Lecture 19

CIS 341: COMPILERS

Announcements

- HW5: Oat v. 2.0
  - records, function pointers, type checking, array-bounds checks, etc.
  - typechecker & safety
  - Due: Wednesday, April 13th
  - *Please start soon (if you haven't already!)*

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 341;}
}
```

- 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

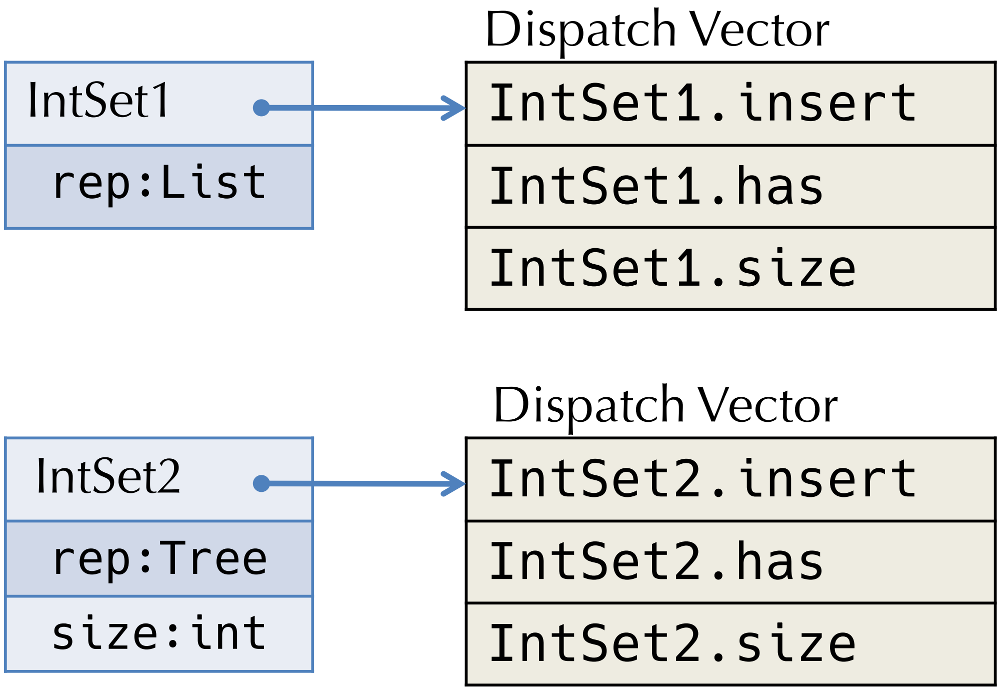
- Consider a client program that uses the IntSet interface:

```
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.



IntSet1

`rep:List`

Dispatch Vector

`IntSet1.insert`

`IntSet1.has`

`IntSet1.size`

IntSet2

`rep:Tree`

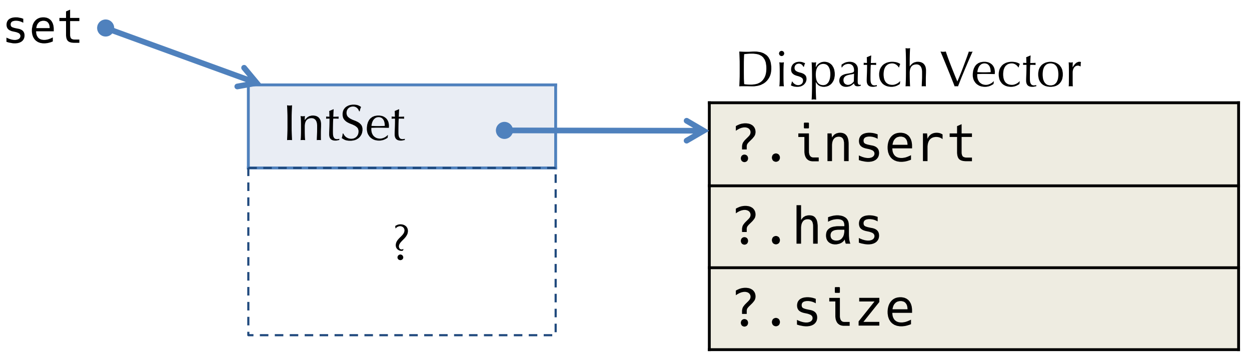
`size:int`

Dispatch Vector

`IntSet2.insert`

`IntSet2.has`

`IntSet2.size`



`set`

IntSet

?

Dispatch Vector

`?.insert`

`?.has`

`?.size`

Method Dispatch (Single Inheritance)

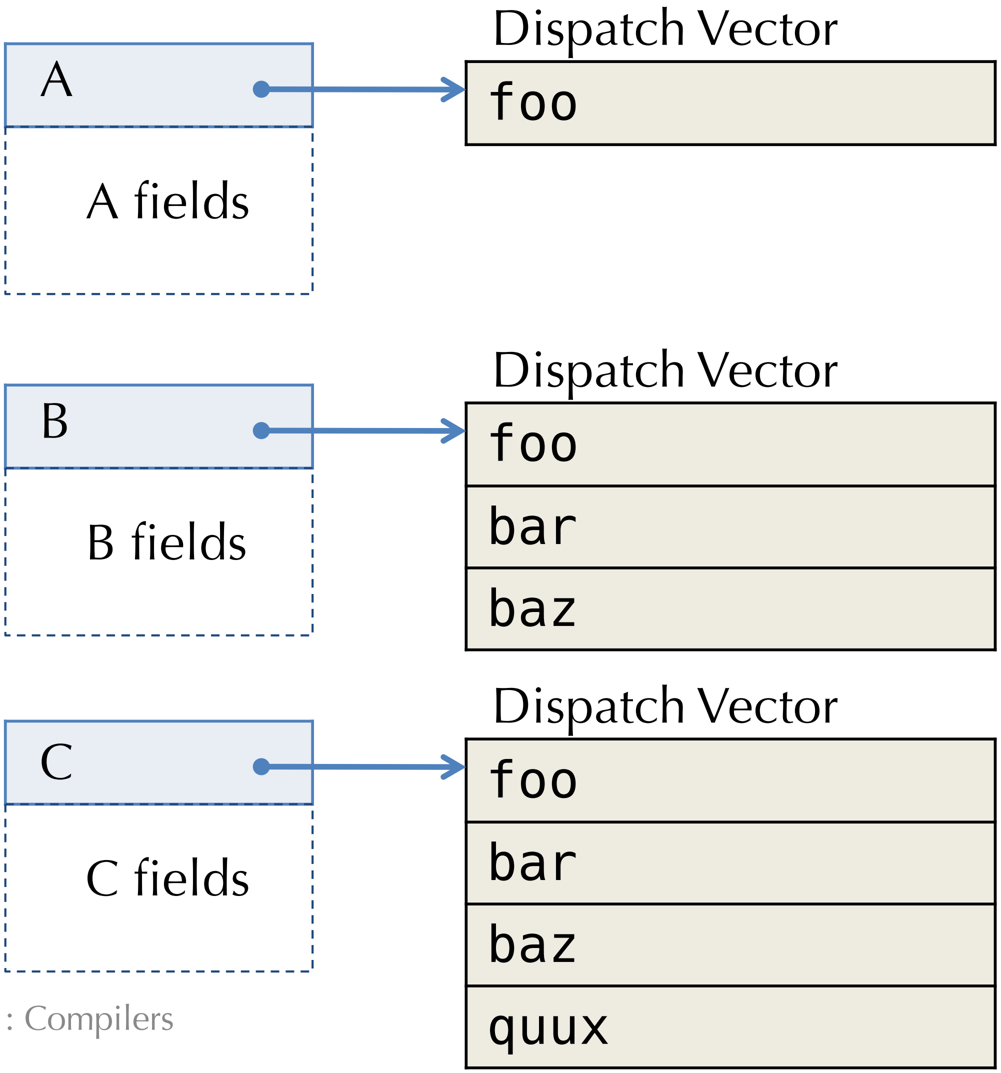
- Idea: every method has its own small integer index.
- Index is used to look up the method in the dispatch vector.

```
                                      Index
interface A {
  void foo();                           0
}

interface B extends A {
  void bar(int x);                      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:

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

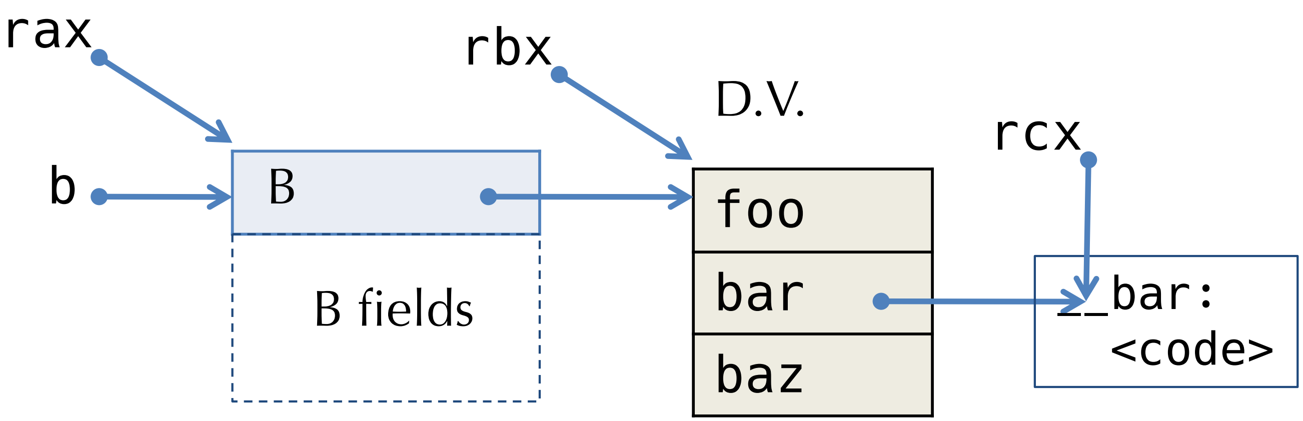
- First, compile $[\![H;G;L \vdash \texttt{e : C}]\!]$
  to get a (pointer to) an object value of class type C
  - Call this value `%obj_ptr`
- Use `getelementptr` to extract the vtable pointer from `%obj_ptr`
- `load` the vtable pointer
- Use `getelementptr` to extract the address of the function pointer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

  $\texttt{call (cmp\_typ t) m(obj\_ptr, } [\![e_1]\!], \ldots, [\![e_n]\!])$

- 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

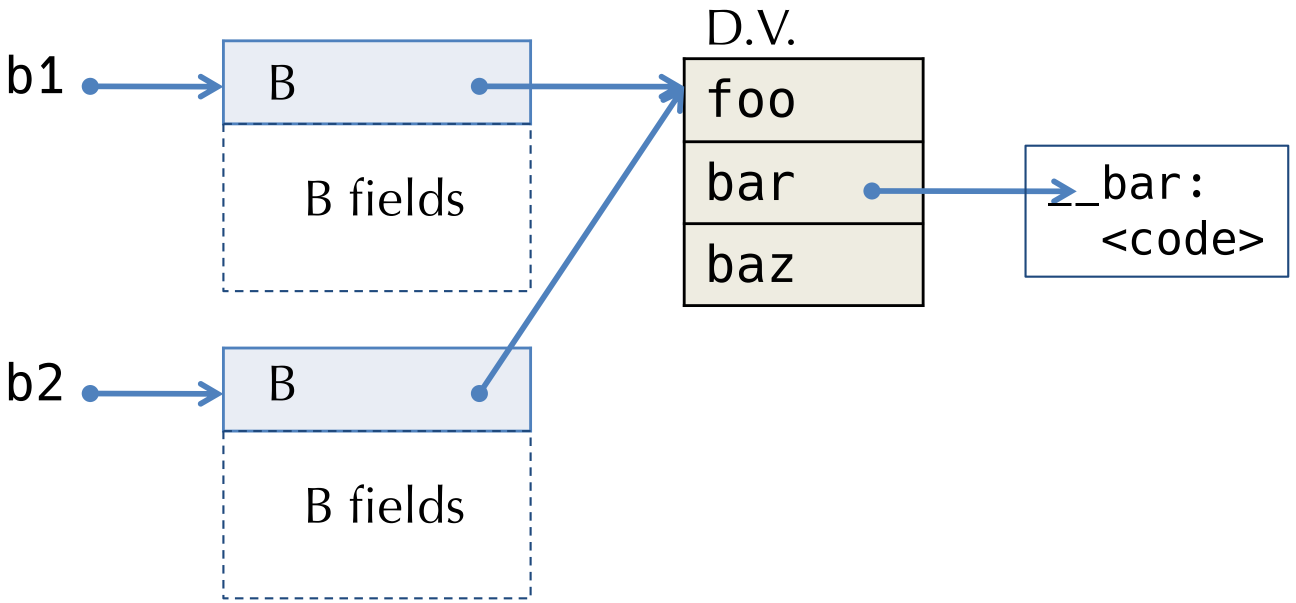
- Suppose `b : B`
- What code for `b.bar(3)`?
  - `bar` has index 1
  - Offset = 8 * 1



```
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.



b1

B

B fields

b2

B

B fields

D.V.

foo

bar

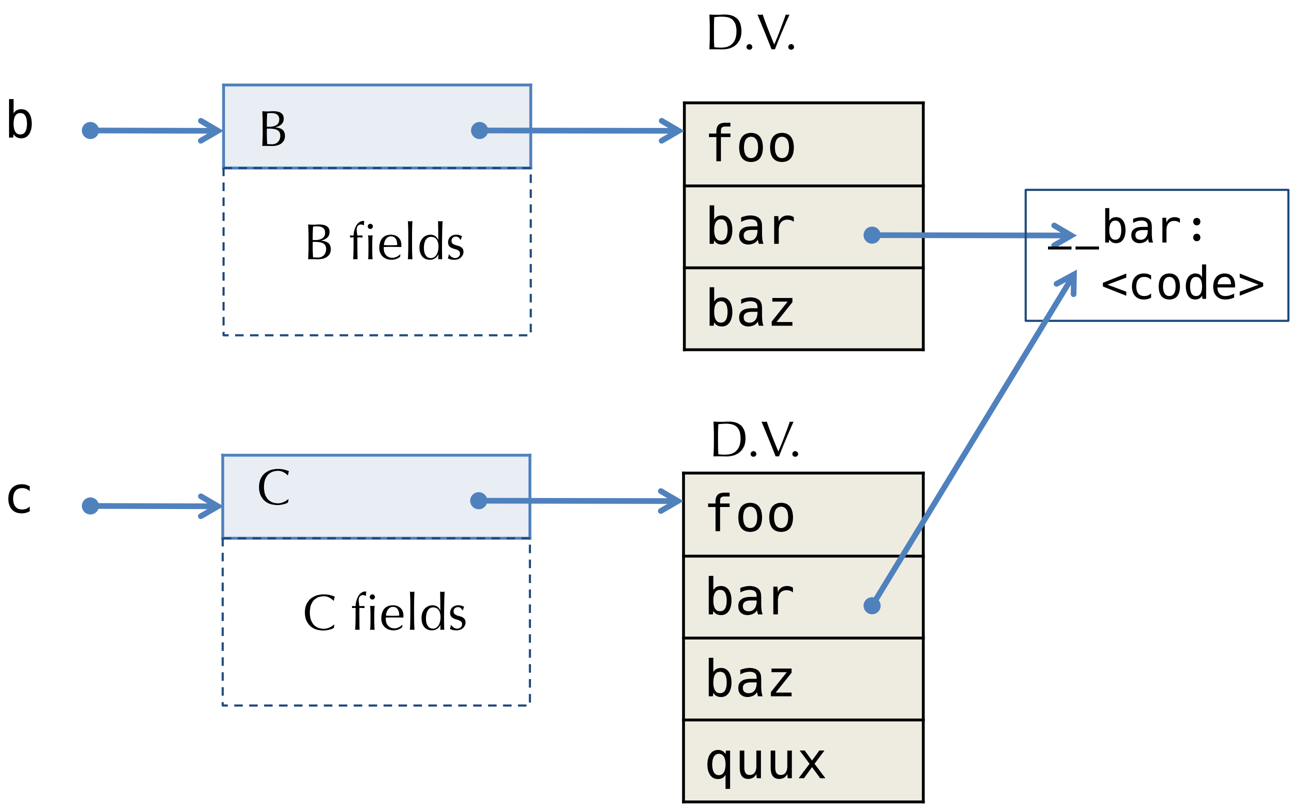
baz

__bar:
<code>

- 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?

Compiling Checked Casts

- How do we compile downcast in general? Consider this generalization of Oat's checked cast:

```
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

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?
}
```

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

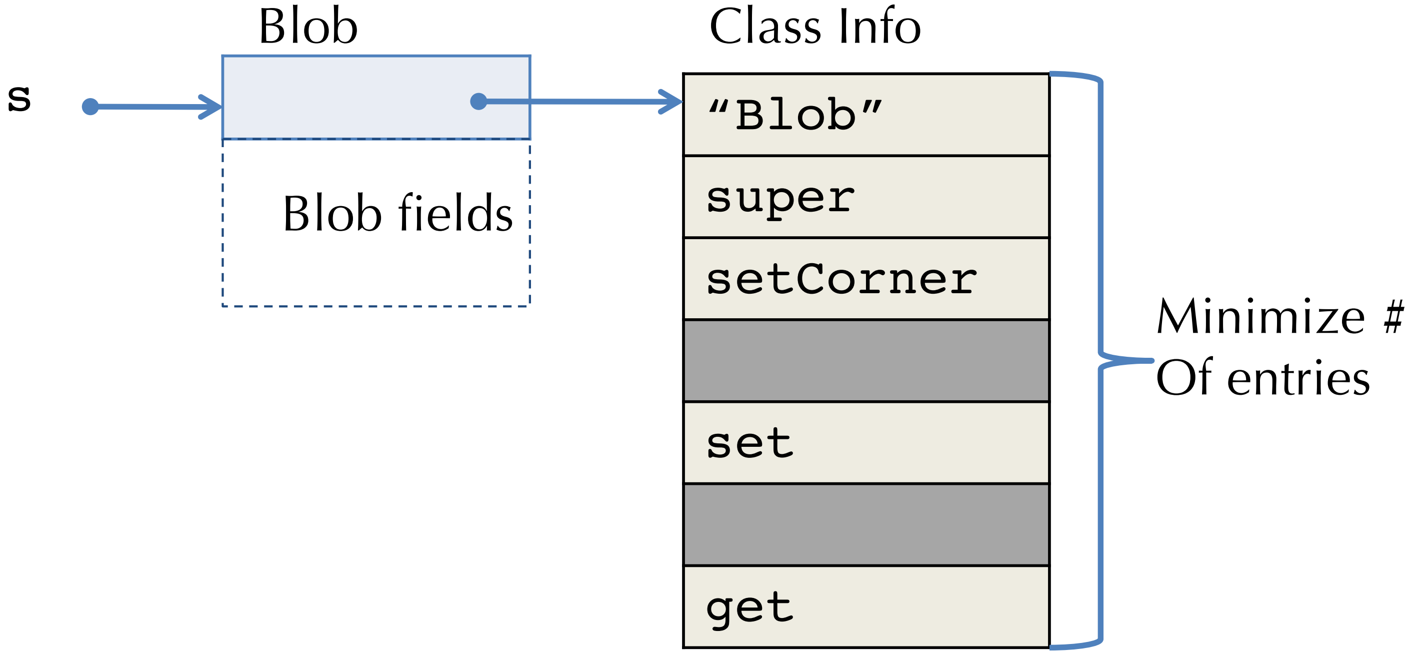
- Give up on separate compilation…
- Now we have access to the whole class hierarchy.

- 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

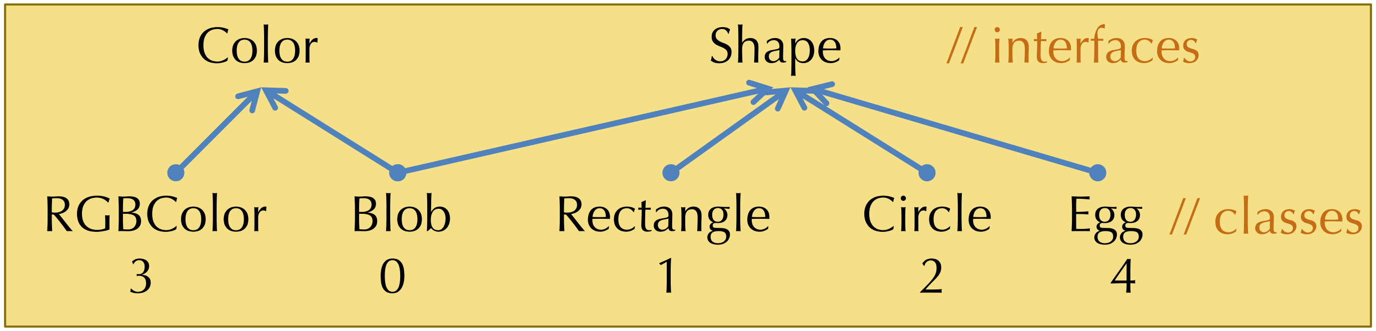


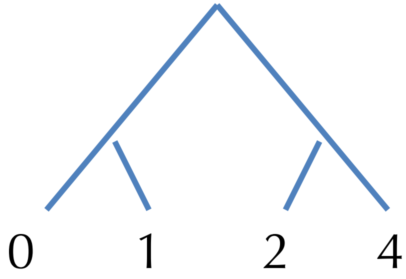
Option 2 variant 2: Binary Search Trees

- 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

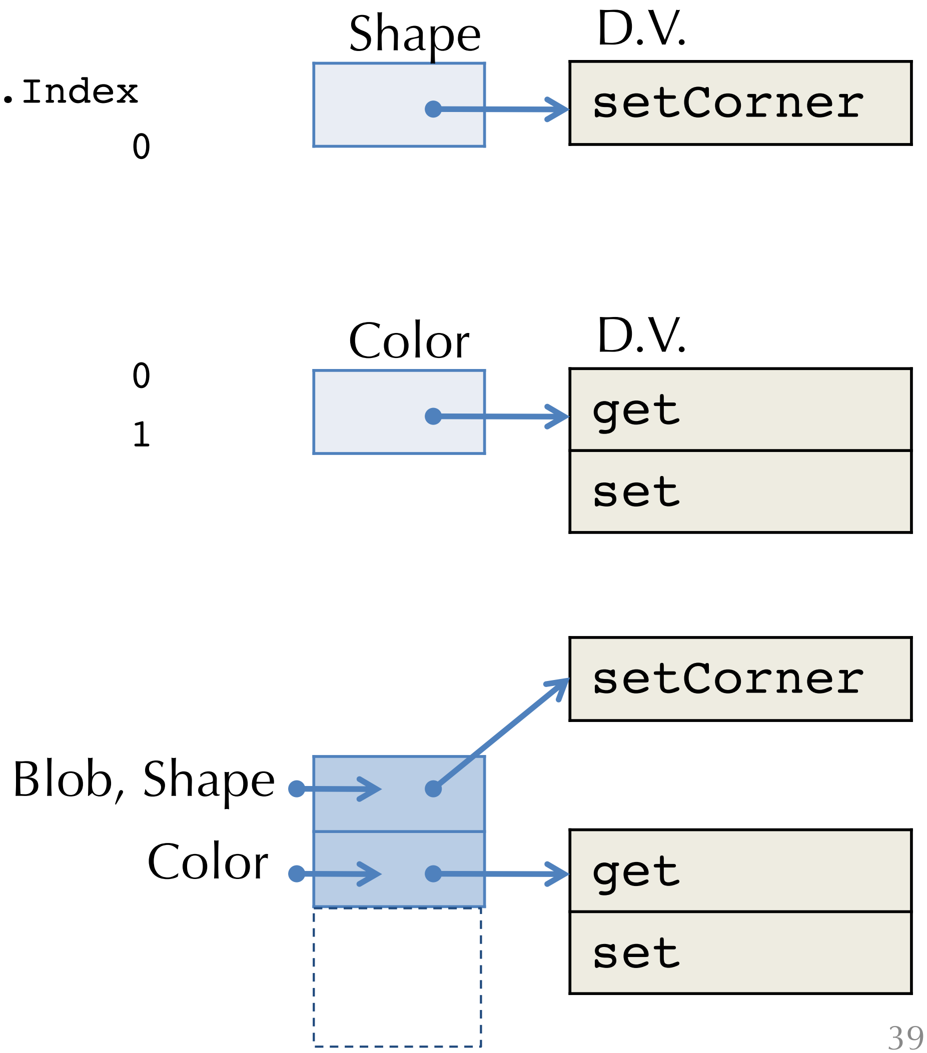
Search Tree Tradeoffs

- 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.

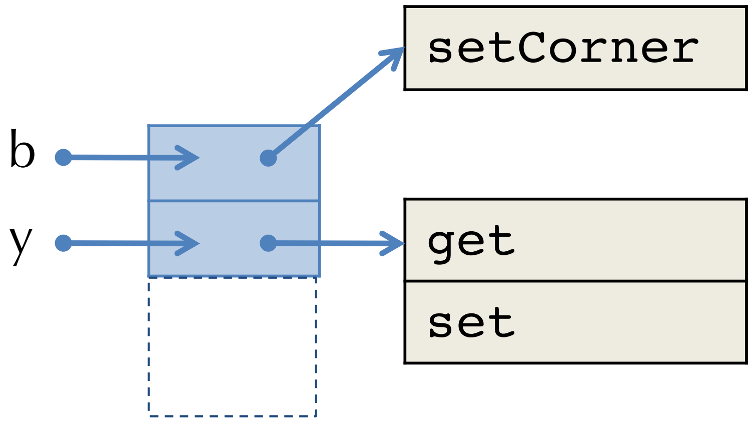
```
Blob b = new Blob();
Color y = b;     // implicit cast!
```

- Compile

```
Color y = b;
```

As

```
Movq ⟦b⟧ + 8 , y
```



Multiple D.V. Summary

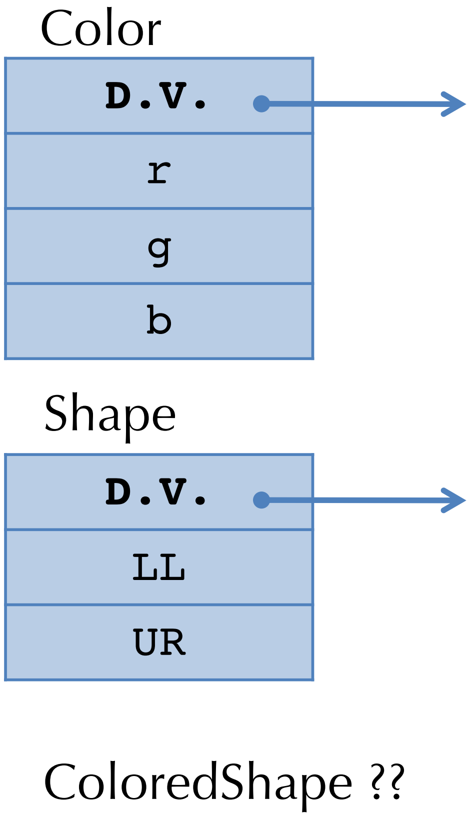
- Benefit: Efficient dispatch, same cost as for multiple inheritance
- Drawbacks:
  - Cast has a runtime cost
  - More complicated programming model… hard to understand/debug?

- 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;
}
```



Color

D.V.

r

g

b

Shape

D.V.

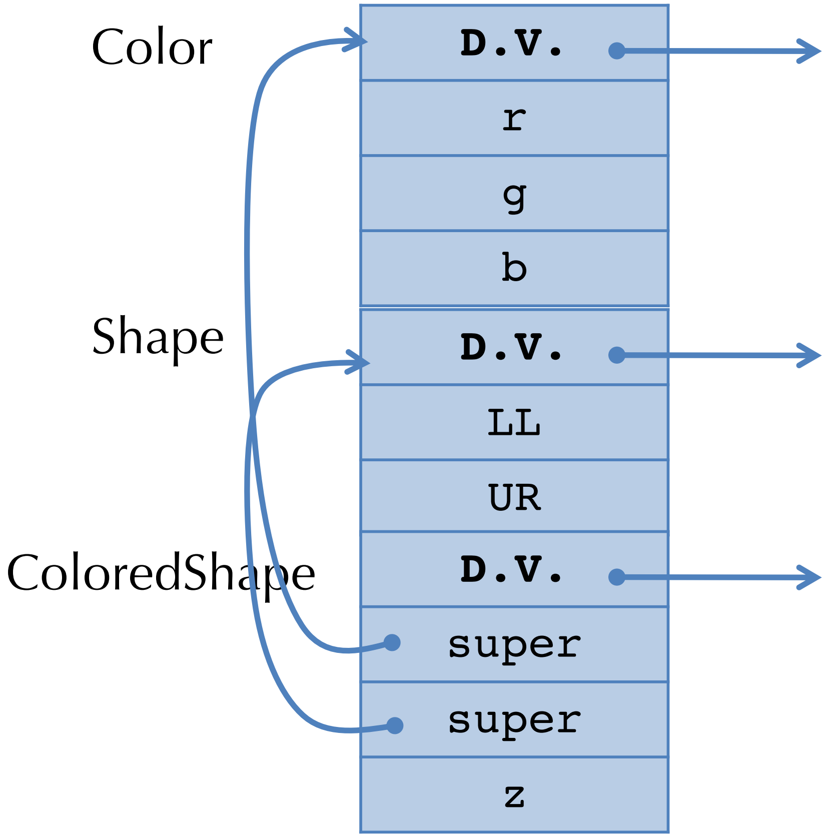
LL

UR

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.

See: fun.ml "closure-based" interpreter
cc.ml

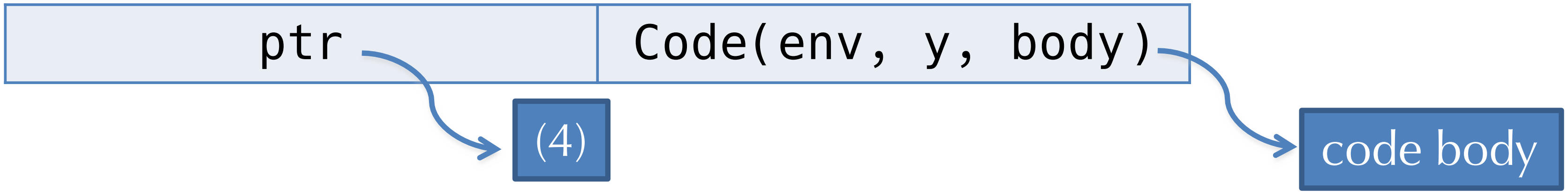
CODE EXAMPLE

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`

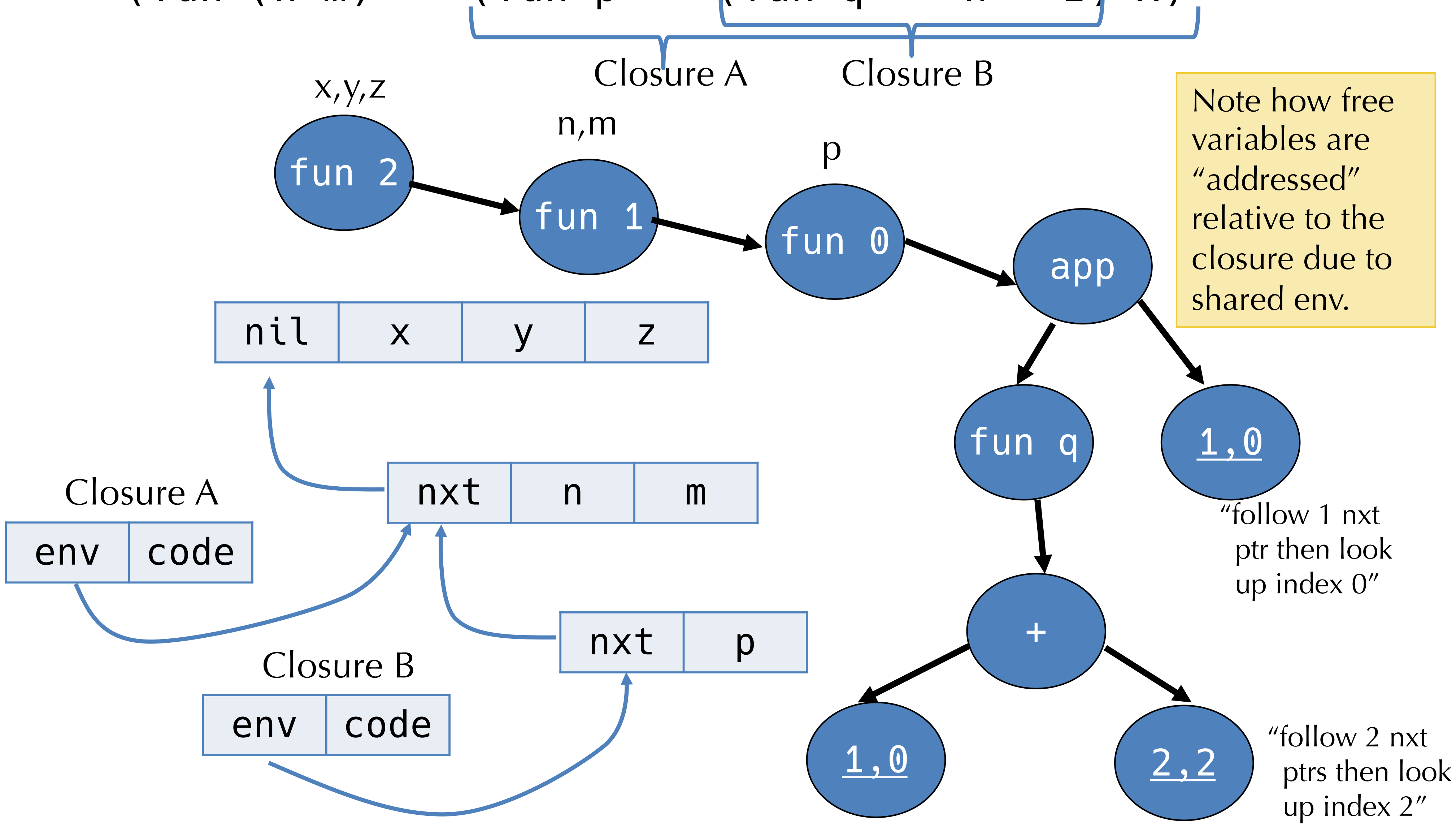
Representing Closures

- 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)
```



nil | x | y | z

nxt | n | m

nxt | p

env | code

env | code

"follow 1 nxt ptr then look up index 0"

"follow 2 nxt ptrs then look up index 2"

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

Compiling Closures to LLVM IR

- The "types" of the environment data structures are generic tuples
  - The tuples contain a mix of int and closure values
  - We know statically what the tuple-type of the environment should be
  - LLVM IR doesn't have generic types

- Type translations:
  - $[\![-]\!]$ for "intepretation" that retains type information
    $[\![\text{int}]\!] = \text{i64}$
    $[\![(t1, \ldots, tn)]\!] = \{[\![t1]\!], \ldots, [\![tn]\!]\}^*$
    $[\![t1 \to t2]\!] = [\![t1 \to t2]\!]_C$
  - $[\![t1 \to t2]\!]_C = \{\text{i8}^*, ((\text{i8}^*, [\![t1]\!]) \to [\![t2]\!])^*\}^*$ "Closure Representation"

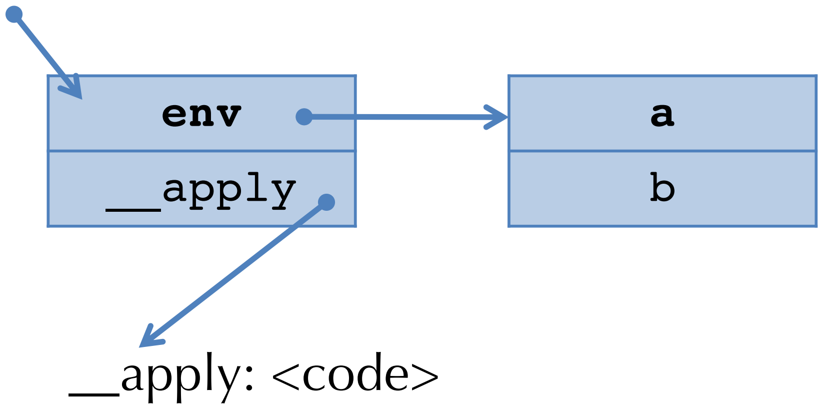
- Rough sketch:
  - Allocation & uses of objects us the "interpretation" translation
  - Anywhere an environment is passed or stored, use i8* and bitcast to/from the translation type.

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
  }
}
```



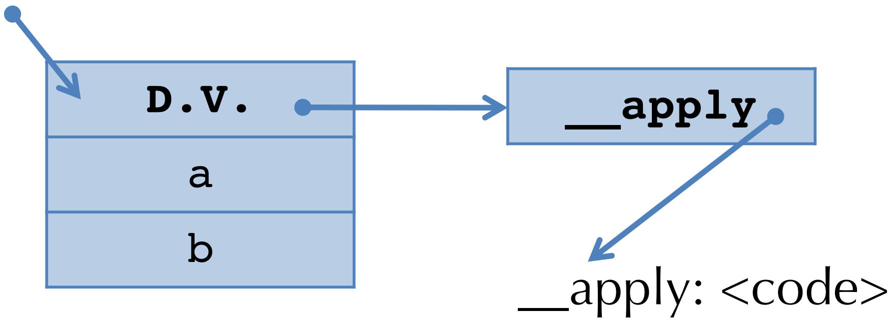
env

__apply

a

b

__apply: <code>



D.V.

a

b

__apply

__apply: <code>