Lecture 17

# CIS 341: COMPILERS

# Announcements / Plan

- HW5: OAT – typechecking, structs, function pointers
  - Available soon
  - Due: Thursday, April 13
- HW6: LLVM Optimization: analysis and register allocation
  - Due: Wednesday, April 26
- FINAL EXAM: Thursday, May 4th noon – 2:00p.m.

# 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…

Beyond describing “structure”… describing “properties”

Types as sets

Subsumption

# TYPES, MORE GENERALLY

# 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…

# Arrays

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

NEW

$$\frac{E \vdash e_1 : \text{int}}{E \vdash \text{new } T[e_1] : T[]}$$

$e_1$ is the size of the newly allocated array.

INDEX

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

UPDATE

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

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

# NULL

- What is the type of `null`?
- Consider:

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

$$\boxed{\text{NULL}} \qquad \frac{}{E \vdash \text{null} : r}$$

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

# 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"

# 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:

$\boxed{\text{P-INT}}$

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

$\boxed{\text{N-INT}}$

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

$\boxed{\text{ZERO}}$

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

- Same for booleans:

$\boxed{\text{TRUE}}$

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

$\boxed{\text{FALSE}}$

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

# What about "if"?

- Two cases are easy:

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

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

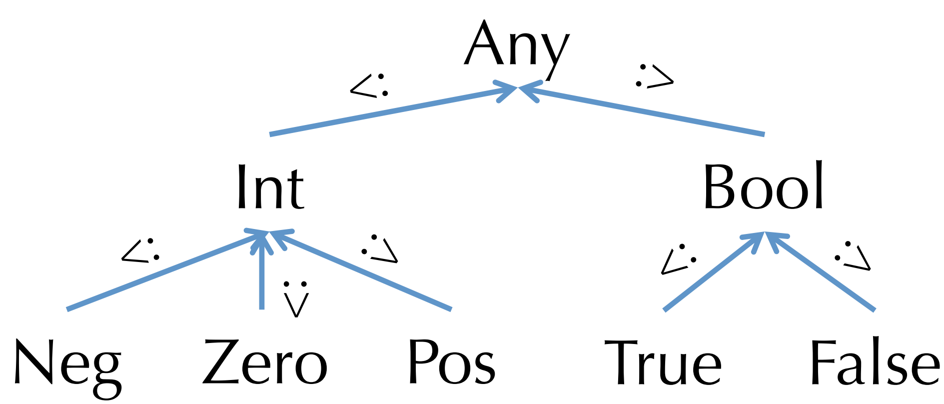
- What happens when we don't know statically which branch will be taken?
- Consider the typechecking problem:

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

- 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:

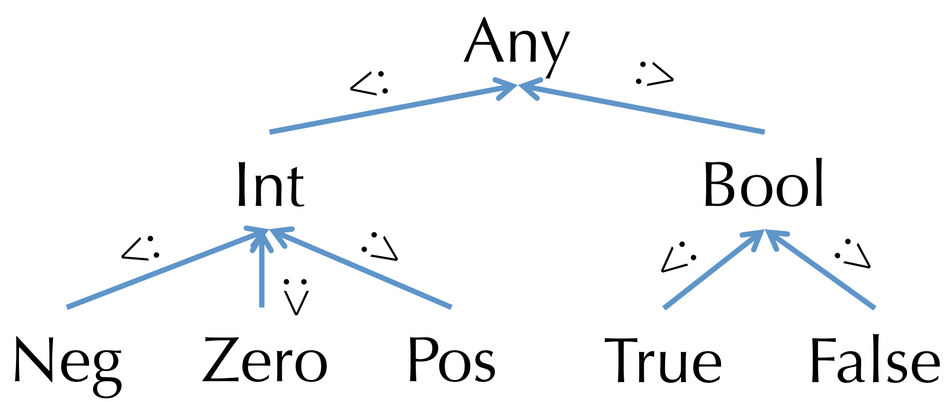
IF-BOOL

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

- Note that $\text{LUB}(T_1, T_2)$ 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, 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: $[\![\text{LUB}(\text{Zero, Pos})]\!] = [\![\text{Int}]\!] = \{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \supseteq$
    $\{0,1,2,3,\ldots\} = \{0\} \cup \{1,2,3,\ldots\} = [\![\text{Zero}]\!] \cup [\![\text{Pos}]\!]$

- 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

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

# Subsumption Rule

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

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

- 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.

# 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"

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

- 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
  - We could give integer division the type: Int -> NonZero -> Int