Lecture 16

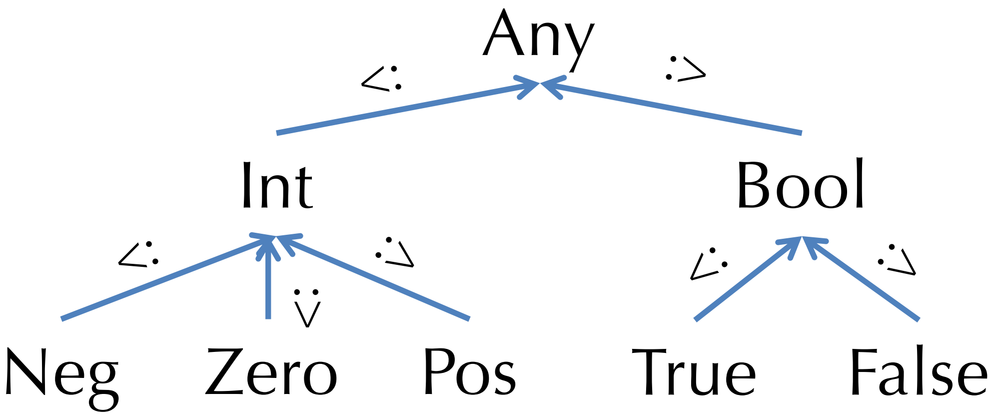
COMPILER DESIGN

Announcements

- **HW4**: OAT v. 1.0
  - Parsing & basic code generation

Subtyping and Upper Bounds

- If we view types as sets of values, there is 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***



Any

<: :>

Int Bool

<: <: :> <: :>

Neg Zero Pos True False

- For types T_1,T_2, define their ***least upper bound*** (LUB) wrt the hierarchy
  - Example: LUB(True, False) = Bool, LUB(Int, Bool) = Any
  - Note: may want to add types for "NonZero", "NonNegative", "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

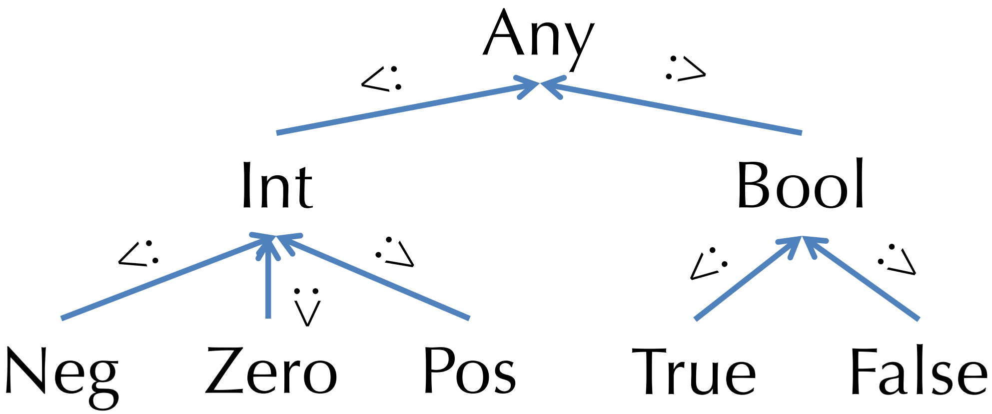
IF-BOOL

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

- Note LUB(T_1, T_2) is the most precise type (wrt 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 | ⊢ v : T}
  - e.g. ⟦Zero⟧ = {0}, ⟦Pos⟧ = {1, 2, 3, …}

- If T_1 <: T_2 implies $[\![T_1]\!] \subseteq [\![T_2]\!]$, then T_1 <: T_2 is sound
  - Ex 1: Pos <: Int is sound, since {1,2,3,…} ⊆ {…,-3,-2,-1,0,1,2,3,...}
  - Ex 2: Int <: Pos is not sound, since it is ***not*** the case that {…,-3,-2,-1,0,1,2,3,...}⊆ {1,2,3,…}

Soundness of LUBs

- Whenever we have a sound subtyping relation, it follows that
  $$\llbracket \mathrm{LUB}(T_1, T_2) \rrbracket \supseteq \llbracket T_1 \rrbracket \cup \llbracket T_2 \rrbracket$$
  - Note that the LUB is an over approximation of the "semantic union"
  - Example: $\llbracket \mathrm{LUB}(\mathrm{Zero}, \mathrm{Pos}) \rrbracket = \llbracket \mathrm{Int} \rrbracket = \{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \supseteq \{0,1,2,3,\ldots\} = \{0\} \cup \{1,2,3,\ldots\} = \llbracket \mathrm{Zero} \rrbracket \cup \llbracket \mathrm{Pos} \rrbracket$

- 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

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

Subsumption Rule

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

SUBSUMPTION

$$\frac{E \vdash e : T \qquad T <: S}{E \vdash e : S}$$

- 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
  - See for example 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"

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

- 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 such rules force the programmer to add the appropriate checks
  - We could give integer division the type: Int -> NonZero -> Int

SUBTYPING OTHER TYPES

Extending Subtyping to Other Types

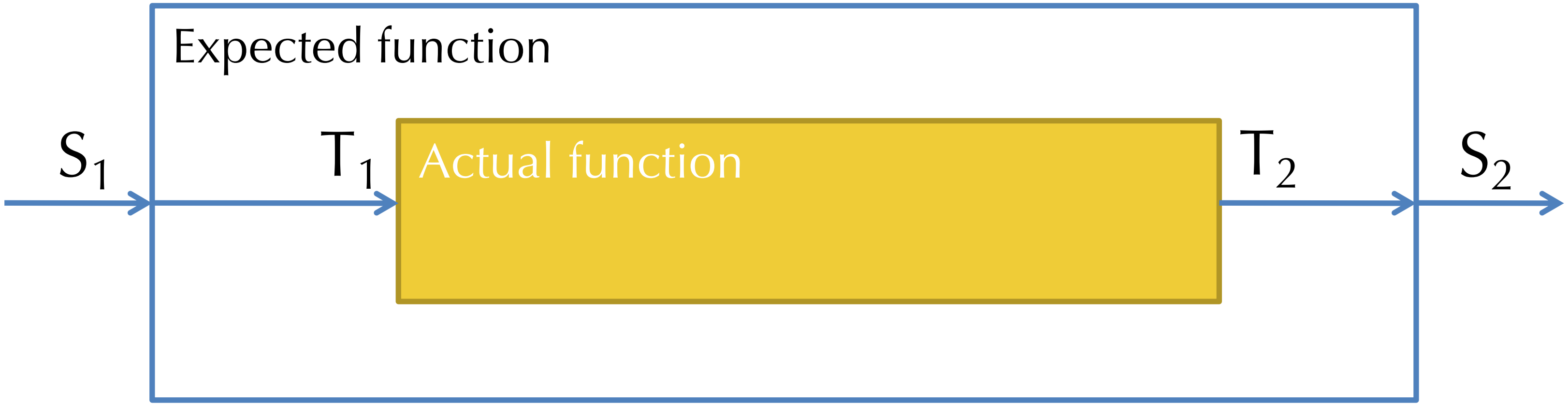
- What about subtyping for tuples?
  - Intuition: whenever a program expects something of type $S_1 * S_2$, it is sound to give it a $T_1 * T_2$
  - Example: (Pos * Neg) <: (Int * Int)

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

- What about functions?
- When is $T_1 \texttt{->} T_2 <: S_1 \texttt{->} S_2$?

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*

$$\frac{S_1 <: T_1 \quad T_2 <: S_2}{(T_1 \text{ -> } T_2) <: (S_1 \text{ -> } S_2)}$$

Immutable Records

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

RECORD

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

PROJECTION

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

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

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

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 well

Immutable Record Subtyping (cont'd)

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

  {x:int; y:int} ≠ {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

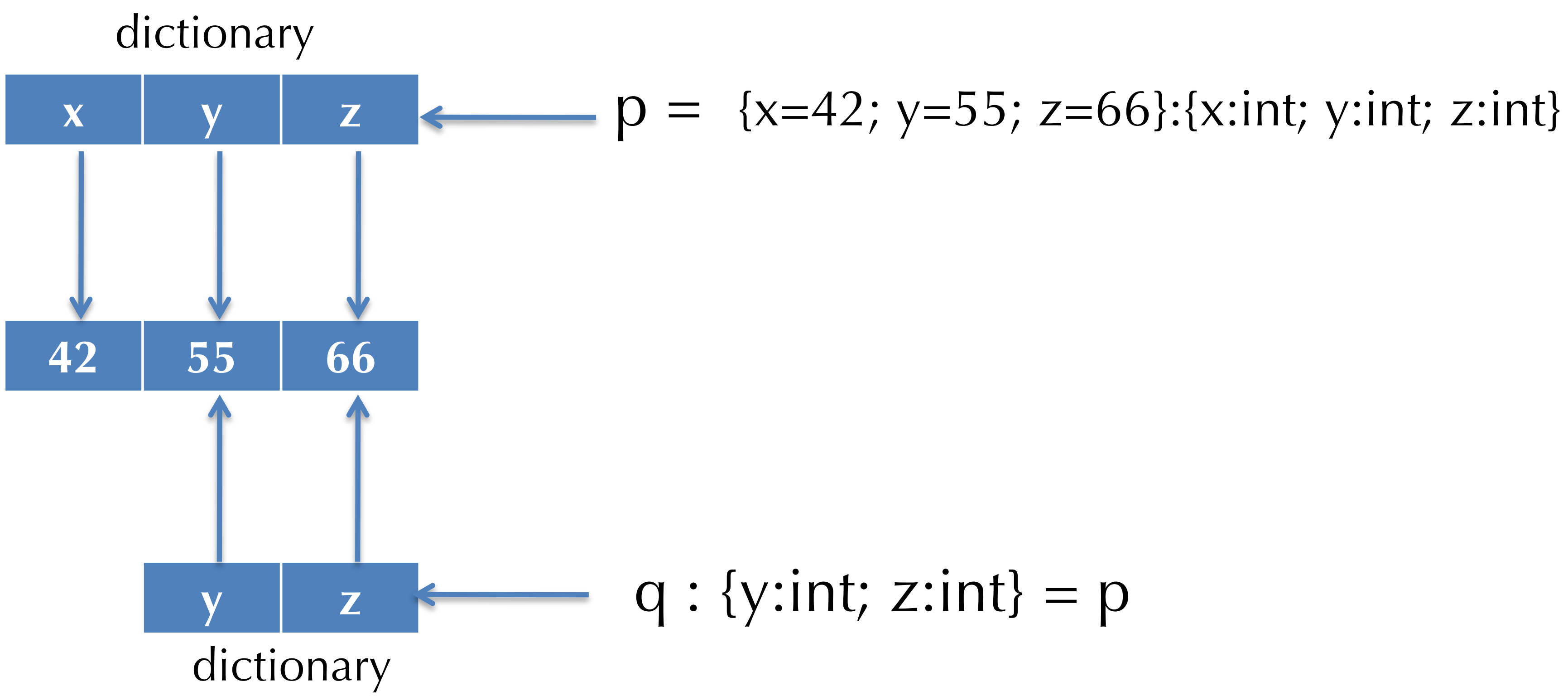
  {x:int; y:int} = {y:int; x:int}

  - 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 we want both …

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



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?
```

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

- 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)

Subtyping and References

- What is the proper subtyping relationship for references and arrays?

- Suppose we have NonZero as a type, 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

Assume: NonZero <: Int => ref Int <: ref NonZero

```
Int ref a;
a := 0;
NonZero ref b;
b = a;
return (1 / b);
```

Mutable Structures are Invariant

- 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!

Assume: B <: A => B[] <: A[]

```
B[] b = new B[5];
A[] a = b;
a[0] = new A;
b[0].someting in b
```

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 ref ≃ {get: unit -> T; set: T -> unit}

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

- When is **T ref <: S ref** ?
- Records, like tuples, subtyping extends pointwise over each component
- {get: unit -> T; set: T -> unit} <: {get: unit -> S; set: S -> unit}
  - get components are subtypes: unit -> T <: unit -> S
    set components are subtypes: T -> unit <: S -> 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 TYPES

Structural vs. Nominal Typing

- Is type equality/subsumption defined by the *structure* or *name* of the data?
- Example 1: 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
  - 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 341;}
}
```

- Similarly for inheritance
  - Programmers must declare the subclass relation via the "`extends`" keyword
  - Type-checker still checks the classes are structurally compatible

Full details later 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 *width* subtyping
  - strings
  - 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 == or !=
  - need special case handling in the type-checker
- Type-annotated null values: t null always has type t?
- Definitely-not-null values => "atomic" array initialization syntax
- As an example
  - null is not allowed as a value of type int[]
  - So to construct a record containing a field of type int[], need to initialize it

OAT "Returns" Analysis

- Typesafe, statement-oriented imperative languages like OAT (or Java) must ensure a function (always) returns a value of the appropriate type
  - Does the returned expr'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?

COMPILING CLASSES AND OBJECTS

Code Generation for Objects

- Classes
  - Generate data structure types
    - For objects that are instances of the class and for the class tables
  - Generate the class tables for dynamic dispatch
- Methods
  - Method body code is similar to functions/closures
  - Method calls require *dispatch*
- Fields
  - Issues are the same as for records
  - Generating access code
- Constructors
  - Object initialization
- Dynamic Types
  - Checked downcasts
  - “instanceof” and similar type dispatch

Multiple Implementations

- The same interface can be implemented by multiple classes

```
interface IntSet {
    public IntSet insert(int i);
    public boolean has(int i);
    public int size();
}
```

```
class IntSet1 implements IntSet {
  private List<Integer> rep;
  public IntSet1() {
    rep = new LinkedList<Integer>();}

  public IntSet1 insert(int i) {
    rep.add(new Integer(i));
    return this;}

  public boolean has(int i) {
    return rep.contains(new Integer(i));}

  public int size() {return rep.size();}
}
```

```
class IntSet2 implements IntSet {
  private Tree rep;
  private int size;
  public IntSet2() {
    rep = new Leaf(); size = 0;}

  public IntSet2 insert(int i) {
    Tree nrep = rep.insert(i);
    if (nrep != rep) {
      rep = nrep; size += 1;
    }
    return this;}

  public boolean has(int i) {
    return rep.find(i);}

  public int size() {return size;}
}
```

The Dispatch Problem

- Consider a client program that uses the IntSet interface

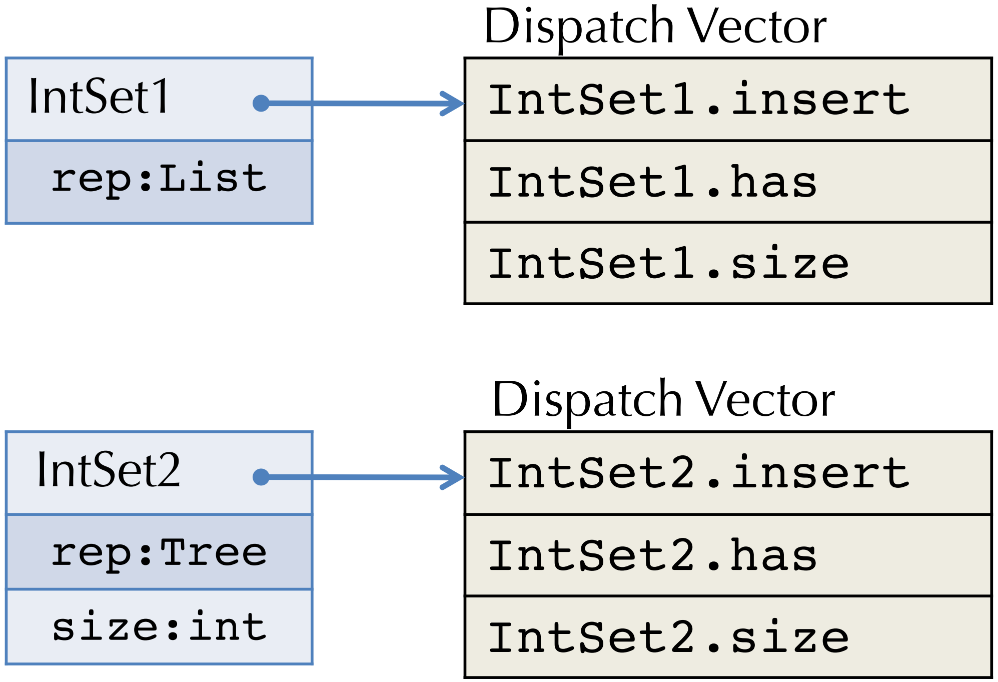
```
IntSet set = …;
int x = set.size();
```

- Which code to call?
  - `IntSet1.size`?
  - `IntSet2.size`?

- Client code doesn’t know the answer
  - So objects must “know” which code to call
  - Invocation of a method must indirect through the object

Compiling Objects

- Objects contain a pointer to a *dispatch vector* (also called a *virtual table* or *vtable*) with pointers to method code
- Code receiving `set:IntSet` only knows that `set` has an initial dispatch vector pointer and the layout of that vector



IntSet1

`rep:List`

Dispatch Vector

`IntSet1.insert`

`IntSet1.has`

`IntSet1.size`

IntSet2

`rep:Tree`

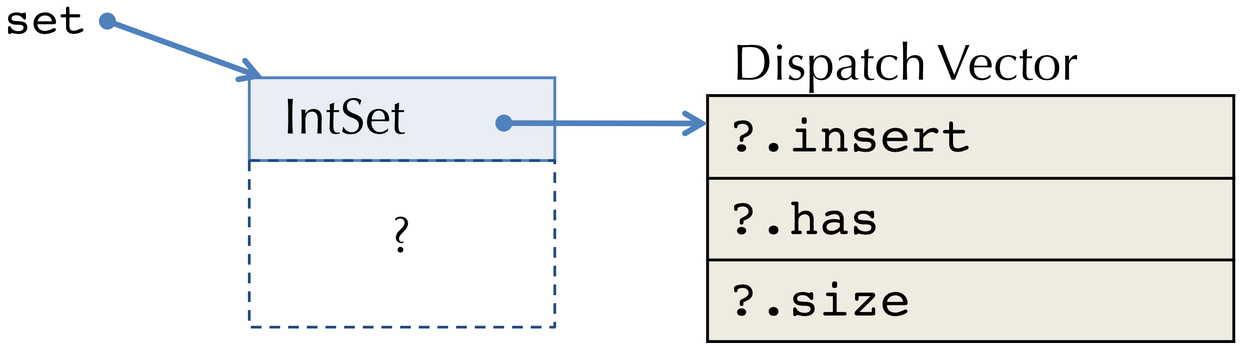
`size:int`

Dispatch Vector

`IntSet2.insert`

`IntSet2.has`

`IntSet2.size`



`set`

IntSet

?

Dispatch Vector

`?.insert`

`?.has`

`?.size`

Method Dispatch (Single Inheritance)

- Idea: every method has its own small integer index
- Index is used to look up the method in the dispatch vector

Index

```
interface A {
  void foo();
}
```

0

```
interface B extends A {
  void bar(int x);
  void baz();
}
```

1
2

Inheritance / Subtyping

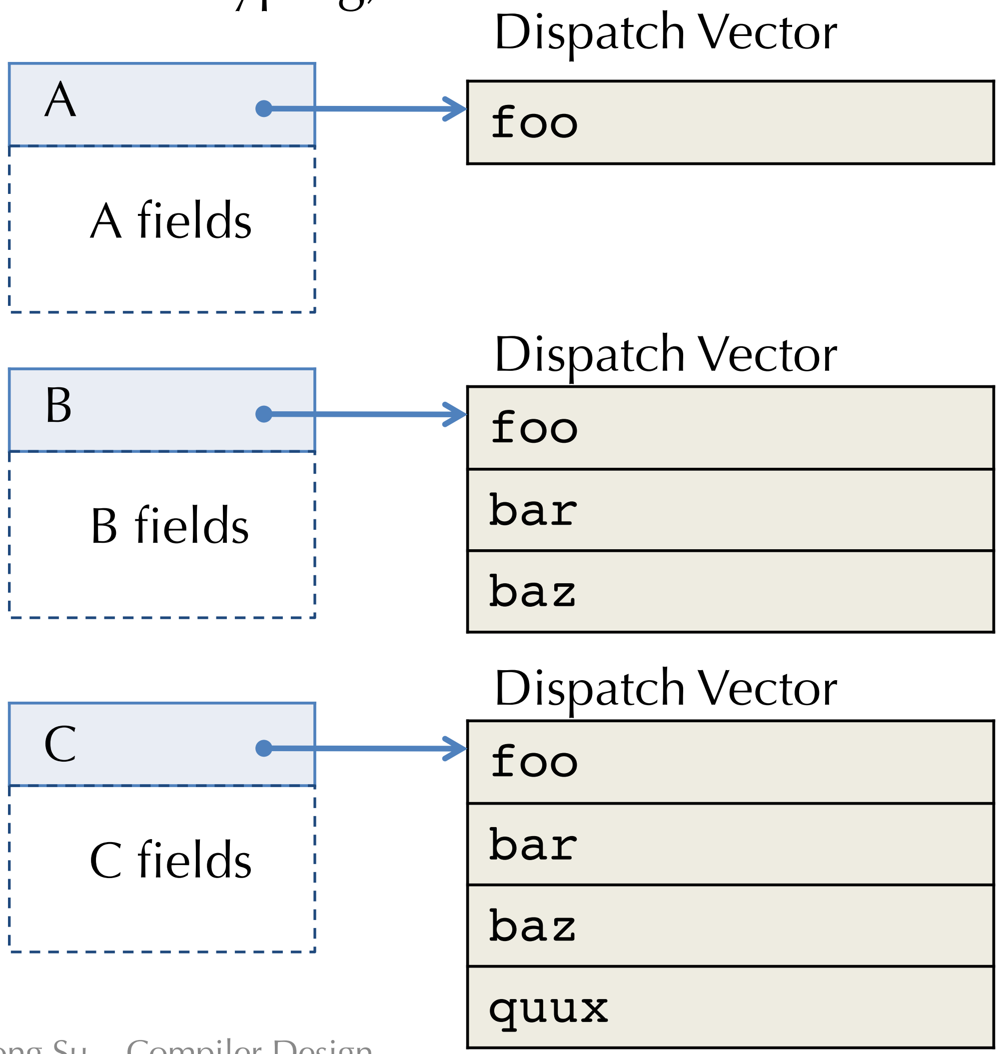
C <: B <: A

```
class C implements B {
  void foo() {…}
  void bar(int x) {…}
  void baz() {…}
  void quux() {…}
}
```

0
1
2
3

Dispatch Vector Layouts

- Each interface and class gives rise to a dispatch vector layout
- Note that inherited methods have identical dispatch indices in the subclass (Width subtyping)



Dispatch Vector

A

A fields

`foo`

Dispatch Vector

B

B fields

`foo`
`bar`
`baz`

Dispatch Vector

C

C fields

`foo`
`bar`
`baz`
`quux`