

# YSC4230: Programming Language Design and Implementation

## Week 10: Type Checking and Typing for Advanced Features

Ilya Sergey

[ilya.sergey@yale-nus.edu.sg](mailto:ilya.sergey@yale-nus.edu.sg)

# Why Types?

- *Type checking* (ensuring that the program is ascribed a “correct” type) catches errors at compile time, eliminating a class of mistakes that would otherwise lead to run-time errors, provided type information
- *Type inference* derives type information from the code (think function parameters in OCaml vs Java)
- *Type information* is sometimes necessary for code generation
  - Floating-point + is not the same instruction as integer + is not the same as pointer/integer +
  - pointer/integer compiled differently depending on pointer type
  - Assignment  $x = y$  compiled differently if  $y$  is an **int** or a **struct**

# Purposes of Type Checking

- *Statically* rule out scoping issues (unresolved identifiers)
- *Statically* rule out multiple runtime errors (what is “a” + 42?)
- Make sure that the ascribed types are a correct approximation of runtime values
- Provide information about types of intermediate operations to the compiler

# What is a type system?

- A type system consists of a system of judgements and inference rules
  - (Extrinsic view) A judgement is a *claim*, which may or may not be valid
    - $\vdash 3 : \text{int}$  - “3 has type integer”
    - $\vdash (1 + 2) : \text{bool}$  - “(1+2) has type boolean”
  - **Inference rules** are used to derive *valid* judgements from other valid judgements.

$$\begin{array}{c} \text{ADD} \\ \vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \\ \hline \vdash e_1 + e_2 : \text{int} \end{array}$$

Read: “If  $e_1$  and  $e_2$  have type  $\text{int}$ , so does  $e_1 + e_2$ ”

- Type system might involve many different kinds of judgement
  - Well-typed expressions
  - Well-formed types
  - Well-formed statements
  - ...

# Inference Rules, General Form

- An *inference rule* consists of a list of premises  $J_1, \dots, J_n$  and one conclusion  $J$  (optionally: a side-condition):

$$\frac{J_1 \quad J_2 \quad \dots \quad J_n}{J} \text{SIDE-CONDITION}$$

- Side-condition: additional premise, but not a judgement
- Read *top-down*: If  $J_1$  and  $J_2$  and ... and  $J_n$  are valid, and the side condition holds, then  $J$  is valid.
- Read *bottom-up*: To prove  $J$  is valid, sufficient to prove  $J_1, J_2, \dots, J_n$  are valid

# Type Judgments

- In the judgment:  $G \vdash e : t$ 
  - $G$  is a *typing environment* or a *type context*
  - $G$  maps variables to types. It is just a set of bindings of the form:  
 $x_1 : t_1, x_2 : t_2, \dots, x_n : t_n$
- A type judgement takes the form  $G \vdash e : t$   
“Under the type environment  $G$ , the expression  $e$  has type  $t$ ”
- For example:  
 $x : \text{int}, b : \text{bool} \vdash \text{if } (b) \ 3 \ \text{else } x : \text{int}$
- What do we need to *check* to decide whether “if (b) 3 else x” has type **int** in the environment **x : int, b : bool**?
  - $b$  must be a bool      i.e.       $x : \text{int}, b : \text{bool} \vdash b : \text{bool}$
  - 3 must be an int      i.e.       $x : \text{int}, b : \text{bool} \vdash 3 : \text{int}$
  - $x$  must be an int      i.e.       $x : \text{int}, b : \text{bool} \vdash x : \text{int}$

# Simply-typed Lambda Calculus with Integers

INT

$G \vdash i : \text{int}$

VAR

$x : T \in G$

$G \vdash x : T$

ADD

$G \vdash e_1 : \text{int}$

$G \vdash e_2 : \text{int}$

$G \vdash e_1 + e_2 : \text{int}$

FUN

$G, x : T \vdash e : S$

$G \vdash \text{fun } (x:T) \rightarrow e : T \rightarrow S$

APP

$G \vdash e_1 : T \rightarrow S$

$G \vdash e_2 : T$

$G \vdash e_1 e_2 : S$

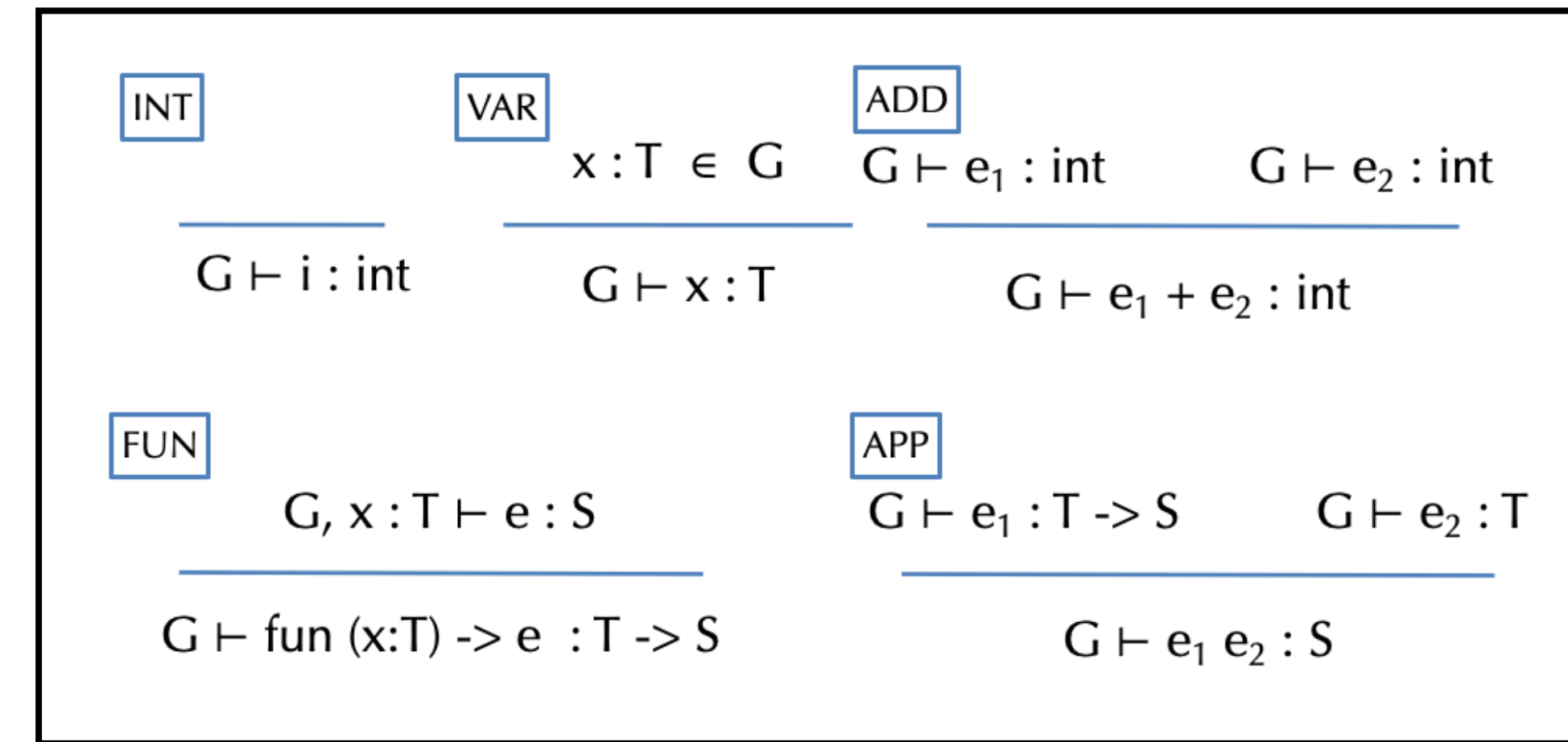
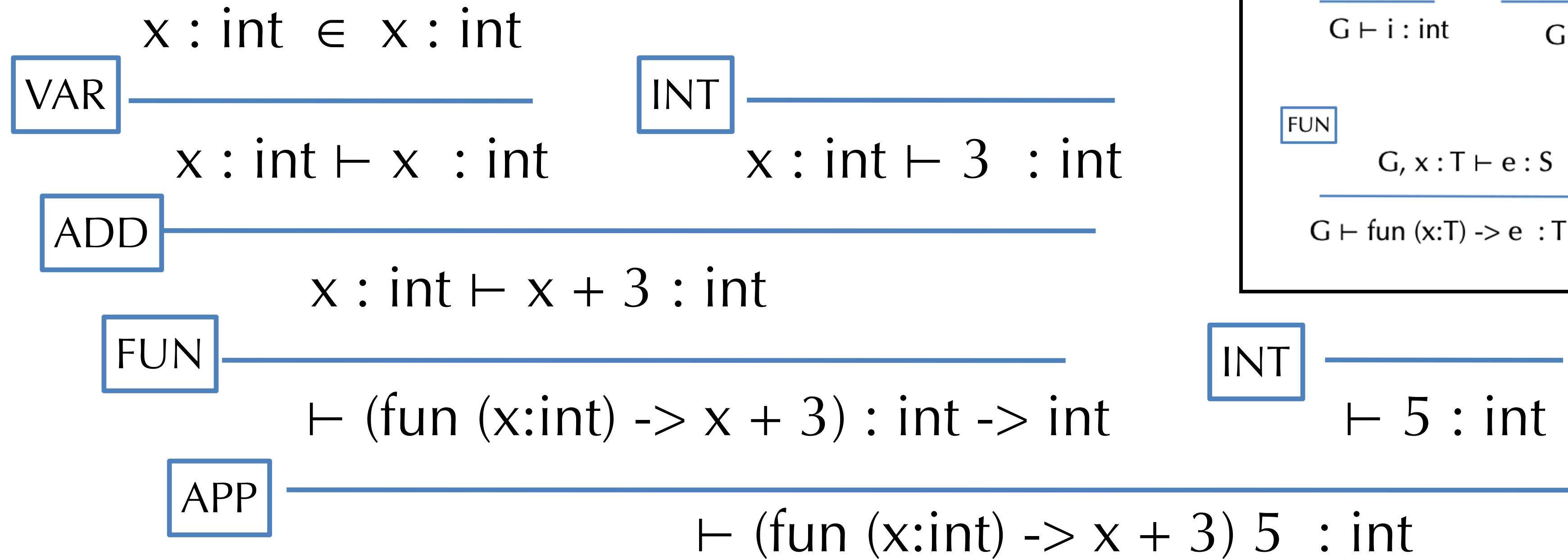
# Model for Type Checking

- A *derivation* or *proof tree* has (instances of) judgments as its nodes and edges that connect premises to a conclusion according to an inference rule.
- Leaves of the tree are *axioms* (i.e. rules with no premises)
  - Example: the INT rule is an axiom
- Goal of the type checker: verify that *such a tree exists*.
- **Example:** Find a tree for the following program using the inference rules on the previous slide:

$$\vdash (\text{fun } (x:\text{int}) \rightarrow x + 3) 5 : \text{int}$$



# Example Derivation Tree



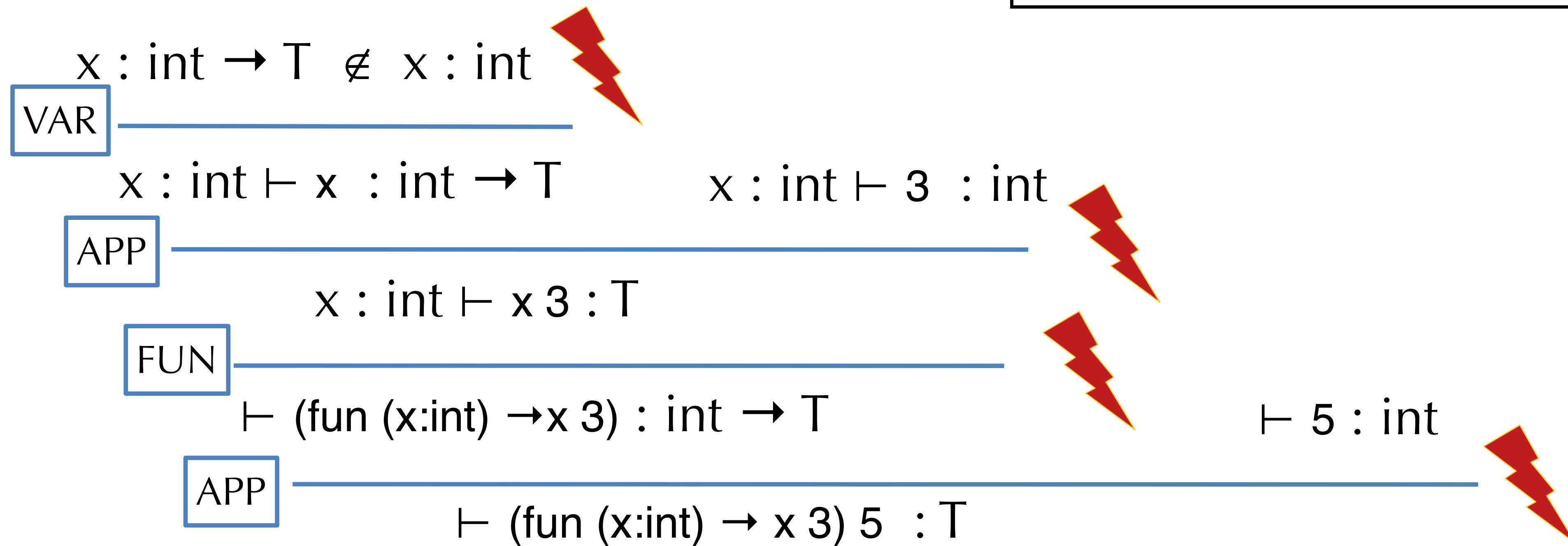
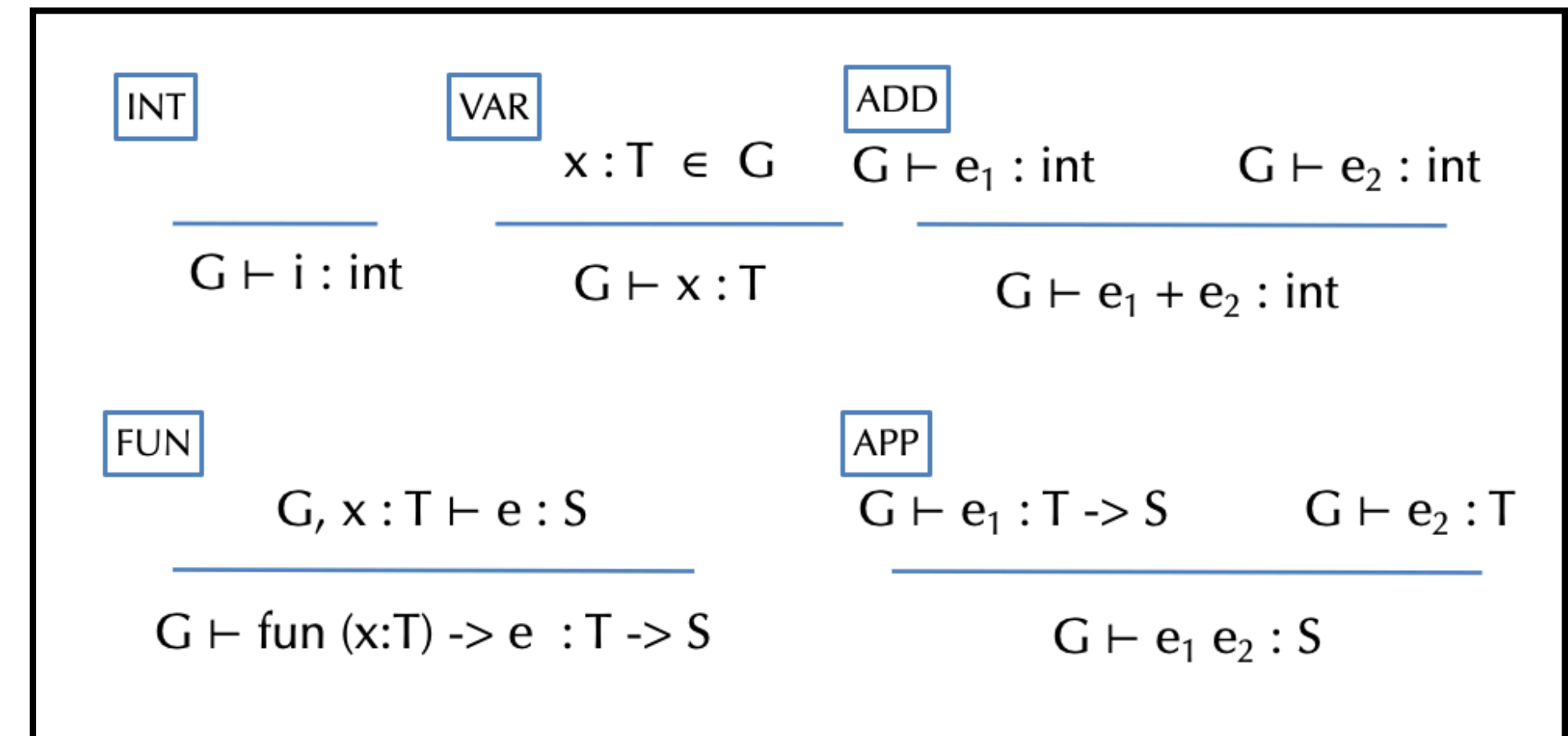
- Note: the OCaml function `typecheck` verifies the existence of this tree. The structure of the recursive calls when running `typecheck` is the same shape as this tree!
- Note that `x : int ∈ E` is implemented by the function `lookup`

# Ill-typed Programs

- Programs without derivations are ill-typed

Example: There is no type  $T$  such that

$$\vdash (\text{fun } (x:\text{int}) \rightarrow x \ 3) \ 5 : T$$



# Simply-typed Lambda Calculus with Integers

- For the language in “stlc.ml” we have five inference rules:

$$\begin{array}{c} \boxed{\text{INT}} \\ \hline G \vdash i : \text{int} \end{array} \quad \begin{array}{c} \boxed{\text{VAR}} \\ x : T \in G \\ \hline G \vdash x : T \end{array} \quad \begin{array}{c} \boxed{\text{ADD}} \\ G \vdash e_1 : \text{int} \quad G \vdash e_2 : \text{int} \\ \hline G \vdash e_1 + e_2 : \text{int} \end{array}$$
  
$$\begin{array}{c} \boxed{\text{FUN}} \\ G, x : T \vdash e : S \\ \hline G \vdash \text{fun } (x:T) \rightarrow e : T \rightarrow S \end{array} \quad \begin{array}{c} \boxed{\text{APP}} \\ G \vdash e_1 : T \rightarrow S \quad G \vdash e_2 : T \\ \hline G \vdash e_1 e_2 : S \end{array}$$

- Note how these rules correspond to the OCaml code.

# Implementing a Type Checker for Lambda Calculus

See [stlc.ml](#)

# Exercise

- Implement the missing parts of the type-checker

# Notes about this Type Checker

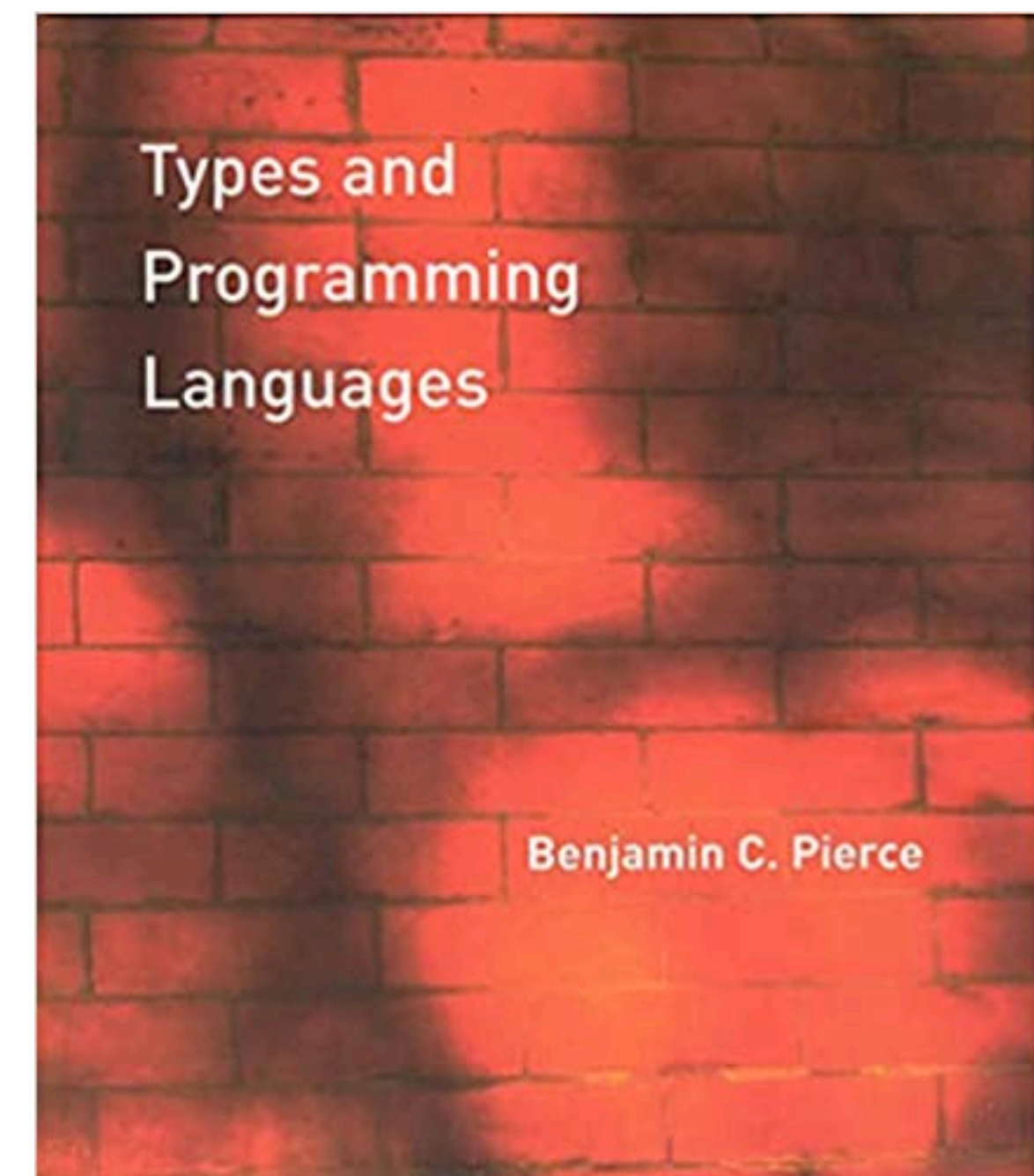
- In the interpreter, we only evaluate the body of a function when it's applied.
- In the type checker, we always check the body of the function (even if it's never applied.)
  - We *assume* the input has some type (say  $t_1$ ) and reflect this in the type of the function ( $t_1 \rightarrow t_2$ ).
- Dually, at a call site  $(e_1 e_2)$ , we don't know what *closure* we're going to get as  $e_1$ .
  - But we can calculate  $e_1$ 's type, check that  $e_2$  is an argument of the right type, and also determine what type will  $(e_1 e_2)$  have.
- **Question:** Why is this a valid approximation of the dynamic program behaviour?

# Contexts and Inference Rules

- Need to keep track of contextual information.
  - What variables are in scope?
  - What are their types?
  - What information do we have about each syntactic construct?
- What relationships are there among the syntactic objects?
  - e.g. is one type a subtype of another?
- How do we describe this information?
  - In the compiler there's a mapping from variables to information we know about them – the "context".
  - The compiler has a collection of (mutually recursive) functions that follow the structure of the syntax.

# Why Inference Rules?

- They are a compact, precise way of specifying language properties.
  - E.g. ~20 pages for full Java vs. 100's of pages of prose Java Language Spec.
  - Check out **oat-v1-typing.pdf**
- Inference rules correspond closely to the *recursive AST traversal* that implements them
- Type checking (and *type inference*) is nothing more than attempting to *prove* a judgment ( $G \vdash e : t$ ) by searching backwards through the rules.
- Strong mathematical foundations
  - The “Curry-Howard correspondence”:
    - Programming Language ~ Logic,
    - Program ~ Proof,
    - Type ~ Proposition
- Talk to me you're interested in type systems!





# Types and Type Safety

# Type Safety

**Theorem:** (type soundness of simply typed lambda calculus with integers)

If  $\vdash e : t$  then there exists a value  $v$  such that  $e \Downarrow v$ .

*"Well typed programs do not go wrong."*

– Robin Milner, 1978

- Note: this is a *very* strong property.
  - Well-typed programs cannot "go wrong" by trying to execute undefined code (such as  $3 + (\text{fun } x \rightarrow 2)$ )
  - Simply-typed lambda calculus is guaranteed to terminate! (i.e. it *isn't* Turing complete)

# Tuples

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

TUPLE

$$G \vdash e_1 : T_1 \quad \dots \quad G \vdash e_n : T_n$$

---

$$G \vdash (e_1, \dots, e_n) : T_1 * \dots * T_n$$

PROJ

$$G \vdash e : T_1 * \dots * T_n \quad 1 \leq i \leq n$$

---

$$G \vdash \#i e : T_i$$

# Arrays

- Array constructs are not hard
- First: add a new type constructor:  $T[]$

$$\boxed{\text{NEW}} \quad \frac{G \vdash e_1 : \text{int} \quad G \vdash e_2 : T}{G \vdash \text{new } T[e_1](e_2) : T[]}$$

$e_1$  is the size of the newly allocated array.  $e_2$  initialises the elements of the array.

$$\boxed{\text{INDEX}} \quad \frac{G \vdash e_1 : T[] \quad G \vdash e_2 : \text{int}}{G \vdash e_1[e_2] : T}$$

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

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

# References

- OCaml-style references (note that in OCaml references are expressions)
- First, add a new type constructor:  $T \text{ ref}$

REF

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

DEREF

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

ASSIGN

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

Note the similarity with the rules for arrays...

# Well-Formed Types

- In languages with type definitions, need additional rules to define well-formed types
- Judgements take the form  $H \vdash t$ 
  - $H$  is set of type names
  - $t$  is a type
  - $H \vdash t$   
means  
*“Assuming  $H$  lists well-formed types,  $t$  is a well-formed type”*

$$\text{INT} \quad \frac{}{H \vdash \text{int}}$$

$$\text{BOOL} \quad \frac{}{H \vdash \text{bool}}$$

$$\text{ARROW} \quad \frac{H \vdash t_1 \quad H \vdash t_2}{H \vdash t_1 \rightarrow t_2}$$

$$\text{NAMED} \quad \frac{}{H \vdash s} \quad s \in H$$

- Note: also need to modify the typing rules and judgements. E.g.,

$$\text{FUN} \quad \frac{H \vdash t_1 \quad H, \Gamma \{x \mapsto t_1\} \vdash e : t_2}{H, \Gamma \vdash \mathbf{fun} (x : t_1) \rightarrow e : t_1 \rightarrow t_2}$$

# Type-Checking Statements

- In languages with statements, need additional rules to define well-formed statements
- E.g., judgements may take the form  $H;G;rt \vdash s$ 
  - $H$  maps type names to their definitions
  - $G$  is a type environment (variables  $\rightarrow$  types)
  - $rt$  is a type
  - $H;G;rt \vdash s$

means

*“with type definitions  $H$ , assuming type environment  $G$ ,  $s$  is a valid statement within the context of a function that returns a value of type  $rt$ ”*

**ASSIGN**

$$\frac{\Gamma \vdash e : \Gamma(x)}{D; \Gamma; rt \vdash x := e}$$

**RETURN**

$$\frac{\Gamma \vdash e : rt}{D; \Gamma; rt \vdash \mathbf{return} e}$$

**DECL**

$$\frac{\Gamma \vdash e : t \quad D; \Gamma\{x \mapsto t\}; rt \vdash s_2}{D; \Gamma; rt \vdash \mathbf{var} x = e; s_2}$$

# Type Safety For General Languages

## Theorem: (Type Safety)

If  $\vdash P : t$  is a well-typed program, then either:

- (a) the program terminates in a well-defined way, or
- (b) the program continues computing forever

- Well-defined termination could include:
  - halting with a return value
  - raising an exception
- Type safety rules out undefined behaviours:
  - abusing "unsafe" casts: converting pointers to integers, etc.
  - treating non-code values as code (and vice-versa)
  - breaking the type abstractions of the language (e.g., via Java/Ruby reflection)
- What is "defined" depends on the language semantics...



# Types as Sets

# 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:

$$\begin{array}{ccc} \boxed{\text{P-INT}} & \boxed{\text{N-INT}} & \boxed{\text{ZERO}} \\ \frac{i > 0}{G \vdash i : \text{Pos}} & \frac{i < 0}{G \vdash i : \text{Neg}} & \frac{}{G \vdash 0 : \text{Zero}} \end{array}$$

- Same for booleans:

$$\begin{array}{cc} \boxed{\text{TRUE}} & \boxed{\text{FALSE}} \\ \frac{}{G \vdash \text{true} : \text{True}} & \frac{}{G \vdash \text{false} : \text{False}} \end{array}$$

# What about “if”?

- Two cases are easy:

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

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

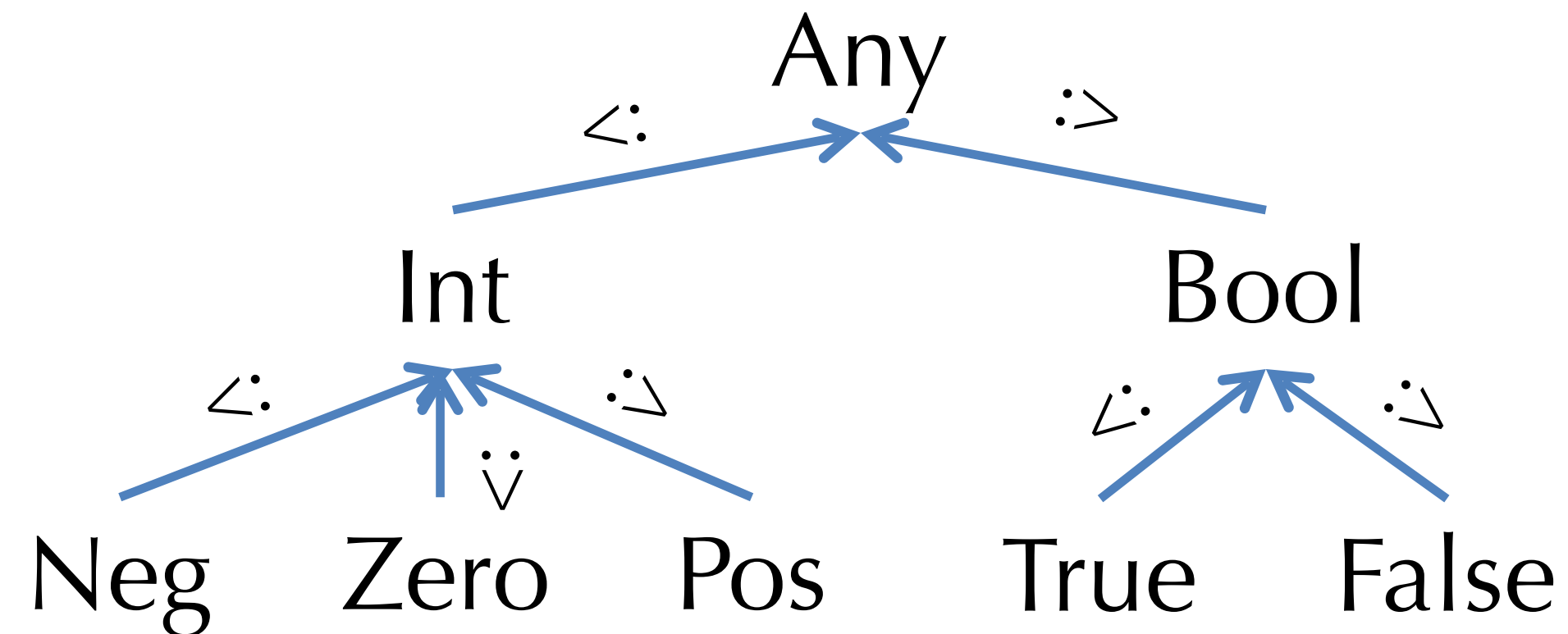
- What happens when we don't know statically which branch will be taken?
- Consider the type checking problem:

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

- 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:  $\text{Pos} \subseteq \text{Int}$
- This subset relation gives rise to a *subtype* relation:  $\text{Pos} <: \text{Int}$  (sometimes also typeset as  $\leq$ )
- 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:  $\text{LUB}(\text{True}, \text{False}) = \text{Bool}$ ,  $\text{LUB}(\text{Int}, \text{Bool}) = \text{Any}$
  - 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

$$G \vdash e_1 : \text{bool} \quad E \vdash e_2 : T_1 \quad G \vdash e_3 : T_2$$

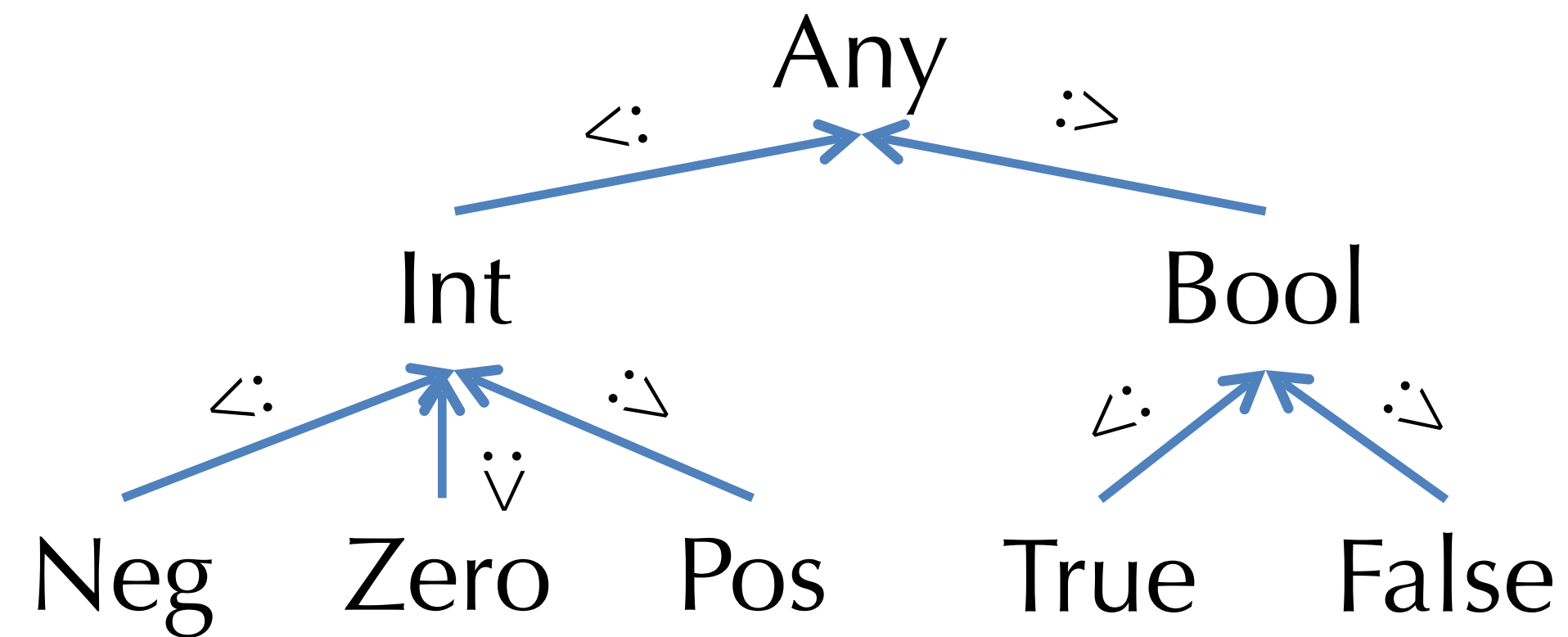
---

$$G \vdash \text{if } (e_1) e_2 \text{ else } e_3 : \text{LUB}(T_1, T_2)$$

- Note that  $\text{LUB}(T_1, T_2)$  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,  $\text{LUB}(T_1, T_2)$  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:  $T \langle: T$  for any type  $T$
  - Transitive:  $T_1 \langle: T_2$  and  $T_2 \langle: T_3$  then  $T_1 \langle: T_3$
  - Antisymmetric: If  $T_1 \langle: T_2$  and  $T_2 \langle: T_1$  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  $\llbracket T \rrbracket$  for the subset of (closed) values of type  $T$ 
  - i.e.  $\llbracket T \rrbracket = \{v \mid \vdash v : T\}$
  - e.g.  $\llbracket \text{Zero} \rrbracket = \{0\}$ ,  $\llbracket \text{Pos} \rrbracket = \{1, 2, 3, \dots\}$
- If  $T_1 <: T_2$  implies  $\llbracket T_1 \rrbracket \subseteq \llbracket T_2 \rrbracket$ , then  $T_1 <: T_2$  is sound.
  - e.g.  $\text{Pos} <: \text{Int}$  is sound, since  $\{1, 2, 3, \dots\} \subseteq \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$
  - e.g.  $\text{Int} <: \text{Pos}$  is not sound, since it is *not* the case that  $\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\} \subseteq \{1, 2, 3, \dots\}$



# Subsumption Rule

- When we add subtyping judgments of the form  $T <: S$  we can uniformly integrate it into the type system generically:

$$\boxed{\text{SUBSUMPTION}} \quad \frac{G \vdash e : T \quad T <: S}{G \vdash e : S}$$

- Subsumption allows any value of type  $T$  to be treated as an  $S$  whenever  $T <: S$ .
- Adding this rule makes the search for typing derivations more difficult – this rule can be applied anywhere, since  $T <: T$ .
  - But careful engineering of the typing system can incorporate the subsumption rule into a deterministic algorithm. (See, e.g., the Oat type system)

# Downcasting

- What happens if we have an Int but need something of type Pos?
  - At compile time, we don't know whether the Int is greater than zero.
  - At run time, we do.

- Add a “checked downcast”

$$G \vdash e_1 : \text{Int} \quad G, x : \text{Pos} \vdash e_2 : T_2 \quad G \vdash e_3 : T_3$$

---

$$G \vdash \text{ifPos } (x = e_1) e_2 \text{ else } e_3 : \text{LUB}(T_2, T_3)$$

- At runtime, ifPos checks whether  $e_1$  is  $> 0$ .  
If so, branches to  $e_2$  and otherwise branches to  $e_3$ .
- Inside the expression  $e_2$ ,  $x$  is the name for  $e_1$ 's value, which is known to be strictly positive because of the dynamic check.
- Note that such rules force the programmer to add the appropriate checks
  - We could give integer division the type:  $\text{Int} \rightarrow \text{NonZero} \rightarrow \text{Int}$

# Subtyping in the Wild

# Extending Subtyping to Other Types

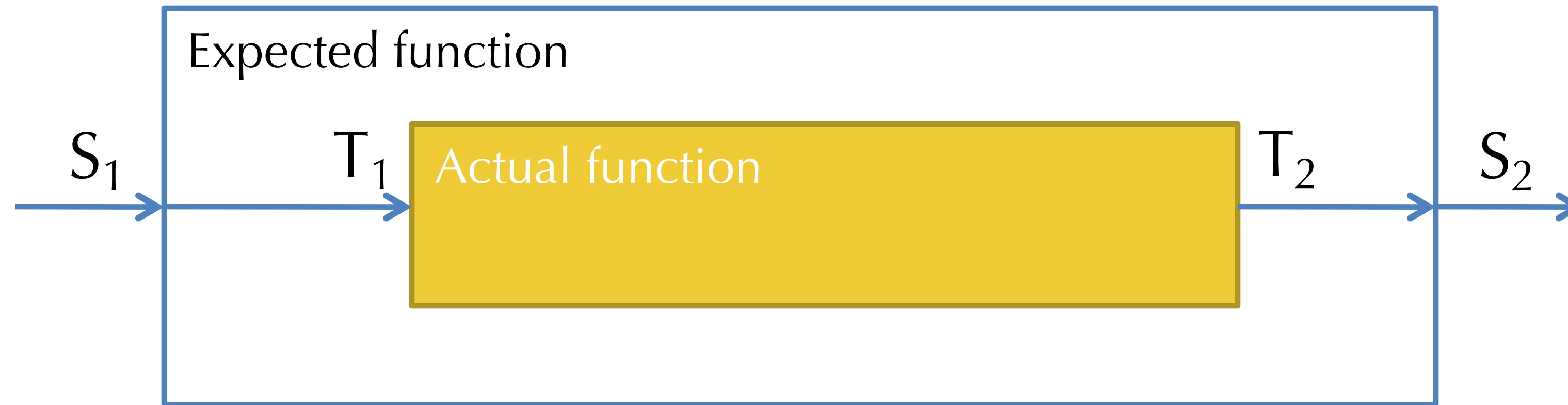
- What about subtyping for tuples?
  - Intuition: whenever a program expects something of type  $S_1 * S_2$ , it is sound to give it a  $T_1 * T_2$ .
  - Example:  $(\text{Pos} * \text{Neg}) <: (\text{Int} * \text{Int})$

$$\frac{T_1 <: S_1 \quad T_2 <: S_2}{(T_1 * T_2) <: (S_1 * S_2)}$$

- What about functions?
- When is  $T_1 \rightarrow T_2 <: S_1 \rightarrow S_2$  ?

# Subtyping for Function Types

- One way to see it:



- Need to convert an  $S_1$  to a  $T_1$  and  $T_2$  to  $S_2$ , so the argument type is *contravariant* and the output type is *covariant*.

$$\frac{S_1 <: T_1 \quad T_2 <: S_2}{(T_1 \rightarrow T_2) <: (S_1 \rightarrow S_2)}$$

# Immutable Records

- Record type:  $\{\text{lab}_1:T_1; \text{lab}_2:T_2; \dots ; \text{lab}_n:T_n\}$ 
  - Each  $\text{lab}_i$  is a label drawn from a set of identifiers.

**RECORD**  $G \vdash e_1 : T_1 \quad G \vdash e_2 : T_2 \quad \dots \quad G \vdash e_n : T_n$

---

$G \vdash \{\text{lab}_1 = e_1; \text{lab}_2 = e_2; \dots ; \text{lab}_n = e_n\} : \{\text{lab}_1:T_1; \text{lab}_2:T_2; \dots ; \text{lab}_n:T_n\}$

**PROJECTION**  $G \vdash e : \{\text{lab}_1:T_1; \text{lab}_2:T_2; \dots ; \text{lab}_n:T_n\}$

---

$G \vdash e.\text{lab}_i : T_i$

# Immutable Record Subtyping

- Depth subtyping:
  - Corresponding fields may be subtypes

$$\boxed{\text{DEPTH}} \quad T_1 <: U_1 \quad T_2 <: U_2 \quad \dots \quad T_n <: U_n$$

---

$$\{\text{lab}_1:T_1; \text{lab}_2:T_2; \dots ; \text{lab}_n:T_n\} <: \{\text{lab}_1:U_1; \text{lab}_2:U_2; \dots ; \text{lab}_n:U_n\}$$

- Width subtyping:
  - Subtype record may have *more* fields:

$$\boxed{\text{WIDTH}} \quad m \leq n$$

---

$$\{\text{lab}_1:T_1; \text{lab}_2:T_2; \dots ; \text{lab}_n:T_n\} <: \{\text{lab}_1:T_1; \text{lab}_2:T_2; \dots ; \text{lab}_m:T_m\}$$

# Mutability and Subtyping



# NULL

- What is the type of null?
- Consider:  
int[] a = null; // OK?  
int x = null; // OK? (nope)  
string s = null; // OK?

NULL       $\frac{}{G \vdash \text{null} : r}$

- Null has **any** *reference type*
  - Null is generic
- What about type safety?
  - Requires defined behavior when dereferencing null  
e.g. Java's NullPointerException
  - Requires a safety check for every dereference operation

# Subtyping and References

- What is the proper subtyping relationship for references and arrays?
- Suppose we have NonZero as a type and the division operation has type:  
 $\text{Int} \rightarrow \text{NonZero} \rightarrow \text{Int}$ 
  - Recall that  $\text{NonZero} <: \text{Int}$
- Should  $(\text{NonZero ref}) <: (\text{Int ref})$  ?
- Consider this program:

```
Int bad(NonZero ref r) {  
  Int ref a = r;      (* OK because (NonZero ref <: Int ref*)  
  a := 0;             (* OK because 0 : Zero <: Int *)  
  return (42 / !r)    (* OK because !r has type NonZero *)  
}
```

# Mutable Structures are Invariant

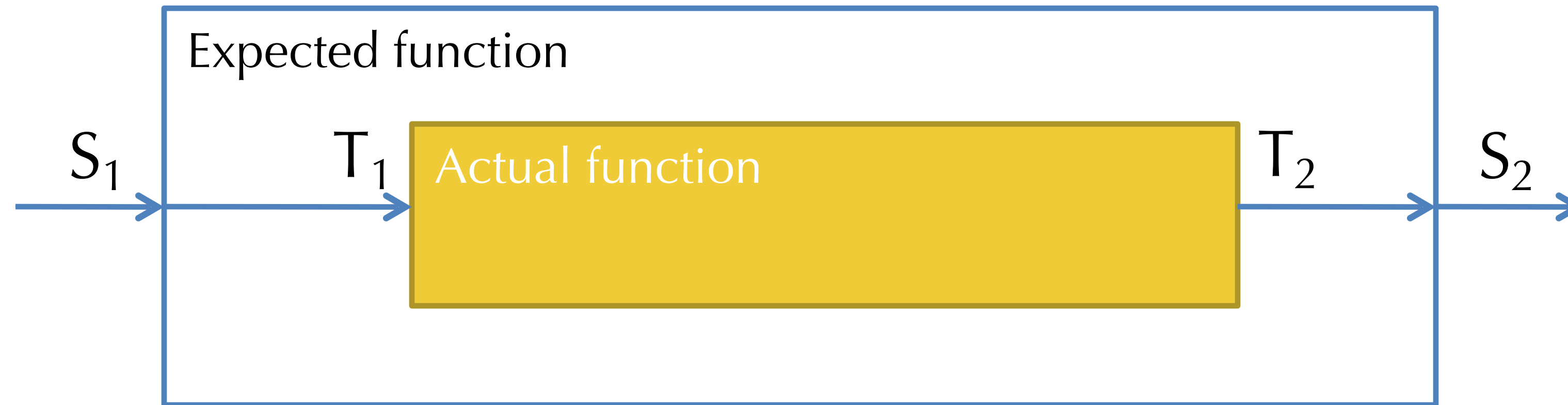
- Covariant reference types are **unsound** (well-typed programs *do go wrong*)
  - As demonstrated in the previous example
- Contravariant reference types are also unsound
  - i.e. If  $T_1 <: T_2$  then  $\text{ref } T_2 <: \text{ref } T_1$  is also unsound
  - Exercise: construct a program that breaks contravariant references.
- Moral: Mutable structures are invariant:
 
$$T_1 \text{ ref } <: T_2 \text{ ref} \quad \text{implies} \quad T_1 = T_2$$
- Same holds for arrays, OCaml-style mutable records, object fields, etc.
  - Note: Java and C# get this wrong. They allow covariant array subtyping, but then compensate by adding a dynamic check on *every* array update!

Let  $\Gamma = [x \mapsto \text{nat array}]$

$$\text{ASSN} \frac{\text{SUB} \frac{\text{VAR} \frac{}{\Gamma \vdash x : \text{nat array}} \quad \text{ARRAY} \frac{\text{NATINT} \frac{}{\text{nat} <: \text{int}}}{\text{nat array} <: \text{int array}}}{\Gamma \vdash x : \text{int array}} \quad \text{NAT} \frac{}{\Gamma \vdash 0 : \text{nat}} \quad \text{INT} \frac{}{\Gamma \vdash -1 : \text{int}}}{\Gamma \vdash x[0] := -1}}$$

# Reminder: Subtyping for Function Types

- One way to see it:



- Need to convert an  $S_1$  to a  $T_1$  and  $T_2$  to  $S_2$ , so the argument type is *contravariant* and the output type is *covariant*.

$$\frac{S_1 <: T_1 \quad T_2 <: S_2}{(T_1 \rightarrow T_2) <: (S_1 \rightarrow S_2)}$$

# Another Way to See It

- We can think of a reference cell as an immutable record (object) with two functions (methods) and some hidden state:

$$\text{T ref} \approx \{\text{get: unit} \rightarrow \text{T}; \text{ set: T} \rightarrow \text{unit}\}$$

- get returns the value hidden in the state.
  - set updates the value hidden in the state.
- When is  $\text{T ref} <: \text{S ref}$ ?
  - Records are like tuples: subtyping extends pointwise over each component.
  - $\{\text{get: unit} \rightarrow \text{T}; \text{ set: T} \rightarrow \text{unit}\} <: \{\text{get: unit} \rightarrow \text{S}; \text{ set: S} \rightarrow \text{unit}\}$ 
    - get components are subtypes:  $\text{unit} \rightarrow \text{T} <: \text{unit} \rightarrow \text{S}$
    - set components are subtypes:  $\text{T} \rightarrow \text{unit} <: \text{S} \rightarrow \text{unit}$
  - From get, we must have  $\text{T} <: \text{S}$  (covariant return)
  - From set, we must have  $\text{S} <: \text{T}$  (contravariant arg.)
  - From  $\text{T} <: \text{S}$  and  $\text{S} <: \text{T}$  we conclude  $\text{T} = \text{S}$ .

# Demo: Arrays in Java

- Check out <https://github.com/ysc4230/week-10-java-arrays>
- The code shows the run-time issue with covariant array subtyping

# Structural vs. Nominal Subtyping

# Structural vs. Nominal Typing

- Is type equality / subsumption defined by the *structure* of the data or the *name* of the data?
- Example: type abbreviations (OCaml) vs. “newtypes” (a la Haskell)

```
(* OCaml: *)  
type cents = int (* cents = int in this scope *)  
type age = int  
  
let foo (x:cents) (y:age) = x + y
```

```
(* Haskell: *)  
newtype Cents = Cents Integer (* Integer and Cents are isomorphic, not identical. *)  
newtype Age = Age Integer  
  
foo :: Cents -> Age -> Int  
foo x y = x + y (* Ill typed! *)
```

- Type abbreviations are treated “structurally” Newtypes are treated “by name”.



# Nominal Subtyping in Java

- In Java, Classes and Interfaces must be named and their relationships *explicitly* declared:

```
(* Java: *)
interface Foo {
    int foo();
}

class C {          /* Does not implement the Foo interface */
    int foo() {return 2;}
}

class D implements Foo {
    int foo() {return 4230;}
}
```

- Similarly for inheritance: the subclass relation must be declared via the “extends” keyword.
  - Typechecker still checks that the classes are structurally compatible

# Oat's Type System

# Oat's Treatment of Types

- Primitive (non-reference) types:
  - int, bool
- Definitely non-null reference types: R
  - (named) mutable structs with (right-oriented) *width* subtyping
  - string
  - arrays (including length information, per HW4)
- Possibly-null reference types: R?
  - Subtyping:  $R <: R?$
  - *Checked downcast* syntax if?:

```
int sum(int[]? arr) {  
    var z = 0;  
    if? (int[] a = arr) {  
        for(var i = 0; i < length(a); i = i + 1;) {  
            z = z + a[i];  
        }  
    }  
    return z;  
}
```

# Full Oat Features

- Named structure types with mutable fields
  - but using structural, width subtyping
- Typed function pointers
- Polymorphic operations: `length` and `== / !=`
  - need special case handling in the typechecker
- Type-annotated null values: `t null` always has type `t`?
- Definitely-not-null values means we need an "atomic" array initialization syntax
  - for example, `null` is not allowed as a value of type `int[]`, so to construct a record containing a field of type `int[]`, we need to initialize it
  - subtlety: `int[][]` cannot be initialized by default, but `int[]` can be

# Oat "Returns" Analysis

- Type-safe, statement-oriented imperative languages like Oat (or Java) must ensure that a function (always) **returns** a value of the appropriate type.
  - Does the returned expression's type match the one declared by the function?
  - Do all paths through the code return appropriately?
- Oat's statement checking judgment
  - takes the expected return type as input: what type should the statement return (or void if none)
  - produces a boolean flag as output: does the statement definitely return?

$$H; G; L_1; rt \vdash stmt \Rightarrow L_2; returns$$

$$\frac{H; G; L \vdash exp : t' \quad H \vdash t' \leq t}{H; G; L; t \vdash \text{return } exp; \Rightarrow L; \top} \text{TYP\_RET T}$$

$$\frac{\begin{array}{l} H; G; L \vdash exp : \text{bool} \\ H; G; L; rt \vdash block_1; r_1 \\ H; G; L; rt \vdash block_2; r_2 \end{array}}{H; G; L; rt \vdash \text{if}(exp) block_1 \text{ else } block_2 \Rightarrow L; r_1 \wedge r_2} \text{TYP\_IF}$$

$$\frac{\begin{array}{l} H; G; L \vdash exp : \text{bool} \\ H; G; L; rt \vdash block; r \end{array}}{H; G; L; rt \vdash \text{while}(exp) block \Rightarrow L; \perp} \text{TYP\_WHILE}$$

$$\frac{\begin{array}{l} H; G; L \vdash exp : (t_1, \dots, t_n) \rightarrow \text{void} \\ H; G; L \vdash exp_1 : t'_1 \quad \dots \quad H; G; L \vdash exp_n : t'_n \\ H \vdash t'_1 \leq t_1 \quad \dots \quad H \vdash t'_n \leq t_n \end{array}}{H; G; L; rt \vdash exp(exp_1, \dots, exp_n); \Rightarrow L; \perp} \text{TYP\_SCALL}$$

**Example: Typing in Oat**

# Checking Derivations

- A *derivation* or *proof tree* has (instances of) judgments as its nodes and edges that connect premises to a conclusion according to an inference rule.
- Leaves of the tree are *axioms* (i.e. rules with no premises)
  - Example: the INT rule is an axiom
- Goal of the type checker: verify that such a tree exists.
- Example 1: Find a tree for the following program using the inference rules in Oat specification

```
var x1 = 0;  
var x2 = x1 + x1;  
x1 = x1 - x2;  
return(x1);
```

Example 2: There is no tree for this ill-typed program:

```
int f() {  
    var x = int[] null;  
    x = new int[] {3,4};  
    return x[0];  
}
```

# Example Derivation

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
```

$D_1$   
 $D_2$   
 $D_2$   
 $D_4$

$$H;G;L_0;rt \vdash_{ss} stmt_1 .. stmt_n \Rightarrow L_n;returns$$

$$H;G;L_0;rt \vdash stmt_1 \Rightarrow L_1;\perp$$

...

$$H;G;L_{n-2};rt \vdash stmt_{n-1} \Rightarrow L_{n-1};\perp$$

$$H;G;L_{n-1};rt \vdash stmt_n \Rightarrow L_n;r$$

$$\frac{H;G;L_{n-1};rt \vdash stmt_n \Rightarrow L_n;r}{H;G;L_0;rt \vdash_{ss} stmt_1 .. stmt_{n-1} stmt_n \Rightarrow L_n;r} \text{ TYP\_STMTS}$$

$$\frac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \mathcal{D}_3 \quad \mathcal{D}_4}{H;G; \cdot ; \mathbf{int} \vdash_{ss} \mathbf{var} x_1 = 0; \mathbf{var} x_2 = x_1 + x_2; x_1 = x_1 - x_2; \mathbf{return} x_1; \Rightarrow x_1:\mathbf{int}, x_2:\mathbf{int}, \cdot; \top} \text{ TYP\_STMTS}$$



# Example Derivation

$$\frac{H;G;L_1 \vdash vdecl \Rightarrow L_2}{H;G;L_1;rt \vdash vdecl; \Rightarrow L_2; \perp} \quad \text{TYP\_STMTDECL}$$

$$\frac{H;G;L \vdash exp : t \quad x \notin L}{H;G;L \vdash \text{var } x = exp \Rightarrow L, x:t} \quad \text{TYP\_DECL}$$

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
```

$$\frac{}{H;G;L \vdash integer : int} \quad \text{TYP\_INT}$$

$\mathcal{D}_1 =$

$$\frac{\frac{\frac{}{H;G;\cdot \vdash 0 : int} \quad \text{TYP\_INT} \quad x_1 \notin \cdot}{H;G;\cdot \vdash \text{var } x_1 = 0 \Rightarrow \cdot, x_1:int} \quad \text{TYP\_DECL}}{H;G;:,int \vdash \text{var } x_1 = 0; \Rightarrow \cdot, x_1:int; \perp} \quad \text{TYP\_STMTDECL}}$$

# Example Derivation

$$\frac{H;G;L_1 \vdash vdecl \Rightarrow L_2}{H;G;L_1;rt \vdash vdecl; \Rightarrow L_2; \perp} \quad \text{TYP\_STMTDECL}$$

$$\frac{H;G;L \vdash exp : t \quad x \notin L}{H;G;L \vdash \text{var } x = exp \Rightarrow L, x:t} \quad \text{TYP\_DECL}$$

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
```

$$\frac{}{H;G;L \vdash \text{integer} : \text{int}} \quad \text{TYP\_INT}$$

$$\frac{id:t \in L}{H;G;L \vdash id : t} \quad \text{TYP\_LOCAL}$$

$\mathcal{D}_2 =$

$$\frac{\frac{\frac{}{\vdash +:(\text{int}, \text{int}) \rightarrow \text{int}} \quad \text{TYP\_INTOPS}}{x_1:\text{int} \in \cdot, x_1:\text{int}} \quad \text{TYP\_LOCAL}}{H;G;\cdot, x_1:\text{int} \vdash x_1 : \text{int}} \quad \text{TYP\_LOCAL} \quad \frac{x_1:\text{int} \in \cdot, x_1:\text{int}}{H;G;\cdot, x_1:\text{int} \vdash x_1 : \text{int}} \quad \text{TYP\_LOCAL}}{H;G;\cdot, x_1:\text{int} \vdash x_1 + x_1 : \text{int}} \quad \text{TYP\_BOP} \quad x_2 \notin \cdot, x_1:\text{int}} \quad \text{TYP\_DECL}}{H;G;\cdot \vdash \text{var } x_2 = x_1 + x_1 \Rightarrow \cdot, x_1:\text{int}, x_2:\text{int}} \quad \text{TYP\_DECL}}{H;G;\cdot, x_1:\text{int}; \text{int} \vdash \text{var } x_2 = x_1 + x_1; \Rightarrow \cdot, x_1:\text{int}, x_2:\text{int}; \perp} \quad \text{TYP\_STMTDECL}$$

# Example Derivation

$$\frac{H;G;L_1 \vdash vdecl \Rightarrow L_2}{H;G;L_1;rt \vdash vdecl; \Rightarrow L_2; \perp} \text{ TYP\_STMTDECL}$$

$$\frac{H;G;L \vdash exp : t \quad x \notin L}{H;G;L \vdash \text{var } x = exp \Rightarrow L, x:t} \text{ TYP\_DECL}$$

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
```

$$\frac{}{H;G;L \vdash integer : int} \text{ TYP\_INT}$$

$$\frac{id:t \in L}{H;G;L \vdash id : t} \text{ TYP\_LOCAL}$$

$\mathcal{D}_5 =$

$$\frac{\frac{\frac{}{\vdash -:(int, int) \rightarrow int} \text{ TYP\_INTOPS}}{x_1:int \in \cdot, x_1:int, x_2:int} \text{ TYP\_LOCAL} \quad \frac{x_2:int \in \cdot, x_1:int, x_2:int}{H;G;\cdot, x_1:int, x_2:int \vdash x_2 : int} \text{ TYP\_LOCAL}}{H;G;\cdot, x_1:int, x_2:int \vdash x_1 : int} \text{ TYP\_LOCAL} \quad \frac{}{H;G;\cdot, x_1:int, x_2:int \vdash x_1 - x_1 : int} \text{ TYP\_BOP}}$$

$\mathcal{D}_3 =$

$$\frac{G \vdash x_1 \text{ not a global function id} \quad \frac{x_1:int \in \cdot, x_1:int, x_2:int}{H;G;\cdot, x_1:int, x_2:int \vdash x_1 : int} \text{ TYP\_LOCAL} \quad \frac{}{H \vdash int \leq int} \text{ SUB\_SUB\_INT} \quad \mathcal{D}_5}{H;G;\cdot, x_1:int, x_2:int; int \vdash x_1 = x_1 - x_2; \Rightarrow \cdot, x_1:int, x_2:int; \perp} \text{ TYP\_ASSN}$$

# Example Derivation

$$\frac{H;G;L \vdash \text{exp} : t' \quad H \vdash t' \leq t}{H;G;L;t \vdash \text{return exp}; \Rightarrow L;\top} \text{TYP\_RET}\top$$

$$\frac{}{H \vdash \text{int} \leq \text{int}} \text{SUB\_SUB\_INT}$$

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
```

$$\frac{id:t \in L}{H;G;L \vdash id : t} \text{TYP\_LOCAL}$$

$\mathcal{D}_4 =$

$$\frac{\frac{x_1:\text{int} \in \cdot, x_1:\text{int}, x_2:\text{int}}{H;G;\cdot, x_1:\text{int}, x_2:\text{int} \vdash x_1 : \text{int}} \text{TYP\_LOCAL} \quad \frac{}{H \vdash \text{int} \leq \text{int}} \text{SUB\_SUB\_INT}}{H;G;\cdot, x_1:\text{int}, x_2:\text{int}; \text{int} \vdash \text{return } x_1; \Rightarrow \cdot, x_1:\text{int}, x_2:\text{int}; \top} \text{TYP\_RET}\top$$

# Example: Ill-Typed Oat Program

```
int f() {  
    var x = int[] null;  
    x = new int[] {3,4};  
    return x[0];  
}
```

# Next in this Module

- Making our programs *faster*