Lecture 20

CIS 4521/5521: COMPILERS

Announcements

- HW5: OAT v. 2.0
  - records, function pointers, type checking, array-bounds checks, etc.
  - Due: Wednesday, April 9th
  - Test cases due: Tuesday, April 8th

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

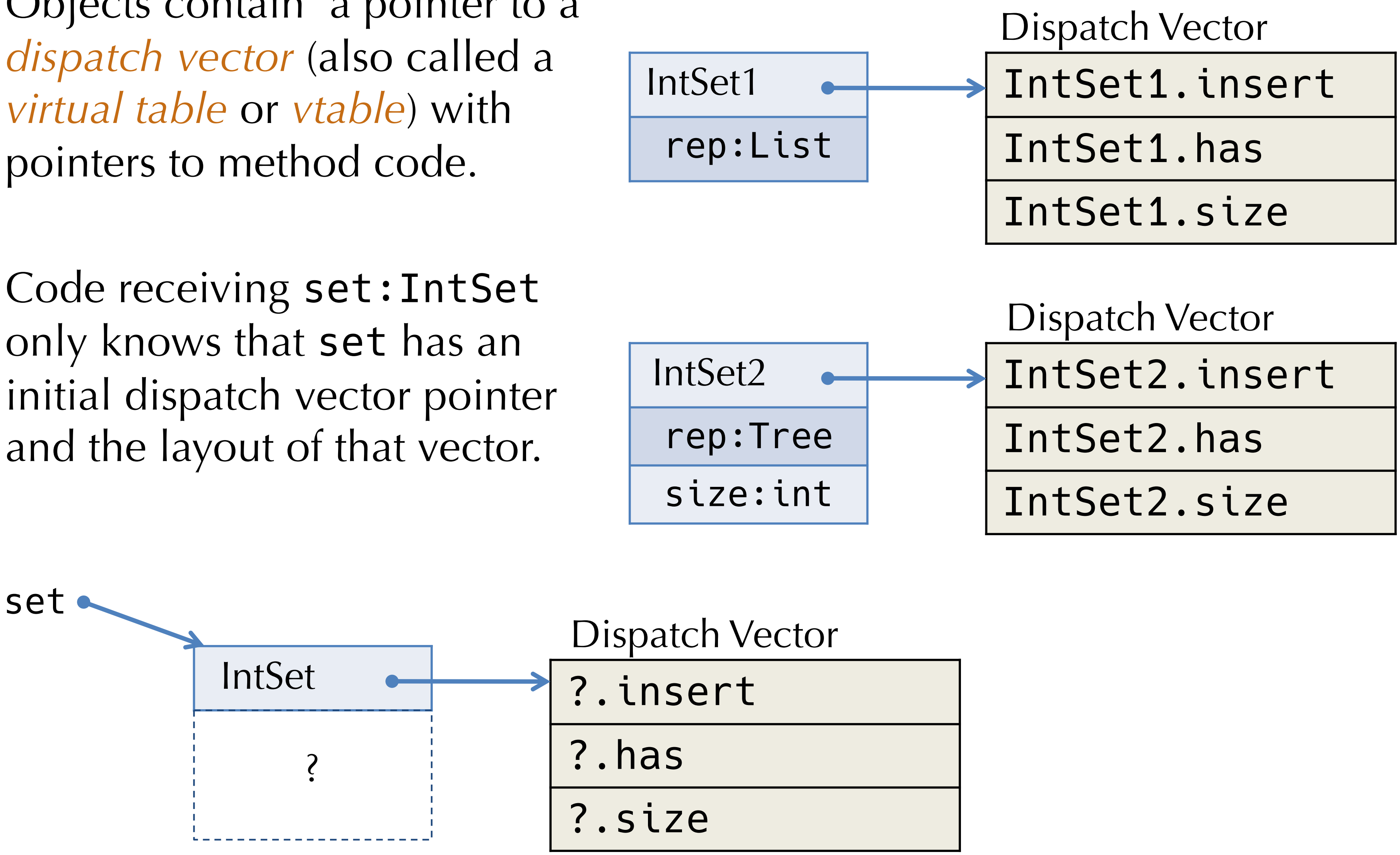
- Consider a client program that uses the IntSet interface:

```
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
}
```

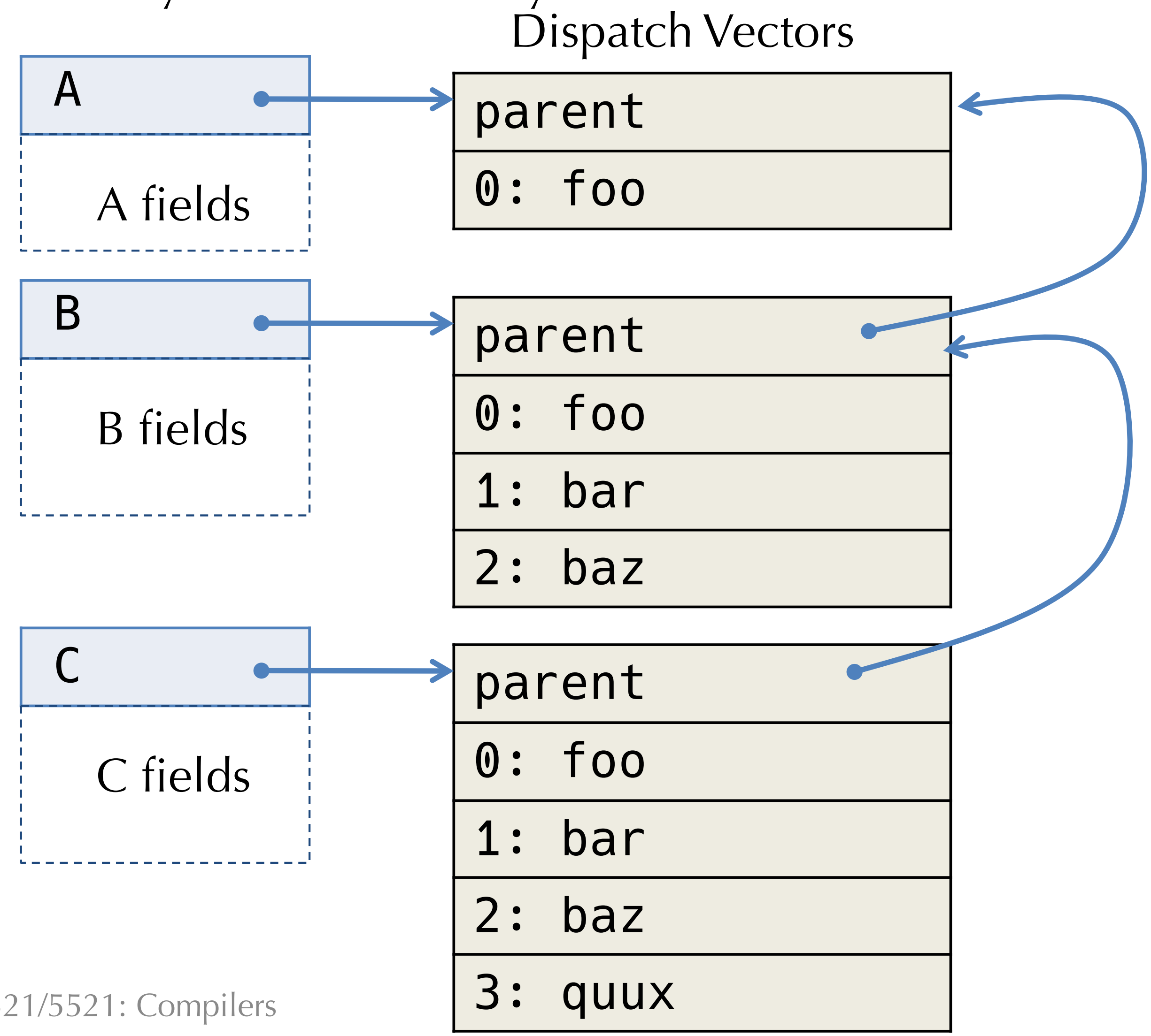
```
interface B extends A {
  void bar(int x);             1
  void baz();                  2
}
```

Inheritance / Subtyping:
C <: B <: A

```
class C implements B {
  void foo() {…}               0
  void bar(int x) {…}          1
  void baz() {…}               2
  void quux() {…}              3
}
```

Dispatch Vector Layouts

- Each interface and class gives rise to a dispatch vector layout.
- Note that inherited methods have identical indices in the subclass.
- Methods added by subclasses only add new rows: ***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 {
  int x;
  A (int x) { super(); this.x = x; } // constructor
  void print() { System.out.print(x); }     // method1
  int blah(A a) { return 0; }  // method2

}

class B extends A {
  int y; int z;  // Added fields
  B (int x, int y, int z){  // constructor
    super(x);
    this.y = y;
    this.z = z;
  }
  void print() { return; }  // overrides A
}

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

Type Translation of a Class

- Each class gives rise to two implementation types at the LLVM IR level:

- 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 structs 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:

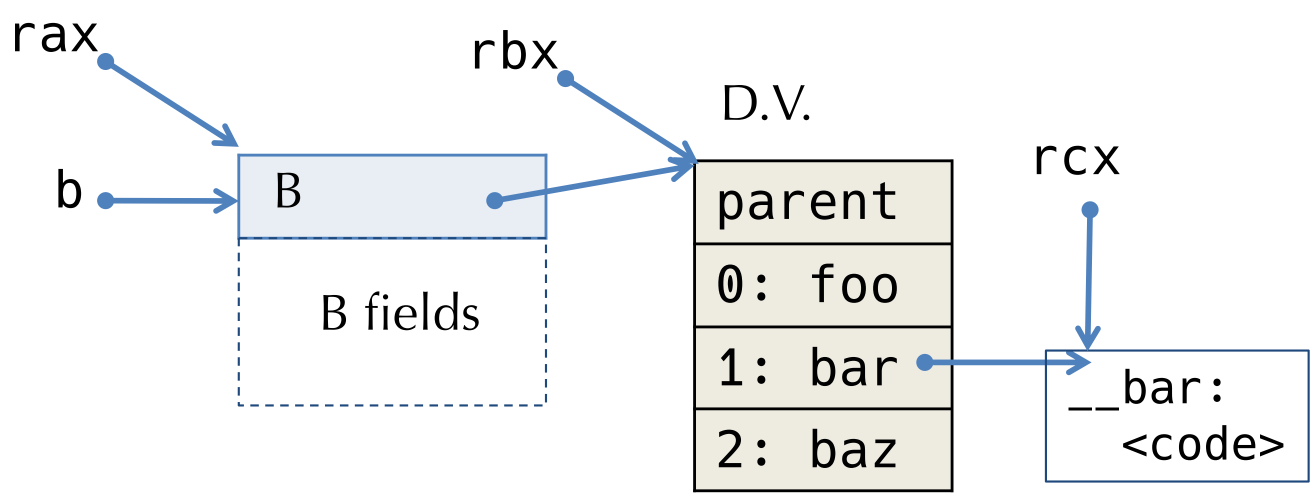
  $[\![H;G;L \vdash$ `e.m(e1,…,en):t` $]\!]$
- First, compile $[\![H;G;L \vdash$ `e : C` $]\!]$
  to get a (pointer to) an object value of class type C
  - Call this value `%obj_ptr`
- Use `getelementptr` to extract the vtable pointer from `%obj_ptr`
- `load` the vtable pointer
- Use `getelementptr` to 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

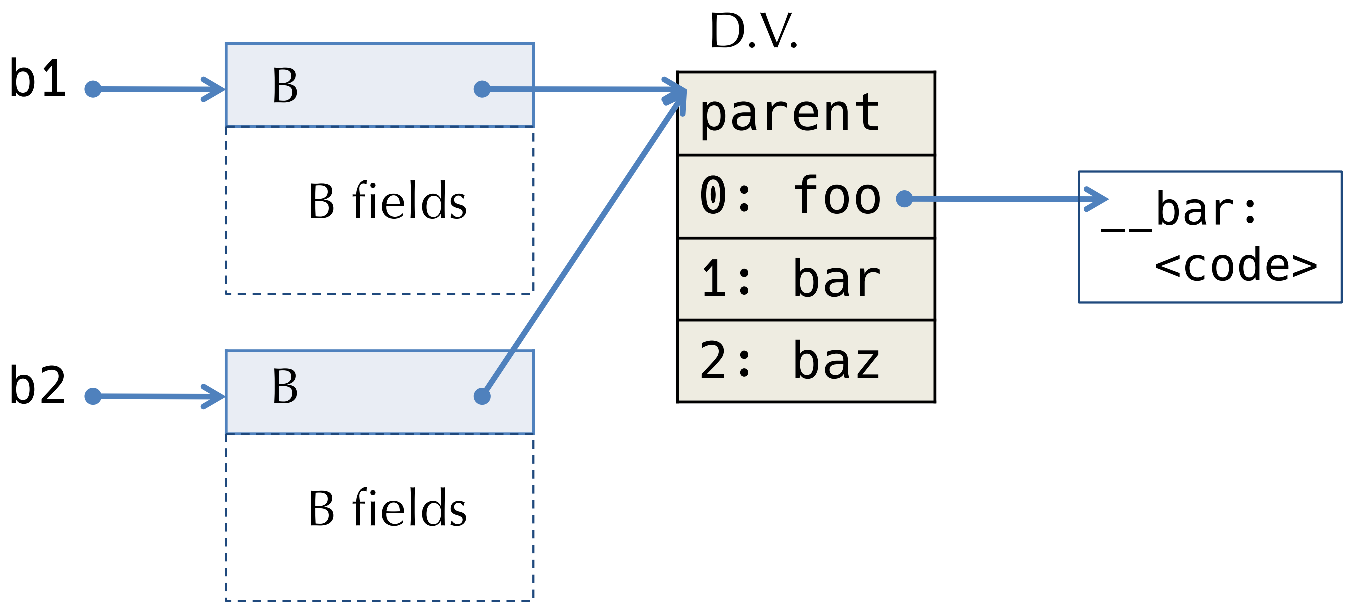
- Suppose `b : B`
- What code for `b.bar(3)`?
  - `bar` has index 1
  - Offset = 8 * (1+1)




```
movq ⟦b⟧, %rax
movq (%rax), %rbx
movq $16(rbx), %rcx      // D.V. + offset
movq %rax, %rdi          // "this" pointer
movq 3, %rsi             // Method argument
call *%rcx               // Indirect call
```

Sharing Dispatch Vectors

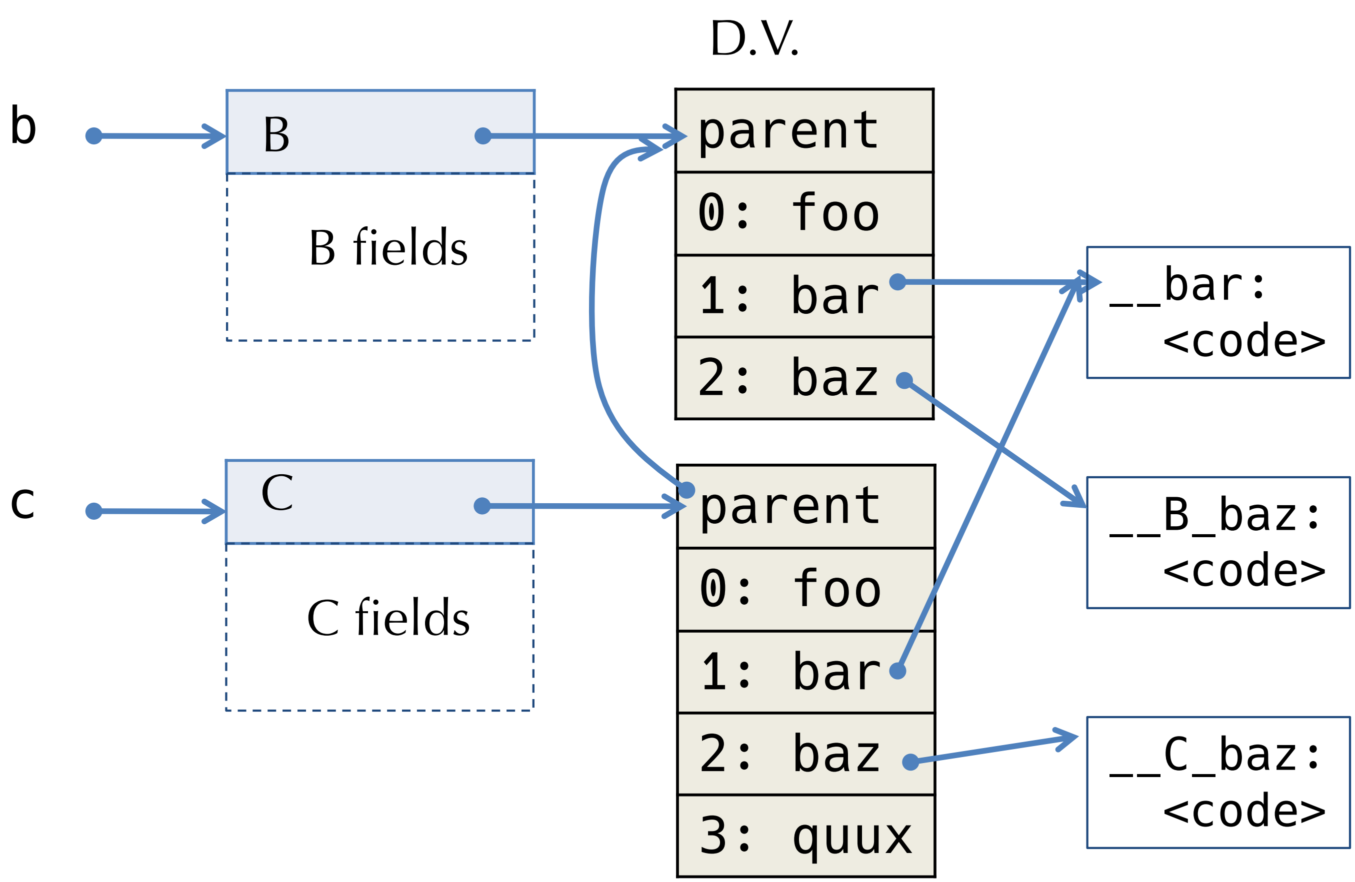
- All instances of a class may share the same dispatch vector.
  - Assuming that methods are immutable.
- Code pointers stored in the dispatch vector are available at link time – dispatch vectors can be built once at link time.



- 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
  - overriden methods have different dispatch pointers
- 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 methods, except:
  - The code pointer to call is determined *statically*
  - 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
    - call the super-class constructor first (to recursively initialize those fields)
    - What methods (if any) are allowed? What is the type of `this` during those calls?
  - The D.V. pointer is initialized last
    - When? After running the initialization code.

Compiling Checked Downcasts

- How do we compile downcast in general? Consider this Java-like generalization of Oat's checked cast, where `t` ranges over Java-style reference types:

```
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

else X == @_vtbl_D, so set X to @_vtbl_E where E is D's parent and goto LOOP

MULTIPLE INHERITANCE

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

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?
}
```

General Approaches

- Can't directly identify methods by position anymore.

- Option 1: Use a level of indirection:
  - Map method identifiers to code pointers (e.g. index by method name)
  - Use a hash table
  - May need to do search up the class hierarchy

- Option 2: Give up separate compilation
  - Use "sparse" dispatch vectors, or binary decision trees
  - Must know then entire class hierarchy

- Option 3: Allow multiple D.V. tables (C++)
  - Choose which D.V. to use based on static type
  - Casting from/to a class may require run-time operations

- Note: many variations on these themes
  - Different Java compilers pick different approaches to options1 and 2…

Option 2 variant 1: Sparse D.V. Tables

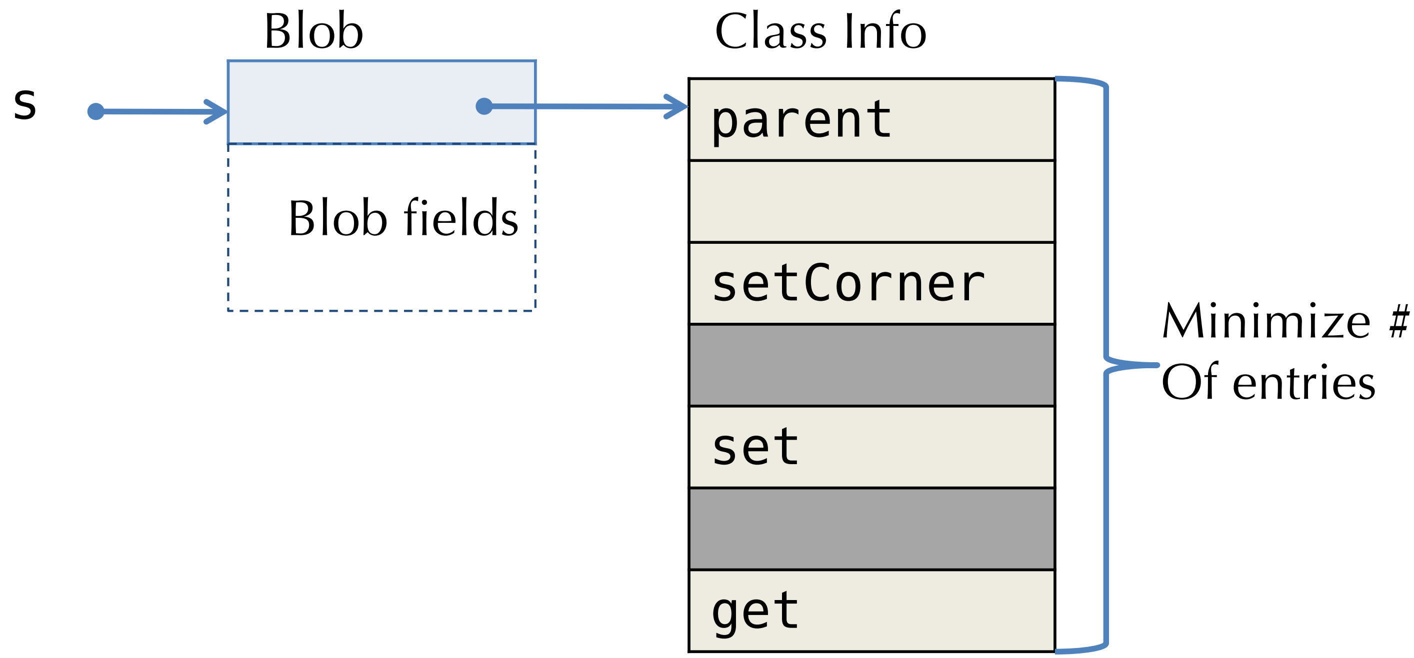
- Give up on separate compilation…
- Now we have access to the whole class hierarchy.

- So: ensure that no two methods in the same class are allocated the same D.V. offset.
  - Allow holes in the D.V. just like the hash table solution
  - Unlike hash table, there is never a conflict!

- Compiler needs to construct the method indices
  - Graph coloring techniques can be used to construct the D.V. layouts in a reasonably efficient way (to minimize size)
  - Finding an optimal solution is NP complete!

Example Object Layout

- Advantage: Identical dispatch and performance to single-inheritance case
- Disadvantage: Must know entire class hierarchy

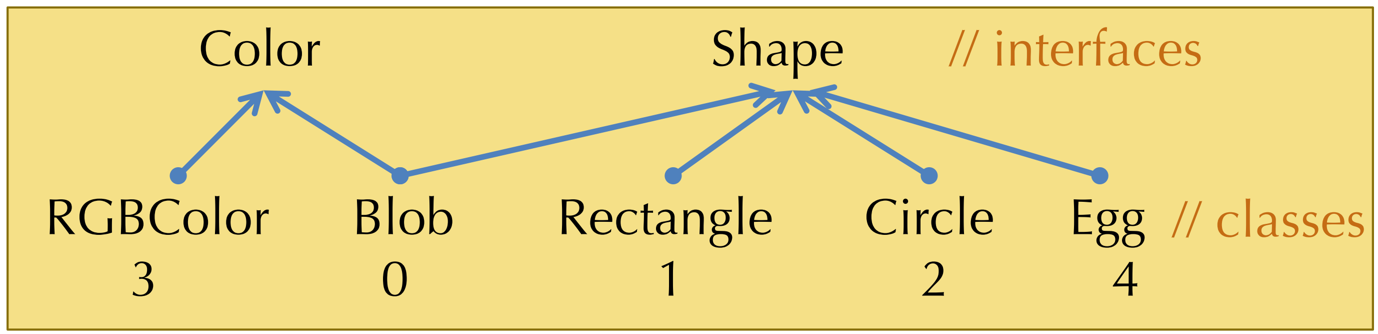


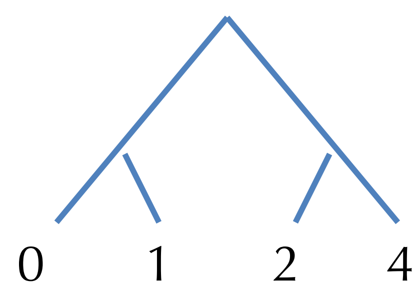
Option 2 variant 2: Binary Search Trees

- Idea: Use conditional branches not indirect jumps
- Each object has a class index (unique per class) as first word
  - Instead of D.V. pointer (no need for one!)
- Method invocation uses range tests to select among *n* possible classes in *lg n* time
  - Direct branches to code at the leaves.

```
Shape x;
x.SetCorner(…);
```

```
  Mov eax, ⟦x⟧
  Mov ebx, [eax]
  Cmp ebx, 1
  Jle  __L1
  Cmp ebx, 2
  Je __CircleSetCorner
  Jmp __EggSetCorner
__L1:
  Cmp ebx, 0
  Je __BlobSetCorner
  Jmp __RectangleSetCorner
```





Decision tree

Search Tree Tradeoffs

- Binary decision trees work well if the distribution of classes that may appear at a call site is skewed.
  - Branch prediction hardware eliminates the branch stall of ~10 cycles (on X86)
- Can use profiling to find the common paths for each call site individually
  - Put the common case at the top of the decision tree (so less search)
  - 90%/10% rule of thumb: 90% of the invocations at a call site go to the same class
- Drawbacks:
  - Like sparse D.V.'s you need the whole class hierarchy to know how many leaves you need in the search tree.
  - Indirect jumps can have better performance if there are >2 classes (at most one mispredict)

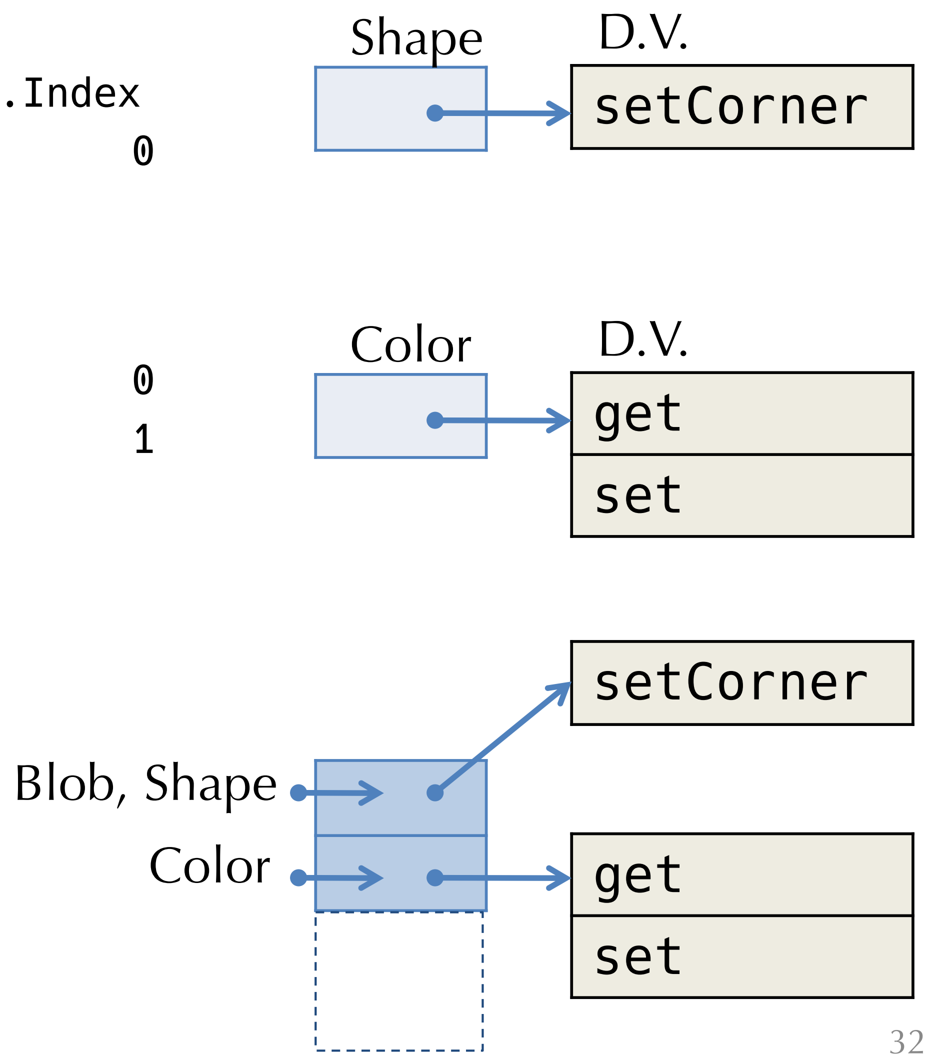
Option 3: Multiple Dispatch Vectors

- Duplicate the D.V. pointers in the object representation.
- Static type of the object determines which D.V. is used.

```
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) {…}
  float get(int rgb) {…}
  void set(int rgb, float value) {…}
}
```



Multiple Dispatch Vectors

- A reference to an object might have multiple "entry points"
  - Each entry point corresponds to a dispatch vector
  - Which one is used depends on the statically known type of the program.

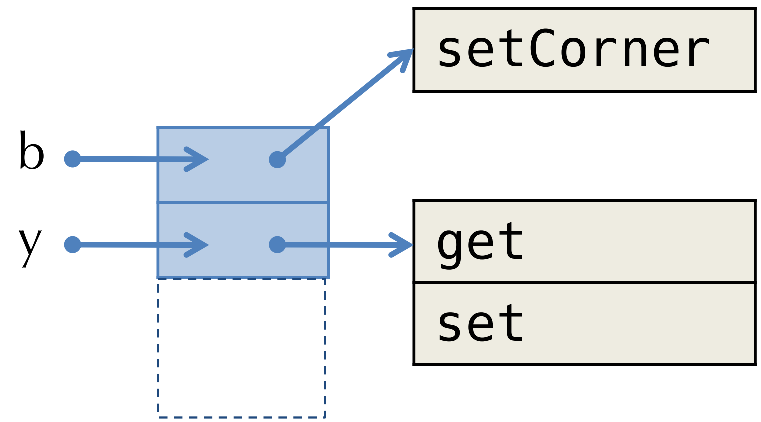
```
Blob b = new Blob( );
Color y = b;     // implicit cast!
```

- Compile

```
Color y = b;
```

As

```
Movq ⟦b⟧ + 8 , y
```



Multiple D.V. Summary

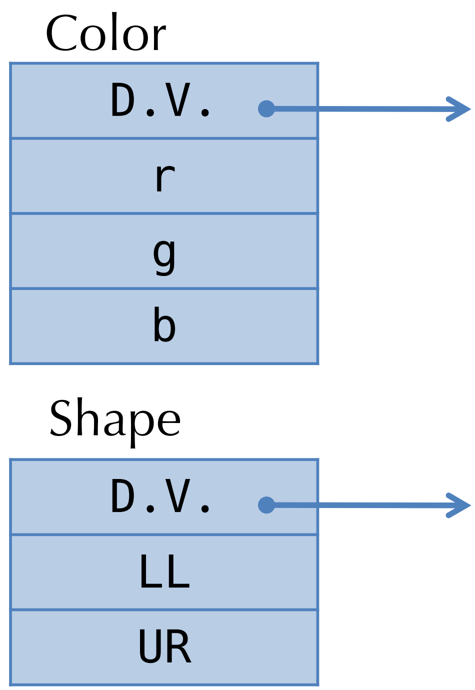
- Benefit: Efficient dispatch, same cost as for multiple inheritance
- Drawbacks:
  - Cast has a runtime cost
  - More complicated programming model… hard to understand/debug?

- What about multiple inheritance and fields?

Multiple Inheritance: Fields

- Multiple supertypes (Java): methods conflict (as we saw)
- Multiple inheritance (C++): fields can also conflict
- Location of the object's fields can no longer be a constant offset from the start of the object.

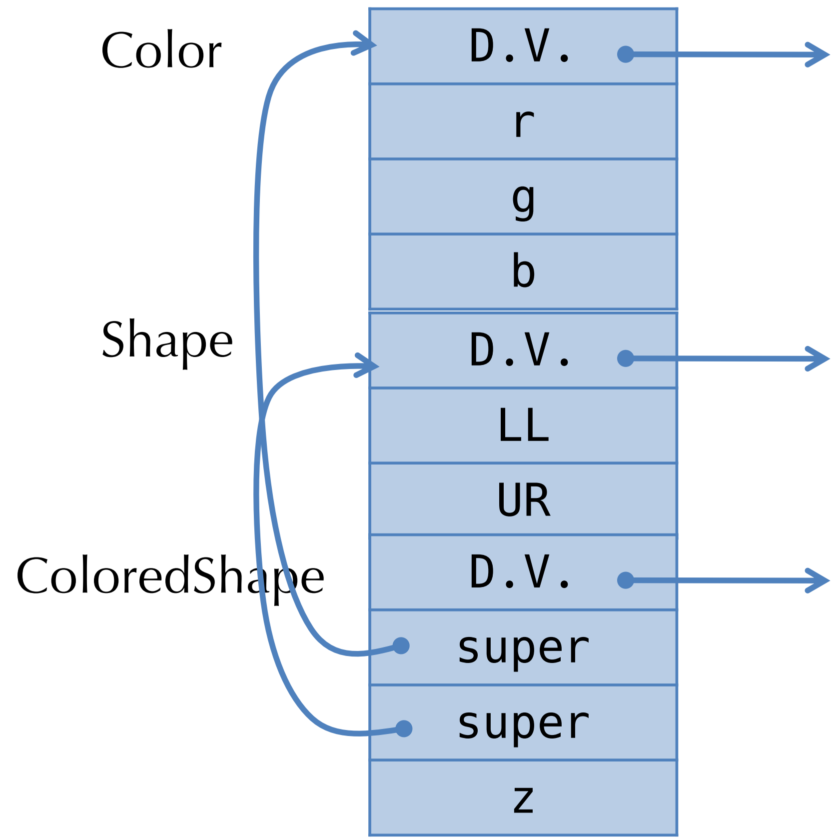
```
class Color {
  float r, g, b; /* offsets: 4,8,12 */
}
class Shape {
  Point LL, UR; /* offsets: 4, 8 */
}
class ColoredShape extends
Color, Shape {
  int z;
}
```



ColoredShape ??

C++ approach:

- Add pointers to the superclass fields
  - Need to have multiple dispatch vectors anyway (to deal with methods)
- Extra indirection needed to access superclass fields
- Used even if there is a single superclass
  - Uniformity



Compiling lambda calculus to straight-line code.

Representing evaluation environments at runtime.

CLOSURE CONVERSION REVISITED

Compiling First-class Functions

- To implement first-class functions on a processor, there are two problems:
  - First: we must implement substitution of free variables
  - Second: we must separate 'code' from 'data'

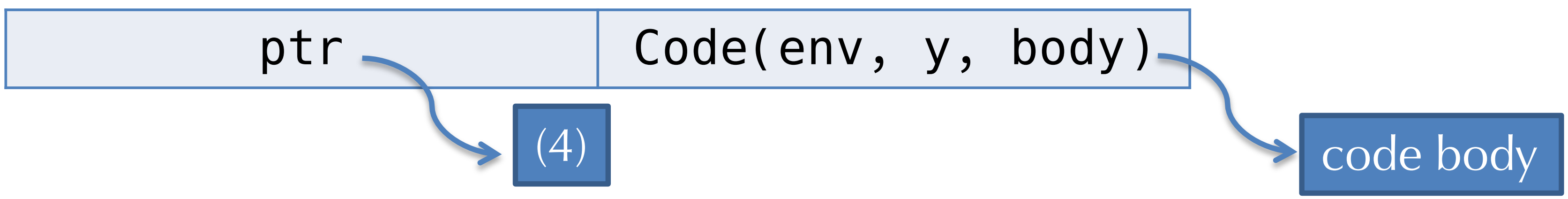
- Reify the substitution:
  - Move substitution from the meta language to the object language by making the data structure & lookup operation explicit
  - The environment-based interpreter is one step in this direction
- Closure Conversion:
  - Eliminates free variables by packaging up the needed environment in the data structure.
- Hoisting:
  - Separates code from data, pulling closed code to the top level.

Example of closure creation

- Recall the "add" function:
  `let add = fun x -> fun y -> x + y`

- Consider the inner function: `fun y -> x + y`

- When run the function application: `add 4`
  the program builds a closure and returns it.
  - The closure is a pair of the environment and a code pointer.



- The code pointer takes a pair of parameters: env and y
  - The function code is (essentially):

```
fun (env, y) -> let x = nth env 0 in x + y
```

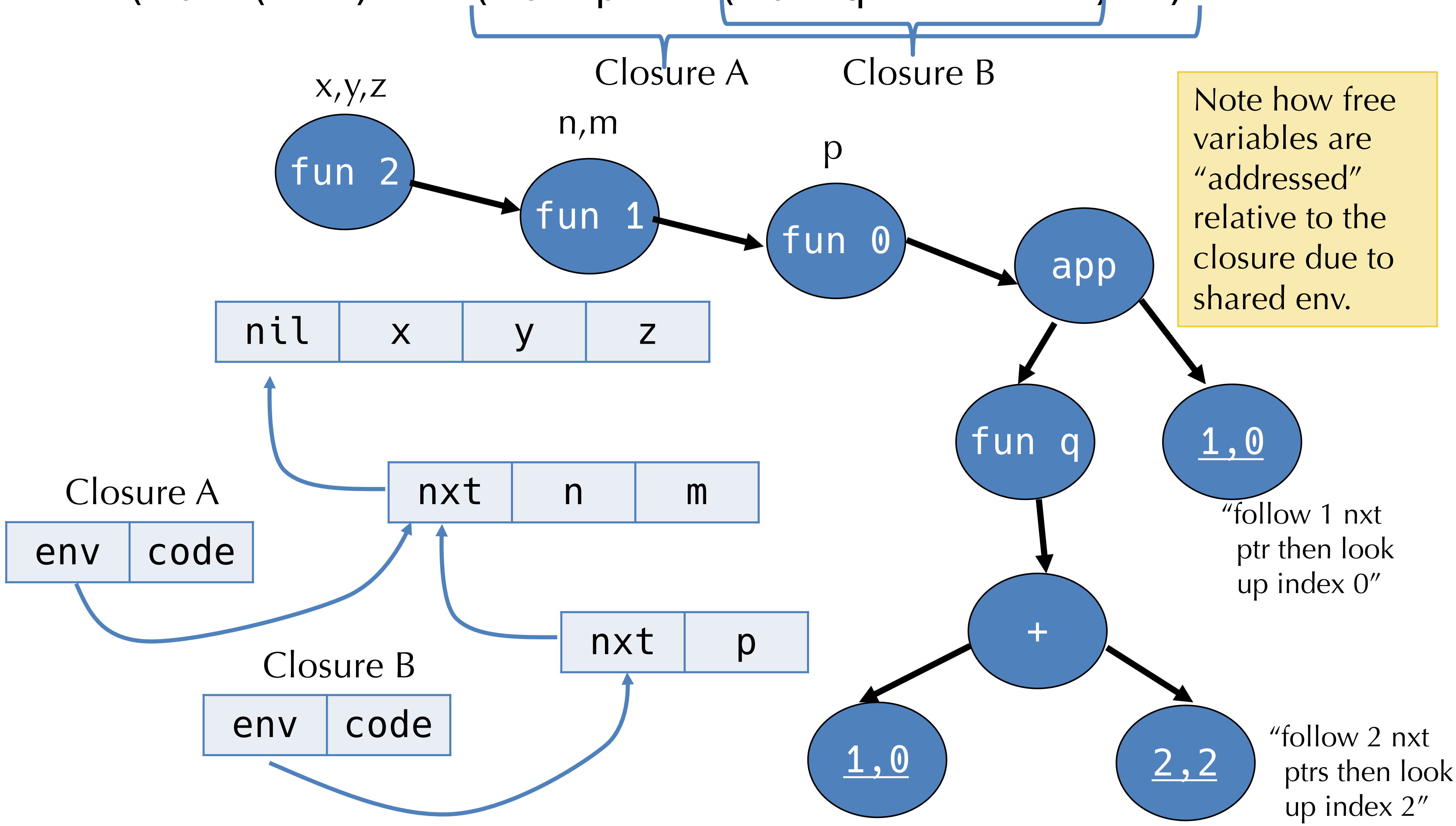
Representing Closures

- As we saw, the simple closure conversion algorithm doesn't generate very efficient code.
  - It stores all the values for variables in the environment, even if they aren't needed by the function body.
  - It copies the environment values each time a nested closure is created.
  - It uses a linked-list datastructure for tuples.

- There are many options:
  - Store only the values for free variables in the body of the closure.
  - Share subcomponents of the environment to avoid copying
  - Use vectors or arrays rather than linked structures

Array-based Closures with N-ary Functions

```
(fun (x y z) ->
   (fun (n m) -> (fun p -> (fun q -> n + z) x)
```



Note how free variables are "addressed" relative to the closure due to shared env.

Observe: Closure ≈ Single-method Object

- Free variables ≈ Fields
- Environment pointer ≈ “this” parameter
- Closure for function: ≈ Instance of this class:

```
fun (x,y) ->
  x + y + a + b
```

```
class C {
  int a, b;
  int apply(x,y) {
    x + y + a + b
  }
}
```