Lecture 19

CS131: COMPILERS

Announcements

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

- Teaching evaluation
  - By January 5th
  - Submit evaluation to reveal final scores.

MUTABILITY & SUBTYPING

NULL

- What is the type of null?
- Consider:

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

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

- 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.)

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:

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

  - 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);
}
```

STRUCTURAL VS. NOMINAL TYPES

Structural vs. Nominal Typing

- Is type equality / subsumption defined by the *structure* of the data or the *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!
```

- OCaml type abbreviations are treated “**structurally**”
  Haskell newtypes are treated “**by name**”

Nominal Subtyping in Java

- Example 2: In Java, Classes and Interfaces must be named and 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 3410;}
}
```

- Similarly for inheritance: programmers must declare the subclass relation via the “extends” keyword.
  – Typechecker still checks that the classes are structurally compatible

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

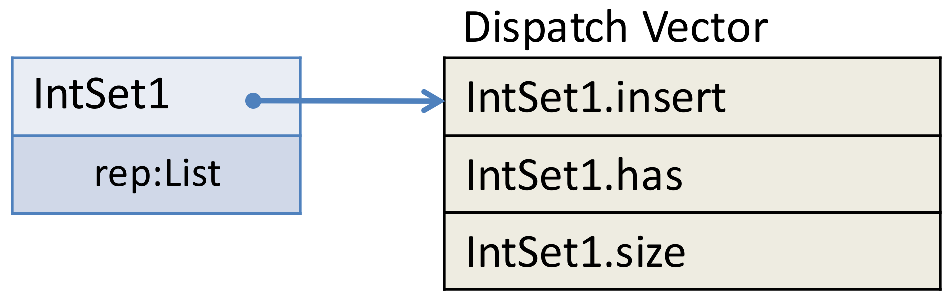
- Consider a client program that uses the IntSet interface:

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

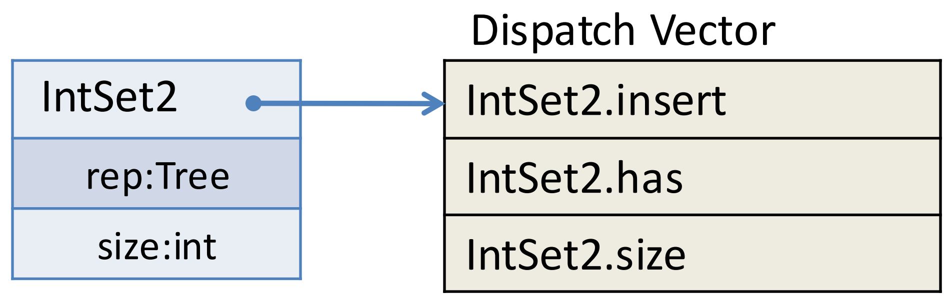
- Which code to call?
  - IntSet1.size ?
  - IntSet2.size ?
- Client code doesn’t know the answer.
  - So objects must “know” which code to call.
  - Invocation of a method must indirect through the object.

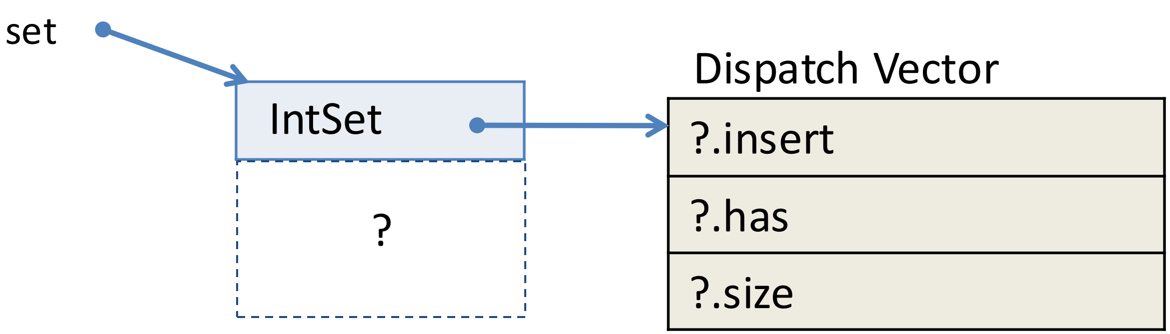
Compiling Objects

- 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.





Method Dispatch (Single Inheritance)

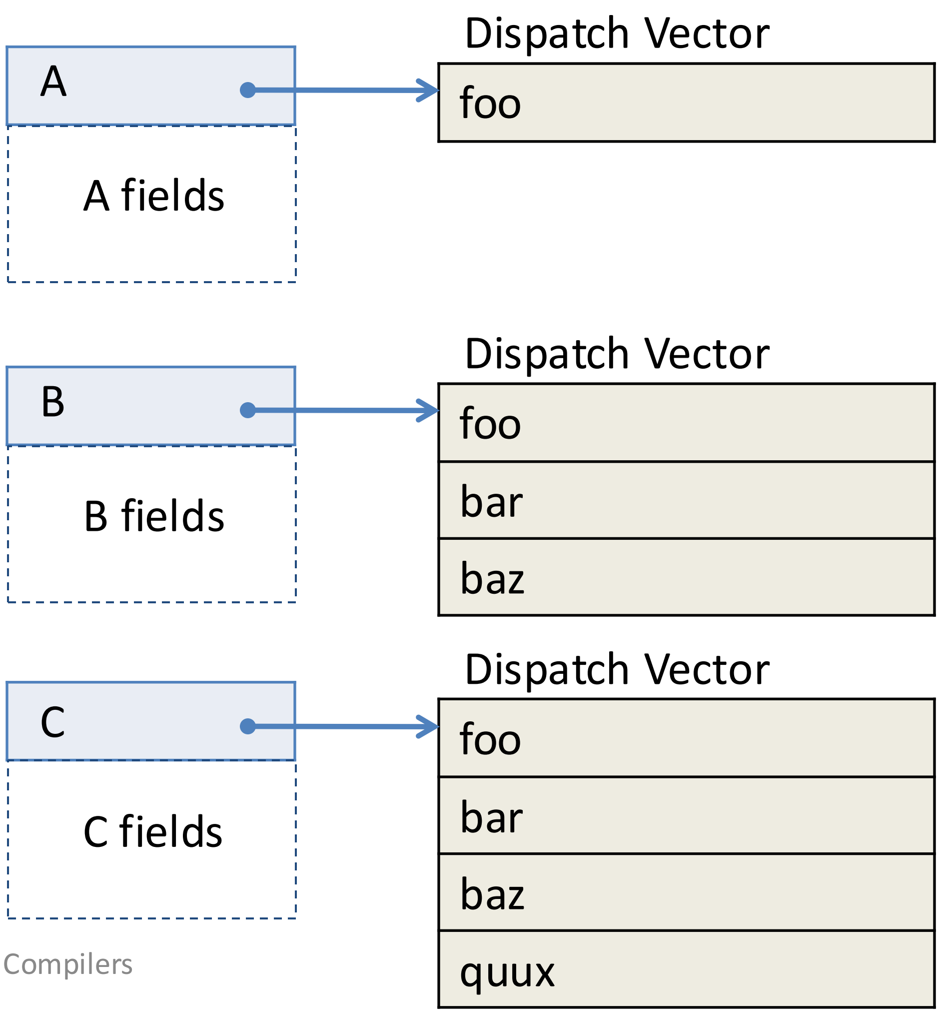
- Idea: every method has its own small integer index.
- Index is used to look up the method in the dispatch vector.

	Index
interface A {	
void foo();	0
}	
interface B extends A {	
void bar(int x);	1
void baz();	2
}	
class C implements B {	
void foo() {…}	0
void bar(int x) {…}	1
void baz() {…}	2
void quux() {…}	3
}	

Inheritance / Subtyping:
C <: B <: A

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*)



Representing Classes in the LLVM

- During typechecking, create a *class hierarchy*
  - Maps each class to its interface:
    - Superclass
    - Constructor type
    - Fields
    - Method types (plus whether they inherit & which class they inherit from)

- Compile the class hierarchy to produce:
  - An LLVM IR struct type for each object instance
  - An LLVM IR struct type for each vtable (a.k.a. class table)
  - Global definitions that implement the class tables

Example OO Code (Java)

```
class A {
  A (int x)                    // constructor
  { super(); int x = x; }

  void print() { return; }     // method1
  int blah(A a) { return 0; }  // method2

}

class B extends A {
  B (int x, int y, int z){
    super(x);
    int y = y;
    int z = z;
  }

  void print() { return; }   // overrides A
}

class C extends B {
  C (int x, int y, int z, int w){
    super(x,y,z);
    int w = w;
  }
  void foo(int a, int b) {return;}
  void print() {return;}    // overrides B
}
```

Type Translation of a Class

- Each class gives rise to two implementation types:

- Object Instance Type
  - pointer to the dispatch vector
  - fields of the class

- Dispatch Vector Type
  - pointer to the superclass dispatch vector
  - pointers to methods of the class

- The inheritance hierarchy is used to statically construct the global class tables
  - which are records that have Dispatch Vector Types

Example OO Hierarchy in LLVM

Object instance types

Class table types

```
%Object = type { %_class_Object* }
%_class_Object = type {  }

%A = type { %_class_A*, i64 }
%_class_A = type { %_class_Object*, void (%A*)*, i64 (%A*, %A*)* }

%B = type { %_class_B*, i64, i64, i64 }
%_class_B = type { %_class_A*, void (%B*)*, i64 (%A*, %A*)* }

%C = type { %_class_C*, i64, i64, i64, i64 }
%_class_C = type { %_class_B*, void (%C*)*, i64 (%A*, %A*)*, void (%C*, i64, i64)* }
```

```
@_vtbl_Object = global %_class_Object {  }

@_vtbl_A = global %_class_A { %_class_Object* @_vtbl_Object,
                  void (%A*)* @print_A,
                  i64 (%A*, %A*)* @blah_A }

@_vtbl_B = global %_class_B { %_class_A* @_vtbl_A,
                  void (%B*)* @print_B,
                  i64 (%A*, %A*)* @blah_A }

@_vtbl_C = global %_class_C { %_class_B* @_vtbl_B,
                  void (%C*)* @print_C,
                  i64 (%A*, %A*)* @blah_A,
                  void (%C*, i64, i64)* @foo_C }
```

Class tables (structs containing function pointers)

Method Arguments

- Methods bodies are compiled just like top-level procedures…
- … except that they have an implicit extra argument: this (or self)
  - Historically (Smalltalk), these were called the “receiver object”
  - Method calls were thought of a sending “messages” to “receivers”

A method in a class...

```
class IntSet1 implements IntSet {
  …
  IntSet1 insert(int i) { <body> }
}
```

… is compiled like this (top-level) procedure:

```
IntSet1 insert(IntSet1 this, int i) { <body> }
```

- Note 1: the type of “this” is the class containing the method.
- Note 2: references to fields inside <body> are compiled like this.field

LLVM Method Invocation Compilation

- Consider method invocation:

  $[\![H;G;L \vdash e.m(e_1,\ldots,e_n):t]\!]$

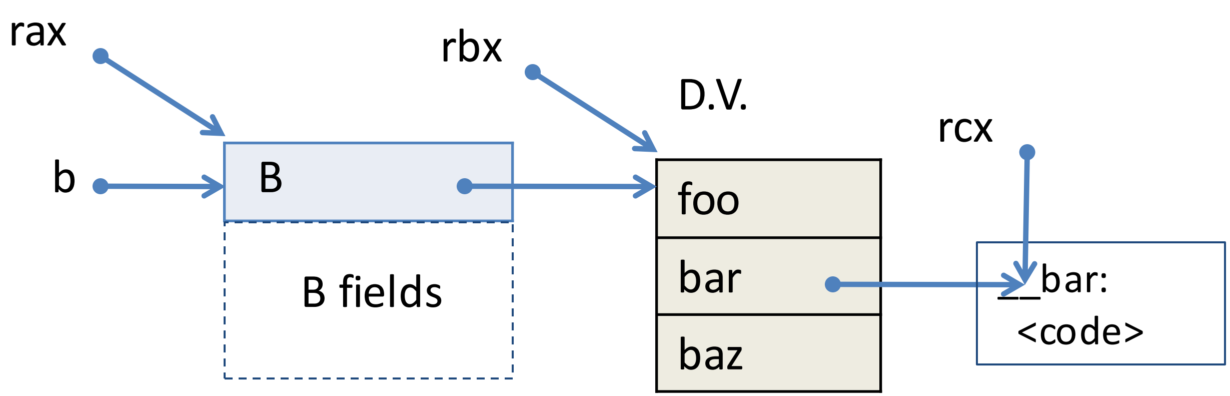
- First, compile $[\![H;G;L \vdash e : C]\!]$
  to get a (pointer to) an object value of class type C
  - Call this value %obj_ptr
- Use getelementptr to extract the vtable pointer from %obj_ptr
- load the vtable pointer
- Use getelementptr to extract the address of the function pointer from the vtable
  - using the information about C in H
- load the function pointer
- Call through the function pointer, passing '%obj_ptr' for this:

  call (cmp_typ t) m(obj_ptr, $[\![e_1]\!]$, …, $[\![e_n]\!]$)

- In general, function calls may require bitcast to account for subtyping: arguments may be a subtype of the expected "formal" type

X86 Code For Dynamic Dispatch

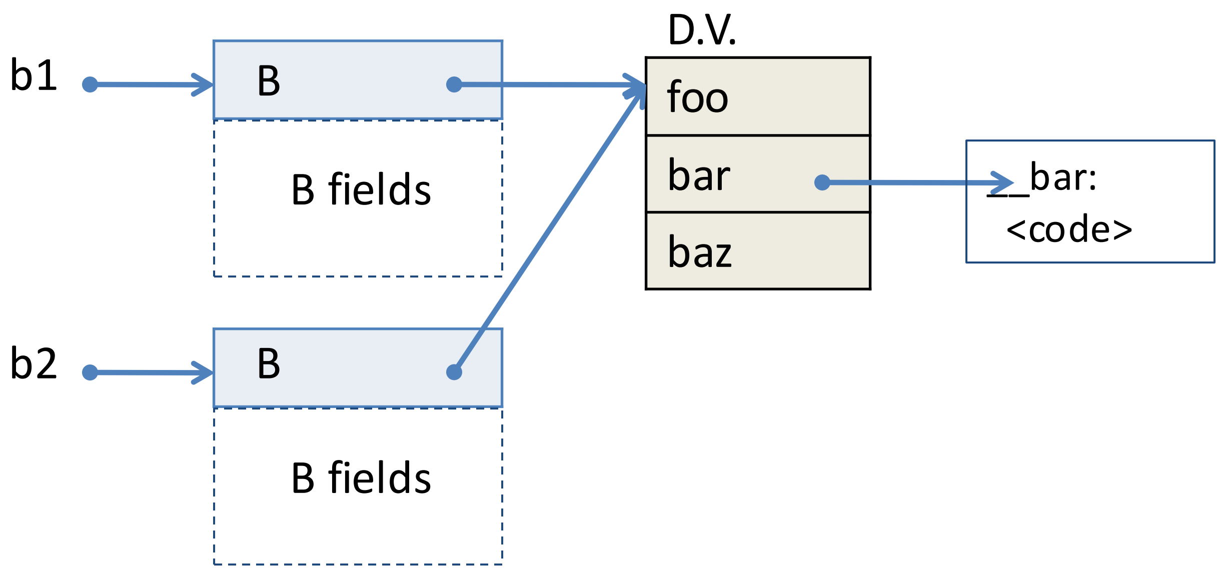
- Suppose b : B
- What code for b.bar(3)?
  - bar has index 1
  - Offset = 8 * 1



movq ⟦b⟧, %rax
movq [%rax], %rbx
movq [rbx+8], %rcx *// D.V. + offset*
movq %rax, %rdi *// "this" pointer*
movq 3, %rsi *// Method argument*
call %rcx *// Indirect call*

Sharing Dispatch Vectors

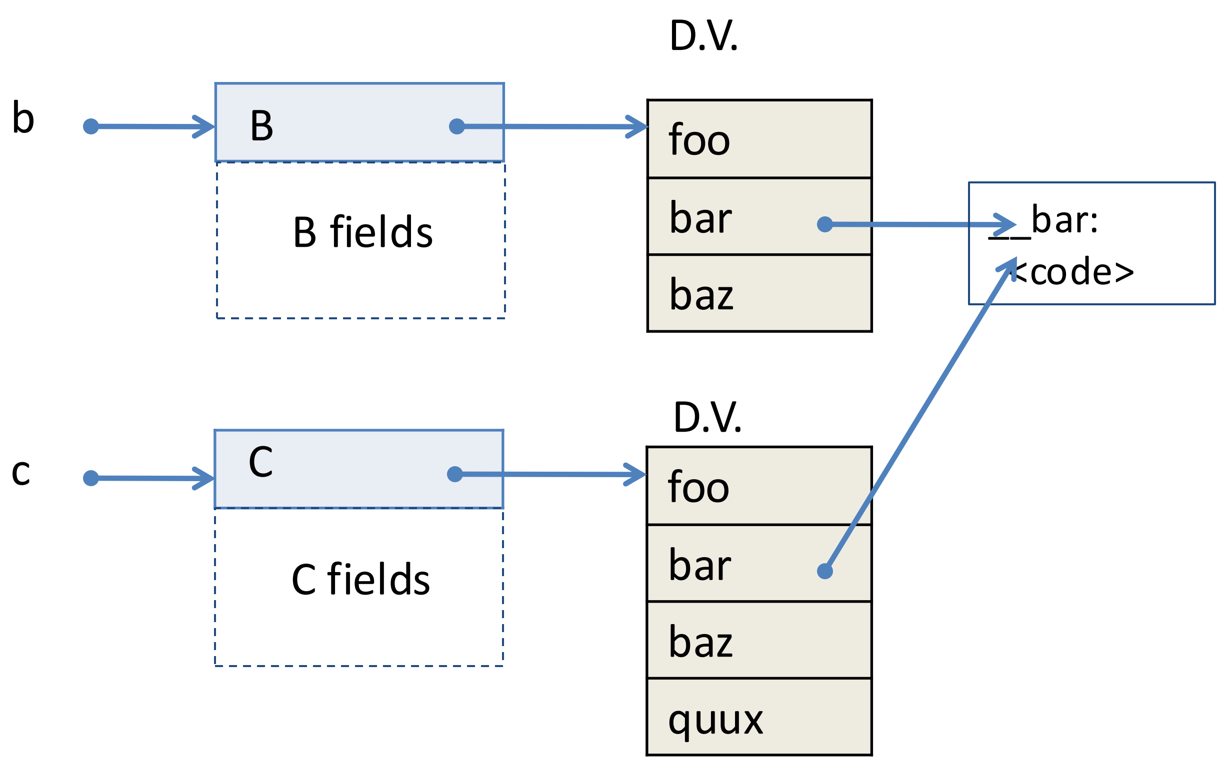
- All instances of a class may share the same dispatch vector.
  - Assuming that methods are immutable.
- Code pointers stored in the dispatch vector are available at link time – dispatch vectors can be built once at link time.



- One job of the object constructor is to fill in the object's pointer to the appropriate dispatch vector.
- Note: The address of the D.V. *is* the run-time representation of the object's type.

Inheritance: Sharing Code

- Inheritance: Method code "copied down" from the superclass
  - If not overridden in the subclass
- Works with separate compilation – superclass code not needed.



Compiling Static Methods

- Java supports *static* methods
  - Methods that belong to a class, not the instances of the class.
  - They have no "this" parameter (no receiver object)
- Compiled exactly like normal top-level procedures
  - No slots needed in the dispatch vectors
  - No implicit "this" parameter
- They're not really methods
  - They can only access static fields of the class

Compiling Constructors

- Java and C++ classes can declare constructors that create new objects.
  - Initialization code may have parameters supplied to the constructor
  - e.g. new Color(r,g,b);

- Modula-3: object constructors take no parameters
  - e.g. new Color;
  - Initialization would typically be done in a separate method.

- Constructors are compiled just like static methods, except:
  - The "this" variable is initialized to a newly allocated block of memory big enough to hold D.V. pointer + fields according to object layout
  - Constructor code initializes the fields
    - What methods (if any) are allowed?
  - The D.V. pointer is initialized
    - When? Before/After running the initialization code?