CS4212: Compiler Design

Week 11: Type Safety
Typing for Advanced Features

Ilya Sergey

ilya@nus.edu.sg

ilyasergey.net/CS4212/

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:
    $x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n$
- A type judgement takes the form $G \vdash e : t$
  "Under the type environment G, the expression e has type t"
- For example:
  x : int, b : bool ⊢ if (b) 3 else x : 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 : int, b : bool ⊢ b : bool
  - 3 must be an int i.e. x : int, b : bool ⊢ 3 : int
  - x must be an int i.e. x : int, b : bool ⊢ x : int

Simply-typed Lambda Calculus with Integers

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

INT

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

VAR

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

ADD

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

FUN

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

APP

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

- Note how these rules correspond to the OCaml code.

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.

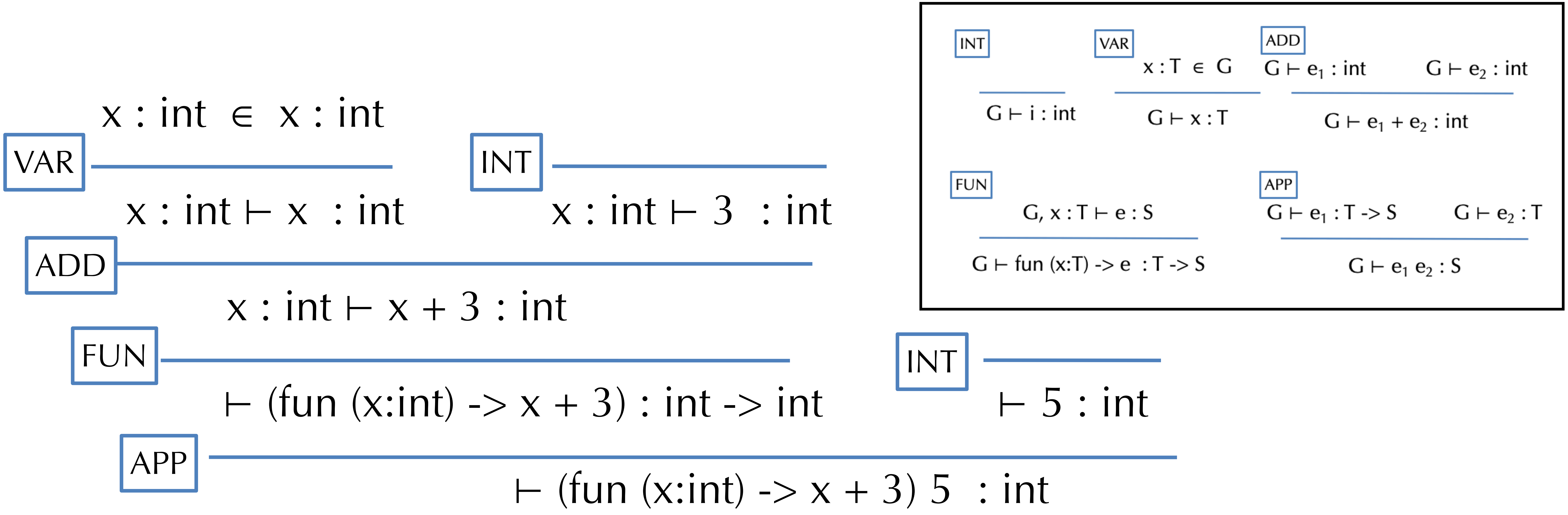
- Leaves of the tree are *axioms* (i.e. rules with no premises)
  - Example: the INT rule is an axiom

- 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:

$$\vdash \text{(fun (x:int) -> x + 3) 5} \;\; : \text{int}$$

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

  $\vdash$ (fun (x:int) → x 3) 5 : T

INT
$$\frac{}{G \vdash i : int}$$

VAR
$$\frac{x : T \in G}{G \vdash x : T}$$

ADD
$$\frac{G \vdash e_1 : int \qquad G \vdash e_2 : int}{G \vdash e_1 + e_2 : int}$$

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

APP
$$\frac{G \vdash e_1 : T \text{ -> } S \qquad G \vdash e_2 : T}{G \vdash e_1\, e_2 : S}$$

VAR
$$\frac{x : int \rightarrow T \;\notin\; x : int}{x : int \vdash x \;:\; int \rightarrow T}$$

APP
$$\frac{x : int \vdash x \;:\; int \rightarrow T \qquad x : int \vdash 3 \;:\; int}{x : int \vdash x\,3 : T}$$

FUN
$$\frac{x : int \vdash x\,3 : T}{\vdash \text{(fun (x:int)} \rightarrow \text{x 3)} : int \rightarrow T}$$

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

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 -> 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 doe 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?

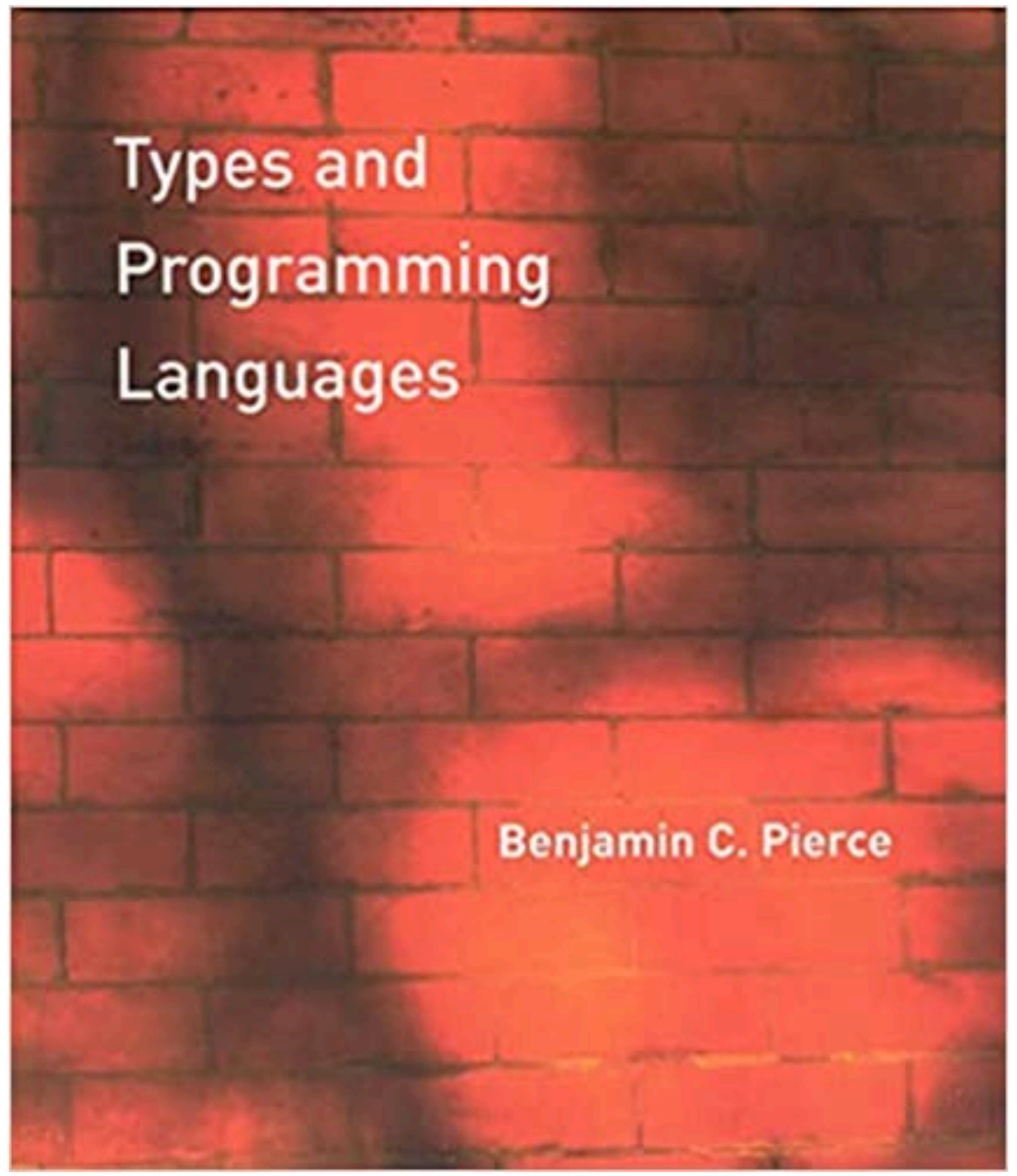
- They are a compact, precise way of specifying language properties.
  - E.g. ~20 pages for full Java vs. 100's of pages of prose Java Language Spec.
  - Check out **oat-v1-typing.pdf**

- 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 ⊢ 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 + (fun x -> 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: $T_1 * \ldots * T_n$

TUPLE

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

PROJ

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

A note on Curry-Howard Correspondence

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

VAR
$$\frac{x : T \in G}{G \vdash x : T}$$

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

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

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

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

PROJ
$$\frac{G \vdash e : T_1 * \ldots * T_n \qquad 1 \leq i \leq n}{G \vdash \#i\ e \ : \ T_i}$$

Arrays

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

NEW

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

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

INDEX

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

UPDATE

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

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 ref

REF

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

DEREF

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

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 ⊢ t
  - H is set of type names
  - t is atype
  - H ⊢ t
    means
    "Assuming H lists well-formed types, t is a well-formed type"

INT
$$\frac{}{H \vdash \texttt{int}}$$

BOOL
$$\frac{}{H \vdash \texttt{bool}}$$

ARROW
$$\frac{H \vdash t_1 \qquad H \vdash t_2}{H \vdash t_1 \to t_2}$$

NAMED
$$\frac{}{H \vdash s}\; s \in H$$

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

FUN
$$\frac{H \vdash t_1 \qquad H, \Gamma\{x \mapsto t_1\} \vdash e : t_2}{H, \Gamma \vdash \textbf{fun}\ (x : t_1)\texttt{->}e : t_1 \to t_2}$$

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 ⊢ s
  - H maps type names to their definitions
  - G is a type environment (variables -> types)
  - rt is a type
  - H;G;rt ⊢ s
    means
    "with type definitions H, assuming type environment G, s is avalid 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
$$\frac{\Gamma \vdash e : rt}{D; \Gamma; rt \vdash \textbf{return } e}$$

DECL
$$\frac{\Gamma \vdash e : t \qquad D; \Gamma\{x \mapsto t\}; rt \vdash s_2}{D; \Gamma; rt \vdash \textbf{var } x = e; s_2}$$

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…

A good place for a break

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:

P-INT

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

N-INT

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

ZERO

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

- Same for booleans:

TRUE

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

FALSE

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

What about "if"?

- Two cases are easy:

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

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

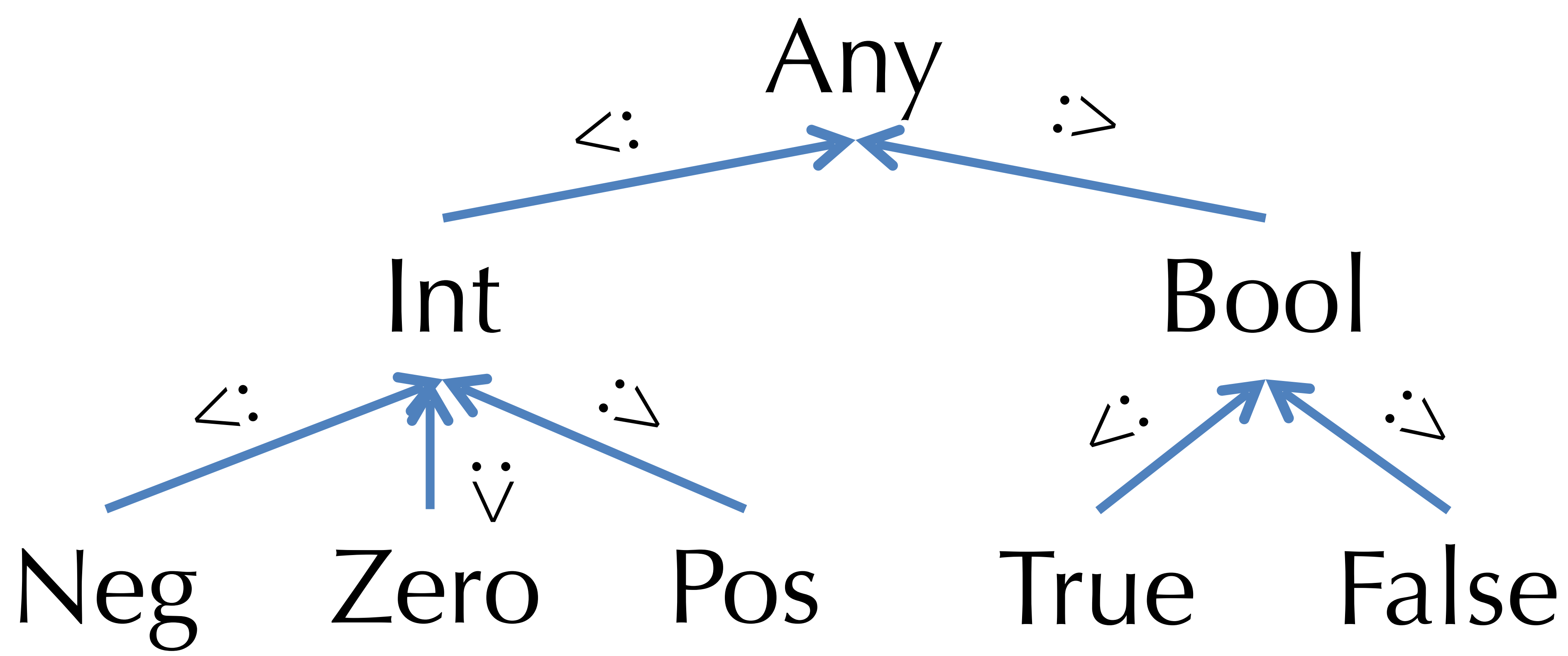
- What happens when we don't know statically which branch will be taken?
- Consider the type checking problem:

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

- 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: Pos $\subseteq$ Int
- This subset relation gives rise to a *subtype* relation: Pos <: 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: 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:

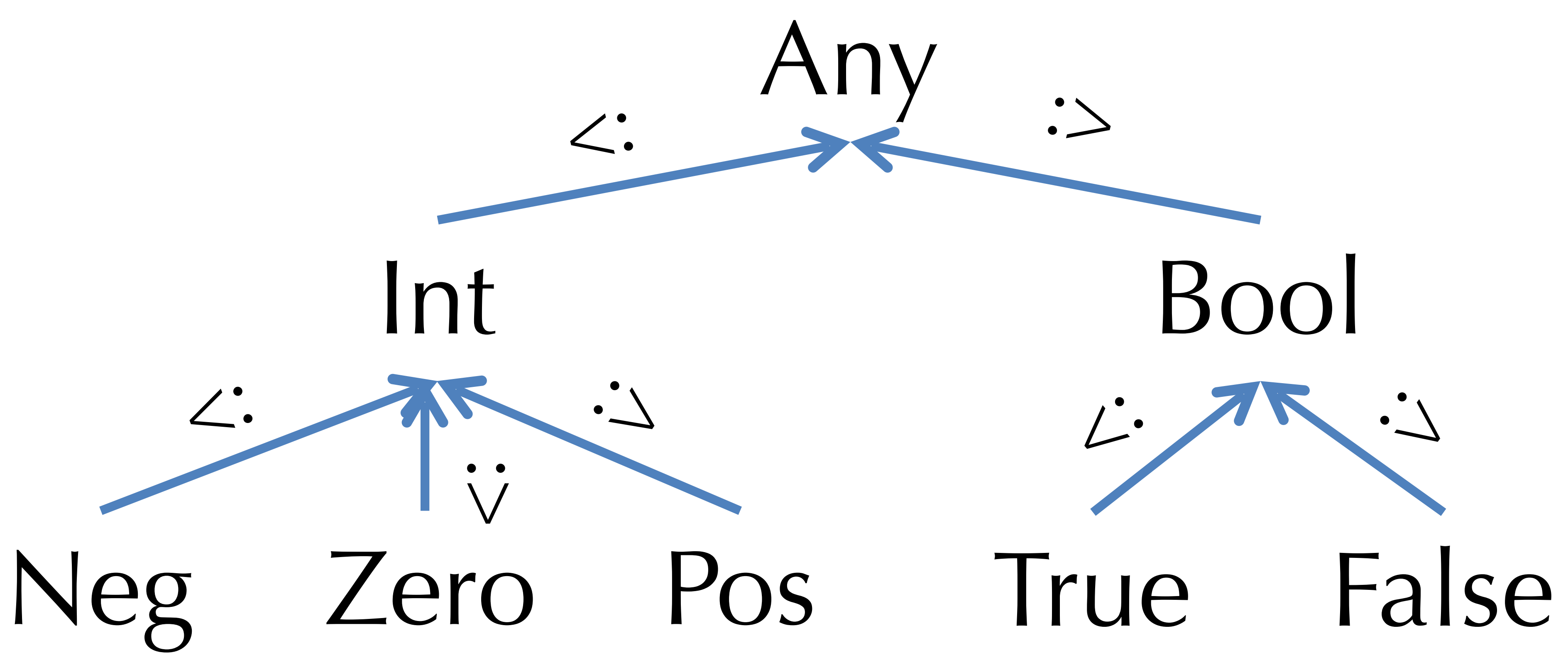
IF-BOOL

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

- 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 $\{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \subseteq \{1,2,3,\ldots\}$

Subsumption Rule

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

SUBSUMPTION

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

- 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)

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: (Pos * Neg) <: (Int * Int)

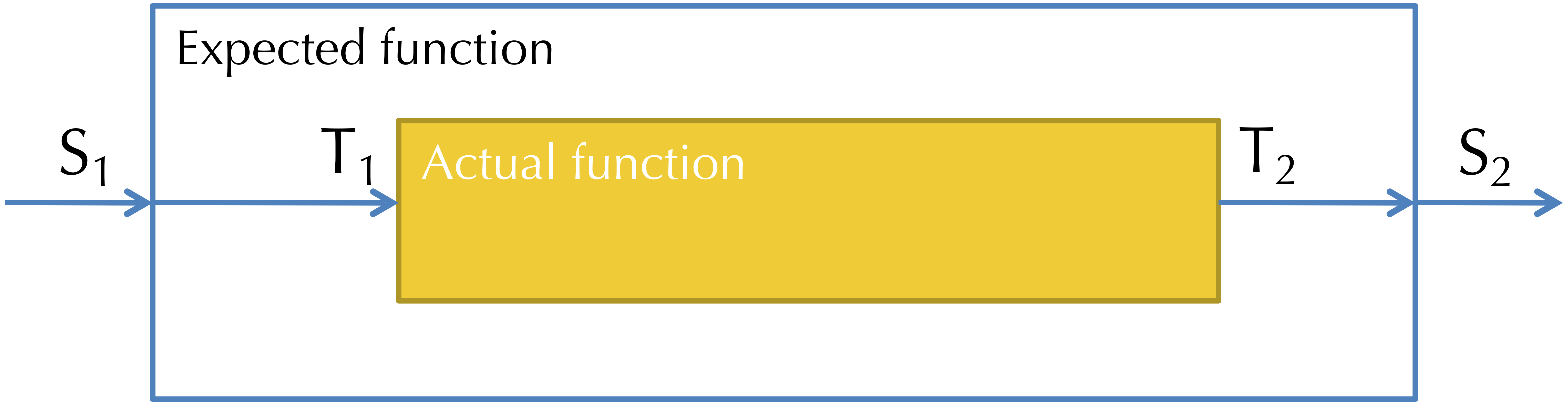
$$\frac{T_1 <: S_1 \quad T_2 <: S_2}{(T_1 * T_2) <: (S_1 * S_2)}$$

- 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 S1 to a T1 and T2 to S2, so the argument type is *contravariant* and the output type is *covariant*.

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

Immutable Records

- Record type: $\{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}$
  - Each lab_i is a label drawn from a set of identifiers.

RECORD

$$\frac{G \vdash e_1 : T_1 \qquad G \vdash e_2 : T_2 \qquad \ldots \qquad G \vdash e_n : T_n}{G \vdash \{lab_1 = e_1;\ lab_2 = e_2;\ \ldots\ ;\ lab_n = e_n\} : \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}}$$

PROJECTION

$$\frac{G \vdash e : \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}}{G \vdash e.lab_i : T_i}$$

Immutable Record Subtyping

- Depth subtyping:
  - Corresponding fields may be subtypes

DEPTH

$$\frac{T_1 <: U_1 \quad T_2 <: U_2 \quad \dots \quad T_n <: U_n}{\{lab_1{:}T_1;\ lab_2{:}T_2;\ \dots\ ;\ lab_n{:}T_n\} <: \{lab_1{:}U_1;\ lab_2{:}U_2;\ \dots\ ;\ lab_n{:}U_n\}}$$

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

WIDTH

$$\frac{m \leq n}{\{lab_1{:}T_1;\ lab_2{:}T_2;\ \dots\ ;\ lab_n{:}T_n\} <: \{lab_1{:}T_1;\ lab_2{:}T_2;\ \dots\ ;\ lab_m{:}T_m\}}$$

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: Int → NonZero → Int
  - Recall that NonZero <: Int
- Should (NonZero ref) <: (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 ref $T_2 <:$ ref T_1 is also unsound
  - Exercise: construct a program that breaks contravariant references.
- Moral: Mutable structures are invariant:

  $$T_1 \text{ ref} <: T_2 \text{ ref} \quad \text{implies} \quad T_1 = T_2$$

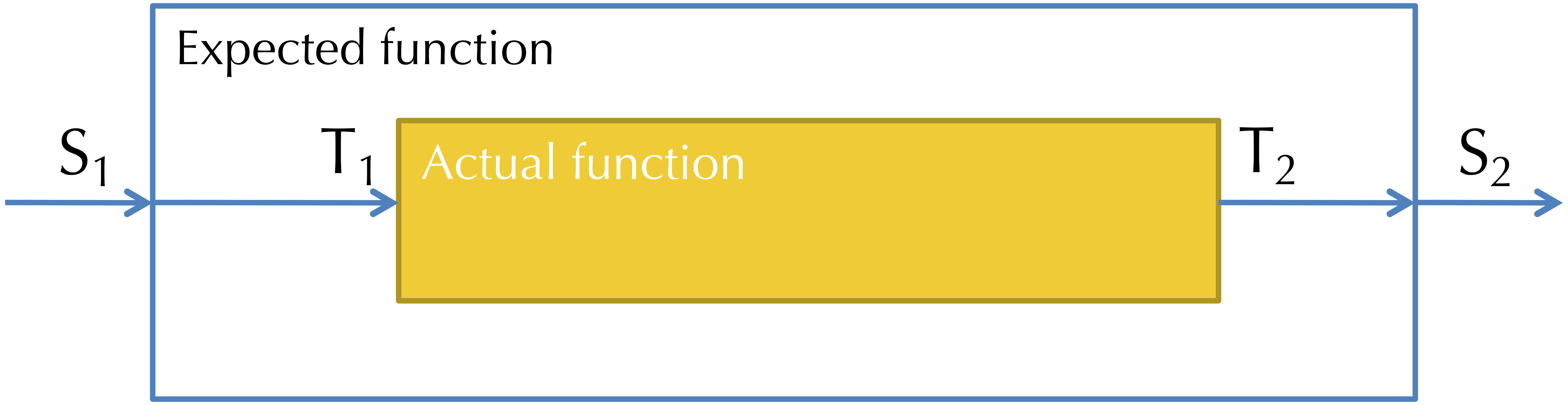
- Same holds for arrays, OCaml-style mutable records, object fields, etc.
  - Note: Java and C# get this wrong. They allows covariant array subtyping, but then compensate by adding a dynamic check on *every* array update!

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

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : \texttt{nat array}}\textsc{Var} \qquad \dfrac{\dfrac{}{\texttt{nat} <: \texttt{int}}\textsc{NatInt}}{\texttt{nat array} <: \texttt{int array}}\textsc{Array}}{\Gamma \vdash x : \texttt{int array}}\textsc{Sub} \qquad \dfrac{}{\Gamma \vdash 0 : \texttt{nat}}\textsc{Nat} \qquad \dfrac{}{\Gamma \vdash -1 : \texttt{int}}\textsc{Int}}{\Gamma \vdash x[0] := -1}\textsc{Assn}$$

Reminder: Subtyping for Function Types

- One way to see it:



- Need to convert an S1 to a T1 and T2 to S2, 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:

$$T \text{ ref} \;\simeq\; \{\text{get}: \text{unit} \to T;\quad \text{set}: T \to \text{unit}\}$$

  - get returns the value hidden in the state.
  - set updates the value hidden in the state.

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

Demo: Arrays in Java

- Check out https://github.com/cs4212/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?

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

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

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

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

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

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

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

$$\frac{\begin{array}{l} H; G; L_0; rt \vdash \ stmt_1 \ \Rightarrow \ L_1; \bot \\ \ldots \\ H; G; L_{n-2}; rt \vdash \ stmt_{n-1} \ \Rightarrow \ L_{n-1}; \bot \\ H; G; L_{n-1}; rt \vdash \ stmt_n \ \Rightarrow \ L_n; r \end{array}}{H ; G ; L_0 ; rt \vdash_{ss} stmt_1 \mathinner{..} stmt_{n-1}\, stmt_n \Rightarrow L_n ; r} \text{ TYP\_STMTS}$$

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

Example Derivation

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

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

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

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

$\mathcal{D}_1 =$

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

Example Derivation

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

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

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

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

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

$\mathcal{D}_2 =$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash +{:}(\texttt{int},\texttt{int})\ \texttt{->}\ \texttt{int}}\ \text{TYP\_INTOPS} \quad \dfrac{x_1{:}\texttt{int} \in \cdot, x_1{:}\texttt{int}}{H;G;\cdot,x_1{:}\texttt{int} \vdash x_1 : \texttt{int}}\ \text{TYP\_LOCAL} \quad \dfrac{x_1{:}\texttt{int} \in \cdot, x_1{:}\texttt{int}}{H;G;\cdot,x_1{:}\texttt{int} \vdash x_1 : \texttt{int}}\ \text{TYP\_LOCAL}}{H;G;\cdot,x_1{:}\texttt{int} \vdash x_1 + x_1 : \texttt{int}}\ \text{TYP\_BOP} \quad x_2 \notin \cdot, x_1{:}\texttt{int}}{H;G;\cdot \vdash \texttt{var}\ x_2 = x_1 + x_1 \Rightarrow \cdot, x_1{:}\texttt{int}, x_2{:}\texttt{int}}\ \text{TYP\_DECL}}{H;G;\cdot,x_1{:}\texttt{int};\texttt{int} \vdash \texttt{var}\ x_2 = x_1 + x_1; \Rightarrow \cdot, x_1{:}\texttt{int}, x_2{:}\texttt{int};\bot}\ \text{TYP\_STMTDECL}$$

Example Derivation

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

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

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

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

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

$\mathcal{D}_5 =$

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

$\mathcal{D}_3 =$

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

Example Derivation

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

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

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

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

$\mathcal{D}_4 =$

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

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*