CS 4120 Lecture 34 More object features, higher-order functions 16 November 2011

Lecturer: Andrew Myers

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 the 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.

4 Higher-order and first-class functions

Functions (and other kinds of values) are *first-class* if:

1. they can be assigned to variables,
2. they can be passed as function arguments,
3. they can be returned as function results,
4. and they can be constructed at run time.

Functions are first-class in quite a few programming languages, usually those considered "functional": ML, Haskell, Scheme, Common Lisp. They are not first-class in Java, though objects create many of the same issues, particularly with nested classes. In the languages C and C++, conditions 1–3 are met, but functions cannot be created at run time. Although it is not part of the standard, `gcc` does allow new functions to be created at run time, by generating code at run time.

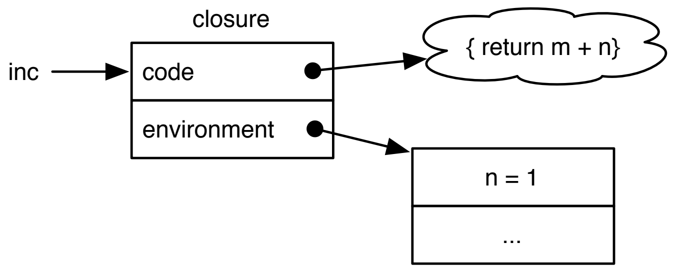
If functions can be passed as arguments and returned as results, we can implement *higher-order functions* in which this is done. Imagining that we extend Xi with first-class functions, we can implement a function that returns another function:

```
addfun(n: int): int->int {
  f(m: int): int {
    return m+n
  }
  return f
}
inc: int->int = addfun(1)
x: int = inc(5)
// x == 6
```

The variable `inc` is bound here to a function value that is created inside the call to `addfun`. What's happening at the assembly code or IR level? In a language like C, we can represent a function value simply as a pointer to the machine code implementing the function. In this example, the machine code doesn't suffice because the body of `f` has a *free variable* `n` that is not bound by `f`. Code that has unbound variables is called *open* code. It comes from an outer scope. How is the code to know that `n` is supposed to be 1 when it is invoked via the variable `inc`?

5 Closures and closure conversion

The standard implementation of a first-class function is as a *closure* that binds together the code of the function with a record containing definitions for its free variables. For example, the closure for `inc` might be represented in memory as shown in this figure:



The closure is a way of representing open code as closed code in which there are no free variables. The process of converting open code to closed code is called *closure conversion*. Closure conversion effectively adds an extra environment argument to every function that is used in a first-class way. For example, the function `f` has a signature like this when closed (note that `env` represents the memory address of the environment here):

```
f(f_environment env, int m) {
    return m + env.n
}
```

A call to a first-class function, such as `inc(5)`, is then implemented as follows:

```
inc.code(inc.environment, 5)
```

or, in abstract assembly, something like:

```
mov 0(inc), t1
mov 8(inc), %rdi
mov $5, %rsi
call t1
```

This is clearly more expensive than calling the code of `f` directly.

Note that it is possible to avoid the extra calling cost in the case where the function to be called is known at compile time, and is already closed. In this case the caller and callee can have their own special calling convention—a specialized version of the callee, if you like.

The environment that is part of the closure contains bindings for free variables of the code. We assume *lexical scoping* in which the variables in scope in a given piece of code are apparent in the lexical representation of the program: to find a binding for a variable, we walk outward through the text of the program to find the closest enclosing binding for the variable sought. The most common alternative is *dynamic scoping* in which free variables are evaluated by walking up the heap until a stack frame defining the variable is found. While some languages still do support dynamic scoping (notably Perl), it's usually considered to be bad idea.

Since bindings in the environment mention variables that are in scope in the body of the function, but are not defined in the function, they are variables from *outer scopes* such as containing functions (e.g, `addfun`). Before we introduced first-class functions, variables would be stored on the stack (or, as an optimization, in registers). However, in general, the variables from outer scopes can't be located on the stack any more.

In the `inc` example, the variable `n` is captured by the function returned. Although `f` doesn't assign to `n`, assignments to `n` should still be possible within `f` even after `addfun` returns. And if `addfun` had constructed multiple closures capturing `n`, the different closures should see each other's assignments. So `n` cannot be stored on the stack.

The variable `n` is an example of an *escaping variable*: a variable that is referenced by a function (`f`) that *escapes* the function in which the variable is defined. Escaping variables need to be stored on the heap.