Lecture 17

COMPILER DESIGN
• **HW4**: due soon

• **HW5**: OAT v. 2.0
  – Records, function pointers, type checking, array-bounds checks, etc.
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();
}

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

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

Index

<table>
<thead>
<tr>
<th>Method</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>0</td>
</tr>
<tr>
<td>bar(int x)</td>
<td>1</td>
</tr>
<tr>
<td>baz</td>
<td>2</td>
</tr>
<tr>
<td>quux</td>
<td>3</td>
</tr>
</tbody>
</table>

Inheritance / Subtyping

```
C <: B <: A
```
• Each interface and class gives rise to a dispatch vector layout
• Note that inherited methods have identical dispatch indices in the subclass (Width subtyping)
MULTIPLE INHERITANCE
Multiple Inheritance

- C++: a class may declare more than one superclass

- Semantic problem: *ambiguity*

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

- Same problem can happen with fields

- In C++, fields/methods can be duplicated when such ambiguities arise
  - Explicit sharing can also be declared
Multiple Inheritance

• Java: a class may implement more than one interface

• No semantic ambiguity when two interfaces declare the same method
  – The class will implement a single method

```java
interface A { int m(); }
interface B { int m(); }
class C implements A,B { int m() {...} } // only one m
```
interface Shape {
    void setCorner(int w, Point p);
}

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

class Blob implements Shape, Color {
    void setCorner(int w, Point p) {...}
    float get(int rgb) {...}
    void set(int rgb, float value) {...}
}
**General Approaches**

**Problem:** Cannot directly identify methods by position anymore

- **Option 1:** Allow multiple D.V. tables (C++)
  - Choose which D.V. to use based on static type
  - Casting from/to a class may require runtime operations

- **Option 2:** Use a level of indirection
  - Map method identifiers to code pointers (e.g. index by method name)
  - Use a hash table
  - May need to search up the class hierarchy

- **Option 3:** Give up separate compilation
  - Use “sparse” dispatch vectors, or binary decision trees
  - Must know the entire class hierarchy

**Note:** many variations on these themes
  - Different Java compilers pick different approaches to options 2 & 3
Option 1: Multiple Dispatch Vectors

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

```java
interface Shape {
    void setCorner(int w, Point p);
}

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

class Blob implements Shape, Color {
    void setCorner(int w, Point p) {...}
    float get(int rgb) {...}
    void set(int rgb, float value) {...}
}
```
Multiple Dispatch Vectors

• An object may have **multiple “entry points”**
  – Each entry point corresponds to a dispatch vector
  – Which one to use depends on the object’s static type

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

• Compile

```
Color y = b;

↓
Movq [b] + 8 , y
```
Multiple D.V. Summary

• **Pros**
  – Efficient dispatch, similar cost as for single inheritance

• **Cons**
  – 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
  - Fields can no longer be constant offsets from the start of the object

```java
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;
}
```
class A {
    public:
        int x;
        virtual void f();
};

class B {
    public:
        int y;
        virtual void g();
        virtual void f();
};

class C: public A, public B {
    public:
        int z;
        virtual void f();
};

C *pc = new C;
B *pb = pc;
A *pa = pc;

Three pointers to the same object, but different static types
Call to `pc->g` can proceed through **C-as-B vtbl**

- Offset `d` in vtbl is used in call to `pb->f`, why?
  - Since `C::f` may refer to `A` data that is above the pointer `pb`
Multiple Inheritance “Diamond”

- Is interface or implementation inherited twice?
- What if definitions conflict?
**Diamond Inheritance in C++**

- **Standard base classes**
  - D members appear twice in C

- **Virtual base classes**
  - class A : public virtual D { ... }
  - Avoid duplication of base class members
  - Require additional pointers so that D part of A, B parts of object can be shared

Multiple inheritance is complicated in C++ due to its desire for efficient lookup.
interface Shape {
    void setCorner(int w, Point p);
}

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

class Blob implements Shape, Color {
    void setCorner(int w, Point p) {...}  
    float get(int rgb) {...}  
    void set(int rgb, float value) {...}
}
Option 2: Search + Inline Cache

- For each class/interface, keep a table: \textit{method names} $\rightarrow$ \textit{method code}
  - Recursively walk up the hierarchy looking for the method name
- Note
  - Identifiers in quotes are not strings
  - In practice, they are some kind of unique identifiers
interface Incrementable {
    public void inc();
}
class IntCounter implements Incrementable {
    public void add(int);
    public void inc();
    public int value();
}
class FloatCounter implements Incrementable {
    public void inc();
    public void add(float);
    public float value();
}

void add2(Incrementable x) { x.inc(); x.inc(); }
Inline Cache Code

- **Optimization**
  - At call site, store class and code pointer in a cache
  - On method call, check whether class matches cached value

- **Compiling:**
  ```
  Shape s = new Blob();  s.get();
  Call site 434
  ```

- Compiler knows that `s` is a `Shape`
  - Suppose `%rax` holds object pointer

- Cached interface dispatch
  ```
  // set up parameters
  movq [%rax], tmp
  cmpq tmp, [cacheClass434]
  Jnz __miss434
  callq [cacheCode434]
  __miss434:
  // do the slow search
  ```
Option 2 variant 2: Hash Table

- Idea: don’t try to give all methods unique indices
  - Resolve conflicts by checking that the entry is correct at dispatch
- Use hashing to generate indices
  - Range of the hash values should be relatively small
  - Hash indices can be pre computed, but passed as an extra parameter

```java
interface Shape {
    void setCorner(int w, Point p);  hash("setCorner") = 11
}

interface Color {
    float get(int rgb);  hash("get") = 4
    void set(int rgb, float value);  hash("set") = 7
}

class Blob implements Shape, Color {
    void setCorner(int w, Point p) {...}  11
    float get(int rgb) {...}             4
    void set(int rgb, float value) {...} 7
}
```
Dispatch with Hash Tables

- What if there is a conflict?
  - Entries containing several methods point to code that resolves conflict (e.g. by searching through a table based on class name)

- Advantage
  - Simple, basic code dispatch is (almost) identical
  - Reasonably efficient

- Disadvantage
  - Wasted space in DV
  - Extra argument needed for resolution
  - Slower dispatch if conflict
Option 3 variant 1: Sparse D.V. Tables

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

• So, ensure no 2 methods in 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 can be used to construct D.V. layouts reasonably efficiently (to minimize size)
  – Finding an optimal solution is NP complete
Example Object Layout

- **Advantage**: Identical dispatch & performance to single-inheritance case
- **Disadvantage**: Must know entire class hierarchy
Option 3 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(...);

```asm
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
```

---

Zhendong Su  Compiler Design
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 branch stall of ~10 cycles (on X86)

• Profiling helps 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% 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)
Observe: Closure $\approx$ Single-method Object

- Free variables
- Environment pointer
- Closure for function:
  \[
  \text{fun } (x, y) \rightarrow x + y + a + b
  \]

$\approx$ Fields

$\approx$ "this" parameter

$\approx$ Instance of this class:

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

Zhendong Su    Compiler Design
CLASSES & OBJECTS IN LLVM
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
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
}
Example OO Hierarchy in LLVM

Object instance 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)* }  

Class table types

_class_Object* @_vtbl_Object = global { }  

_class_A* @_vtbl_A = global { _class_Object* @_vtbl_Object,  
void (%A*)* @print_A,  
i64 (%A*, %A*)* @blah_A }  

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

_class_B* @_vtbl_C = global { _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

```java
class IntSet1 implements IntSet {
    ...
    IntSet1 insert(int i) { <body> }
}
```

... is compiled like this (top-level) procedure

```java
IntSet1 insert(IntSet1 this, int i) { <body> }
```

- Note
  - The type of “this” is the class containing the method
  - References to fields inside <body> are compiled like this.field
LLVM Method Invocation Compilation

Consider method invocation $[H;G;L \vdash e.m(e_1,\ldots,e_n):t]$

1. Compile $[H;G;L \vdash e : C]$  
   - To get a (pointer to) an object value of class type $C$  
   - Call this value $\text{obj_ptr}$
2. Use `Getelementptr` to extract the vtable pointer from $\text{obj_ptr}$
3. Load the vtable pointer
4. Use `Getelementptr` to extract the function pointer’s address from vtable  
   - Use the information about $C$ in $H$
5. Load the function pointer
6. Call through the function pointer, passing ‘$\text{obj_ptr}$’ for this

   \[
   \text{call (cmp_typ t) m(\text{obj_ptr}, [e_1], \ldots, [e_n])}
   \]

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

One job of object constructor is to fill in the object’s pointer to the appropriate dispatch vector

Note
  - The address of the D.V. is the runtime 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

  \[ \text{if? } (t \ x = \ exp) \ \{ \ \ldots \ \} \ \text{else} \ \{ \ \ldots \ \} \]

• 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 T to look for C (T is exp's dynamic type)
  – 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 T to look for C (T is exp's dynamic type)
  – 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
  \[
  \%B = \text{type} \{ \%\_class\_B*, i64, i64, i64 \}
  \]

- 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
  \[
  \_\_vtbl\_B = \text{global} \%\_class\_B \{ \%\_class\_A* \_\_vtbl\_A, \\
  \quad \text{void} (%B*)* @print\_B, \\
  \quad i64 (%A*, %A*)* @blah\_A \}
  \]

- 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