CS 412
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 23: Multiple Inheritance
28 March 01

Multiple Inheritance

- Mechanism: a class may declare multiple superclasses (C++)
- Java: may implement multiple interfaces, may inherit code from only one superclass
- Two problems: multiple supertypes, multiple superclasses
- What are implications of multiple supertypes in compiler?

Semantic problems

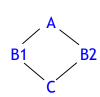
- Problem 1: ambiguity

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

- All methods, fields must be uniquely defined

- Problem 2: field replication

```
class A { int x; }
class B1 extends A { … }
class B2 extends A { … }
class C extends B1, B2 { … }
```



Dispatch vectors break

```
interface Shape {
   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 { …
}
```

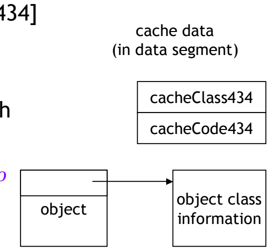

DV alternatives

- Option 1: search with inline cache (Smalltalk, Java)
  - For each class, interface, have table mapping method names to method code. Recursively walk upward in hierarchy looking for method name
  - *Optimization*: at call site, store class and code pointer in call site code (**inline caching**). On call, check whether class matches cache.

Inline cache code

- Let r_o be the receiver object:

```
mov t1, [ro]
cmp t1, [cacheClass434]
jnz miss
call [cacheCode434]
miss: call slowDispatch
```



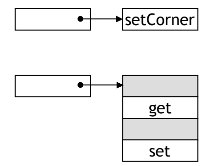
90% of calls from a site go to same code as last call from same site

Option 2: Sparse dispatch vectors

- Make sure that two methods never allocated same offset: give Shape offset 0, Color offsets 1 and 2. Allow holes in DV!
- Some methods can be given same offset since they never occur in the same DV
- *Graph coloring* techniques can be used to compute method indices in reasonably optimal way (finding optimum is NP-complete!)

Sparse Dispatch Vectors

```
interface Shape {
        void setCorner(int w, Point p);   0
}
interface Color {
        float get(int rgb);               1
        void set(int rgb, float value);   3
}
class Blob implements Shape, Color { … }
```



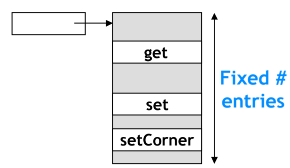
- Advantage: same fast dispatch code as SI case
- Disadvantage: requires knowledge of entire type hierarchy (makes separate compilation, dynamic loading difficult)

Option 3: Hash tables

- Idea: don't try to give all method unique indices; resolve conflicts by checking that entry is correct at dispatch
- Use hashing to generate method indices
  - Precompute hash values!
  - Some Java implementations

```
interface Shape {
        void setCorner(int w, Point p);        11
}
interface Color {
        float get(int rgb);                    4
        void set(int rgb, float value);        7
}
class Blob implements Shape, Color { … }
```


Dispatch with Hash tables

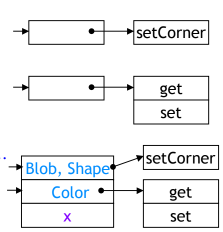


- What if there's a conflict? Entries containing several methods point to resolution code
- Basic dispatch code is (almost) identical!
- Advantage: simple, reasonably fast
- Disadvantage: some wasted space in DV, extra argument for resolution, slower dispatch if conflict

Option 4: Multiple DVs (C++)

- Idea: allow methods to have same offset in DV, have more than one DV when they clash

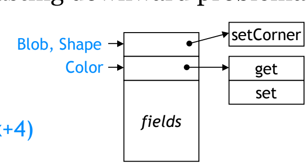
```
interface Shape {
  void setCorner(int w, Point p);
}
interface Color {
  float get(int rgb);
  void set(int rgb, float value);
}
class Blob implements Shape, Color { …
    …
    int x;
}
```



Multiple DV's

- Multiple possible pointers to object!
- Pointer chosen is determined by static type of object reference
- Changing static type requires addition of constant; casting downward problematic

Blob x;
Color y = x;
$\Rightarrow$ MOVE(y, x+4)

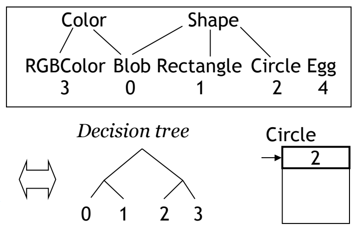


Option 5: Binary decision trees

- Idea: use conditional branches, not indirect jumps
- Unique class index stored in first object word
- Range tests used to select among *n* possible classes at call site in lg *n* time – direct branches to code

```
Shape x;
x.SetCorner(…)

mov ebx, [eax]
cmp ebx, 1
jle L1
cmp ebx, 2
je Circle$setCorner
jmp Egg$setCorner
L1: cmp ebx, 0
je Blob$setCorner
jmp Rect$setCorner
```



Binary decision tree

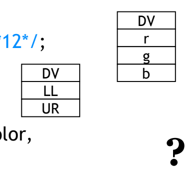
- Works well if distribution of classes is highly skewed: branch prediction hardware eliminates branch stall of ~10 cycles
  - Can use profiling to identify common paths for each call site individually
  - 90%/10% : usually a common path to put at top of decision tree

- Like sparse DVs: need whole-program analysis
- Indirect jump can have better expected execution time for >2 classes: at most one mispredict

Multiple Inheritance

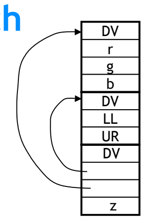
- Multiple supertypes : methods conflict
- Multiple inheritance : *fields* also can conflict
- Location of object fields no longer can be constant offset from start of object

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

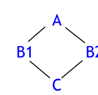


C++ approach

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

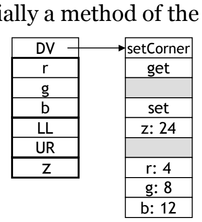


- Add pointers to superclass fields
- Extra indirection required to access superclass fields
- Needed even with single superclass
- Pointers needed to avoid field replication problem



Field offsets in DV

- Another approach: put offset to all fields in DV (offset to field is essentially a method of the class)



(with sparse dispatch vectors)

- Java finesses the whole problem: doesn't allow MI

Multimethods/generic functions

- Most OO languages (e.g. Java): dispatching on a single receiver object
- Dylan, CLOS, Cecil: multimethods (generic functions) exist *independent* of classes
- Calls dispatched to (dynamically) best matching function definition

```
setColor(Shape s, Color c)
setColor(Circle s, Color c)
setColor(Shape s, RGBColor c)
```

- Semantic problem: ambiguity
- Implementation problem: DV's don't work as well (multi-dimensional); binary decision trees are best option

Summary

- Multiple inheritance is expensive!
- Multiple supertypes: inline caching, sparse vectors, hashing, multiple DV's, binary decision trees
- Multiple superclasses: extra indirection for field accesses via inline pointer or DV field offset