Lecture 18

CIS 341: COMPILERS

Announcements

- Reminder: HW4: Compiling OAT v.1
- DUE: *TONIGHT*

- HW5: Full OAT: Objects & Typechecking
  - Implement (parts of) the typechecker and compiler for an OO-language
- Available soon
- DUE: April 6th

TYPECHECKING OAT

OAT Classes

- Class-based Object-Oriented Language
- Classes are named
- Explicit Constructor:
  - Marked with the 'new' keyword
  - Take a list of parameters
  - Explicitly invoke the superclass constructor
  - Fields with initializers declared in a 'constructor block'
  - No other code allowed in the constructor

```
class A {
  new (int x)()  // constructor parameters & superclass call
  { int x = x; } // declare field 'x'

  void print() {  // method
    print_string(string_cat("A: x=", string_of_int(this.x)));
    return;
  }
}
```

OAT Inheritance

- Single, Declared Inheritance
  - Omitted superclass defaults to "Object"
  - Fields are all 'public' scope, children cannot override them
  - Methods are all 'public' scope, children *can* override them
  - No overloading
    - i.e. a method name appears at most once in each class

```
class B <: A {      // extend 'A'
  new (int x, int y, int z)(x){  // invoke superclass with (x)
    int y = y;      // declare two new fields 'y' and 'z'
    int z = z;
  }

  void print() {  // override the 'print' method
    print_string(string_cat("B: x=", string_of_int(this.x)));
    print_string(string_cat("B: y=", string_of_int(this.y)));
    print_string(string_cat("B: z=", string_of_int(this.z)));
  }
}
```

Typechecking

- Hierarchy is single-inheritance
  - Classes inherit only from classes previously declared
- Class scopes are mutually recursive
  - Class fields and methods may mention any class
- Strict distinction between "possibly null" reference types `R?` and "definitely not null" types `R`
- Checked downcast:

```
if? (typ x = exp) { …} else { …}
```

- Questions:
  - How to initialize fields of objects? (what data is in scope?)

```
class A {
  new ()() {
    B b = new B();
  }
}

class B {
  new ()() {
    A? a = null;
  }

  void set(A a) {
    this.a = a;
  }
}
```

Subtyping in Oat: Base Types

$$\frac{}{H \vdash \texttt{int} <: \texttt{int}} \quad \textsc{Typ\_Sub\_Int}$$

$$\frac{}{H \vdash \texttt{bool} <: \texttt{bool}} \quad \textsc{Typ\_Sub\_Bool}$$

$$\frac{}{H \vdash \texttt{null} <: \texttt{null}} \quad \textsc{Typ\_Sub\_Null}$$

Subtyping in Oat: Reference Types

$$\frac{}{H \vdash \texttt{null} <: ref?} \quad \textbf{TYP\_SUB\_NULLS}$$

$$\frac{H \vdash_r ref_1 <: ref_2}{H \vdash ref_1? <: ref_2?} \quad \textbf{TYP\_SUB\_NREF}$$

$$\frac{H \vdash_r ref_1 <: ref_2}{H \vdash ref_1 <: ref_2} \quad \textbf{TYP\_SUB\_REF}$$

$$\frac{H \vdash_r ref_1 <: ref_2}{H \vdash ref_1 <: ref_2?} \quad \textbf{TYP\_SUB\_NRREF}$$

Subtyping in Oat: Reference Types

$$\frac{}{H \vdash_r \texttt{string} <: \texttt{string}} \quad \textsc{Typ\_SubRString}$$

$$\frac{}{H \vdash_r t[] <: t[]} \quad \textsc{Typ\_SubRArray}$$

$$\frac{H \vdash C_1 <:^* C_2}{H \vdash_r C_1 <: C_2} \quad \textsc{Typ\_SubRClass}$$

$$\frac{C <: C_1\ \{\ldots\} \in H}{H \vdash C <:^* C} \quad \textsc{Typ\_ExtRefl}$$

$$\frac{C <: C_1\ \{\ldots\} \in H \quad H \vdash C_1 <:^* C_2}{H \vdash C <:^* C_2} \quad \textsc{Typ\_ExtTrans}$$

Typechecking OAT: Classes & Constructors

$$\frac{G;H;C_2 \vdash ctr \Rightarrow L \quad G;H;L;C_1;C_2 \vdash fdecl_1 \quad .. \quad G;H;L;C_1;C_2 \vdash fdecl_i}{G;H \vdash \texttt{class}\ C_1\ \texttt{<:}\ C_2\ \texttt{\{}\ ctr\ fdecl_1\ ..fdecl_i\ \texttt{\}}\ \ \textbf{ok}}$$

$$\frac{\begin{array}{l} C\ \texttt{<:}\ C'\ \texttt{\{}(t'_1, .., t'_j) \rightarrow C\ fieldtyps\ methodtyps\texttt{\}} \quad \in H \\ G;H;x_1:t_1, .., x_i:t_i \vdash exp_1 : t''_1 \quad .. \quad G;H;x_1:t_1, .., x_i:t_i \vdash exp_j : t''_j \\ H \vdash t''_1\ \texttt{<:}\ t'_1 \quad .. \quad H \vdash t''_j\ \texttt{<:}\ t'_j \\ G;H;x_1:t_1, .., x_i:t_i \vdash_f decls \Rightarrow L \end{array}}{G;H;C \vdash \texttt{new}\ (t_1\ x_1, .., t_i\ x_i)(exp_1, .., exp_j)\texttt{\{}decls\texttt{\}} \quad \Rightarrow L}$$

Typechecking OAT: Downcasts

$$\frac{G;H;L \vdash exp : t' \quad H \vdash t <: t' \quad t \neq t' \quad x \notin L \qquad G;H;L,x:t;rt \vdash block_1 \quad G;H;L;rt \vdash block_2}{G;H;L;rt \vdash \texttt{if?}(t\ x = exp)\ block_1\ \texttt{else}\ block_2 \Rightarrow L}$$

- The checks t <: t′ and t != t′ ensure that this is not a "silly" downcast:
  - If t = t′ then no cast is needed and the 'true' branch will always be taken
  - If t is not a subtype of t′ then the false branch will always be taken

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:

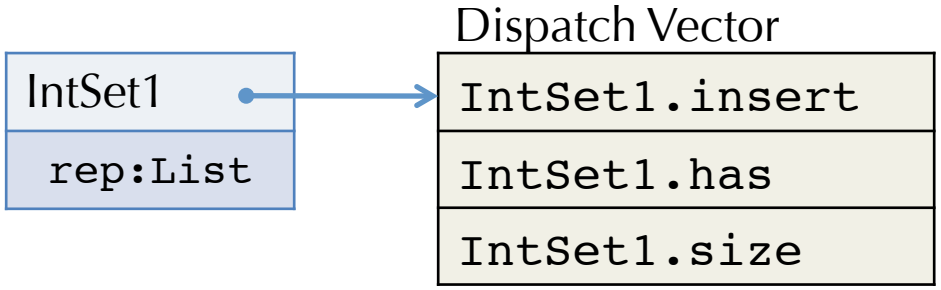
```
IntSet set = …;
int x = set.size();
```

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

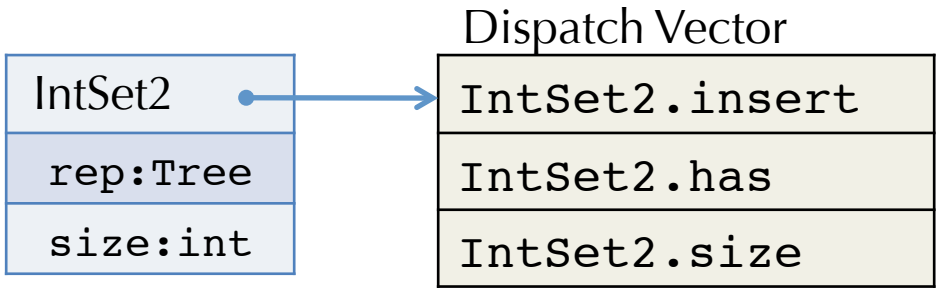
- Client code doesn't know the answer.
  - So objects must "know" which code to call.
  - Invocation of a method must indirect through the object.

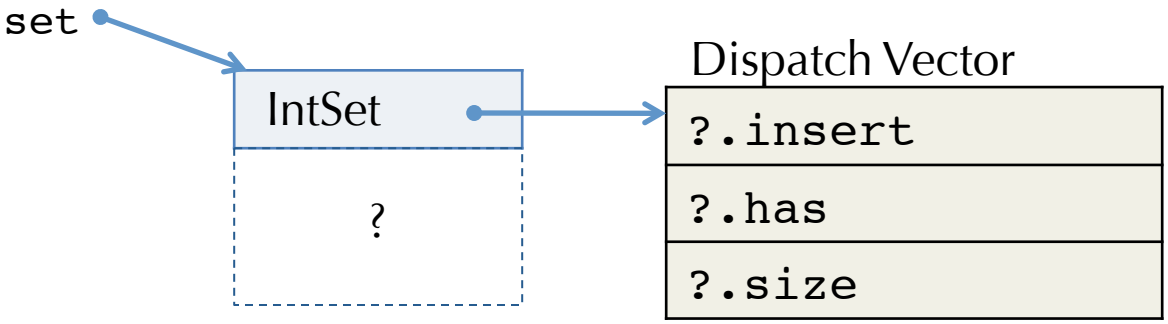
Compiling Objects

- 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.

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

Index

0

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

1
2

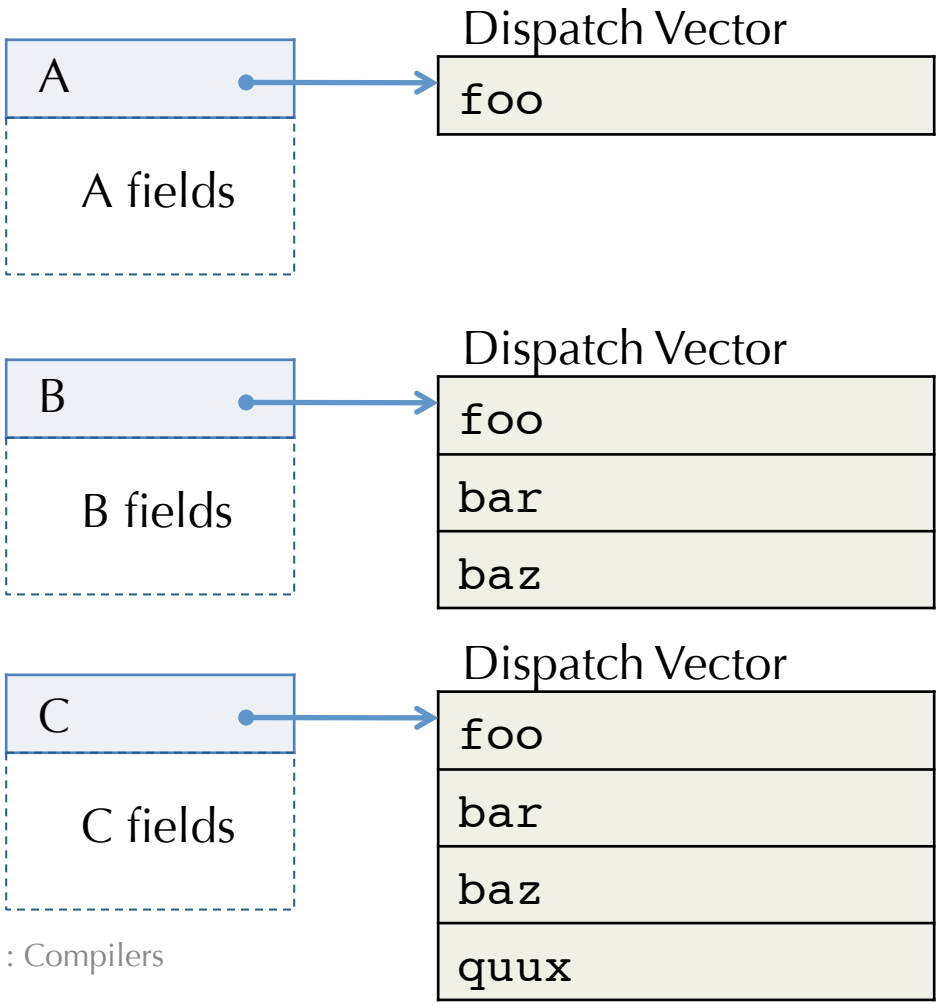
Inheritance / Subtyping:
C <: B <: A

```
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)



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 OAT Code

```
class A {
  new (int x)()
  { int x = x; }

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

}

class B <: A {
  new (int x, int y, int z)(x){
    int y = y;
    int z = z;
  }

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

class C <: B {
  new (int x, int y, int z, int w)(x,y,z){
    int w = w;
  }

  void foo(int a, int b) {return;}
  void print() {return;}    // overrides B
}
```

Example OAT Hierarchy in LLVM

```
%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 }
```

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:

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

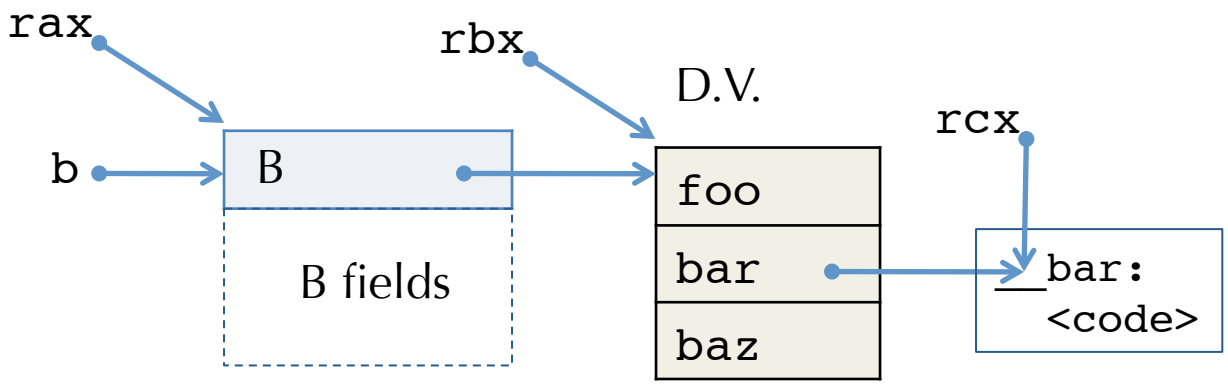
- First, compile $[\![G;H;L \vdash \texttt{e : C}]\!]$
  to get a (pointer to) an object value of class type C
  - Call this value obj_ptr
- Use Getelementptr to extract the vtable pointer from obj_ptr
- Load the vtable pointer
- Use Getelementptr to extract the function poiner from the vtable
  - using 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

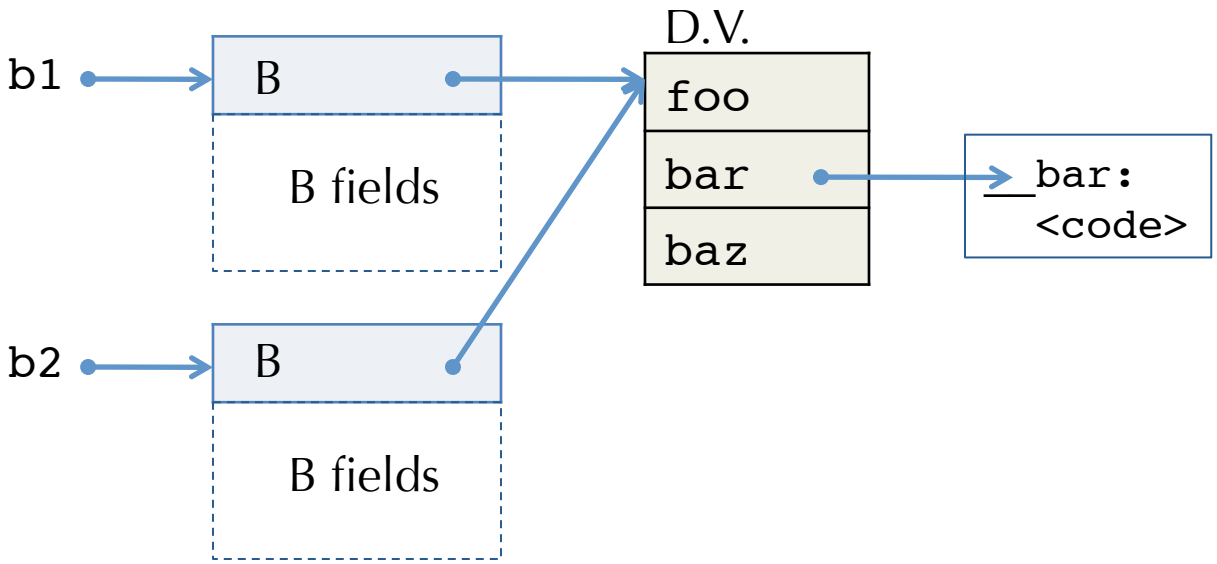
- Suppose `b : B`
- What code for `b.bar(3)`?
  - `bar` has index 1
  - Offset = 8 * 1




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

Sharing Dispatch Vectors

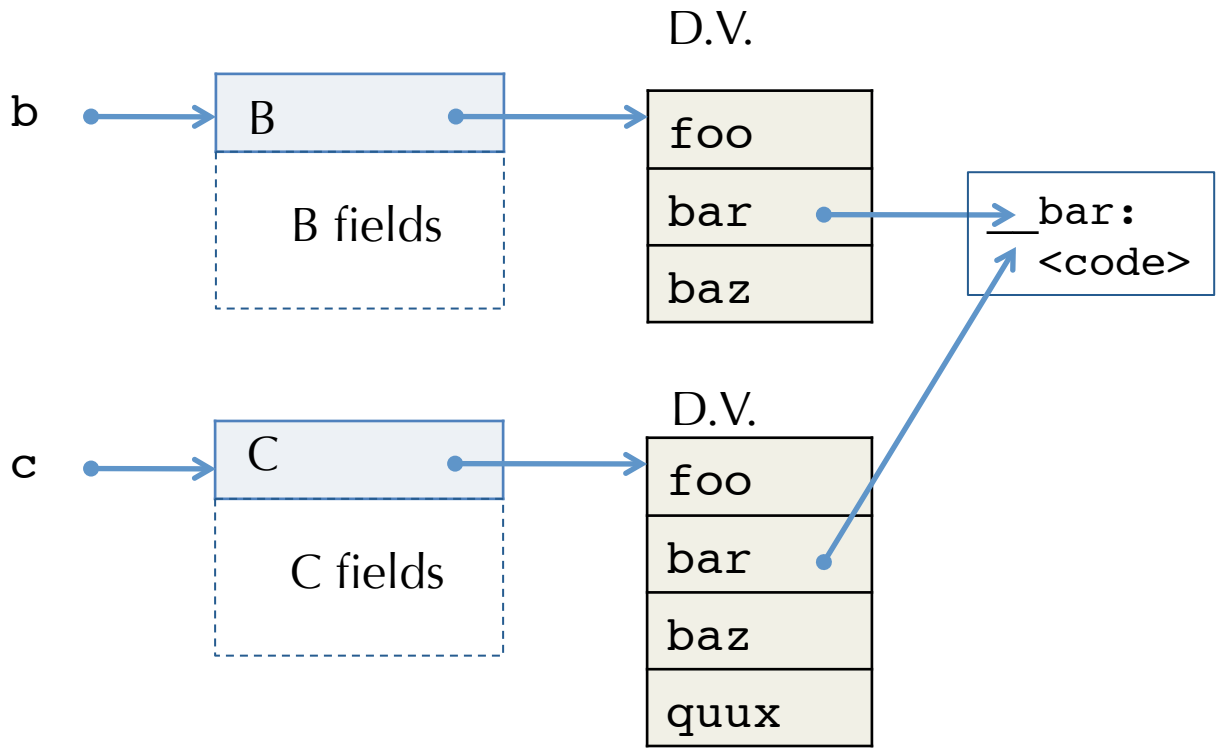
- All instances of a class may share the same dispatch vector.
  - Assuming that methods are immutable.
- Code pointers stored in the dispatch vector are available at link time – dispatch vectors can be built once at link time.



- 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

- OAT, Java, 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?

Compiling Checked Casts

- How do we compile downcast:

```
if? (t x = exp) { … } else { … }
```

- Reason by cases:
  - t must be either null, ref or ref? (can't be just int or bool)
- If t is null:
  - The static type of exp must be ref? for some ref.
  - If exp == null then take the true branch, otherwise take the false branch
- If t is string or t[]:
  - The static type of exp must be the corresponding string? Or t[]?
  - If exp == null take the false branch, otherwise take the true branch
- If t is C:
  - The static type of exp must be D or D? (where C <: D)
  - If exp == null take the false branch, otherwise:
  - emit code to walk up the class hierarchy starting at D, looking for C
  - If found, then take true branch else take false branch
- If t is C?:
  - The static type of exp must be D? (where C <: D)
  - If exp == null take the true branch, otherwise:
  - Emit code to walk up the class hierarchy starting at D, looking for C
  - If found, then take true branch else take false branch

"Walking up the Class Hierarchy"

- A non-null object pointer refers to an LLVM struct with a type like:

```
%B = type { %_class_B*, i64, i64, i64 }
```

- The first entry of the struct is a pointer to the vtable for Class B
  - This pointer *is* the dynamic type of the object.
  - It will have the value @vtbl_B
- The first entry of the class table for B is a pointer to its superclass:

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

- Therefore, to find out whether an unknown type X is a subtype of C:
  - Assume C is not Object (ruled out by "silliness" checks for downcast)

  LOOP:
  - If X == @_vtbl_Object then NO, X is not a subtype of C
  - If X == @_vtbl_C then YES, X is a subtype of C
  - If X = @_vtbl_D, so set X to @_vtbl_E where E is D's parent and goto LOOP