Lecture 15

CIS 341: COMPILERS

Announcements

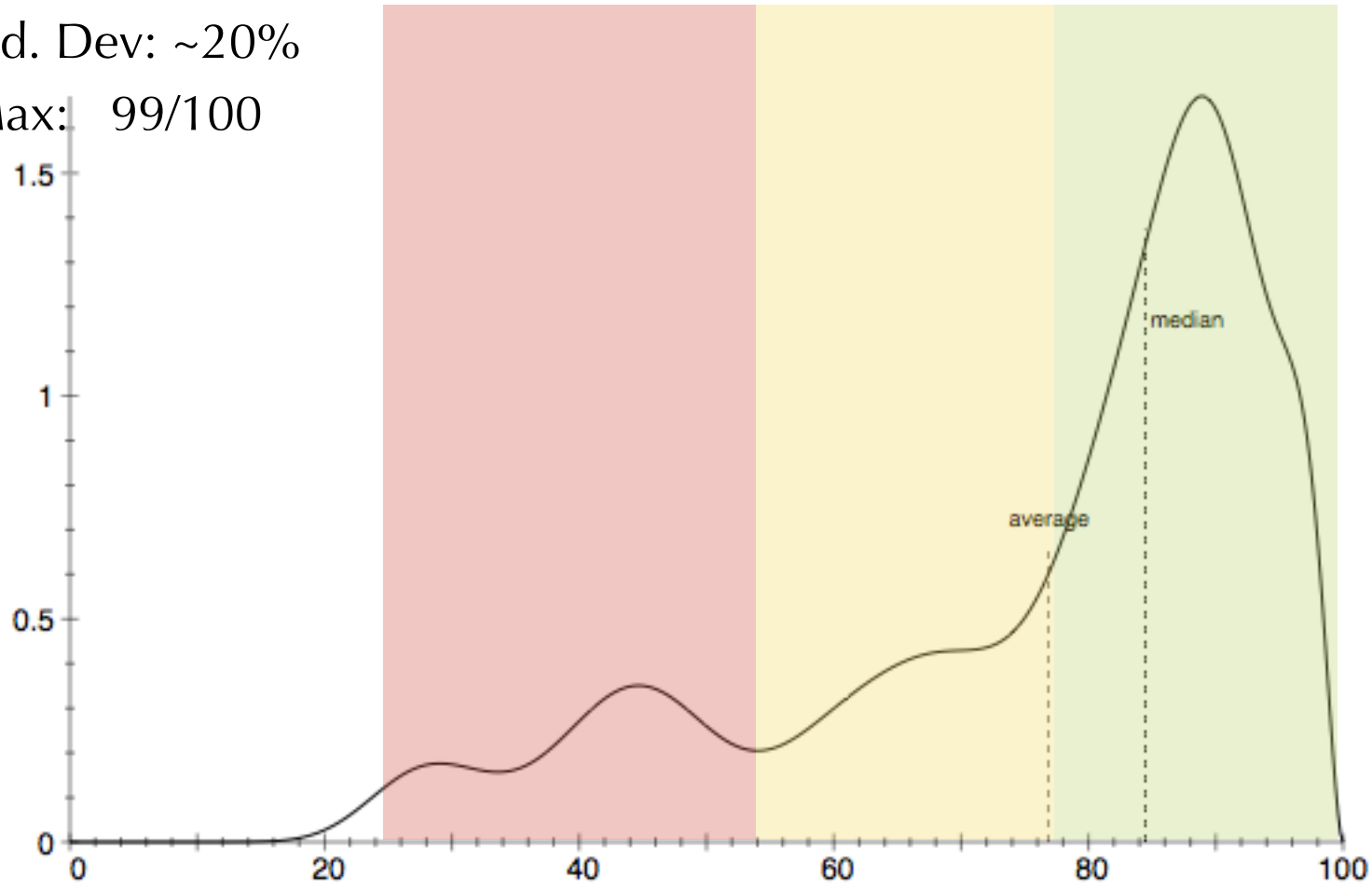
- Midterm Exam:
  - Graded and entered

- Project Grades:
  - We need to propagate the grades from one team member to another.
  - But: for Project 2 we forgot to ask for team.txt, so we need the team-member information. See email/announcement on Piazza for instructions.

- Project 4 is available from the course web pages
  - Due on Thursday, March 21st.
  - As usual, start early and ask questions if you get stuck
  - Note: revised version of LL intermediate representation to be more compliant with "real" LLVM IR

Midterm Exam Grade Distribution

- Average: ~77%
- Median: ~84%
- Std. Dev: ~20%
- Max: 99/100



Compiling lambda calculus to straight-line code.
Representing evaluation environments at runtime.

CLOSURE CONVERSION

Compiling First-class Functions

- To implement first-class functions on a processor, there are two problems:
  - First: we must implement substitution of free variables
  - Second: we must separate 'code' from 'data'

- Closure Conversion:
  - Eliminates free variables by packaging up the needed environment in a data structure.
  - Big idea: push the meta-level environment into the object-level

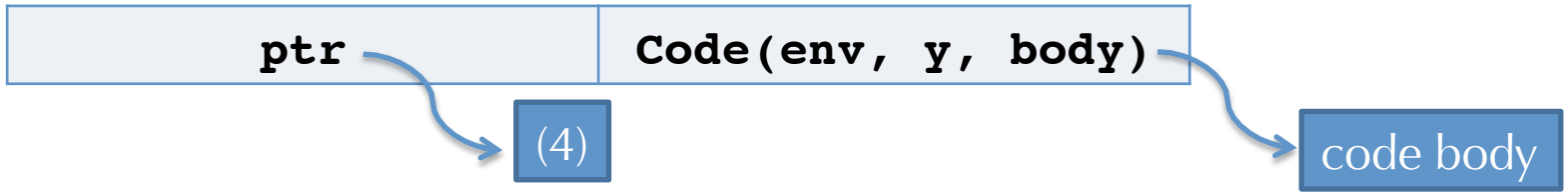
- Hoisting:
  - Separates code from data, pulling closed code to the top level.

Example of closure creation

- Recall the "add" function:
  `let add = fun x -> fun y -> x + y`

- Consider the inner function: `fun y -> x + y`

- When run the function application: `add 4`
  the program builds a closure and returns it.
  - The closure is a pair of the environment and a code pointer.



- The code pointer takes a pair of parameters: env and y
  - The function code is (essentially):
    `fun (env, y) -> let x = nth env 1 in x + y`

Example of Closure Application

- To "invoke" a closure, the semantics of the IL must bake in the projection of the environment and code point from the closure value.

- At the meta-level: App(e1, e2)

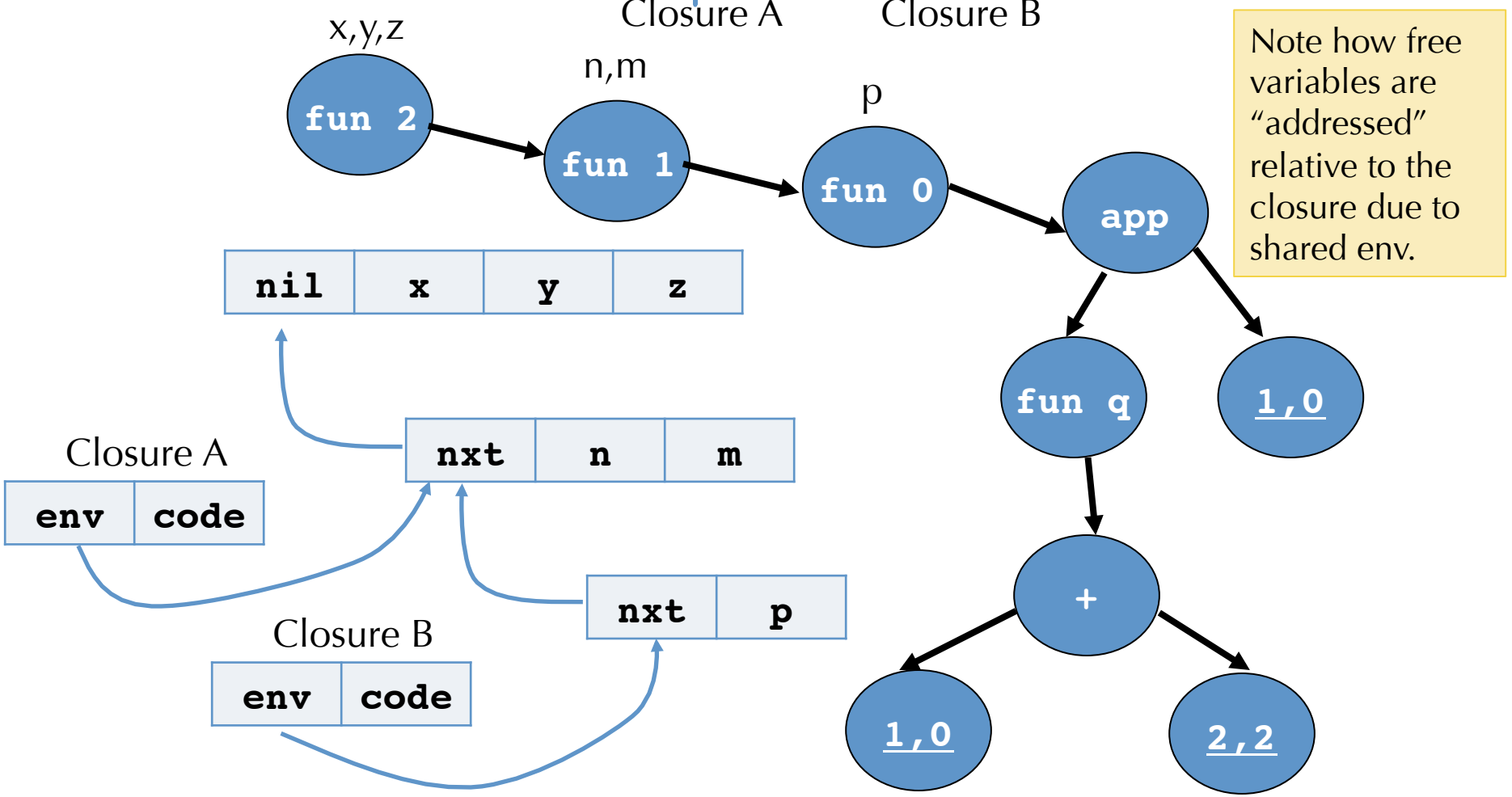
Representing Closures

- The simple closure conversion algorithm in cc.ml isn't very efficient:
  - It stores all the values for variables in the environment, even if they aren't needed.
  - It copies the environment values to a new tuple each time an inner closure is created.

- There are many options:
  - Store only the values for the free variables in the body of the closure.
  - Share subcomponents of the environment to avoid copying
  - Use vectors or arrays rather than linked structures (indexing into the environment becomes more complicated)

Array-based Closures with N-ary Functions

```
(fun (x y z) ->
   (fun (n m) -> (fun p -> (fun q -> n + z) x)
```



Note how free variables are "addressed" relative to the closure due to shared env.

BACK TO TYPECHECKING

Simply-typed Lambda Calculus

- For the language in "tc.ml" we have five inference rules:

INT

$$\frac{}{E \vdash i : \text{int}}$$

VAR

$$\frac{x : T \in E}{E \vdash x : T}$$

ADD

$$\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : \text{int}}{E \vdash e_1 + e_2 : \text{int}}$$

FUN

$$\frac{E, x : T \vdash e : S}{E \vdash \text{fun } (x{:}T) \text{ -> } e \ : T \text{ -> } S}$$

APP

$$\frac{E \vdash e_1 : T \text{ -> } S \qquad E \vdash e_2 : T}{E \vdash e_1\ e_2 : S}$$

Different Kinds of Judgments

- So far, we've been using judgments of the form "e : T" to mean expression e has type T
- For statements, which don't evaluate to values, the judgment form is "s ok", meaning that the evaluation of the statement s doesn't yield any run-time failures.

- Note how this difference mirrors the difference in syntax and semantics
  - expressions evaluate to values
  - statements are evaluated for their side effects

- (Sometimes we omit the keyword 'ok' since it is the same for all statements.)

Adding More Typing Rules

- It is easy to add inference rules for other program constructs:

WHILE

$$\frac{E \vdash e_1 : \text{int} \qquad E \vdash s \text{ ok}}{E \vdash \text{while } (e_1)\ s \ \text{ ok}}$$

Note: If the language has Booleans, we should require: $E \vdash e_1 : \text{bool}$.

VarDecl

$$\frac{E \vdash e_1 : T \quad E, x : T \vdash s \text{ ok}}{E \vdash T\ x = e_1;\ s \ \text{ ok}}$$

Note: We add the assumption x : T to the context when checking e_2 – x is in scope in e_2.

ASSIGN

$$\frac{E \vdash x : T \qquad E \vdash e : T}{E \vdash x = e \ \text{ ok}}$$

Note: We have a choice about the statements vs. expressions. We could follow C-style and make assignment an expression with type 'T'

Arrays

- Array constructs are not hard either, here is one possibility
- First: add a new type constructor: T[]

NEW

$$\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : T}{E \vdash \text{new } T[e_1](e_2) \; : T[]}$$

e_1 is the size of the newly allocated array. e_2 is initializes the elements of the array.

INDEX

$$\frac{E \vdash e_1 : T[] \qquad E \vdash e_2 : \text{int}}{E \vdash e_1[e_2] \; : T}$$

UPDATE

$$\frac{E \vdash e_1 : T[] \quad E \vdash e_2 : \text{int} \quad E \vdash e_3 : T}{E \vdash e_1[e_2] = e_3 \text{ ok}}$$

Note: These rules don't ensure that the array index is in bounds – that should be checked dynamically.

Tuples

- ML-style tuples with statically known number of products:
- First: add a new type constructor: $T_1 * \ldots * T_n$

TUPLE

$$\frac{E \vdash e_1 : T_1 \quad \ldots \quad E \vdash e_n : T_n}{E \vdash (e_1, \ldots, e_n) : T_1 * \ldots * T_n}$$

PROJ

$$\frac{E \vdash e : T_1 * \ldots * T_n \quad 1 \leq i \leq n}{E \vdash \#i\ e\ :\ T_i}$$

References

- ML-style references (note that ML uses only expressions)
- First, add a new type constructor: T ref

REF

$$\frac{E \vdash e : T}{E \vdash \text{ref } e : T \text{ ref}}$$

DEREF

$$\frac{E \vdash e : T \text{ ref}}{E \vdash !e : T}$$

ASSIGN

$$\frac{E \vdash e_1 : T \text{ ref} \qquad E \vdash e_2 : T}{E \vdash e_1 := e_2 : \text{unit}}$$

Note the similarity with the rules for arrays…

Recursive Definitions

- Consider the ML factorial function:

```
let rec fact (x:int) : int =
 if (x == 0) 1 else x * fact(x-1)
```

- Note that the function name `fact` appears inside the body of `fact`'s definition!
- To typecheck the body of fact, we must assume that the type of fact is already known.

$$\frac{E, \text{fact} : \text{int} \to \text{int}, x : \text{int} \vdash e_{body} : \text{int}}{E \vdash \text{int fact(int x) (} e_{body}\text{)} : \text{int} \to \text{int}}$$

- In general: Collect the names and types of all mutually recursive definitions, add them all to the context E before checking any of the definition bodies.
- Often useful to separate the "global context" from the "local context"

oat.pdf (Project 4 version)

OAT TYPING RULES

Beyond describing “structure”… describing “properties”

Types as sets

Subsumption

TYPES, MORE GENERALLY

What are types, anyway?

- A *type* is just a predicate on the set of values in a system.
  - For example, the type "int" can be thought of as a boolean function that returns "true" on integers and "false" otherwise.
  - Equivalently, we can think of a type as just a *subset* of all values.

- For efficiency and tractability, the predicates are usually taken to be very simple.
  - Types are an *abstraction* mechanism

- We can easily add new types that distinguish different subsets of values:

```
type tp =
     | IntT                    (* type of integers *)
     | PosT | NegT | ZeroT     (* refinements of ints *)
     | BoolT                   (* type of booleans *)
     | TrueT | FalseT          (* subsets of booleans *)
     | AnyT                    (* any value *)
```

Modifying the typing rules

- We need to refine the typing rules too…
- Some easy cases:
  - Just split up the integers into their more refined cases:

P-INT

$$\frac{i > 0}{E \vdash i : Pos}$$

N-INT

$$\frac{i < 0}{E \vdash i : Neg}$$

ZERO

$$\frac{}{E \vdash 0 : Zero}$$

- Same for booleans:

TRUE

$$\frac{}{E \vdash true : True}$$

FALSE

$$\frac{}{E \vdash false : False}$$

What about "if"?

- Two cases are easy:

IF-T
$$\frac{E \vdash e_1 : \text{True} \quad E \vdash e_2 : T}{E \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : T}$$

IF-F
$$\frac{E \vdash e_1 : \text{False} \quad E \vdash e_3 : T}{E \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : T}$$

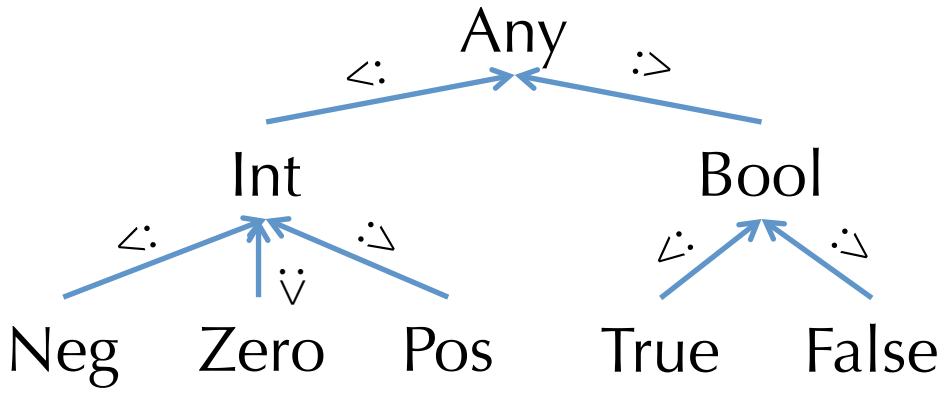
- What happens when we don't know statically which branch will be taken?
- Consider the typechecking problem:

$$x{:}\text{bool} \vdash \text{if } (x)\ 3 \text{ else } -1 : ?$$

- The true branch has type Pos and the false branch has type Neg.
  - What should be the result type of the whole if?

Subtyping and Upper Bounds

- If we think of types as sets of values, we have a natural inclusion relation: Pos ⊆ Int
- This subset relation gives rise to a *subtype* relation: Pos <: Int
- Such inclusions give rise to a *subtyping hierarchy*:



- Given any two types T_1 and T_2, we can calculate their *least upper bound* (LUB) according to the hierarchy.
  - Example: LUB(True, False) = Bool, LUB(Int, Bool) = Any
  - Note: might want to add types for "NonZero", "NonNegative", and "NonPositive" so that set union on values corresponds to taking LUBs on types.

"If" Typing Rule Revisited

- For statically unknown conditionals, we want the return value to be the LUB of the types of the branches:

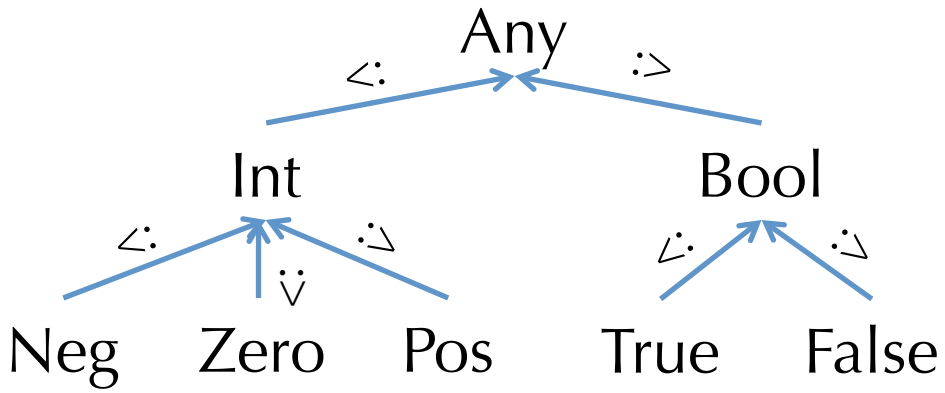
IF-BOOL

$$\frac{E \vdash e_1 : \text{bool} \quad E \vdash e_2 : T_1 \quad E \vdash e_3 : T_2}{E \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : \text{LUB}(T_1, T_2)}$$

- Note that LUB(T_1, T_2) is the most precise type (according to the hierarchy) that is able to describe any value that has either type T_1 or type T_2.
- In math notation, LUB(T1, T2) is sometimes written $T_1 \vee T_2$
- LUB is also called the *join* operation.

Subtyping Hierarchy

- A *subtyping hierarchy*:



- The subtyping relation is a *partial order*:
  - Reflexive: $T <: T$ for any type T
  - Transitive: $T_1 <: T_2$ and $T_2 <: T_3$ then $T_1 <: T_3$
  - Antisymmetric: It $T_1 <: T_2$ and $T_2 <: T_1$ then $T_1 = T_2$

Soundness of Subtyping Relations

- We don't have to treat *every* subset of the integers as a type.
  - e.g., we left out the type NonNeg

- A subtyping relation $T_1 <: T_2$ is *sound* if it approximates the underlying semantic subset relation.
- Formally: write $[\![T]\!]$ for the subset of (closed) values of type T
  - i.e. $[\![T]\!] = \{v \mid \vdash v : T\}$
  - e.g. $[\![\text{Zero}]\!] = \{0\}$, $[\![\text{Pos}]\!] = \{1, 2, 3, \ldots\}$

- If $T_1 <: T_2$ implies $[\![T_1]\!] \subseteq [\![T_2]\!]$, then $T_1 <: T_2$ is sound.
  - e.g. Pos <: Int is sound, since $\{1,2,3,\ldots\} \subseteq \{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$
  - e.g. Int <: Pos is not sound, since it is *not* the case that $\{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \subseteq \{1,2,3,\ldots\}$

Soundness of LUBs

- Whenever you have a sound subtyping relation, it follows that:

$$[\![\text{LUB}(T_1, T_2)]\!] \supseteq [\![T_1]\!] \cup [\![T_2]\!]$$

  - Note that the LUB is an over approximation of the "semantic union"
  - Example: $[\![\text{LUB}(\text{Zero}, \text{Pos})]\!] = [\![\text{Int}]\!] = \{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \supseteq \{0,1,2,3,\ldots\} = \{0\} \cup \{1,2,3,\ldots\} = [\![\text{Zero}]\!] \cup [\![\text{Pos}]\!]$

- Using LUBs in the typing rules yields sound approximations of the program behavior (as if the IF-B rule).
- It just so happens that LUBs on types <: Int correspond to +

ADD

$$\frac{E \vdash e_1 : T_1 \quad E \vdash e_2 : T_2 \quad T_1 <: \text{Int} \quad T_2 <: \text{Int}}{E \vdash e_1 + e_2 : T_1 \vee T_2}$$