Section 7

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

Deriving $e \Downarrow v$.

Rules used:

$$\frac{}{i \Downarrow i}\ \text{Int} \qquad \frac{e_1 \Downarrow i \quad e_2 \Downarrow j \quad (k = i \square j)}{e_1 \square e_2 \Downarrow k}\ \text{Arith} \qquad \frac{e_1 \Downarrow v_1 \quad e_2[x \to v_1] \Downarrow v_2}{\text{let } x = e_1 \text{ in } e_2 \Downarrow v_2}\ \text{Let}$$

$$\dfrac{\dfrac{\dfrac{}{2 \Downarrow 2}\text{Int} \quad \dfrac{}{3 \Downarrow 3}\text{Int}\ 2\times3=6}{2*3 \Downarrow 6} \qquad (5+x)[x \to 6] = 5+6 \qquad \dfrac{\dfrac{}{5 \Downarrow 5}\text{Int} \quad \dfrac{}{6 \Downarrow 6}\text{Int}\ 5+6=11}{5+6 \Downarrow 11}}{\text{let } x = \underbrace{2*3}_{e_1} \text{ in } \underbrace{5+x}_{e_2} \Downarrow 11}\ \text{Let}$$

Type Checking

We already know how to execute / run / interpret a program using $e \Downarrow v$.

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 $e \Downarrow v$???

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. ⇒ "is-value" relation
   i.e. $e \Downarrow e$?

$$\frac{}{i \text{ val}} \qquad \frac{}{\lambda x.e \text{ val}}$$

$$\frac{}{\text{true val}} \qquad \frac{}{\text{false val}}$$

$$\frac{}{\text{nil val}} \qquad \frac{v_1 \text{ val} \quad v_2 \text{ val}}{v_1 :: v_2 \text{ val}}$$

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

3. Does $e \Downarrow \ldots$ 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. No one is there to help you.

Example 2 λ^+ also has undefined behaviors, e.g. "true + 1".
However, we'll design a type system that predicts whether $e \Downarrow \ldots$ will get stuck w/o actually running e.
Type system is there to help you.

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 to terminate.

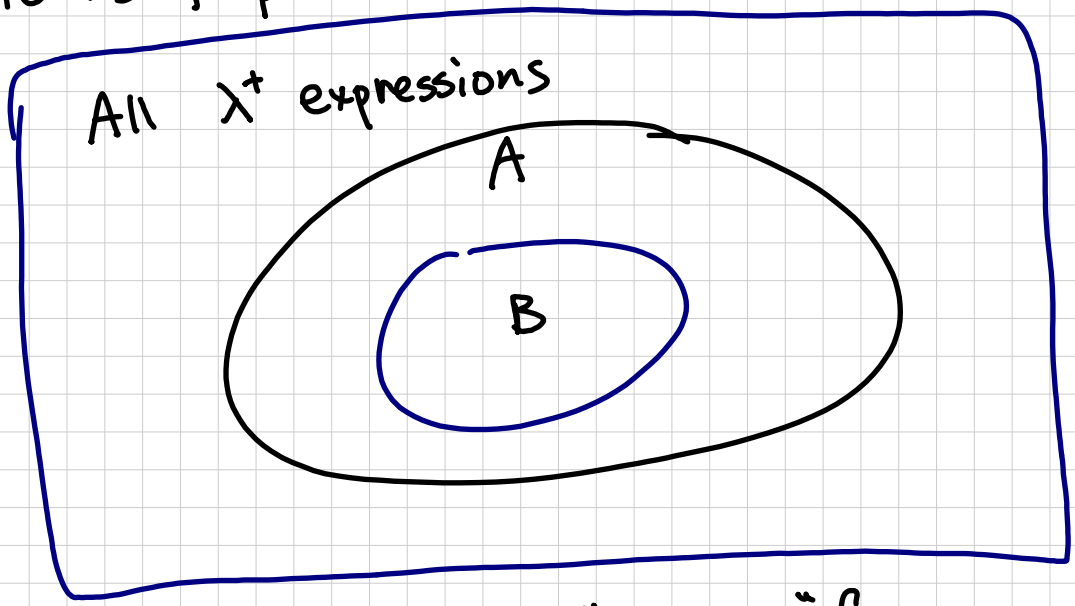
- At-home Covid tests strive to be complete: try to catch as many cases as possible, but may have "false positives".
  ↳ just go to the hospital & get a more accurate test!

- 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:



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

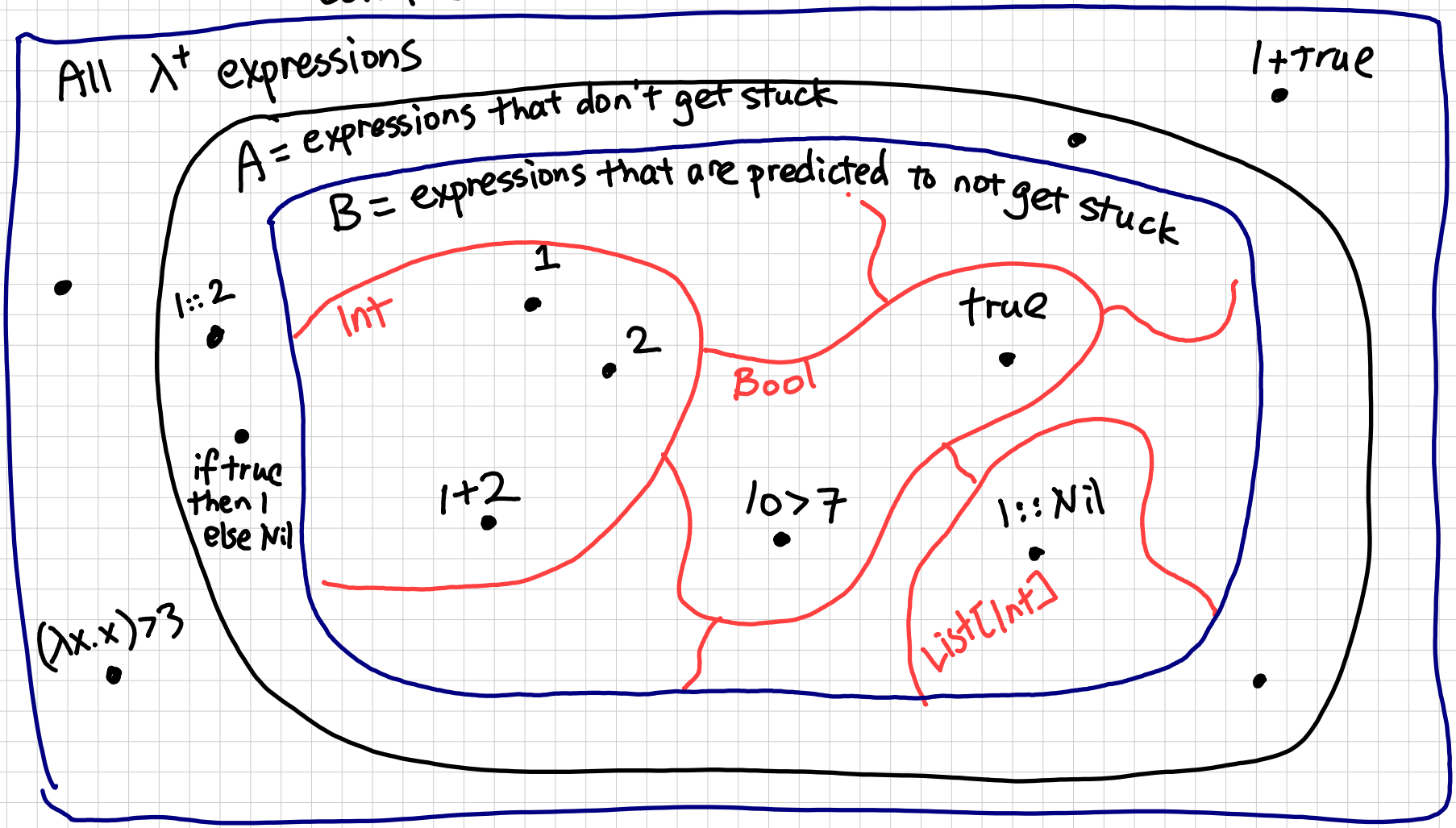
B: expressions that are predicted to not get stuck, i.e. well-typed expressions.

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

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 $e \Downarrow v$

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

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

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

Abstract Eval $e : T$

(*: 1st attempt INCORRECT! but close)

abstract value

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

$i : \text{Int}$ ← $\{0, 1, -1, 2, -2, \ldots\}$

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

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

Problem: Let's try:

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

Abstract values crept into the expression language!

The problem is, if we immediately do the substitution $e_2[x \to T_1]$, we would accidentally bloat our λ^t language! (What does "Int+1" even 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.

immediately do the substitution

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

record & delay the substition

$$\frac{\Gamma \vdash e : T_1 \qquad x : T_1, \Gamma \vdash e_2 : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2}\ \text{T-Let}$$

incoming environment

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

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

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!)

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

"querying key x in Γ gives data T."

lookup order: left-to-right

$$\frac{(x:\text{Int}, x:\text{Boo})(x) = \text{Int}}{x:\text{Int}, x:\text{Bool} \vdash x:\text{Int}}\ \text{T-VAR} \qquad \frac{}{x:\text{Int}, x:\text{Bool} \vdash 1:\text{Int}}\ \text{T-Int}$$

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

new entry

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

inherited

$$\frac{}{\vdash \text{true} : \text{Bool}}\ \text{T-Bool} \qquad \frac{\cdots}{x:\text{Bool} \vdash \text{let } x = \underbrace{2}_{e_1} \text{ in } \underbrace{x+1}_{e_2} : \text{Int}}\ \text{T-Let}$$

still empty

new entry

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

initially, Γ is empty!