Implementing Object-Oriented Languages

Key features:
- inheritance (possibly multiple)
- subtyping & subtype polymorphism
- message passing, dynamic binding, run-time type testing

Subtype polymorphism is the key problem
- support uniform representation of data
  (analogous to boxing for polymorphic data)
- store the class of each object at a fixed offset
- organize layout of data to make instance variable access and method lookup & invocation fast
- code compiled expecting an instance of a superclass still works if run on an instance of a subclass
- multiple inheritance complicates this
- perform static analysis to bound polymorphism
- perform transformations to reduce polymorphism

Implementing instance variable access

Key problem: subtype polymorphism
Solution: prefixing
- layout of subclass has layout of superclass as a prefix
- code that accesses a superclass will access the superclass part of any subclass properly, transparently
  + access is just a load or store at a constant offset

```java
class Point {
    int x;
    int y;
}
class ColorPoint extends Point {
    Color color;
}
```

// OK: subclass polymorphism
Point p = new ColorPoint(3,4,Blue);

// OK: x and y have same offsets in all Point subclasses
int manhattan_distance = p.x + p.y;

Implementing dynamic dispatching (virtual functions)

How to find the right method to invoke for a dynamically dispatched message `rcvr.Message(arg1, ...)`?

Option 1: search inheritance hierarchy, starting from run-time class of `rcvr`
  - very slow, penalizes deep inheritance hierarchies

Option 2: use a hash table
  - can act like a cache on the front of Option 1
    - still significantly slower than a direct procedure call
    - but used in early Smalltalk systems!

Option 3: store method addresses in the receiver objects, as if they were instance variables
  - each message/generic function declares an instance variable
  - each inheriting object stores an address in that instance variable
  - invocation = load + indirect jump!
    + good, constant-time invocation,
      independent of inheritance structure, overriding, ...
    - much bigger objects

Option 4: factor out class-invariant parts into shared object
  - each instance variables whose values are common across all instances of a class (e.g. method addresses) are moved out to a separate object
    - historically called a virtual function table (vtbl)
    - each instance contains a single pointer to the vtbl
    - combine with (or replace) class pointer
    - layout of subclass’s vtbl has layout of superclass’s vtbl as a prefix

  + dynamic dispatching is fast & constant-time
    - but an extra load
  + no space cost in object
    - aside from vtbl/class pointer

Virtual function tables

Observation: in Option 3, all instances of a given class will have identical method addresses
Example of virtual function tables

class Point {
    int x;
    int y;
    void draw();
    int distance2origin();
}

class ColorPoint extends Point {
    Color color;
    void draw();
    void reverse_video();
}

Multiple inheritance

Problem: prefixing doesn’t work with multiple inheritance

class Point {
    int x;
    int y;
    ...
}

class ColoredThing {
    Color color;
    ...
}

class ColorPoint extends Point, ColoredThing {
    ...
}

Some solutions

Option 1: stick with single inheritance [e.g. Smalltalk]
- some examples really benefit from MI

Option 2: distinguish classes from interfaces [e.g. Java, C#]
- only single inheritance below classes
  ⇒ if rcvr statically of class type, then can exploit prefixing for its instance variable accesses and message sends
- disallow instance variables in interfaces
  ⇒ no problems accessing them!
- only messages to receivers of interface type are unresolved
  ⇒ much smaller problem; can use e.g. hashing

Option 3: compute offset of a field in rcvr by sending rcvr a message [Cecil/Vortex]
- reduced problem to dynamic dispatching
- apply CHA etc. to optimize (all) dispatches
  ⇒ for fields whose offsets never change, static binding + inlining reduces dispatches to constant

Another solution

Option 4: embedding + pointer shifting [C++]
- concatenate superclass layouts, extend with subclass data
- when upcasting to a superclass, shift pointer to point to where superclass is embedded
  ⇒ downcasting does the reverse
- virtual function calls may need to shift rcvr pointers
  ⇒ "trampolines" may get inserted
+ gets back to constant-time access in most cases
- very complicated, lots of little details
- some things (e.g. casting) may now have run-time cost
- does poorly if using "virtual base classes", i.e., diamond-shaped inheritance hierarchies
- some sensible programs now disallowed
  ⇒ e.g. casting through void*, downcasting from virtual base class
- interior pointers may complicate GC, equality testing, debugging, etc.
Example

class Point {
    int x;
    int y;
}

class ColoredThing {
    Color color;
}

class ColorPoint extends Point, ColoredThing {
}

ColorPoint cp = new ColorPoint(3, 4, Blue);
Point p = cp; // OK
ColoredThing t = cp; // OK: adds 8 to cp
// now this works:
ColorPoint cp2 =
    new ColorPoint(p.x, p.y, t.color);
// this works, too:
ColorPoint cp3 =
    (ColorPoint) t; // subtracts 8 from t

Example of virtual function tables

class Point {
    int x;
    int y;
    void draw();
    int d2o();
}

class ColoredThing {
    Color color;
    void reverse_video();
    int d2o();
}

class ColorPoint extends Point, ColoredThing {
    void draw();

Limitations of table-based techniques

Table-based techniques only work well when:

- have static type information to use to map message/instance variable names to offsets in tables/objects
- not true in dynamically typed languages
- cannot extend classes with new operations except via subclassing
- not true in languages with open classes (e.g. MultiJava [Clifton et al. 00]) or multiple dispatching (e.g. CLOS, Dylan, Cecil)
- cannot modify classes dynamically
- not true in fully reflective languages (e.g. Smalltalk, Self, CLOS)
- memory loads and indirect jumps are inexpensive
- may not be true with heavily pipelined hardware

Dynamic table-based implementations

Standard implementation: global hash table in runtime system
- indexed by class x msg
- filled dynamically as program runs
- can be flushed after reflective operations
  + reasonable space cost
  + incremental
  - fair average-case dispatch time, poor worst-case time

Refinement: hash table per message name
- each call site knows statically which table to consult
  + faster dispatching
**Inline caching**

Give each dynamically-dispatched call site its own small method lookup cache
+ call site knows its message name
+ cache is isolated from other call sites

Trick: use machine call instruction itself as a one-element cache
- initially: call runtime system's `Lookup` routine
- `Lookup` routine patches call instruction to branch to invoked method
- record receiver class
- next time through, jump directly to expected target method
- method checks whether current receiver class is same as last receiver class
- if so, then cache hit (90-95% frequency, for Smalltalk)
- if not, then call `Lookup` and rebind cache

+ fast dispatch sequence if cache hit (∼4 instructions plus call)
  + hardware call prefetching works well
  - exploits self-modifying code
  - low performance if not a cache hit

[Deutsch & Schiffman 84]

---

**Example of inline caching**

Initially:

```
... 
call Lookup
msg: "draw"
class: 
...
```

After caching target method:

```
... 
call Lookup
msg: "draw"
class: CPt
...
```

ColorPoint::draw()

```
if cache.class ≠ self.class then
call Lookup
...regular code...
```

---

**Polymorphic inline caching (PIC)**

Idea: support a multi-element cache by generating a call-site-specific dispatcher stub
+ fast dispatching even if several classes are common
  - still slow performance if many classes equally common
  - some space cost

Foreshadowing:
dispatching stubs record dynamic profile data of which receiver classes occur at which call sites

[Hölzl et al. 91]

---

**Example of polymorphic inline caching**

After a few receiver classes:

```
switch (self.class) {
case ColorPt:
case ColorPt3D:
case Point:
default: call Lookup
}
```

ColorPt::draw()

```
Point::draw()
```

---
Implementing the dispatcher stub switch

In original PIC design, switch implemented with a linear chain of class identity tests

Alternatively, can implement with a binary search, exploiting ordering of integer class IDs or addresses
+ avoid worst-case behavior of long linear searches
+ a single test can direct many classes to same target method
− requires global knowledge to construct dispatchers

In traditional compilers, switch implemented with a jump table, akin to C++ dispatch tables

Can blend table-based lookups, linear search, and binary search [Chambers & Chen 99]
• exploit available static analysis of possible receiver classes, profile information of likely receiver classes
• construct dispatcher best balancing expected dispatching speed against dispatch space cost

Handling multiple dispatching

Languages with multimethods (e.g. CLOS, Dylan, Cecil) allow methods to dispatch on the run-time classes of any of the arguments
• call sites do not know statically which arguments may be dispatched upon

Implementation schemes:
• hash table indexed by N keys [Kiczales & Rodriguez 89]
• N-deep tree of hash tables, each indexed by 1 key [Dussud 89]
• can stop dispatching at any subtree if all remaining arguments undispatched
• N-deep DAG of 1-key dispatches [Chen & Turau 94, Chambers & Chen 99]
• compressed N+1-dimensional dispatch table [Amiel et al. 94, Pang et al. 99]

Probably more efficient to support multimethods directly than if simulated with double-dispatching [Ingalls 86] or visitor pattern [Gamma et al. 95]