

1 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.

2 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.

3 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), MultiJava, Dylan, Diesel, 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*, 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.