

Section 7

- HW4 out. Due next Friday.
- Review:
 - Drawing derivation tree for $e \Downarrow v$
 - Type checking .

Deriving $e \Downarrow v$.

Rules used:

$$\frac{}{i \Downarrow i} \text{Int}$$

$$\frac{e_1 \Downarrow i \quad e_2 \Downarrow j \quad (k = i \square j)}{e_1 \square e_2 \Downarrow k} \text{Arith}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2[x \rightarrow v_1] \Downarrow v_2}{\text{let } x = e_1 \text{ in } e_2 \Downarrow v_2} \text{Let}$$

$$\frac{\frac{2 \Downarrow 2}{2 \Downarrow 2} \text{Int} \quad \frac{3 \Downarrow 3}{3 \Downarrow 3} \text{Int}}{2 * 3 \Downarrow 6} \quad 2 * 3 = 6$$

$$\frac{\frac{5 \Downarrow 5}{5 \Downarrow 5} \text{Int} \quad \frac{6 \Downarrow 6}{6 \Downarrow 6} \text{Int}}{5 + 6 \Downarrow 11} \quad 5 + 6 = 11$$

$$\text{let } x = \frac{2 * 3}{e_1} \text{ in } \frac{5 + x}{e_2} \Downarrow 11$$

Let

Type Checking

We already know how to execute/run/interpret a program using eval.

If we want to know the behavior of a program e , all we need to do is to eval it.

⇒ Why bother doing something other than eval???

Motivation :

It may be expensive / dangerous / impossible to obtain such info by running the program.

Example (Aviation Software) If you want to know whether a program crashes, you could load the program onto an airplane, fly it, and see if the plane crashes.

Thus, we would like to predict program behaviors before they happen!

Different kinds of predictions

1. Given e , does e evaluate to itself.
i.e. $e \Downarrow e$? \Rightarrow "is-value" relation

$i\ val$	$\lambda x. e\ val$
$true\ val$	$false\ val$
$nil\ val$	$v_1, v_2\ val$ $v_1 :: v_2\ val$

2. Does $e \Downarrow \dots$ halt?

3. Does $e \Downarrow \dots$ get stuck?

↳ AKA does e lead to "undefined behavior"

Example 1. C++ has > 200 undefined behaviors.

If you're the programmer, you're responsible for knowing all > 200 cases, and make sure they don't happen.

Example 2 λ^+ also has undefined behaviors, e.g. $true + 1$.

However, we'll design a type system that predicts whether $e \Downarrow \dots$ will get stuck w/o actually running e .

Due to Rice's Theorem,
our predictions can't be 100% accurate.

Trade-offs: Sound vs complete.

In a prediction system, we ask a binary question:

Is... true or false? / Does ... happen or not?

= Ground-truth : The actual positive/negative answer.

= Prediction : What we predict is the positive/negative answer.

Soundness : Positive predictions don't lie.

Completeness : No true positive is missed.

Example "Do I have Covid?"

A covid test is a prediction system.

- A sound covid test means: "If the test says positive, then I have Covid".

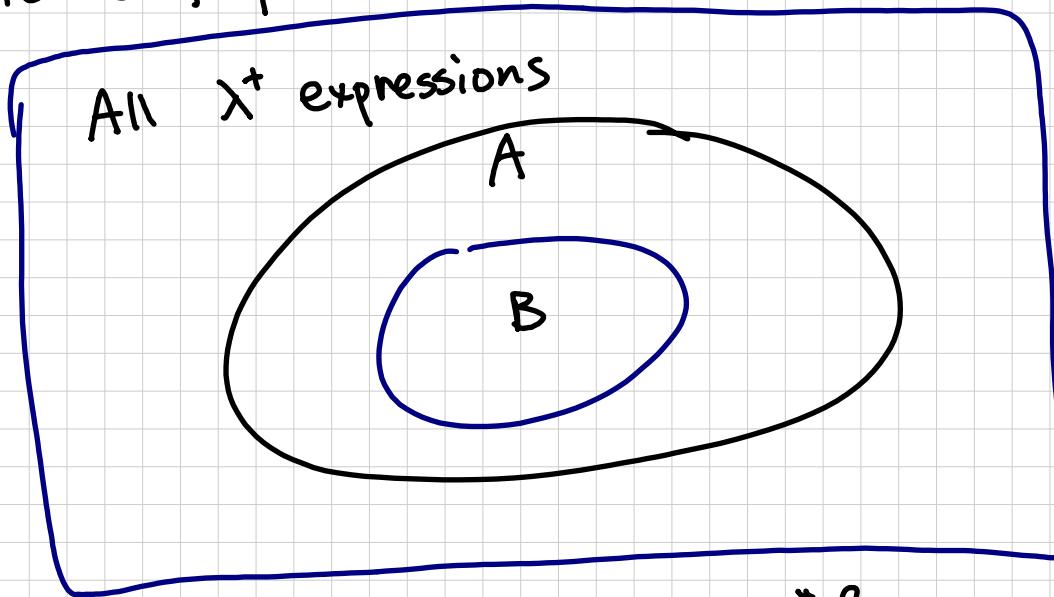
- A complete covid test means: "If I have Covid, then the test must be positive."

- A system can be
- trade-offs! {
1. sound & complete (best case)
 2. complete but unsound (inexpensive at-home Covid tests)
 3. Sound but incomplete (good type systems)
 4. neither (many software analysis tools)
- Due to Rice's Theorem, 1 is impossible if you want your prediction algorithm terminates.
 - At-home Covid tests strive to be complete: try to catch as many cases as possible, but may have "false positives".
 ↳ just go to the hospital & get a more accurate test.
 - Most (good) type systems strive to be sound, because there's no better "hospital" people can go to, other than falling back to actually running the program (BAD).

Our question: Does a λ^+ -program not get stuck,
i.e. evaluate smoothly?

Sound means: if the type checker predicts (positive prediction)
 e won't get stuck
then e for sure doesn't get stuck during
runtime.
(positive behavior)

In terms of picture:



If the type system predicts "positive" for e ,
we say e is well-typed.

A: expressions that
don't get stuck
(actual behavior)

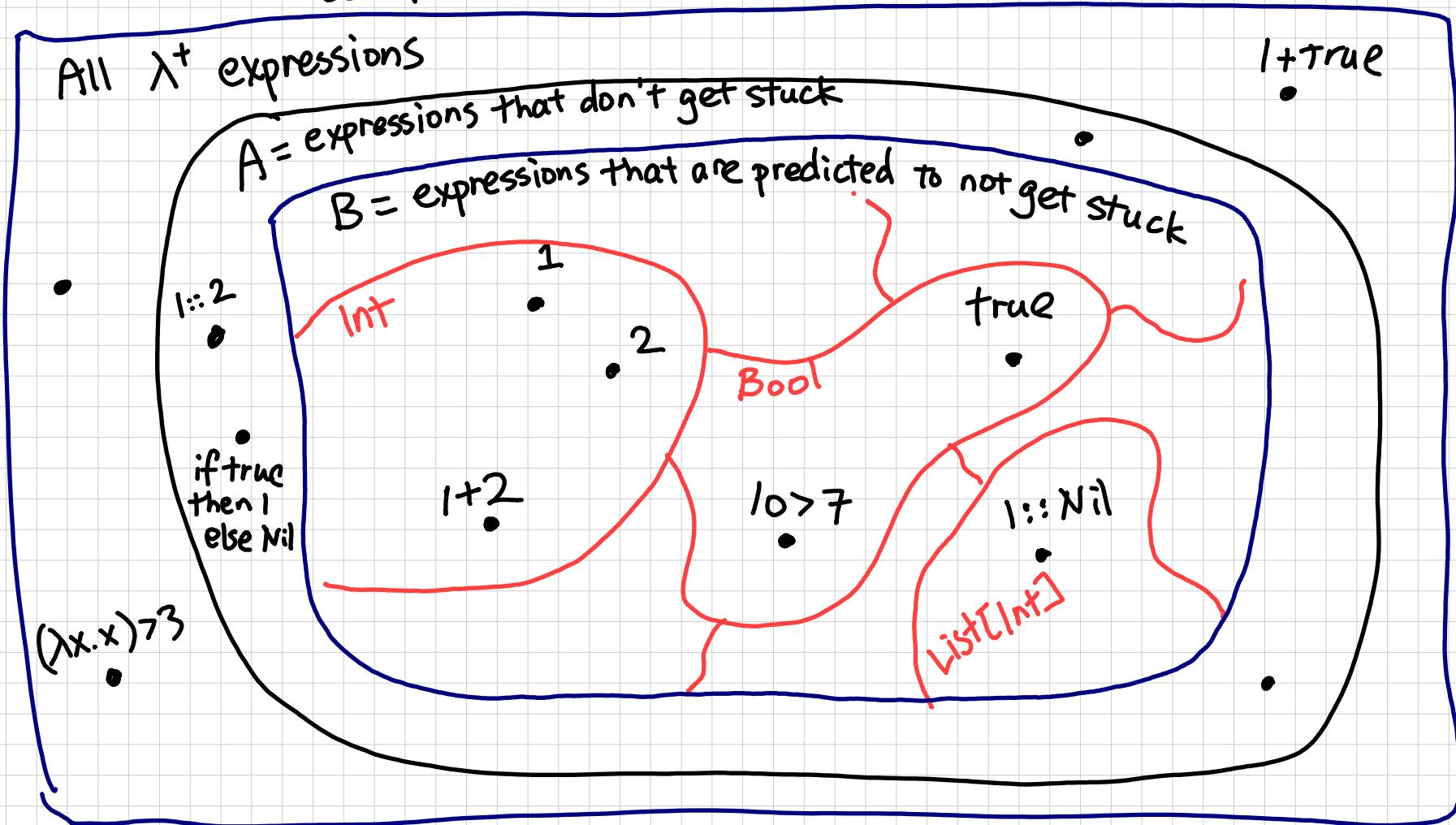
B: expressions that
are predicted to
not get stuck,
i.e. well-typed
expressions.

Why type checking? → Predict runtime behavior, in a sound way.

How? → Abstraction / over-approximation.

Collapse "1", "2", "1+2", ... into a single abstract value **Int**

Collapse "true", "10>7", ... into abstract value **Bool**.



(Concrete) Eval

$e \Downarrow v$

$$\frac{}{i \Downarrow i} \text{Int}$$

$$\frac{e_1 \Downarrow i \quad e_2 \Downarrow j \quad (k = i \square j)}{e_1 \square e_2 \Downarrow k} \text{Arith}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2[x \rightarrow v_1] \Downarrow v_2}{\text{let } x = e_1 \text{ in } e_2 \Downarrow v_2} \text{Let}$$

Abstract Eval

(*: 1st attempt
INCORRECT! but close)

$e : T$

abstract value

$$\frac{}{i : \text{Int}} \text{T-Int}^*$$

$\nwarrow \{0, 1, -1, 2, -2, \dots\}$

$$\frac{e_1 : \text{Int} \quad e_2 : \text{Int}}{e_1 \square e_2 : \text{Int}} \text{T-Arith}^*$$

$$\frac{e_1 : T_1 \quad e_2[x \rightarrow T_1] : T_2}{\text{let } x = e_1 \text{ in } e_2 : T_2} \text{T-Let}^*$$

Problem: Let's try:

$$\frac{\frac{1 : \text{Int} \quad (x+1)[x \rightarrow \text{Int}] = \text{Int} + 1}{\text{let } x = 1 \text{ in } x+1} \quad ?}{\text{let } x = 1 \text{ in } x+1} \text{T-Let}^*$$

Abstract values crepted into the expression!
grammar?

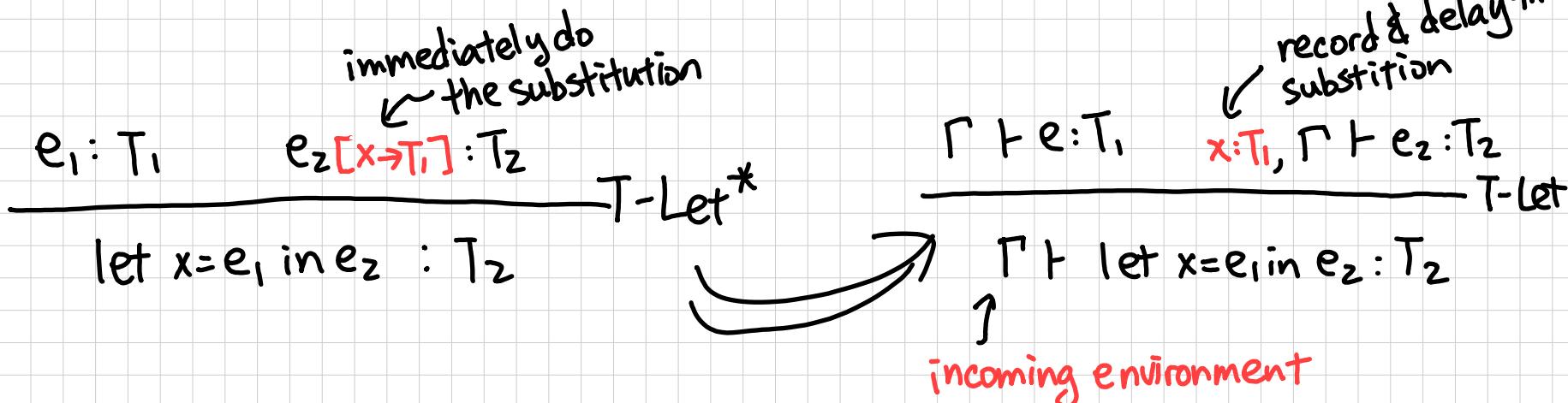
The problem is, if we immediately do the substitution $e_2[x \rightarrow T_1]$, we accidentally bloated our λ^+ language! (What does "Int + 1" mean?)

Solution Be lazy. Don't go ahead and substitute.

Just record the substitution & promise:

"Ok, I will do it eventually if I need to".

Conventionally, people use the Greek symbol Γ (Gamma) to record lazy substitutions, and call it the typing environment.



$$\frac{\overline{i : \text{Int}}}{\text{T-Int}^*} \xrightarrow{\quad} \frac{\Gamma \vdash i : \text{Int}}{\text{T-Int}}$$

$$\frac{e_1 : \text{Int} \quad e_2 : \text{Int}}{e_1 \square e_2 : \text{Int}} \xrightarrow{\quad} \frac{\text{T-Arith}^*}{\frac{\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 \square e_2 : \text{Int}}} \text{T-Arith}$$

We just pass the current environment to sub-expressions
in most cases.

Thus, T-Let adds a new entry $x:T$ to the current Γ .

But what if we encounter free variable x in e_2 ?

↳ That's when we do lookup (can't be lazy anymore!)

"querying key x in Γ gives data T ."

$$\frac{\Gamma(x)=T}{\Gamma \vdash x:T} \text{ T-Var}$$

lookup order: left-to-right

$$(x:\text{Int}, x:\text{Bool})(x) = \text{Int}$$

$$\frac{\text{TVAR}}{x:\text{Int}, x:\text{Bool} \vdash x:\text{Int} \quad x:\text{Int}, x:\text{Bool} \vdash 1:\text{Int}} \text{ T-Int}$$

$x:\text{Int}, x:\text{Bool} \vdash x+1:\text{Int}$ T-Arith

inherited

$$x:\text{Bool} \vdash 2:\text{Int} \text{ T-Int}$$

still empty

$$\vdash \text{true}: \text{Bool} \text{ T-Bool}$$

new entry

$$\frac{x:\text{Bool} \vdash \text{let } x=2 \text{ in } x+1}{\text{T-Let}}$$

initially, Γ is empty!

$$\frac{\vdash \text{let } x=\underline{\text{true}} \text{ in } \underline{\text{let } x=2 \text{ in } x+1} : e_1 \quad \vdash \text{let } x=2 \text{ in } x+1 : e_2}{\text{T-Let}}$$