Lecture 19

CIS 4521/5521: COMPILERS

Announcements

- HW5: OAT v. 2.0
  - records, function pointers, type checking, array-bounds checks, etc.
  - Due: Wednesday, April 9th
  - Available now
  - **Start Early!**

See oat.pdf in HW5

OAT'S TYPE SYSTEM

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);
}
```

COMPILING WITH TYPES

Compilation As Translating Judgments

- Consider the source typing judgment for source expressions:

$$C \vdash e : t$$

- How do we interpret this information in the target language?

$$[\![C \vdash e : t]\!] = \ ?$$

- $[\![C]\!]$ translates contexts
- $[\![t]\!]$ is a target type
- $[\![e]\!]$ translates to a (potentially empty) stream of instructions, that, when run, computes the result into some operand

- INVARIANT: if $[\![C \vdash e : t]\!] = ty$, operand , stream
  then the type (at the target level) of the operand is $ty = [\![t]\!]$

Example

- C ⊢ 4521 + 5 : int what is ⟦ C ⊢ 4521 + 5 : int⟧ ?

⟦ ⊢ 4521 : int ⟧ = `(i64, Const 4521, [])` ⟦⊢ 5 : int⟧ = `(i64, Const 5, [])`

⟦C ⊢ 4521 : int⟧ = `(i64, Const 4521, [])` ⟦C ⊢ 5 : int⟧ = `(i64, Const 5, [])`

⟦C ⊢ 4521 + 5 : int⟧ = `(i64, %tmp, [%tmp = add i64 (Const 4521) (Const 5)])`

What about the Context?

- What is ⟦C⟧?
- Source level C has bindings like: x:int, y:bool
  - We think of it as a finite map from identifiers to types

- What is the interpretation of C at the target level?

- ⟦C⟧ maps source identifiers, "x" to target types and ⟦x⟧

- What is the interpretation of a variable ⟦x⟧ at the target level?
  - How are the variables used in the type system?

Interpretation of Contexts

- ⟦C⟧ = a map from source identifiers to types and target identifiers

- INVARIANT:
  x:t ∈ C means that

  (1) lookup ⟦C⟧ x = (⟦t⟧*, `%id_x`)
  (2) the (target) type of `%id_x` is ⟦t⟧* (a pointer to ⟦t⟧)

Interpretation of Variables

$$\left[\!\!\left[\frac{x : t \in L}{H; G; L \vdash_{lhs} x : t; \top} \right]\!\!\right]$$

as addresses
(which can be assigned)

```
= (T, true, %id_x, [])
```

where (T, `%id_x`) = lookup ⟦L⟧ x
and by invariant: T = ⟦t⟧*

$$\left[\!\!\left[\frac{H; G; L \vdash_{lhs} lhs : t; r}{H; G; L \vdash lhs : t} \right]\!\!\right]$$

lhs as expressions
(which are values)

```
= (%tmp, T,
   stream @
   [%tmp = load T* %ptr])
```

where
⟦H;G;L ⊢$_{lhs}$ lhs : t ; T⟧ = (⟦t⟧, true, `ptr`, stream)

Interpretation of Assignment Stmts

$$\left[\!\!\left[\frac{H;G;L \vdash_{lhs} lhs : t; \top \quad H;G;L \vdash exp : t' \quad H \vdash t' \leq t}{H;G;L;rt \vdash lhs = exp; \Rightarrow L; \bot} \right]\!\!\right]$$

```
= (⟦H;G;L⟧,
   ptr_code @
   exp_code @
   [store T %e_op, %ptr])
```

assignment to a lhs

where

⟦H;G;L ⊢$_{lhs}$ lhs : t ; T⟧ = (⟦t⟧, `true, ptr, ptr_code`)

and

⟦H;G;L ⊢$_{lhs}$ exp : t'⟧ = (⟦t'⟧, `%e_op, exp_code`)

Other Judgments?

- Statement:
  ⟦H;G;L; rt ⊢ stmt ⇒ C′⟧ = ⟦C′⟧ , stream

- Declaration:
  ⟦H;G;L ⊢ var x = exp ⇒ G;L,x:t ⟧ = ⟦G;L,x:t⟧, stream

  INVARIANT: stream is of the form:

```
stream' @
[E %id_x = alloca ⟦t⟧;
 I store ⟦t⟧ opn, ⟦t⟧* %id_x ]
```

  and ⟦H;G;L ⊢ exp : t ⟧ = (⟦t⟧, `opn`, stream′)

- Rest follow similarly

COMPILING CONTROL

Translating while

- Consider translating "`while(e) s`":
  - Test the conditional, if true jump to the body, else jump to the label after the body.

⟦`C;rt ⊢ while(e) s ⇒ C'`⟧ = ⟦`C'`⟧,

```
lpre:
    opn = ⟦C ⊢ e : bool⟧
    %test = icmp eq i1 opn, 0
    br %test, label %lpost, label %lbody
lbody:
     ⟦C;rt ⊢ s ⇒ C'⟧
     br %lpre
lpost:
```

- Note: writing `opn =` ⟦`C ⊢ e : bool`⟧ is pun
  - translating ⟦C ⊢ e : bool⟧ generates *code* that puts the result into `opn`
  - In this notation there is implicit collection of the code

Translating if-then-else

- Similar to while except that code is slightly more complicated because if-then-else must reach a merge and the else branch is optional.

$[\![C;rt \vdash \texttt{if (}e_1\texttt{) } s_1 \texttt{ else } s_2 \Rightarrow C']\!] = [\![C']\!]$

```
    opn = ⟦C ⊢ e : bool⟧
    %test = icmp eq i1 opn, 0
    br %test, label %else, label %then
then:
     ⟦C;rt ⊢ s1 ⇒ C'⟧
     br %merge
else:
    ⟦C; rt s2 ⇒ C'⟧
     br %merge
merge:
```

Connecting this to Code

- Instruction streams:
  - Must include labels, terminators, and “hoisted” global constants
- Must post-process the stream into a control-flow-graph
- See frontend.ml from HW4

OPTIMIZING CONTROL

Standard Evaluation

- Consider compiling the following program fragment:

```
if (x & !y | !w)
  z = 3;
else
  z = 4;
return z;
```

```
    %tmp1 = icmp Eq ⟦y⟧, 0        ; !y
    %tmp2 = and ⟦x⟧ ⟦tmp1⟧
    %tmp3 = icmp Eq ⟦w⟧, 0
    %tmp4 = or %tmp2, %tmp3
    %tmp5 = icmp Eq %tmp4, 0
    br %tmp4, label %else, label %then

then:
    store ⟦z⟧, 3
    br %merge

else:
    store ⟦z⟧, 4
    br %merge

merge:
    %tmp5 = load ⟦z⟧
    ret %tmp5
```

Observation

- Usually, we want the translation ⟦e⟧ to produce a value
  - ⟦C ⊢ e : t⟧ = (ty, operand, stream)
  - e.g. ⟦C ⊢ e_1 + e_2 : int⟧ = (i64, `%tmp`, [`%tmp = add i64` ⟦e_1⟧ ⟦e_2⟧])

- But when the boolean expression we're compiling appears in a test, the program jumps to one label or another after the comparison but otherwise never uses the value.

- In many cases, we can avoid "materializing" the value (i.e., storing it in a temporary) and thus produce better code.
  - This idea also lets us implement different functionality too: e.g. short-circuiting Boolean expressions

- Make up new "judgement" that is similar to ⟦C ⊢ e : bool⟧ but has a different semantics. Call it ⟦C ⊢ e : bool@⟧

Idea: Use a different translation for tests

Usual Expression translation:

$[\![C \vdash e : t]\!] = (ty, operand, stream)$

Conditional branch translation of booleans, without materializing the value:

$[\![C \vdash e : bool@]\!]\ ltrue\ lfalse = stream$

$[\![C, rt \vdash \texttt{if (e) then s1 else s2} \Rightarrow C']\!] = [\![C']\!],$

```
    insns3
then:
    ⟦s1⟧
    br %merge
else:
    ⟦s2⟧
    br %merge
merge:
```

where

$[\![C, rt \vdash s_1 \Rightarrow C']\!] = [\![C']\!], insns_1$

$[\![C, rt \vdash s_2 \Rightarrow C'']\!] = [\![C'']\!], insns_2$

$[\![C \vdash e : bool@]\!]$ `then else` $= insns_3$

Notes:

- takes two extra arguments: a "true" branch label and a "false" branch label.
- Doesn't "return a value"
- Aside: this is a form of continuation-passing translation…

Short Circuit Compilation: Expressions

- ⟦C ⊢ e : bool@⟧ ltrue lfalse = insns

$$\frac{}{⟦C ⊢ \text{false} : \text{bool@}⟧\ l_{true}\ l_{false} = \texttt{[br \%lfalse]}}\ \text{FALSE}$$

$$\frac{}{⟦C ⊢ \text{true} : \text{bool@}⟧\ l_{true}\ l_{false} = \texttt{[br \%ltrue]}}\ \text{TRUE}$$

$$\frac{⟦C ⊢ e : \text{bool@}⟧\ l_{false}\ l_{true} = insns}{⟦C ⊢ !e : \text{bool@}⟧\ l_{true}\ l_{false} = insns}\ \text{NOT}$$

Short Circuit Evaluation

Idea: build the logic into the translation

$$\frac{[\![C \vdash e1 : bool@]\!]\ ltrue\ right = insns_1 \quad [\![C \vdash e2 : bool@]\!]\ ltrue\ lfalse = insns_2}{[\![C \vdash e1 \,|\, e2 : bool@]\!]\ ltrue\ lfalse =}$$

```
    insns1
right:
    insn2
```

$$\frac{[\![C \vdash e1 : bool@]\!]\ right\ lfalse = insns_1 \quad [\![C \vdash e2 : bool@]\!]\ ltrue\ lfalse = insns_2}{[\![C \vdash e1\&e2 : bool@]\!]\ ltrue\ lfalse =}$$

```
    insns1
right:
    insn2
```

where `right` is a fresh label

Short-Circuit Evaluation

- Consider compiling the following program fragment:

```
if (x & !y | !w)
  z = 3;
else
  z = 4;
return z;
```

```
    %tmp1 = icmp Eq ⟦x⟧, 0
    br %tmp1, label %right2, label %right1

right1:
    %tmp2 = icmp Eq ⟦y⟧, 0
    br %tmp2, label %then, label %right2

right2:
    %tmp3 = icmp Eq ⟦w⟧, 0
    br %tmp3, label %then, label %else

then:
    store ⟦z⟧, 3
    br %merge

else:
    store ⟦z⟧, 4
    br %merge

merge:
    %tmp5 = load ⟦z⟧
    ret %tmp5
```

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 that for 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:

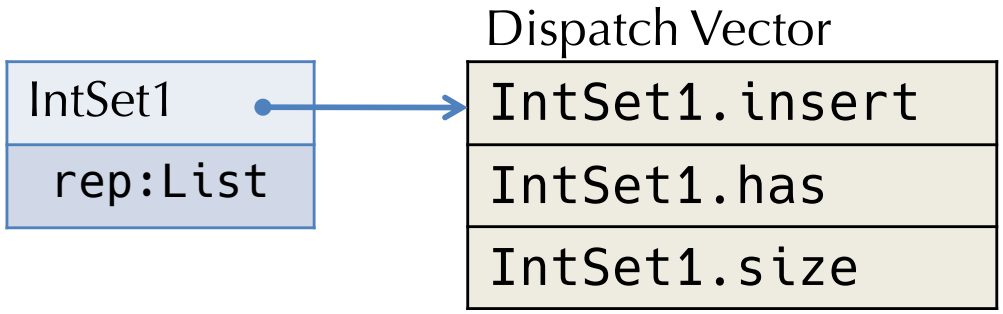
```
void someMethod(IntSet set) {
   int x = set.size();  // What code gets run?
    …
```

- Which code to call?
  - `IntSet1.size` ?
  - `IntSet2.size` ?
  - some other implementation's size method?

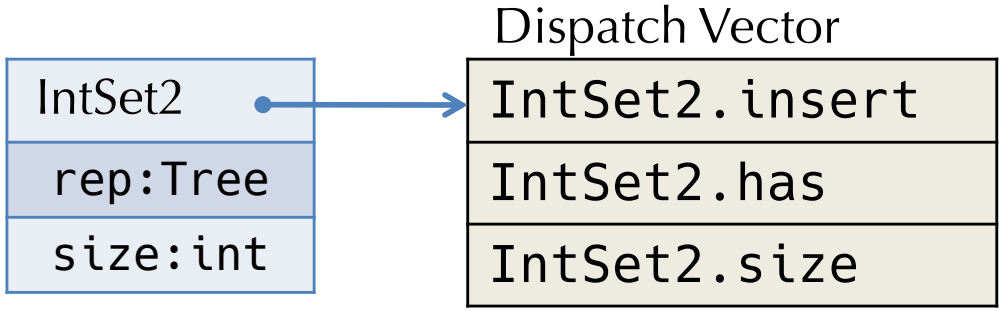
- Client code doesn't know the answer.
  - So, objects must "know" which code to call.
  - Invocation of a method must indirect through the object.

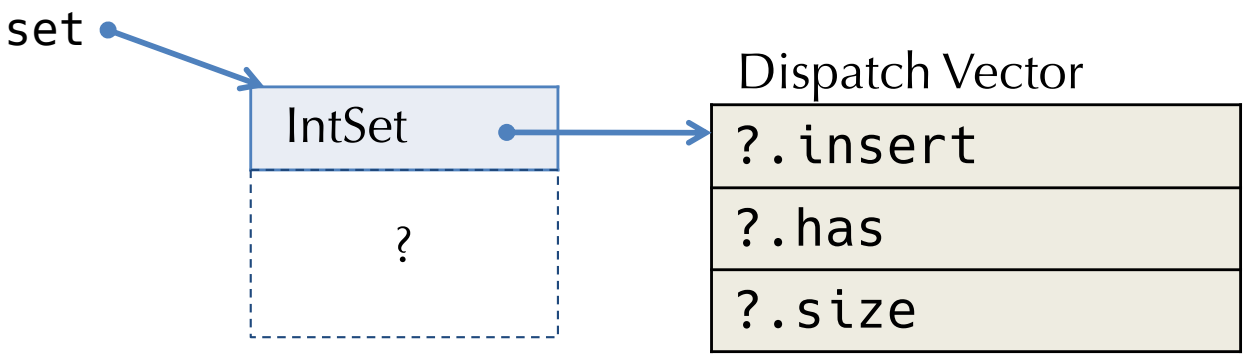
Compiling Objects

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

1
2

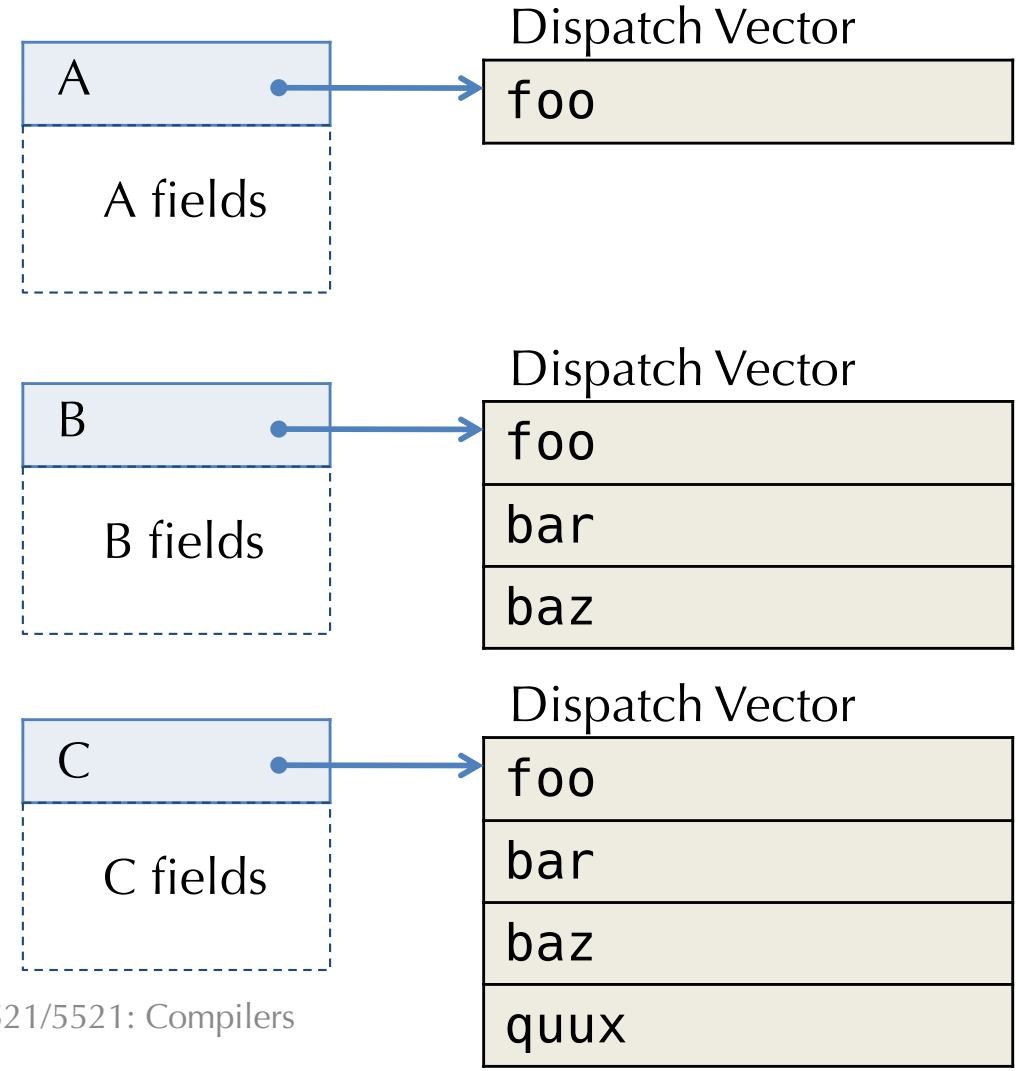
Inheritance / Subtyping:
C <: B <: A

```
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 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

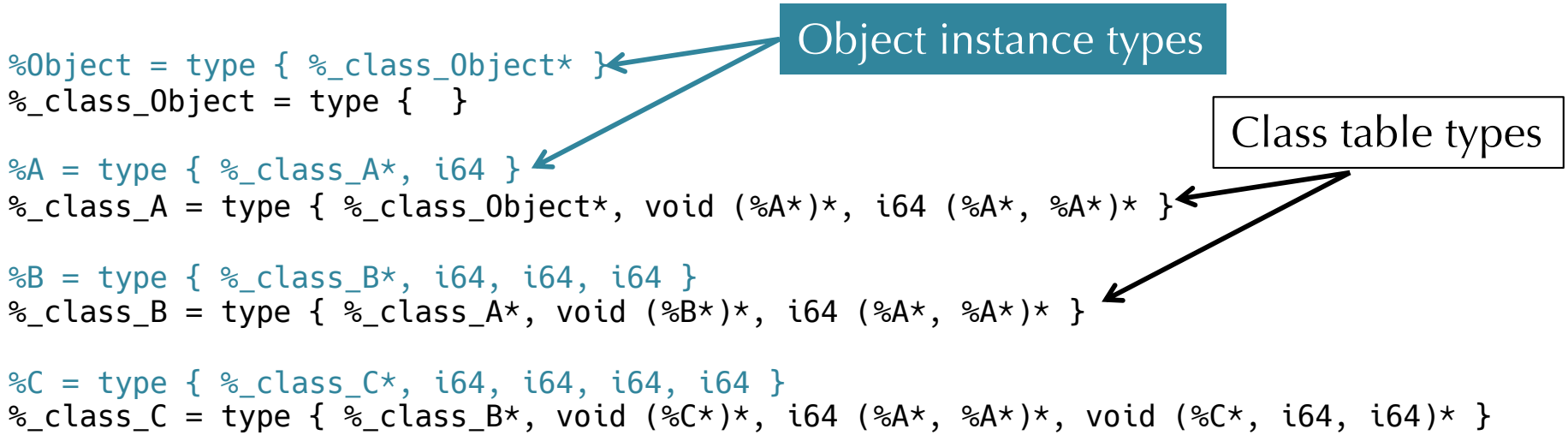
- Each class gives rise to two implementation types:

- 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 = 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`

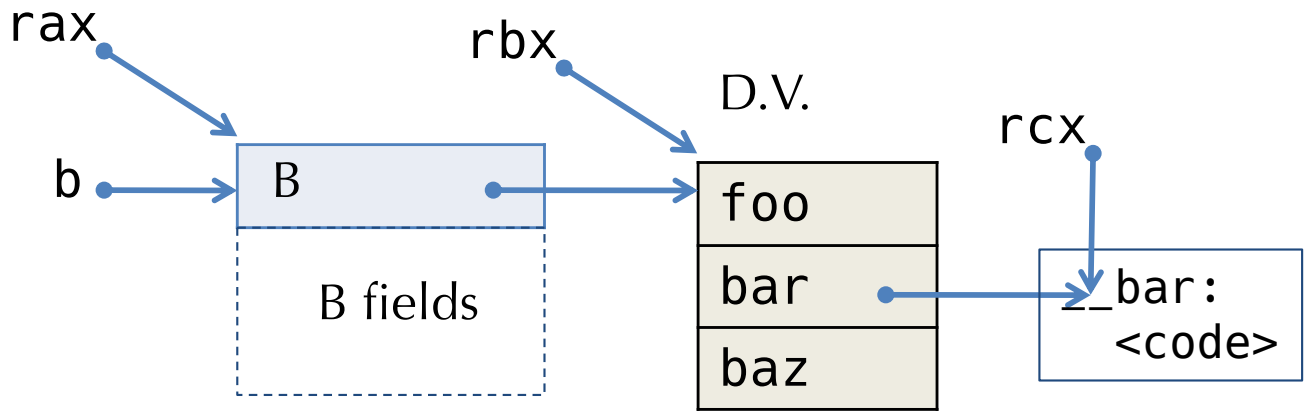
LLVM Method Invocation Compilation

- 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:
  `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

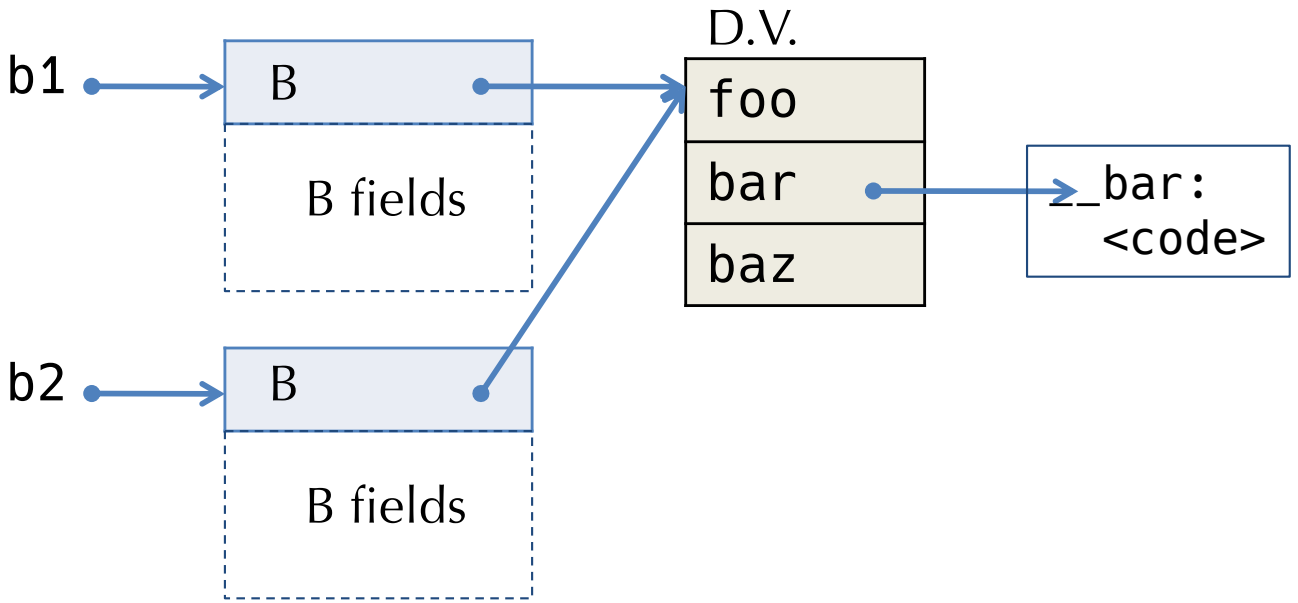
- Suppose `b : B`
- What code for `b.bar(3)`?
  - `bar` has index 1
  - Offset = 8 * 1




```
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

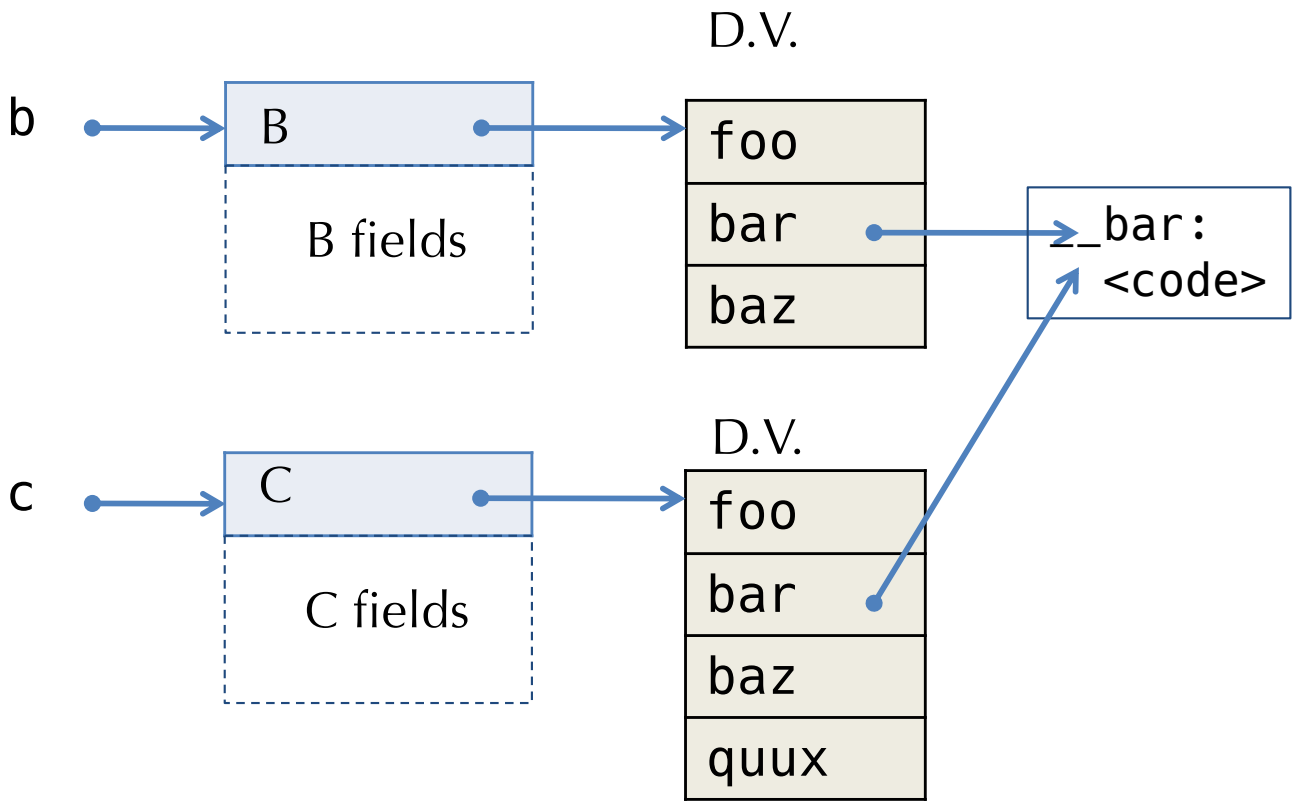
- All instances of a class may share the same dispatch vector.
  - Assuming that methods are immutable.
- Code pointers stored in the dispatch vector are available at link time – dispatch vectors can be built once at link time.



- 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 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