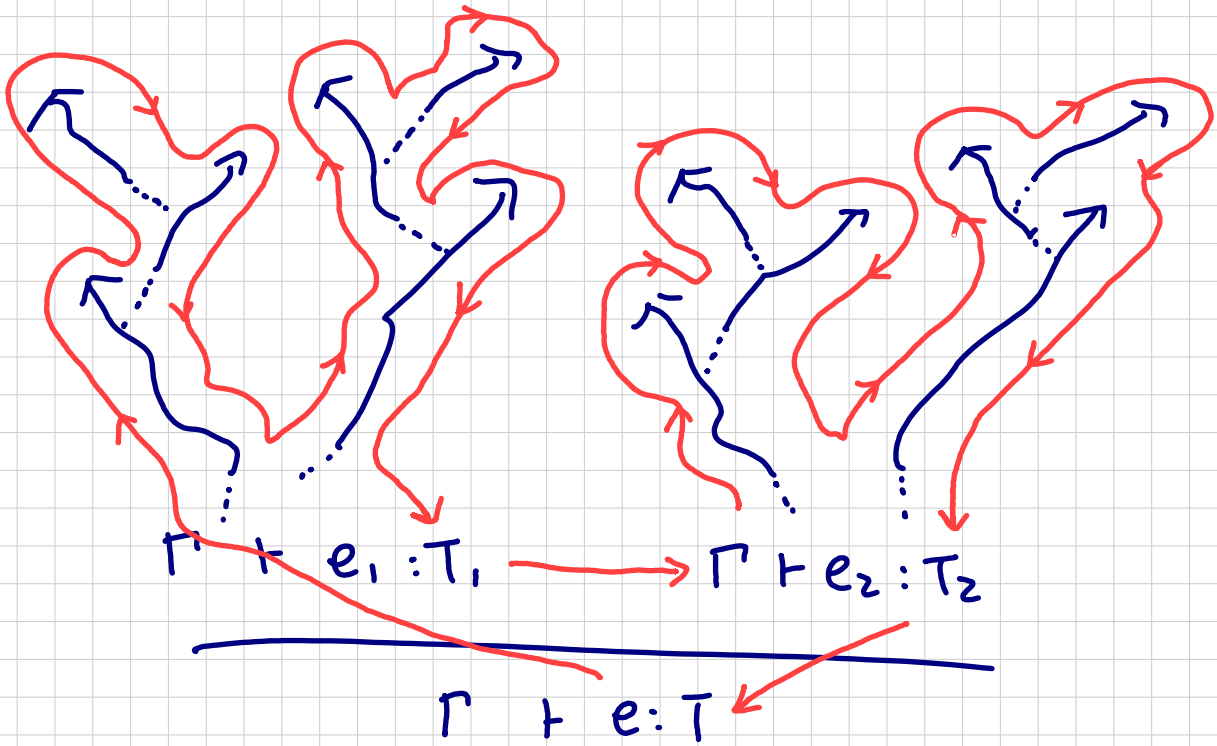
1. Too much annotation burden on the programmer
2. It's way too restrictive: a lot of interesting programs are in $A \setminus B$.

Goal: Enlarge B into C using type inference.

To motivate our type inference algorithm, let's consider:

WHY is B so small compared to A?

To answer this question, consider a typical type checking derivation (which is basically what you implemented in HW4).



Direction in which the derivation tree grows: →

Direction in which type information flows: →→

Problem: Type information ONLY flows with the depth-first traversal of the derivation tree.

But a lot of times, type information needs to flow in the reverse direction.

Examples 1) $\lambda x.\ x+1$ — type of x comes from the body

$$\frac{\vdots \quad \vdash x:\text{Int} \qquad \vdash 1:\text{Int}}{x:? \vdash x+1:\text{Int}}$$
$$\vdash \lambda x.\ x+1 : \text{Int}\to\text{Int}$$

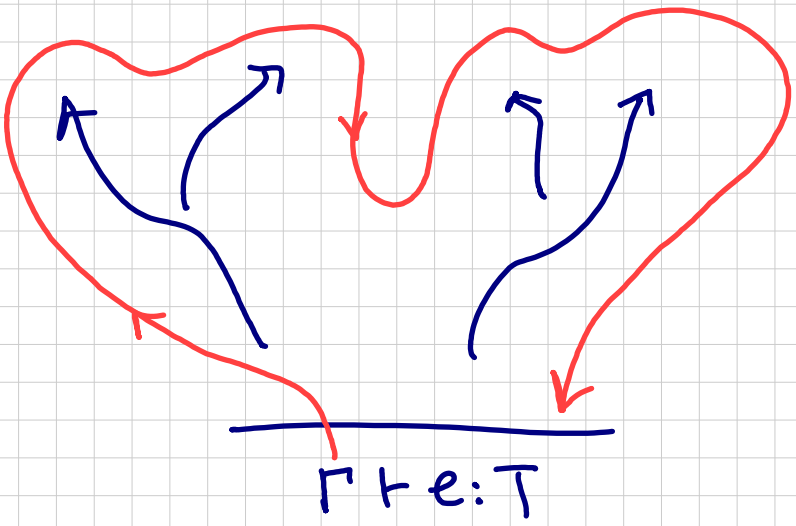
Int can't flow to ? since info is traveling in the WRONG direction.

2) $(\lambda x.\ x)\ 1$ — type of x comes from the supplied arg.

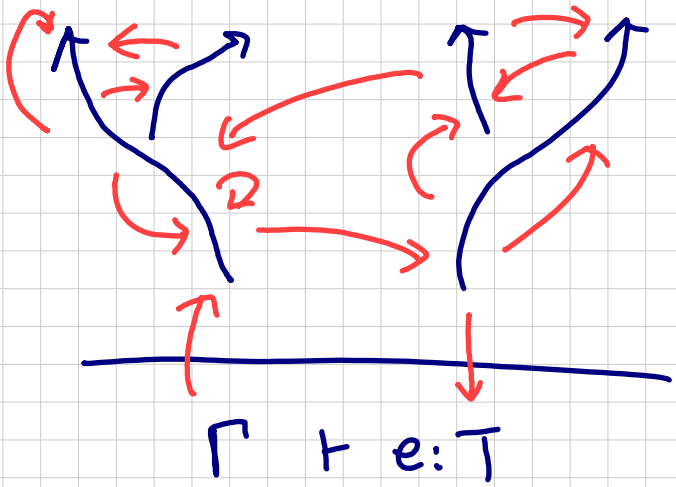
$$\frac{\vdots \quad \vdash \lambda x.x:?\to? \qquad \vdash 1:\text{Int}}{\vdash (\lambda x.x)\ 1 : \text{Int}}$$

Int can't flow to ? again due to WRONG direction.

What we have (type checking)

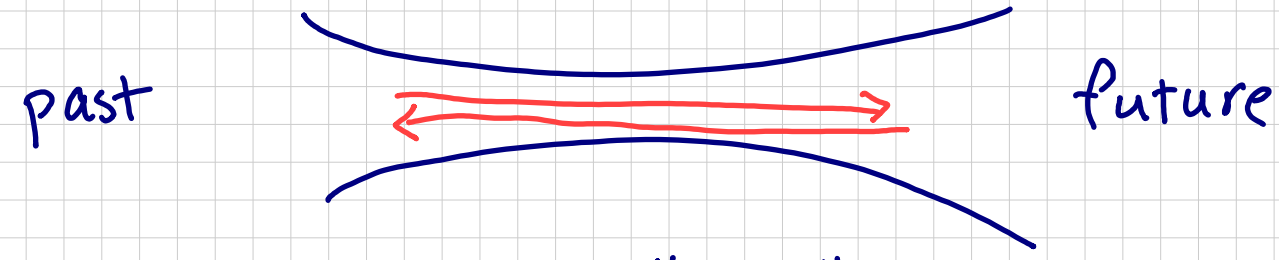


What we want (type inference)



I.e. we want to break causality and enable type info to TIME TRAVEL!

To time-travel, we need "wormholes" to tunnel the future with the past.



Let's use the symbol "$=\!=$" to represent a wormhole connecting two types.

Let's invent type variables X, Y, Z, etc to designate the end-points of wormholes that we have NO INFO about (yet).

Our type inference algorithm has 2 phases:

1. Constraint generation: "Placing down" wormholes at the "right" places.
2. Constraint inference: Figure out all unknown endpoints of the wormholes by examing how the wormholes are connected

Example: Let's place down wormholes for

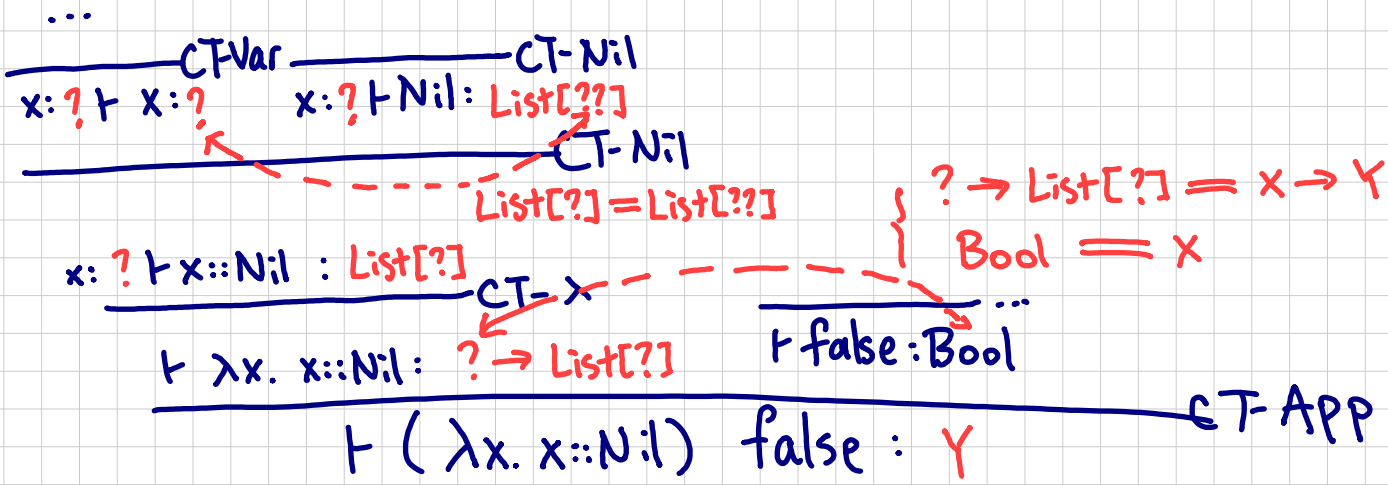
"$(\lambda x.\ x :: Nil)$ false"

Unknown wormhole endpoints (type variables)

$?, ??, X, Y$

Wormholes placed (generated constraints)

$$\begin{cases} List[?] = List[??] \\ ? \to List[?] = X \to Y \\ Bool = X \end{cases}$$



So we're done with placing the wormholes, and here's what we got:

$$\begin{cases} \text{List}[?] = \text{List}[??] \\ ? \to \text{List}[?] = X \to Y \\ \text{Bool} = ? \end{cases}$$

Based on the connections, we can infer what ?, ??, X, Y should be.

This is called constraint-solving.

This paradigm of solving a problem by

constraint-generation + constraint-solving

is extremely powerful beyond type inference. (Take CS292C next quarter to find out!)

The particular solving algorithm we'll use is called unification (which is just high-school math).

Review: How to solve a system of linear equation (w/o matrices)?

$$\begin{cases} 2x = 14y \\ 3z = 4a \\ a = 5y + x \end{cases}$$

Phase 1: Alternate between "unify" {
1) simplification (to isolate a var)
2) if equation has the form var = ... then do forward subst
}

Phase 2: Back Subst

Example (Phase 1)

		Partial solution σ
	$\{ 2x = 14y \leftarrow$ focus	
(1)	$\quad 3z = 4a$	ϕ
	$\quad a = 5y + x$	
	$\{ x = 7y \leftarrow$ focus	
(2)	$\quad 3z = 4a$	ϕ
	$\quad a = 5y + x$	
	$\{ 3z = 4a \leftarrow$ focus	
(1)	$\quad a = 5y + \cancel{x}\ 7y$	$\{ x \mapsto 7y$
	$\{ z = 4/3\, a \leftarrow$ focus	$\{ x \mapsto 7y$
(2)	$\quad a = 5y + 7y$	$\quad z \mapsto 4/3\, a$
		$\quad x \mapsto 7y$
	$\{ a = 5y + 7y \leftarrow$ focus	$\{ a \mapsto 12y$
(2)	$\quad \phi \leftarrow$ focus	$\quad z \mapsto 4/3\, a$
		$\quad x \mapsto 7y$

Phase 2 (Back subst)

$$\sigma = \begin{cases} a \mapsto 12y \\ z \mapsto 4/3\ a \\ x \mapsto 7y \end{cases}$$, we need to

"fully substitute" the right-hand side of the $\mapsto$'s to get

$$\sigma' = \begin{cases} a \mapsto 12y \\ z \mapsto 16y \\ x \mapsto 7 \end{cases}$$ (since $a \mapsto 12y$ and $4/3 \cdot 12y = 16y$)

In general, back subst should output a σ such that any type var that appears on the left of σ does NOT appear on the right. I.e. if $X \in \text{keys}(\sigma)$, then $X \notin \bigcup_{T \in \text{values}(\sigma)} FV(T)$.

2 algorithms can help you ensure this:

1) fixed-point algorithm

2) "actual" back subst (which you learned in linear algebra)

Fixed-point algorithm

Idea Since we want to get rid of any RHS variable X for which σ already has an entry $X \mapsto T$, we can just use $X \mapsto T$ to replace all X appearing on the right.

$$\sigma' = \sigma(\sigma)$$

"apply σ to itself"

meaning for every

$Y \mapsto T$ in σ,

replace it with $Y \mapsto \sigma(T)$.

But doing this once might not be enough.

Example

$$\sigma = \begin{cases} X \mapsto Y \\ Y \mapsto (Z \to A) \\ Z \mapsto A \end{cases}$$

Then

$$\sigma(\sigma) = \begin{cases} X \mapsto (Z \to A) \\ Y \mapsto A \to A \\ Z \mapsto A \end{cases}$$

still not substituted.

So we repeat this:

$$\sigma \to \sigma(\sigma) \to \sigma(\sigma(\sigma)) \to \cdots$$

until we get some σ' for which

$\sigma(\sigma') = \sigma'$, i.e.

we reached a fixedpoint. Then we return σ'.

Proper back subst algorithm

Idea: If σ looks like

$$\begin{cases} X_1 \mapsto T_1 \\ X_2 \mapsto T_2 \\ \vdots \\ \boxed{X_i \mapsto T_i} \\ \vdots \\ X_n \mapsto T_n. \end{cases}$$

Then, for any $X_i \mapsto T_i$,

- X_i can only appear as a FV in $T_{i+1}, T_{i+2}, \ldots, T_n$, and
- X_i cannot appear as FV in $X_1, \ldots, X_{i-1}$. (Think about why).

Thus, we can start at the first entry of σ.

do $[X_1 \mapsto T_1]$ for $T_2, \ldots, T_n$, giving us:

$$\begin{cases} X_2 \mapsto T_2' = T_2[X_1 \mapsto T_1] \\ \vdots \\ X_n \mapsto T_n' = T_n[X_1 \mapsto T_1]. \end{cases}$$

Then use $[X_2 \mapsto T_2']$ and do it on $T_2', \ldots, T_n'$.

Until we go through every X_i. Hint: Use List.map & List.fold