Lecture 20

CIS 341: COMPILERS

Announcements / Plan

- HW5: OAT – typechecking, structs, function pointers
  - Due: Thursday, April 13

  As always, *start early*!

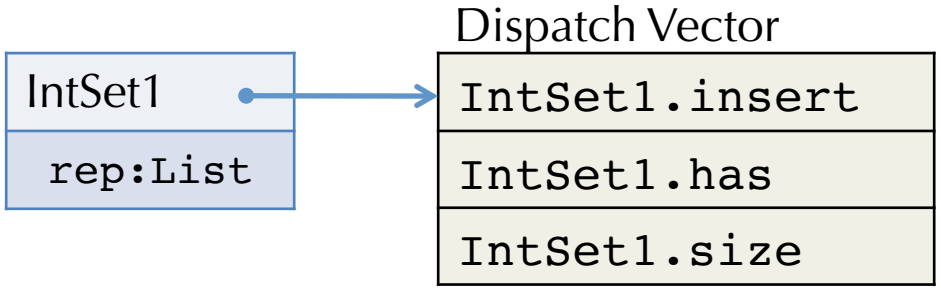
- HW6: LLVM Optimization: analysis and register allocation
  - Due: Wednesday, April 26

- FINAL EXAM: Thursday, May 4th noon – 2:00p.m.

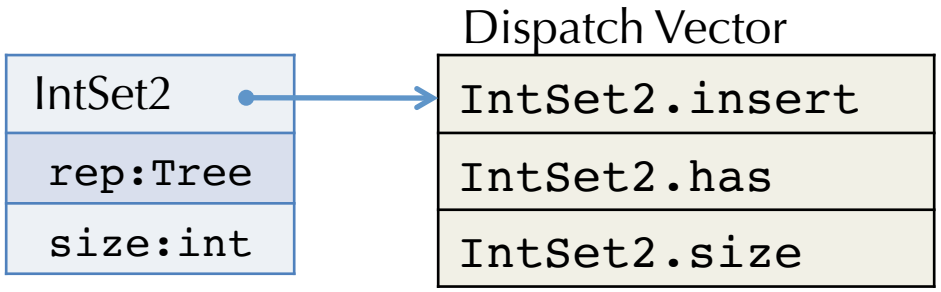
COMPILING CLASSES AND OBJECTS

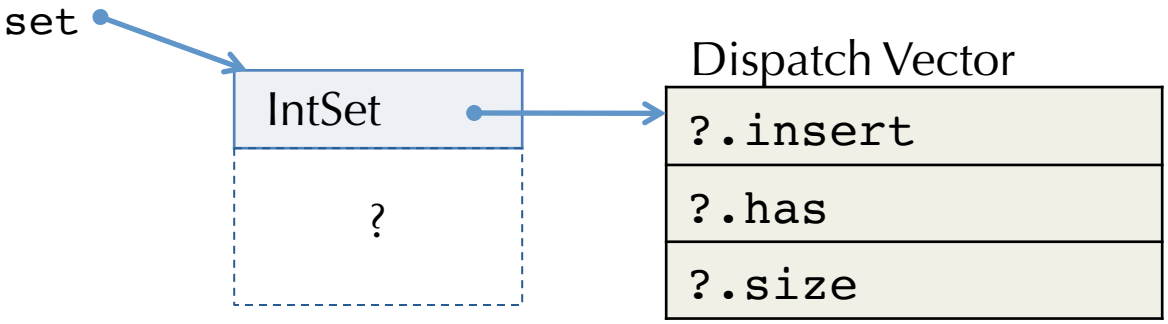
Compiling Objects

- 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);
  void baz();
}
```

1
2

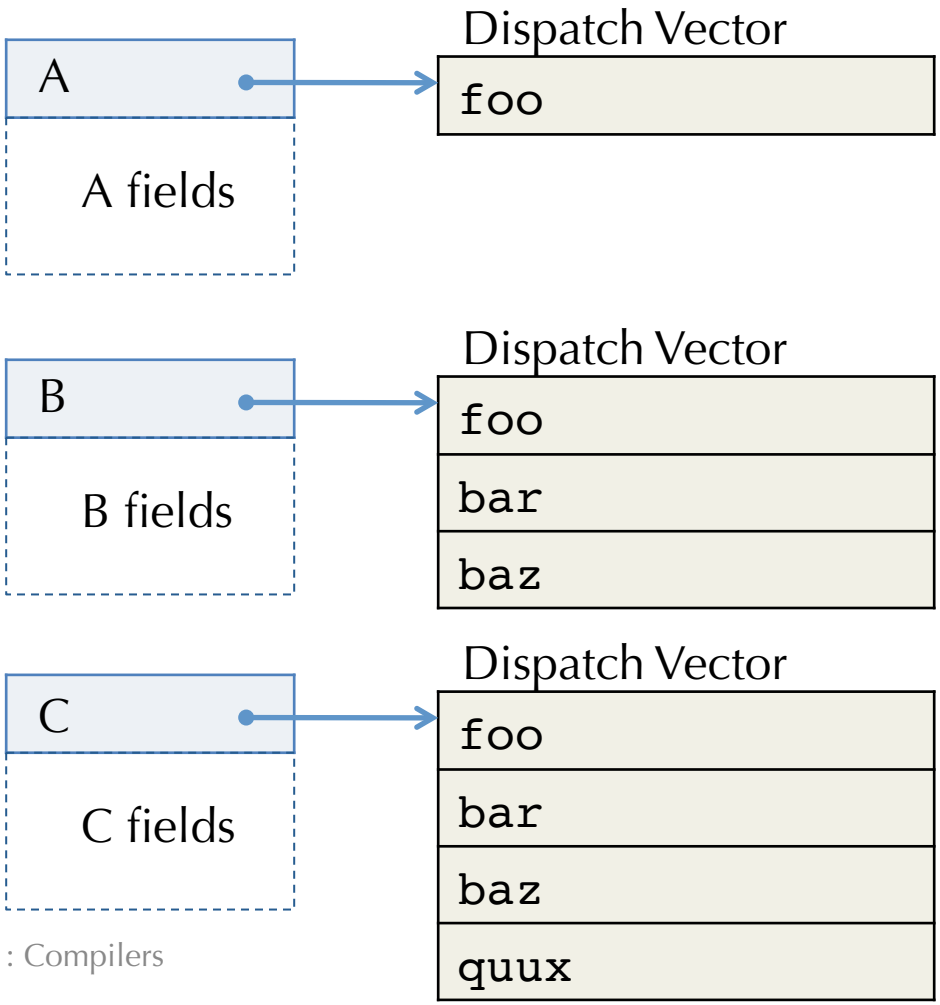
```
class C implements B {
  void foo() {…}
  void bar(int x) {…}
  void baz() {…}
  void quux() {…}
}
```

0
1
2
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

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

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

}

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 OO 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:

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

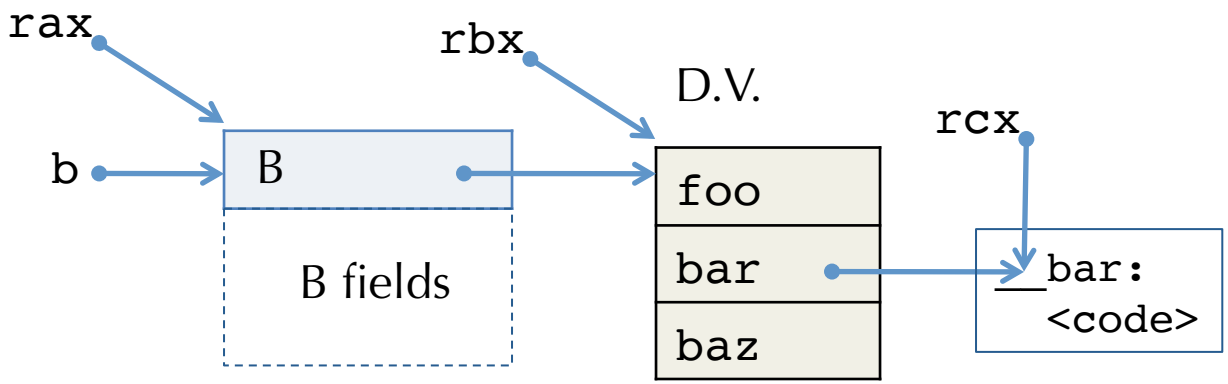
- First, compile $[\![G;H;L \vdash \texttt{e : C}]\!]$
  to get a (pointer to) an object value of class type C
  - Call this value obj_ptr
- Use Getelementptr to extract the vtable pointer from obj_ptr
- Load the vtable pointer
- Use Getelementptr to extract the function poiner from the vtable
  - using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

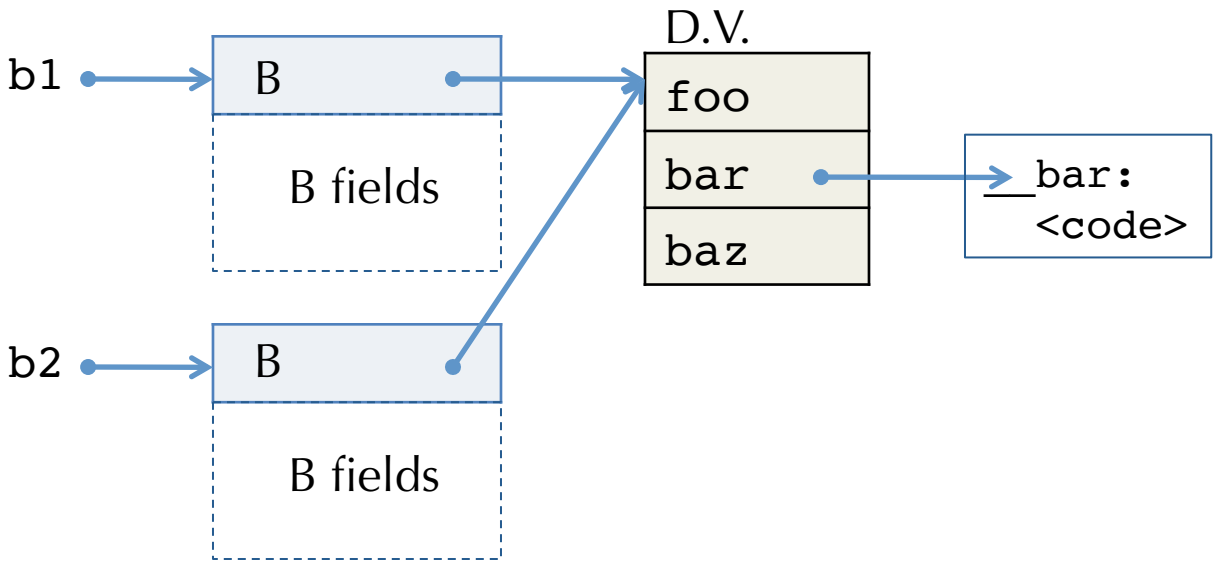
- Suppose `b : B`
- What code for `b.bar(3)`?
  - `bar` has index 1
  - Offset = 8 * 1




```
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

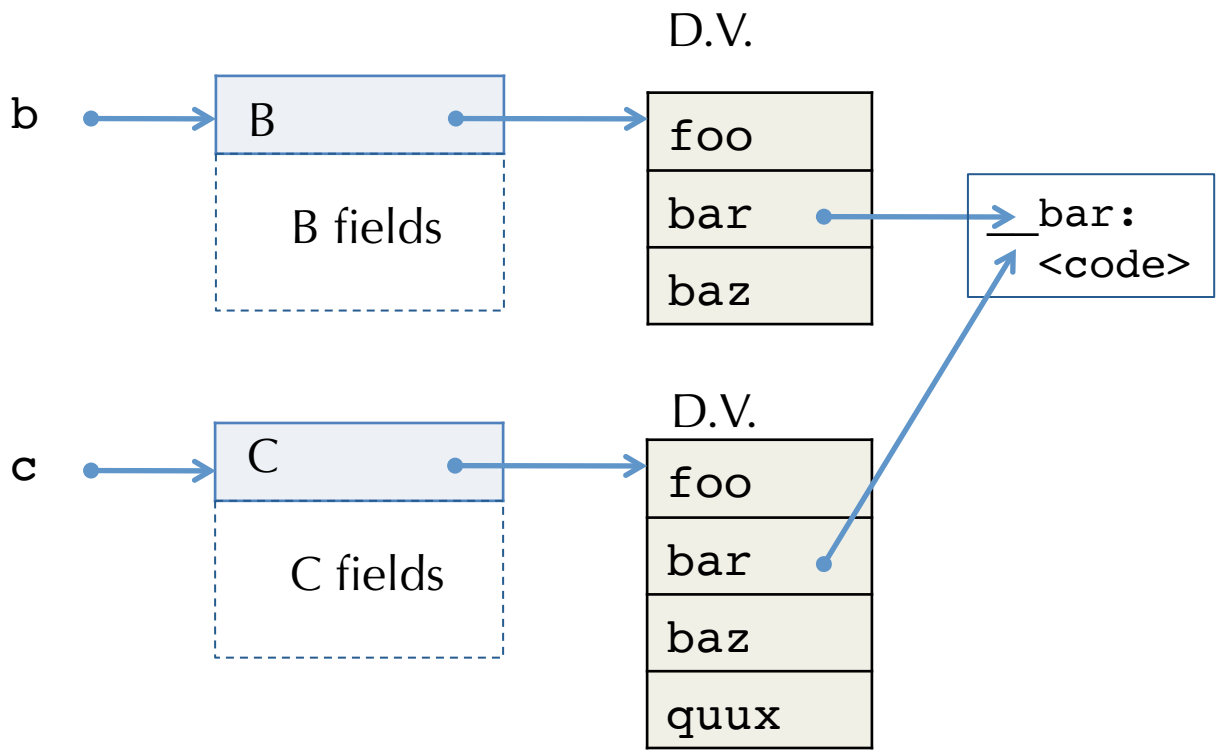
- All instances of a class may share the same dispatch vector.
  - Assuming that methods are immutable.
- Code pointers stored in the dispatch vector are available at link time – dispatch vectors can be built once at link time.



- 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

- 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 and instanceof? E.g. in Java:

```
C c = …    // create an object
D d = (D)c;
// or …
boolean b = c instanceof D;
```

- The static type C must such that D <: C
  - otherwise this is a "silly" cast [i.e. it is guaranteed to fail]
- If `c == null` then
  - cast immediately succeeds
  - `instanceof` immediately returns false
- Otherwise, the dynamic class of c must be some C' <: C
  - dynamically search up the class hierarchy to try to find D

"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

MULTIPLE INHERITANCE

Multiple Inheritance

- C++: a class may declare more than one superclass.
- Semantic problem: Ambiguity

```
class A { int m(); }
class B { int m(); }
class C extends A,B {…}     // which m?
```

  - Same problem can happen with fields.
  - In C++, fields and methods can be duplicated when such ambiguity arises (though explicit sharing can be declared too)

- Java: a class may implement more than one interface.
  - No semantic ambiguity: if two interfaces contain the same method declaration, then the class will implement a single method

```
interface A { int m(); }
interface B { int m(); }
class C implements A,B {int m() {…}}     // only one m
```

Dispatch Vector Layout Strategy Breaks

```
interface Shape {                                  D.V.Index
  void setCorner(int w, Point p);                      0
}

interface Color {
  float get(int rgb);                                  0
  void set(int rgb, float value);                      1
}

class Blob implements Shape, Color {
  void setCorner(int w, Point p) {…}                   0?
  float get(int rgb) {…}                               0?
  void set(int rgb, float value) {…}                   1?
}
```