1 Decision trees

Another approach to dispatch that is both general and potentially efficient is to construct decision trees. It is a fundamentally different approach from the dispatch mechanisms we have been discussing so far, which directly look up the address of method code to jump to. A call of this this form is a form of indirect jump—a jump to a computed address—which stalls the processor pipeline unless the hardware can predict where the jump is going. To do this prediction, modern processors use a *branch target buffer* (BTB), which records the target address of indirect jumps. Since the BTB stores a whole target address, an entry in the BTB is significantly more expensive than the hardware tables used to predict conditional branches.

The idea of dispatching via decision trees is to handle the dispatch entirely with conditional branches. Since there can be more than two targets for a given method call, in general the compiler needs to generate a decision tree. A simple form of decision tree relies on the first word of an object storing a class identifier—perhaps a small integer. The decision tree can then branch on the class identifier to find the right method code.

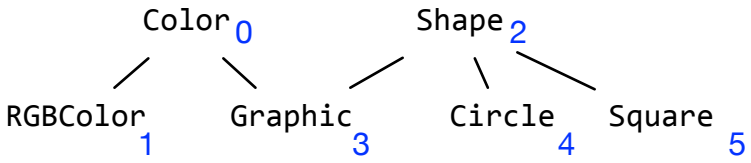


Figure 1: A numbered class hierarchy

For example, consider the class hierarchy shown in Figure 1, where each of the classes has been assigned an identifier rather arbitrarily based on a traversal of the class hierarchy (some coherence in the numbering of classes will help keep the decision tree small). Suppose that `RGBColor` inherits its implementation of a method from `Color` and that `Color` and `Square` inherit it from `Shape`. Then the decision tree for dispatching might look as shown in Figure 2. In this case, the indirect jump is replaced by two conditional branches.

Notice that this approach is quite general—it can handle complex hierarchies. It is also probably the best approach for more complex OO features like multimethods, where dispatch tables tend to blow up in size. However, dispatch code depends on knowing the entire class hierarchy, so this approach is more challenging to use with separate compilation.

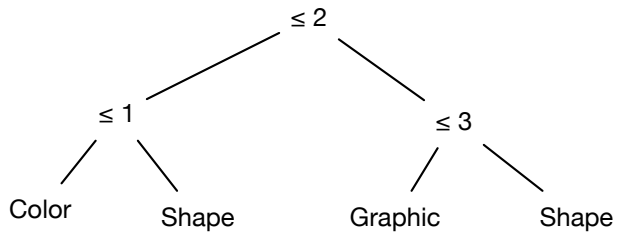


Figure 2: Decision tree for dispatching in hierarchy of Figure 1

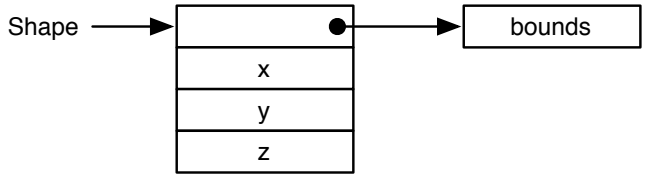
2 Using multiple dispatch tables for separate compilation

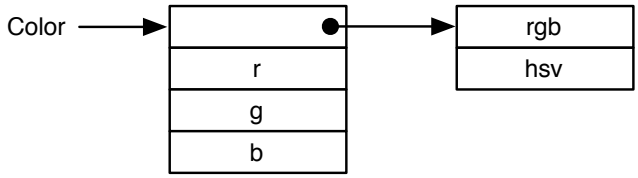
To deal with method index conflicts among superclasses, C++ may use multiple dispatch tables per object, and multiple references to the object. Which dispatch is to be used depends on which references to the object is used.

Different C++ implementations use different object layouts, but here is one possibility. Consider the following three classes:

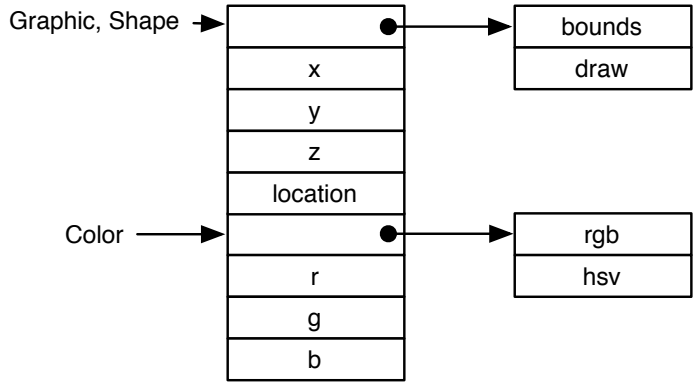
```
class Shape {
  bounds()
  x,y,z: num
}
class Color {
  rgb()
  hsv()
  r,g,b: num
}
class Graphics extends Shape, Color {
  draw()
  location: int
}
```

For separate compilation the method indices for `Shape` and `Color` have to both be assigned independently. So both start methods in their dispatch table at zero:





We can merge both these layouts into a single object, but we need separate dispatch tables because `bounds()` and `rgb()` use the same method index:



There are two distinct "views" of the object, one as either a `Graphic` or a `Shape`, and one as a `Color`. To switch between these views, some computation is required. For example, we might subsumption to view a `Graphic` as a `Color`:

```
Graphic = new Graphic();
Color c = g;
```

We might expect that the assignment `c=g` involves no computation, but in fact it is necessary to add 40 to the address of `g`.

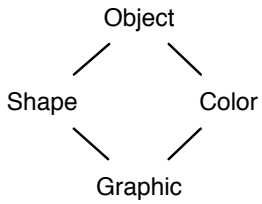
The result is fast dispatch in the usual case, but high per-object overhead, since we have two dispatch table pointers per object rather than just one. Supporting pointers to the interior of objects makes garbage collectors more complex and probably a little slower.

It's possible to put the methods of `Color` also into the `Graphic` dispatch table, but since different class code expects different views of the receiver object, a *trampoline* is needed to bump the receiver pointer to the correct view.

This layout merges the dispatch tables for `Graphic` and `Shape`. In general, one can merge a class with that of one of its superclasses or implemented interfaces. There are more complex schemes for merging multiple dispatch tables more effectively, such as bidirectional dispatch tables. With a bidirectional layout [Mye95], a class hierarchy that uses multiple inheritance only to allow a class to extend one other class and to implement one interface requires only a single dispatch table. This is possible by having the dispatch table grow in opposite directions.

3 Fields and multiple inheritance

Even the offsets to fields can conflict with multiple inheritance. For example, consider this inheritance hierarchy:

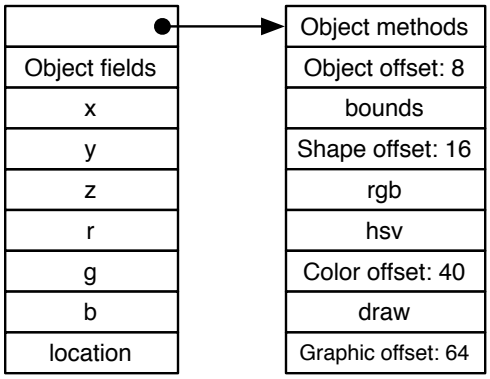


The code of both `Shape` and `Color` might need access to the fields of `Object`. But in a `Graphic` object, those fields can't be located at the same offset from the `Shape` and `Color` fields as in the `Shape` and `Color` object layouts.[1]

One way to solve the problem is to introduce internal pointers within the object between different views of the same object. This gives fast access to the fields of the current class view, and imposes no space or time overhead when inheritance is not being used. However, it has high per-object overhead even when single inheritance is being used. And internal pointers are a demanding feature that probably make the garbage collector slower.

A probably better idea is to store the offsets to fields in the dispatch table. Each field has a dispatch table index that can be consulted to find the field. Dispatch table indices can be assigned using graph coloring or by using multiple dispatch tables. For example, the following figure shows how the object layout might look assuming that dispatch table indices are assigned using graph coloring, so that there is a single dispatch table.

[1] Actually, C++ offers a version of "non-virtual" inheritance in which the fields *are* located at the same offset, but at the cost of duplicating the `Object` fields, which has strange semantics.



The sequence to access a field is more expensive than the usual indexed load. Before multiple inheritance, an access like `o.f` could be implemented as `mov`k_f`(o)`, t, where k_f is a known constant offset. With this object layout, accesses are more complex:

```
mov (o), t2
mov m_f(t2), t3
mov k_f(t2, t3), t1
```

The offset m_f is the location in the dispatch table of the field offset; the offset k_f is the offset within the subobject of the particular field. Since the offset `t3` is a constant, CSE can be used to avoid fetching it more than once.

This approach has much lower space overhead than using internal pointers, and access to fields from other class views are faster. However, access to fields of the current class view is slower, and there is a performance penalty even when inheritance is not being used.

4 Avoiding dispatch

We've talked about layouts and algorithms to speed up method dispatch. But the fastest way to do method dispatch is not to do it at all. If we can determine that there is only possible implementation of the method that is being invoked, the generated code can simply jump directly to the method code. Or the method code can be inlined at the call site, possibly enabling other optimizations.

Given a call `o.m()` where `o` has type T, when can we avoid method dispatch? One simple case is the following. If we know from inspection of the class hierarchy that there is one class C implementing T implements `m` (that is, all subtypes of T inherit from C), the method code C.`m` is the only possible code that dispatch could reach. This optimization does require knowledge of the whole class hierarchy, so it is not compatible with separate compilation. It is also not compatible with dynamic linking, which might cause new implementations of T to show up at run time. To support dynamic linking, it is necessary to have a run-time system, such as the HotSpot JVM, that can invalidate and regenerate code when the assumptions on which the original code is based are violated by newly loaded code.

A more sophisticated way to avoid dispatch is by acquiring more precise information about the class of an object than is present in the declared type. A variable has *exact type* C if it is an instance exactly of C and not of any subclass of C. If we know the exact type of an expression, then any method call using the expression can be resolved at compile time, avoiding dispatch. For example, consider this code:

```
x: C = new C()
x.m()
```

Because of the constructor call, we know `x` has exact type `C`, sometimes written as `x: C!`.

An *exact type analysis* finds exact types for expressions in the program, by propagating information from `new` expressions to possible uses. This can be done by building directly on an inclusion-based pointer analysis, since each `new` allocation is a distinct "object" in pointer analysis. If the set of objects that a given

pointer can point to all have the same class, the exact type of the pointer is known. Even if not, we may be able to determine that there is only one implementation for a given method. The analysis will probably need to be interprocedural to be effective.

5 Specialization

Inheritance is usually implemented by having the code for an inherited method shared across all classes that inherit it. Sometimes it is better to *specialize* the method code for particular inheriting classes, however. For example, consider two classes `A` and `B`:

```
class A {
  f() { ... g() ... }
  g() { A.g code }
}
class B {
    g() { B.g code }
}
```

Ordinarily we'd share the code for `A.f` with `B`. However, consider what happens if we instead specialize the method `f` to both `A` and `B`. Assuming there are no other implementations of `g()` in other subclasses, the version of `A.f` specialized to `A` knows that the exact type of `this` is `A!`. Therefore the call to `g()` must go to the "`A.g` code". Similarly the version specialized to `B` can call the "`B.g` code" directly. The code for `g` can even be inlined inside `f`, possibly enabling further optimizations.

This optimization is a space–time tradeoff. If `f` is called infrequently, we don't want to waste memory and cache space on storing multiple versions of it. If `f` is frequently used and its code is not large, then it makes sense to specialize it. It is a good idea to couple this optimization to some method for determining which methods are "hot"—either a program analysis or, even better, run-time profiling.

6 Multimethods

In most object-oriented languages, method code are chosen according to the class of the receiver object. The receiver object is an argument to the method; why not choose method code based on the other arguments as well? This is the idea behind *multimethods*, also known as *generic functions*. Multimethods are a feature in Common Lisp (CLOS) [DG87], MultiJava [CLCM00], Dylan [Sha96], Cecil [Cha93], and other languages. CLOS is quite widely used in industry.

Multimethods are helpful for so-called "binary" methods, in which there is an explicit argument with the same type as the class. For example, suppose we want to implement a class `Shape` with an a method `intersect(s: Shape): bool`, where `Shape` has various subclasses: `Box`, `Circle`, `Triangle`, and so on. With multimethods, we can think of this method as a generic function of two arguments: `intersect(Shape receiver, Shape s): bool`.

We can imagine wanting to implement different algorithms for different combinations of shapes. For example, when intersecting two boxes, we can use this test:

```
b1.x0 <= b2.x1 && b2.x0 <= b1.x1 && b1.y0 <= b2.y1 && b2.y0 <= b1.y1
```

To test whether two circles intersect, we test whether their centers are closer than the sum of their radii. And so on. With multimethods, we can add new shapes and new intersection algorithms in a modular way, e.g.:

```
intersect(b1: Box, b2: Box) : bool {
    return b1.x0 <= b2.x1 && b2.x0 <= b1.x1
        && b1.y0 <= b2.y1 && b2.y0 <= b1.y1
}
```

Another place where multimethods are handy is in fact for compilers. Recall that visitors are an answer to the problem of how to write the code for a compiler pass in a modular way. With multimethods, we can define a function `visit(Node, Pass)` that specifies the boilerplate traversal behavior in the base implementation. We then override it in a modular way for particular (`Node`, `Pass`) pairs where there is something interesting going on. In fact, MultiJava has been used to build compilers in this way.

A related idea to multimethods is *predicate dispatch* [EKC98], in which methods are chosen based on arbitrary properties of objects, rather than just their run-time class. The two ideas can be naturally combined to select on arbitrary properties of multiple arguments. For example, we might override `intersect` to give code that just works on squares:

```
intersect(b1: Box, b2: Box) : boolean
    where b1.width == b1.height && b2.width == b2.height
{...}
```

If the method is called on two boxes that don't satisfy the `where` clause, the ordinary `intersect(Box, Box)` implementation is called instead.

Predicate dispatch allows dispatching to acquire the power of pattern matching, though arguably in a more modular way, since the code for handling the different pattern cases can be implemented separately.

One problem with multimethods is implementing them efficiently. If there are N classes and k arguments, we can implement multimethods with a selector table containing N^k entries. For $k > 1$, this usually doesn't work very well, though these tables are highly compressible and there has been some work on compressing them.

The usual approach to implementing multimethods, though, is to implement the generic function as a decision tree (or DAG). Building the decision tree requires knowing about all the possible implementations. A decision tree also enables testing on conditions other than the run-time class, so it works for predicate dispatch too. For reasonable programs, the overhead of a decision tree is space and time is reasonable, no worse than implementing the dispatch in other ways.

References

[Cha93] Craig Chambers. The Cecil language: Specification and rationale. Technical report, University of Washington, 1993.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *15th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 130–145, 2000.

[DG87] L. DeMichiel and R. Gabriel. The Common Lisp Object System: An overview. In *Proceedings of European Conference on Object-Oriented Programming*, 1987.

[EKC98] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *12th European Conf. on Object-Oriented Programming*, ECOOP '98, pages 186–211, London, UK, 1998. Springer-Verlag.

[Mye95] Andrew C. Myers. Bidirectional object layout for separate compilation. In *10th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 124–139, October 1995. ACM SIGPLAN Notices 30(10).

[Sha96] A. Shalit. *The Dylan Reference Manual*. Addison-Wesley, 1996.