Lecture 19

EECS 483: COMPILER CONSTRUCTION

Announcements

- HW4: OAT v. 1.0
  - **Deadline Extended to Friday, 29th**
- HW5: OAT v. 2.0
  - Spec and partial starter code available now
  - Remaining starter code released on Monday, April 1
  - Due on Friday, April 12
- HW6: Analysis & Optimization
  - Released on Friday, April 12
  - Due on Friday, May 2 (after the final)
- Guest lectures
  - Professor New will not be lecturing for at least 2 weeks
    - Only remote office hours for Prof. New, no appointments
  - Next week's lectures: Eric
  - No class on April 8 (eclipse)

Beyond describing “structure”… describing “properties”

Types as sets

Subsumption

TYPES, MORE GENERALLY

What are types, anyway?

- A *type* is can often be thought of as 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:

$$\boxed{\text{P-INT}}\quad \frac{i > 0}{G \vdash i : \mathrm{Pos}} \qquad \boxed{\text{N-INT}}\quad \frac{i < 0}{G \vdash i : \mathrm{Neg}} \qquad \boxed{\text{ZERO}}\quad \frac{}{G \vdash 0 : \mathrm{Zero}}$$

- Same for booleans:

$$\boxed{\text{TRUE}}\quad \frac{}{G \vdash \mathrm{true} : \mathrm{True}} \qquad \boxed{\text{FALSE}}\quad \frac{}{G \vdash \mathrm{false} : \mathrm{False}}$$

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

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

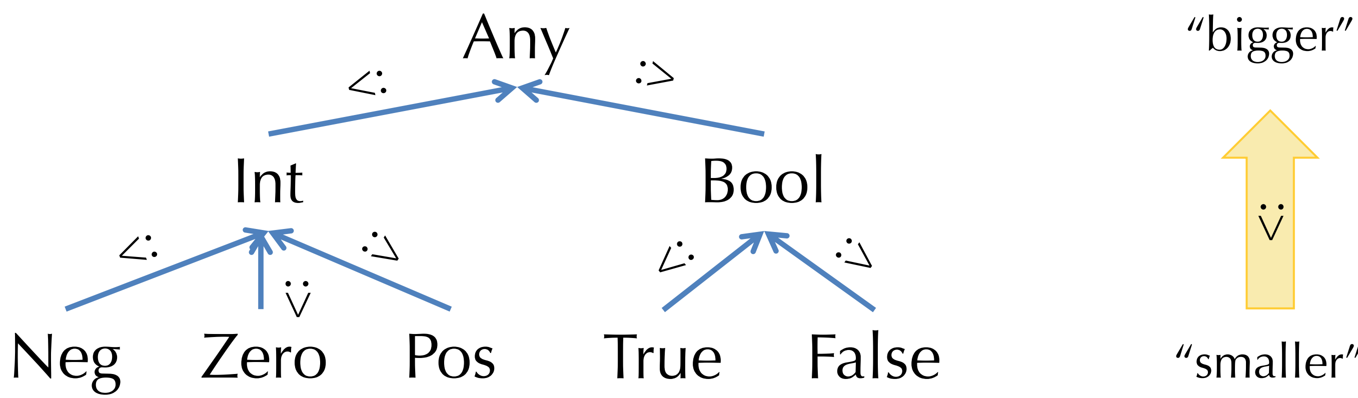
$$x{:}\text{bool} \vdash \text{if } (x)\ 3 \text{ else } {-1} : ?$$

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 $\subseteq$ 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:

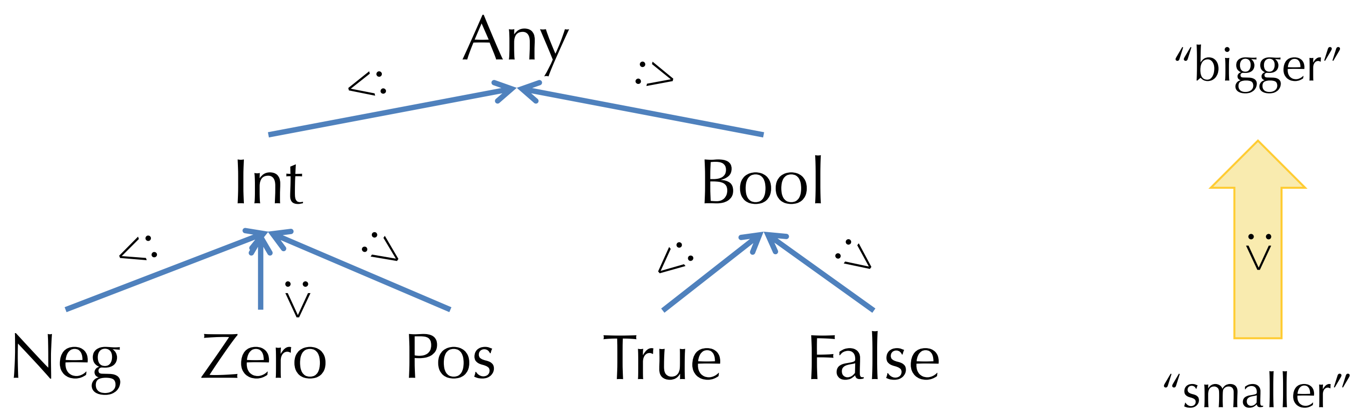
IF-BOOL

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

- 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.*, $[\![Zero]\!] = \{0\}$, $[\![Pos]\!] = \{1, 2, 3, \ldots\}$

- 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:
  $$[\![\mathrm{LUB}(T_1, T_2)]\!] \supseteq [\![T_1]\!] \cup [\![T_2]\!]$$
  - Note that the LUB is an over approximation of the "semantic union"
  - Example: $[\![\mathrm{LUB}(\mathrm{Zero}, \mathrm{Pos})]\!] = [\![\mathrm{Int}]\!] = \{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \supseteq$
    $\{0,1,2,3,\ldots\} = \{0\} \cup \{1,2,3,\ldots\} = [\![\mathrm{Zero}]\!] \cup [\![\mathrm{Pos}]\!]$

- 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 +

$$\boxed{\text{ADD}}\quad \frac{G \vdash e_1 : T_1 \qquad G \vdash e_2 : T_2 \qquad T_1 <: \mathrm{Int} \quad T_2 <: \mathrm{Int}}{G \vdash e_1 + e_2 : T_1 \vee T_2}$$

Subsumption Rule

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

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

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"

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

- 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)

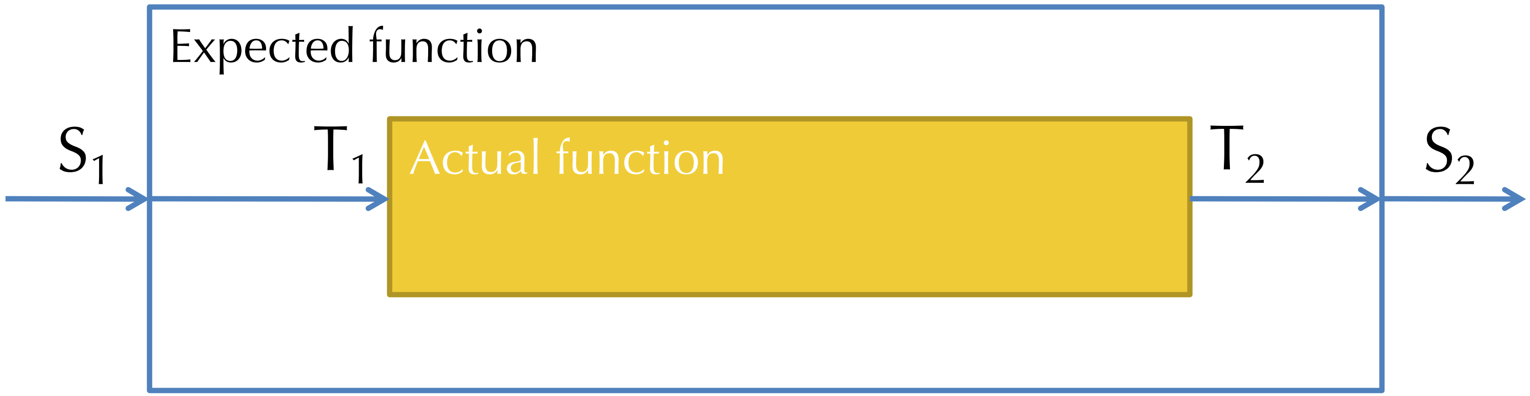
$$\frac{T_1 <: S_1 \quad T_2 <: S_2}{(T_1 * T_2) <: (S_1 * S_2)}$$

Subtyping for Function Types

- In class exercise: which of the following subtyping relationships should hold?
  - Int -> Pos <: Int -> Int
  - Int -> Int <: Int -> Pos
  - Pos -> Int <: Int -> Int
  - Int -> Int <: Pos -> Int
- What is the general principle? When is $T_1 \to T_2 \ <: \ S_1 \to S_2$?

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***.

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

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

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

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

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

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:

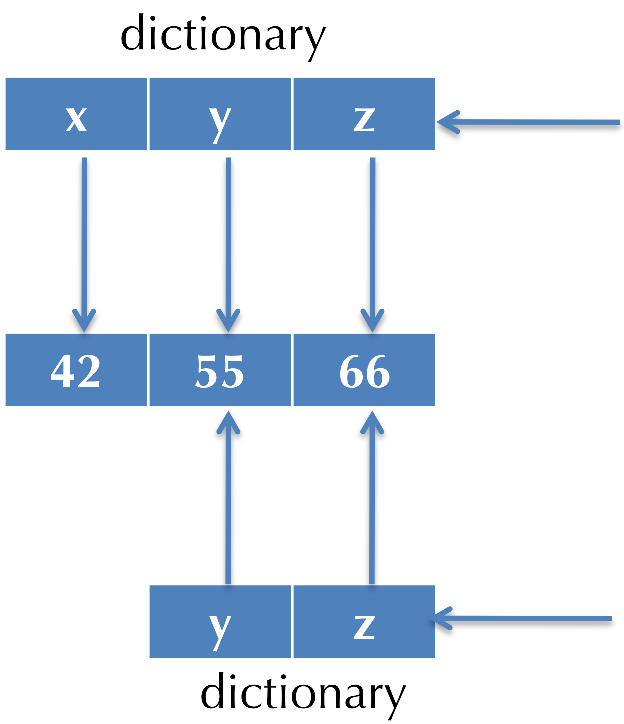
  {x:int; y:int} = {y:int; x:int}

  - 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} </: {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?
```

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

- 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.)

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
  - 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 ref $<: T_2$ ref implies $T_1 = T_2$

- 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!

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:

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

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

- When is T ref <: S ref?
- Records with depth subtyping:
  - extends pointwise over each component.

$$\{\text{get: unit} \to T; \text{ set: } T \to \text{unit}\} <: \{\text{get: unit} \to S; \text{ set: } S \to \text{unit}\}$$

  - 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.

See oat.pdf in HW5

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;
}
```

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: R null always has type R?
- Definitely-not-null values means we need an "atomic" array initialization syntax
  - 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

- Typesafe, 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?

Example OAT code

```
struct Base {          /* struct type with function field */
   int a;
   bool b;
   (int) -> int f
}

struct Extend {        /* structural subtype of Base via width subtyping */
   int a;
   bool b;
   (int) -> int f;
   string c;           /* added field and method */
   (int) -> int g
}

int neg(int x) { return -x; }
int inc(int x) { return x+1; }

int f(Base? x, int y){   /* function that expects a (possibly null) Base */
   if?(Base b = x){
      return b.f(y);
   } else {
      return -1;
   }
}

int program(int argc, string[] argv) {
   var s = new Extend[5]{x -> new Extend{a=3; b=true; c="hello"; f=neg; g=inc}};
   return f(s[2], -3);
}
```