Lecture 18

CS 131: COMPILERS

Announcements

- HW5: OAT v. 2.0
  - records, function pointers, type checking, array-bounds checks, etc.
  - Due: December 13th.
  - Available soon after the class.
  - **Start Early!**

oat.pdf

TYPECHECKING 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.
- Example1: Find a tree for the following program using the inference rules in oat.pdf:

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

Example2: There is no tree for this ill-scoped program:

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

Example Derivation

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

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \mathcal{D}_3 \quad \mathcal{D}_4}{G_0\,;\,\cdot\,;\mathtt{int} \vdash \mathtt{var}\ x_1\ \mathtt{=}\ 0;\ \mathtt{var}\ x_2\ \mathtt{=}\ x_1 + x_1\,;\,x_1\ \mathtt{=}\ x_1\ \mathtt{-}\ x_2\,;\,\mathtt{return}\ x_1\,; \Rightarrow \cdot, x_1 : \mathtt{int}, x_2 : \mathtt{int}}\,[\text{STMTS}]}{\vdash \mathtt{var}\ x_1\ \mathtt{=}\ 0;\ \mathtt{var}\ x_2\ \mathtt{=}\ x_1 + x_1\,;\,x_1\ \mathtt{=}\ x_1\ \mathtt{-}\ x_2\,;\,\mathtt{return}\ x_1\,;}\,[\text{PROG}]$$

Example Derivation

$$\mathcal{D}_1 \quad = \quad \dfrac{\dfrac{\dfrac{\dfrac{}{G_0\,;\cdot \vdash 0 : \mathtt{int}}\,[\text{INT}]}{G_0\,;\cdot \vdash 0 : \mathtt{int}}\,[\text{CONST}]}{G_0\,;\cdot \vdash \mathtt{var}\ x_1 = 0 \Rightarrow \cdot, x_1 : \mathtt{int}}\,[\text{DECL}]}{G_0\,;\cdot\,;\mathtt{int} \vdash \mathtt{var}\ x_1 = 0; \Rightarrow \cdot, x_1 : \mathtt{int}}\,[\text{SDECL}]$$

$$\mathcal{D}_2 \quad = \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash + : (\mathtt{int},\mathtt{int}) \to \mathtt{int}}\,[\text{ADD}] \quad \dfrac{x_1 : \mathtt{int} \in \cdot, x_1 : \mathtt{int}}{G_0\,;\cdot, x_1 : \mathtt{int} \vdash x_1 : \mathtt{int}}\,[\text{VAR}] \quad \dfrac{x_1 : \mathtt{int} \in \cdot, x_1 : \mathtt{int}}{G_0\,;\cdot, x_1 : \mathtt{int} \vdash x_1 : \mathtt{int}}\,[\text{VAR}]}{G_0\,;\cdot, x_1 : \mathtt{int} \vdash x_1 + x_1 : \mathtt{int}}\,[\text{BOP}]}{G_0\,;\cdot, x_1 : \mathtt{int}\,;\mathtt{int} \vdash \mathtt{var}\ x_2 = x_1 + x_1\,; \Rightarrow \cdot, x_1 : \mathtt{int}, x_2 : \mathtt{int}}\,[\text{DECL}]}{G_0\,;\cdot, x_1 : \mathtt{int}\,;\mathtt{int} \vdash \mathtt{var}\ x_2 = x_1 + x_1\,; \Rightarrow \cdot, x_1 : \mathtt{int}, x_2 : \mathtt{int}}\,[\text{SDECL}]$$

Example Derivation

$$\mathcal{D}_3 = \cfrac{\cfrac{}{\vdash \texttt{-} : (\texttt{int},\texttt{int}) \to \texttt{int}}[\textsc{add}] \quad \cfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 : \texttt{int}}[\textsc{var}] \quad \cfrac{x_2 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_2 : \texttt{int}}[\textsc{var}]}{\cfrac{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 \texttt{ - } x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} ; \texttt{int} \vdash x_1 \texttt{ = } x_1 \texttt{ - } x_2\texttt{;} \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}[\textsc{assn}]}[\textsc{bop}]$$

$$\mathcal{D}_4 = \cfrac{\cfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 : \texttt{int}}[\textsc{var}]}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} ; \texttt{int} \vdash \texttt{return } x_1\texttt{;} \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}[\textsc{Ret}]$$

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 behaviors:
  - abusing "unsafe" casts: converting pointers to integers, etc.
  - treating non-code values as code (and vice-versa)
  - breaking the type abstractions of the language
- What is "defined" depends on the language semantics…

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.

- 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 different judgment ($E \vdash e : t$) by searching backwards through the rules.

- Compiling in a context is nothing more than a collection of inference rules specifying yet a different judgment ($G \vdash src \Rightarrow target$)
  - Moreover, the compilation rules are very similar in structure to the typechecking rules

- Strong mathematical foundations
  - The "Curry-Howard correspondence": Programming Language ~ Logic, Program ~ Proof, Type ~ Proposition
  - See UPenn CIS 5000 if you're interested in type systems!

B

MORE TYPES

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

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

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 initializes 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

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

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

References

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

REF

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

DEREF

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

ASSIGN

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

Note the similarity with the rules for arrays…

COMPILING WITH TYPES

Compilation As Translating Judgments

- Consider the source typing judgment for source expressions:

$$C \vdash e : t$$

- How do we interpret this information in the target language?

$$[\![C \vdash e : t]\!] = \ ?$$

- $[\![C]\!]$ translates contexts
- $[\![t]\!]$ is a target type
- $[\![e]\!]$ translates to a (potentially empty) stream of instructions, that, when run, computes the result into some operand

- INVARIANT: if $[\![C \vdash e : t]\!]$ = ty, operand , stream
  then the type (at the target level) of the operand is ty=$[\![t]\!]$

Example

- $C \vdash 3410 + 5 : int$ what is $[\![C \vdash 3410 + 5 : int]\!]$?

$$\frac{\dfrac{[\![\vdash 3410 : int]\!] = (i64, Const\ 3410, [])}{[\![C \vdash 3410 : int]\!] = (i64, Const\ 3410, [])} \quad \dfrac{[\![\vdash 5 : int]\!] = (i64, Const\ 5, [])}{[\![C \vdash 5 : int]\!] = (i64, Const\ 5, [])}}{[\![C \vdash 3410 + 5 : int]\!] = (i64, \%tmp, [\%tmp = add\ i64\ (Const\ 3410)\ (Const\ 5)])}$$

What about the Context?

- What is ⟦C⟧?
- Source level C has bindings like: x:int, y:bool
  - We think of it as a finite map from identifiers to types
- What is the interpretation of C at the target level?
- ⟦C⟧ maps source identifiers, "x" to target types and ⟦x⟧
- What is the interpretation of a variable ⟦x⟧ at the target level?
  - How are the variables used in the type system?

$$\frac{x : t \in L}{G ; L \vdash x : t}\ \text{TYP\_VAR}$$

as expressions
(which denote values)

$$\frac{x : t \in L \quad G ; L \vdash exp : t}{G ; L ; rt \vdash x = exp ; \Rightarrow L}\ \text{TYP\_ASSN}$$

as addresses
(which can be assigned)

Interpretation of Contexts

- ⟦C⟧ = a map from source identifiers to types and target identifiers

- INVARIANT:
  x:t ∈ C   means that

  (1) lookup ⟦C⟧ x = (⟦t⟧*, %id_x)
  (2) the (target) type of %id_x is ⟦t⟧*   (a pointer to ⟦t⟧)

Interpretation of Variables

- Establish invariant for expressions:

$$\left[\!\left[\frac{x : t \in L}{G ; L \vdash x : t}\ \text{TYP\_VAR} \right]\!\right] = \text{(\%tmp, [\%tmp = load T \%id\_x])}$$

as expressions
(which denote values)

where (T, %id_x) = lookup ⟦L⟧ x
by invariant: T = ⟦t⟧*

- What about statements?

$$\left[\!\left[\frac{x : t \in L \quad G ; L \vdash exp : t}{G ; L ; rt \vdash x = exp ; \Rightarrow L}\ \text{TYP\_ASSN} \right]\!\right] = \text{stream @ [store T opn, T* \%id\_x]}$$

as addresses
(which can be assigned)

where (T*, %id_x) = lookup ⟦L⟧ x
and ⟦G;L ⊢ exp : t⟧ = (⟦t⟧, opn, stream)

Other Judgments?

- Statement:
  $[\![C;\ rt \vdash stmt \Rightarrow C']\!]$ = $[\![C']\!]$, stream

- Declaration:
  $[\![G;L \vdash t\ x = exp \Rightarrow G;L,x{:}t\ ]\!]$ = $[\![G;L,x{:}t]\!]$, stream

  INVARIANT: stream is of the form:
  stream' @
  [%id_x = alloca $[\![t]\!]$;
  store $[\![t]\!]$ opn, $[\![t]\!]$* %id_x]

  and $[\![G;L \vdash exp : t\ ]\!]$ = ($[\![t]\!]$, opn, stream')

- Rest follow similarly

COMPILING CONTROL

Translating while

- Consider translating "while(e) s":
  - Test the conditional, if true jump to the body, else jump to the label after the body.

$[\![C;rt \vdash \text{while(e) s} \Rightarrow C']\!] \;=\; [\![C']\!],$

```
lpre:
        opn = ⟦C ⊢ e : bool⟧
        %test = icmp eq i1 opn, 0
        br %test, label %lpost, label %lbody
lbody:
    ⟦C;rt ⊢ s ⇒ C'⟧
    br %lpre
lpost:
```

- Note: writing opn = $[\![C \vdash e : \text{bool}]\!]$ is pun
  - translating $[\![C \vdash e : \text{bool}]\!]$ generates *code* that puts the result into opn
  - In this notation there is implicit collection of the code

Translating if-then-else

- Similar to while except that code is slightly more complicated because if-then-else must reach a merge and the else branch is optional.

$[\![C;rt \vdash \text{if } (e_1)\ s_1 \text{ else } s_2 \Rightarrow C']\!] = [\![C']\!]$

```
        opn = ⟦C ⊢ e : bool⟧
        %test = icmp eq i1 opn, 0
        br %test, label %else, label %then
then:
    ⟦C;rt ⊢ s1 ⇒ C'⟧
    br %merge
else:
        ⟦C; rt s2 ⇒ C'⟧
    br %merge
merge:
```

Connecting this to Code

- Instruction streams:
  - Must include labels, terminators, and "hoisted" global constants
- Must post-process the stream into a control-flow-graph
- See frontend.ml from HW4

OPTIMIZING CONTROL

Standard Evaluation

- Consider compiling the following program fragment:

```
if (x & !y | !w)
  z = 3;
else
  z = 4;
return z;
```

```
        %tmp1 = icmp Eq ⟦y⟧, 0      ; !y
        %tmp2 = and ⟦x⟧ ⟦tmp1⟧
        %tmp3 = icmp Eq ⟦w⟧, 0
        %tmp4 = or %tmp2, %tmp3
        %tmp5 = icmp Eq %tmp4, 0
        br %tmp4, label %else, label %then

then:
        store ⟦z⟧, 3
        br %merge

else:
        store ⟦z⟧, 4
        br %merge

merge:
        %tmp5 = load ⟦z⟧
        ret %tmp5
```

Observation

- Usually, we want the translation $[\![e]\!]$ to produce a value
  - $[\![C \vdash e : t]\!]$ = (ty, operand, stream)
  - e.g. $[\![C \vdash e_1 + e_2 : \text{int}]\!]$ = (i64, %tmp, [%tmp = add i64 $[\![e_1]\!]$ $[\![e_2]\!]$])
- But when the expression we're compiling appears in a test, the program jumps to one label or another after the comparison but otherwise never uses the value.
- In many cases, we can avoid "materializing" the value (i.e., storing it in a temporary) and thus produce better code.
  - This idea also lets us implement different functionality too: e.g. short-circuiting Boolean expressions
- Make up new "judgement" that is similar to $[\![C \vdash e : \text{bool}]\!]$ but has a different semantics. Call it $[\![C \vdash e : \text{bool@}]\!]$

Idea: Use a different translation for tests

Usual Expression translation:

$[\![C \vdash e : t]\!] = (ty, operand, stream)$

Conditional branch translation of booleans, without materializing the value:

$[\![C \vdash e : bool@]\!]$ ltrue lfalse = stream

Notes:

- takes two extra arguments: a "true" branch label and a "false" branch label.
- Doesn't "return a value"

- Aside: this is a form of continuation-passing translation…

$[\![C, rt \vdash \text{if (e) then s1 else s2} \Rightarrow C']\!] = [\![C']\!],$

```
        insns3
then:
    [[s1]]
    br %merge
else:
        [[s2]]
    br %merge
merge:
```

where

$[\![C, rt \vdash s_1 \Rightarrow C']\!] = [\![C']\!], insns_1$

$[\![C, rt \vdash s_2 \Rightarrow C'']\!] = [\![C'']\!], insns_2$

$[\![C \vdash e : bool@]\!]$ then else = $insns_3$

Short Circuit Compilation: Expressions

- ⟦C ⊢ e : bool@⟧ ltrue lfalse = insns

$$\frac{}{\text{⟦C ⊢ false : bool@⟧ ltrue lfalse = [br \%lfalse]}}\ \text{FALSE}$$

$$\frac{}{\text{⟦C ⊢ true : bool@⟧ ltrue lfalse = [br \%ltrue]}}\ \text{TRUE}$$

$$\frac{\text{⟦C ⊢ e : bool@⟧ lfalse ltrue = insns}}{\text{⟦C ⊢ !e : bool@⟧ ltrue lfalse = insns}}\ \text{NOT}$$

Short Circuit Evaluation

Idea: build the logic into the translation

$$\frac{[\![C \vdash e1 : bool@]\!]\ ltrue\ right = insns_1 \quad [\![C \vdash e2 : bool@]\!]\ ltrue\ lfalse = insns_2}{[\![C \vdash e1|e2 : bool@]\!]\ ltrue\ lfalse = \begin{array}{l} \quad\quad insns_1 \\ right: \\ \quad insn_2 \end{array}}$$

$$\frac{[\![C \vdash e1 : bool@]\!]\ right\ lfalse = insns_1 \quad [\![C \vdash e2 : bool@]\!]\ ltrue\ lfalse = insns_2}{[\![C \vdash e1\&e2 : bool@]\!]\ ltrue\ lfalse = \begin{array}{l} \quad\quad insns_1 \\ right: \\ \quad insn_2 \end{array}}$$

where right is a fresh label

Short-Circuit Evaluation

- Consider compiling the following program fragment:

```
if (x & !y | !w)
  z = 3;
else
  z = 4;
return z;
```

```
        %tmp1 = icmp Eq ⟦x⟧, 0
        br %tmp1, label %right2, label %right1

right1:
        %tmp2 = icmp Eq ⟦y⟧, 0
        br %tmp2, label %then, label %right2

right2:
        %tmp3 = icmp Eq ⟦w⟧, 0
        br %tmp3, label %then, label %else

then:
        store ⟦z⟧, 3
        br %merge

else:
        store ⟦z⟧, 4
        br %merge

merge:
        %tmp5 = load ⟦z⟧
        ret %tmp5
```

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

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

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

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

IF-F

$$\frac{G \vdash e_1 : \text{False} \qquad E \vdash e_3 : T}{G \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : T}$$

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

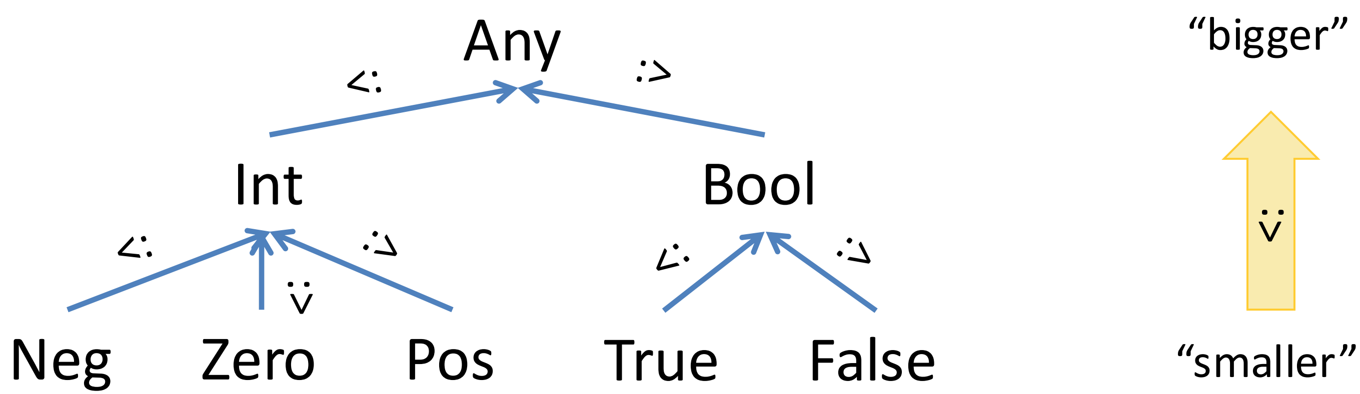
$$x\text{:bool} \vdash \text{if } (x)\ 3 \text{ else } -1 : ?$$

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.
  - Definition: LUB(T_1, T_2) is the smallest T such that T_1 <: T and T_2 <: T
  - 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:

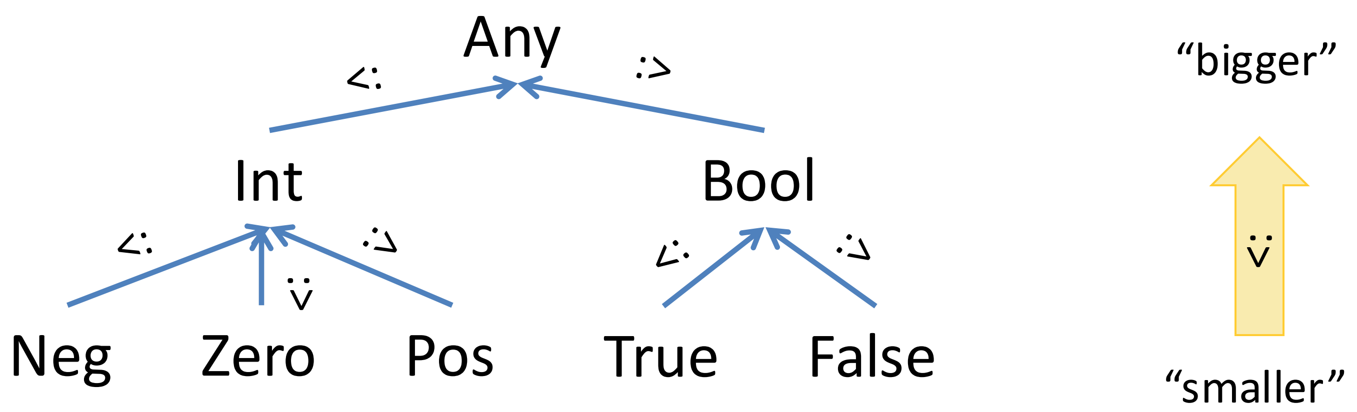
IF-BOOL

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

- Note: $\text{LUB}(T_1, T_2)$ is the most precise type (according to the hierarchy) that can 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: $T <: T$ 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:

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

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

- 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 these specific subtypes of Int are *sound* for +

ADD

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

Subsumption Rule

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

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

- 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"

$$\frac{G \vdash e_1 : \mathrm{Int} \qquad G, x : \mathrm{Pos} \vdash e_2 : T_2 \qquad G \vdash e_3 : T_3}{G \vdash \mathrm{ifPos}\ (x = e_1)\ e_2\ \mathrm{else}\ e_3 : T_2 \vee T_3}$$

- 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, and can be used in other contexts too:
  - We could give integer division the type: Int → NonZero → Int

SUBTYPING OTHER TYPES

Extending Subtyping to Other Types

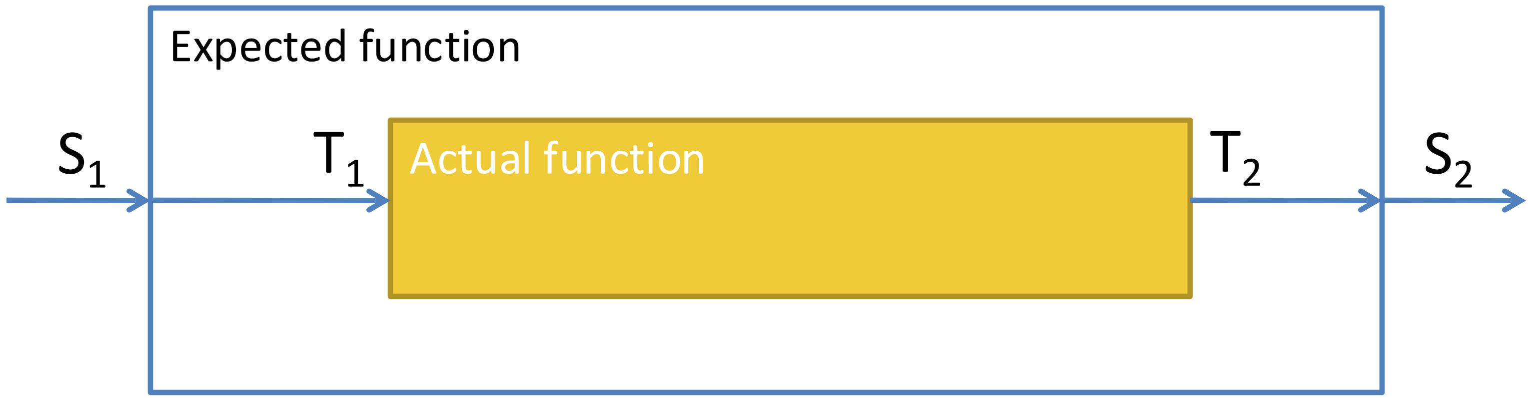
- What about subtyping for tuples?
  - Intuition: whenever a program expects something of type $S_1 * S_2$, it is sound to give it a $T_1 * T_2$.
  - Example: (Pos * Neg) <: (Int * Int)

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

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

RECORD

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

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

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

- Width subtyping:
  - Subtype record may have *more* fields on the right:

WIDTH

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

Depth & Width Subtyping vs. Layout

- Width subtyping (without depth) is compatible with "inlined" record representation as with C structs:

  {x:int; y:int; z:int}   <:   {x:int; y:int}   [Width Subtyping]

  – The layout and underlying field indices for 'x' and 'y' are identical.
  – The 'z' field is just ignored

- Depth subtyping (without width) is similarly compatible, assuming that the space used by A is the same as the space used by B whenever A <: B
- But… they don't mix without more work

Immutable Record Subtyping (cont'd)

- Width subtyping assumes an implementation in which order of fields in a record matters:

  $\{x{:}int;\ y{:}int\} \neq \{y{:}int;\ x{:}int\}$

- But: $\{x{:}int;\ y{:}int;\ z{:}int\} <: \{x{:}int;\ y{:}int\}$
  - Implementation: a record is a struct, subtypes just add fields at the *end* of the struct.

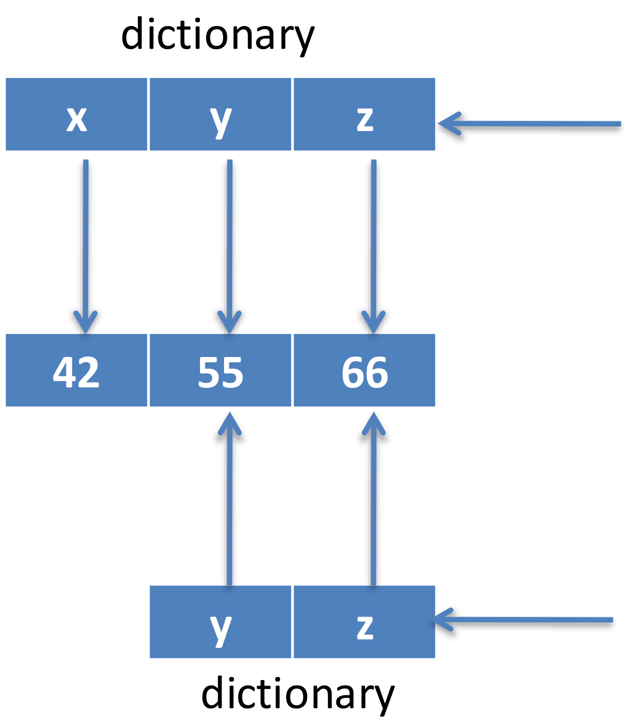
- Alternative: allow permutation of record fields:

  $\{x{:}int;\ y{:}int\} = \{y{:}int;\ x{:}int\}$
  - Implementation: compiler sorts the fields before code generation.
  - Need to know *all* of the fields to generate the code
- Permutation is not directly compatible with width subtyping:

  $\{x{:}int;\ z{:}int;\ y{:}int\} = \{x{:}int;\ y{:}int;\ z{:}int\} \not<: \{y{:}int;\ z{:}int\}$

If you want both:

- If you want permutability & dropping, you need to either copy (to rearrange the fields) or use a dictionary like this:



p = {x=42; y=55; z=66}:{x:int; y:int; z:int}

q : {y:int; z:int} = p

MUTABILITY & SUBTYPING

NULL

- What is the type of null?
- Consider:

```
        int[] a = null; // OK?
        int x   = null;   // not OK?
  string s = null;      // OK?
```

$\boxed{\text{NULL}}$ $\dfrac{}{G \vdash null : r}$

- 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
    (typically implemented using low-level hardware "trap" mechanisms.)