

# CIS 341: Compilers

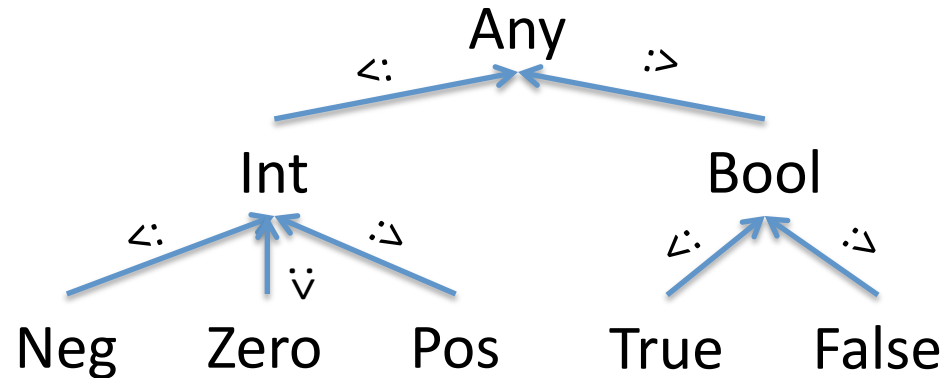
## Lecture 19

# The Plan

- Today:
  - Subtyping
- Project 4: Code generation & typechecking for the “AT” (arrays and types) language
  - Due: October 24<sup>th</sup> at 11:59 pm
- Midterm is Monday, Oct. 20<sup>th</sup> in class.
  - Prof. Zdancewic will be out of town
  - Topics: all of the course up to & including today’s lecture
  - Especially: Parsing, Context-Free Grammars, Ambiguity, x86 C-style procedure calling conventions, closure conversion, type checking

# 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: If  $T_1 <: T_2$  and  $T_2 <: T_1$  then  $T_1 = T_2$

# Subsumption Rule

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

SUBSUMPTION

$$E \vdash e : T \quad T <: S$$

---

$$E \vdash e : S$$

- 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.

# 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 value of type  $T_1 * T_2$ .
  - Example:  $(\text{Pos} * \text{Neg}) <: (\text{Int} * \text{Int})$
  - So,  $(3, -1)$  can be treated as  $(\text{Int} * \text{Int})$

$$T_1 <: S_1 \quad T_2 <: S_2$$

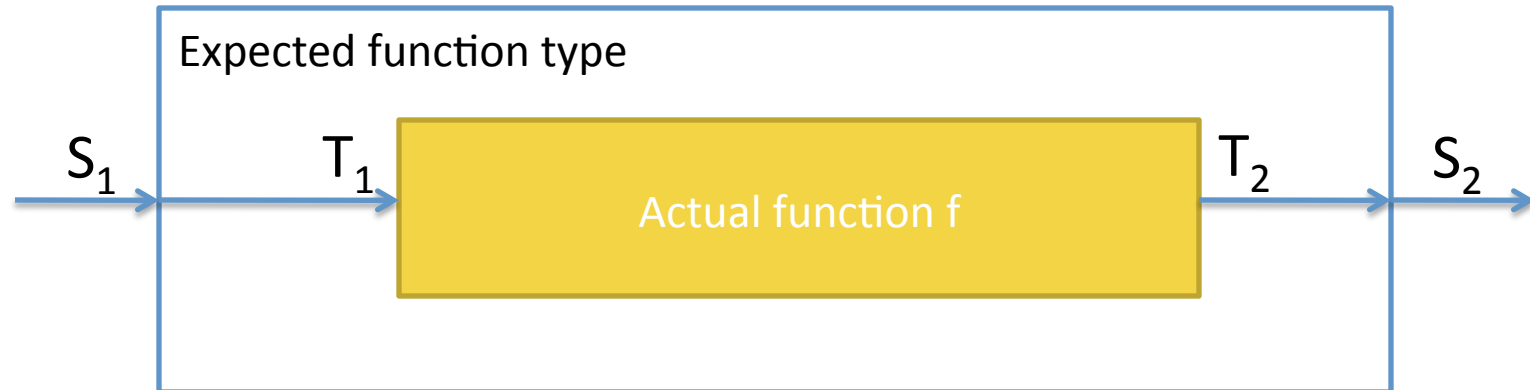
---

$$(T_1 * T_2) <: (S_1 * S_2)$$

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

# Subtyping for Function Types

- One way to see it: consider a function  $f : T_1 \rightarrow T_2$



- 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

$E \vdash e_1 : T_1 \quad E \vdash e_2 : T_2 \quad \dots \quad E \vdash e_n : T_n$

---

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

PROJECTION

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

---

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

# Immutable Record Subtyping

- Depth subtyping:
  - Corresponding fields may be subtypes

DEPTH

$$T_1 <: U_1 \quad T_2 <: U_2 \quad \dots \quad T_n <: U_n$$

---

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

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

WIDTH

$$m \leq n$$

---

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

## Immutable Record Subtyping (cont'd)

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

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

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

- Alternative: allow permutation of record fields:

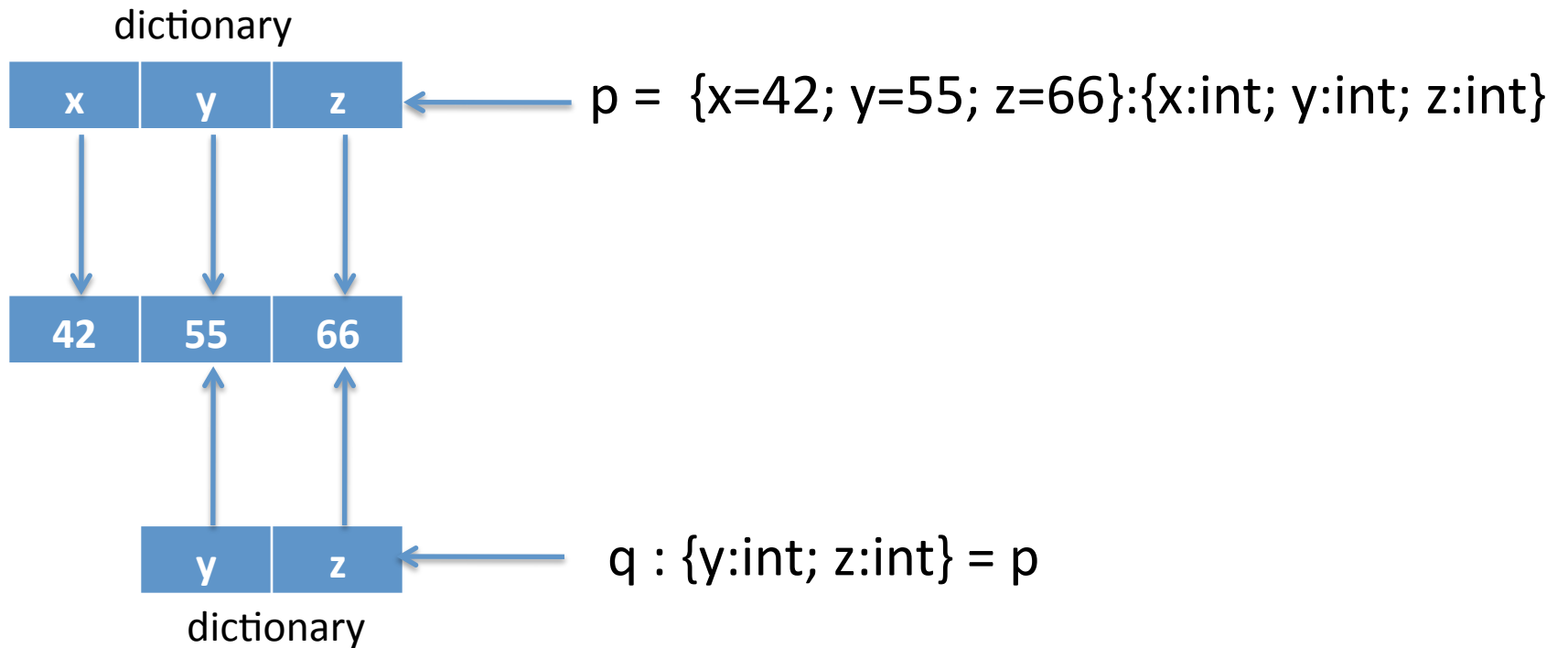
$$\{x:\text{int}; y:\text{int}\} = \{y:\text{int}; x:\text{int}\}$$

- Implementation: compiler sorts the fields before code generation.
  - Need to know *all* of the fields to generate the code
- Permutation not directly compatible with width subtyping:

$$\{x:\text{int}; z:\text{int}; y:\text{int}\} = \{x:\text{int}; y:\text{int}; z:\text{int}\} </: \{y:\text{int}; z:\text{int}\}$$

## If you want both:

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



# 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 *)  
  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  $\text{ref } T_2 <: \text{ref } T_1$  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 gets this wrong. It allows covariant array subtyping, but then it compensates 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:

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

- get returns the value hidden in the state.
  - set updates the value hidden in the state.
- When is  $T \text{ ref} <: S \text{ ref}$ ?
- Records are like tuples: subtyping extends pointwise over each component.
- $\{\text{get: unit} \rightarrow T; \text{set: } T \rightarrow \text{unit}\} <: \{\text{get: unit} \rightarrow S; \text{set: } S \rightarrow \text{unit}\}$ 
  - get components are subtypes:  $\text{unit} \rightarrow T <: \text{unit} \rightarrow S$
  - set components are subtypes:  $T \rightarrow \text{unit} <: S \rightarrow \text{unit}$
- 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  $T = S$ .

# Structural vs. Nominal

- Some languages define subtyping (or type equality) *implicitly*, based on the structure of the data.
  - Terminology: *structural equality* or *structural subtyping*
  - Example:  $\{x:\text{int}; y:\text{int}; z:\text{int}\} <: \{x:\text{int}; y:\text{int}\}$  via structural subtyping
  - Languages like Modula 3 have structural equality and subtyping
  - Sometimes called “duck” equality (“if it looks like a duck and quacks like a duck...”)
  - Advantage: relationships among types can be *inferred* from the structure
- Some languages define subtyping (or type equality) *explicitly*, based on names for the types.
  - Terminology: *nominal equality* or *nominal subtyping*
  - OCaml and Java have nominal type equality
  - Note: Of course declared subtypes are still subject to structural constraints
  - Advantage: Types can make finer distinctions. Example?

# Nominal vs. Structural Equality

- OCaml:

```
module M = struct
  type t1 = {x:int; y:int}
end
module N = struct
  type t2 = {x:int; y:int}
end
```

```
let a:M.t1 =
  {N.x = 3; N.y = 4}
```

Type Error!

- rhs has type N.t2, but is expected to have type M.t1

- Modula 3:

```
TYPE t1 = OBJECT
  x, y : INTEGER
END
TYPE t2 = OBJECT
  x, y : INTEGER
END
```

```
a:t1 = NEW t2
```

Success!

# Explicitly (Declared) vs. Implicit Subtyping

- Java: explicit

```
class C1 {  
    int x, y;  
}  
class C2 extends C1 {  
    int z;  
}
```

```
C1 a = new C2();
```

- Modula 3: implicit

```
TYPE t1 = OBJECT  
    x, y : INTEGER  
END  
TYPE t2 = OBJECT  
    x, y, z : INTEGER  
END
```

```
a:t1 = NEW t2
```