14

Object-Oriented Languages

ob-ject: to feel distaste for something

Webster's Dictionary

A useful software-engineering principle is *information hiding* or *encapsulation*. A module may provide values of a given type, but the representation of that type is known only to the module. Clients of the module may manipulate the values only through *operations* provided by the module. In this way, the module can assure that the values always meet consistency requirements of its own choosing.

Object-oriented programming languages are designed to support information hiding. Because the "values" may have internal state that the operations will modify, it makes sense to call them *objects*. Because a typical "module" manipulates only one type of object, we can eliminate the notion of module and (syntactically) treat the operations as fields of the objects, where they are called *methods*.

Another important characteristic of object-oriented languages is the notion of *extension* or *inheritance*. If some program context (such as the formal parameter of a function or method) expects an object that supports methods m_1, m_2, m_3, then it will also accept an object that supports m_1, m_2, m_3, m_4.

14.1 CLASSES

To illustrate the techniques of compiling object-oriented languages I will use a simple class-based object-oriented language called Object-Tiger.

We extend the Tiger language with new declaration syntax to create classes:

dec → *classdec*
classdec → **class** *class-id* **extends** *class-id* { {*classfield* } }
classfield → *vardec*
classfield → *method*
method → **method** *id*(*tyfields*) = *exp*
method → **method** *id*(*tyfields*) : *type-id* = *exp*

The declaration `class B extends A { ··· }` declares a new class `B` that extends the class `A`. This declaration must be in the scope of the `let`-expression that declares `A`. All the fields and methods of `A` implicitly belong to `B`. Some of the `A` methods may be *overridden* (have new declarations) in `B`, but the fields may not be overridden. The parameter and result types of an overriding method must be identical to those of the overridden method.

There is a predefined class identifier `Object` with no fields or methods.

Methods are much like functions, with formal parameters and bodies. However, each method within `B` has an implicit formal parameter `self` of type `B`. But `self` is not a reserved word, it is just an identifier automatically bound in each method.

The responsibility for initializing object data fields rests with the class, not with the client. Thus, object-field declarations look more like variable declarations than like record-field declarations.

We make new expression syntax to create objects and invoke methods:

exp → **new** *class-id*
→ *lvalue* . *id*()
→ *lvalue* . *id*(*exp*{, *exp*})

The expression `new B` makes a new object of type `B`; the data fields are initialized by evaluating their respective initialization expressions from the class declaration of `B`.

The *l*-value `b.x`, where `b` is an *l*-value of type `B`, denotes the field `x` of object `b`; this is similar to record-field selection and requires no new syntax.

The expression `b.f(x,y)`, where `b` is an *l*-value of type `B`, denotes a call to the `f` method of object `b` with explicit actual parameters `x` and `y`, and the value `b` for the implicit `self` parameter of `f`.

Program 14.1 illustrates the use of the Object-Tiger language. Every `Vehicle` is an `Object`; every `Car` is a `Vehicle`; thus every `Car` is also an

```
let start := 10

    class Vehicle extends Object {
       var position := start
       method move(int x) = (position := position + x)
    }
    class Car extends Vehicle {
       var passengers := 0
       method await(v: Vehicle) =
           if (v.position < position)
                 then v.move(position - v.position)
           else self.move(10)
    }
    class Truck extends Vehicle {
       method move(int x) =
          if x <= 55 then position := position + x
    }

   var t := new Truck
   var c := new Car
   var v : Vehicle := c
in
    c.passengers := 2;
    c.move(60);
    v.move(70);
    c.await(t)
end
```

PROGRAM 14.1. An object-oriented program.

`Object`. Every `Vehicle` (and thus every `Car` and `Truck`) has an integer `position` field and a `move` method.

In addition, a `Car` has an integer `passengers` field and an `await` method. The variables in scope on entry to `await` are

`start` By normal Tiger language scoping rules.
`passengers` Because it is a field of `Car`.
`position` Because it is (implicitly) a field of `Car`.
`v` Because it is a formal parameter of `await`.
`self` Because it is (implicitly) a formal parameter of `await`.

In the main program, the expression `new Truck` has type `Truck`, so the type of `t` is `Truck` (in the normal manner of variable declarations in Tiger). Variable `c` has type `Car`, and variable `v` is explicitly declared to have type `Vehicle`. It is legal to use `c` (of type `Car`) in a context where type `Vehicle` is required (the initialization of `v`), because class `Car` is a subclass of `Vehicle`.

```
class A extends Object {
                    var a := 0}
class B extends A {var b := 0
                    var c := 0}
class C extends A {var d := 0}
class D extends B {var e := 0}
```

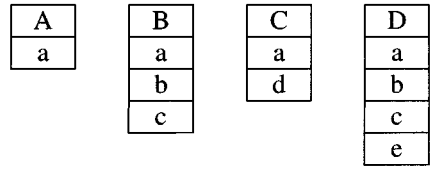


FIGURE 14.2. Single inheritance of data fields.

Class `Truck` overrides the `move` method of `Vehicle`, so that any attempt to `move` a truck "faster" than 55 has no effect.

At the call to `c.await(t)`, the truck `t` is bound to the formal parameter `v` of the `await` method. Then when `v.move` is called, this activates the `Truck_move` method body, not `Vehicle_move`.

We use the notation `A_m` to indicate a *method instance* `m` declared within a class `A`. This is not part of the Object-Tiger syntax, it is just for use in discussing the semantics of Object-Tiger programs. Each different declaration of a method is a different method instance. Two different method instances could have the same method name if, for example, one overrides the other.

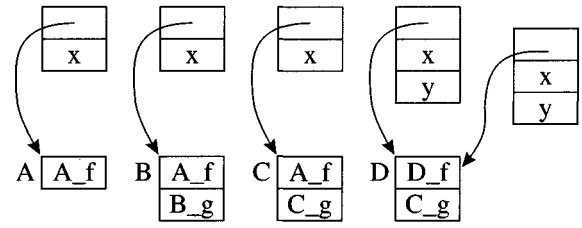
14.2 SINGLE INHERITANCE OF DATA FIELDS

To evaluate the expression `v.position`, where `v` belongs to class `Vehicle`, the compiler must generate code to fetch the field `position` from the object (record) that `v` points to.

This seems simple enough: the environment entry for variable `v` contains (among other things) a pointer to the type (class) description of `Vehicle`; this has a list of fields and their offsets. But at run time the variable `v` could also contain a pointer to a `Car` or `Truck`; where will the `position` field be in a `Car` or `Truck` object?

Single inheritance. For *single-inheritance* languages, in which each class extends just one parent class, the simple technique of *prefixing* works well. Where `B` extends `A`, those fields of `B` that are inherited from `A` are laid out in a `B` record *at the beginning, in the same order they appear in* `A` *records*. Fields of `B` not inherited from `A` are placed afterward, as shown in Figure 14.2.

```
class A extends Object {
                    var x := 0
                    method f() }
class B extends A {method g() }
class C extends B {method g() }
class D extends C {var y := 0
                    method f() }
```



PROGRAM 14.3. Class descriptors for dynamic method lookup.

METHODS

A method instance is compiled much like a function: it turns into machine code that resides at a particular address in the instruction space. Let us say, for example, that the method instance `Truck_move` has an entry point at machine-code label `Truck_move`. In the semantic-analysis phase of the compiler, each variable's environment entry contains a pointer to its class descriptor; each class descriptor contains a pointer to its parent class, and also a list of method instances; each method instance has a machine-code label.

Static methods. Some object-oriented languages allow some methods to be declared *static*. The machine code that executes when `c.f()` is called depends on the type of the *variable* `c`, not the type of the *object* that `c` holds. To compile a method-call of the form `c.f()`, the compiler finds the class of `c`; let us suppose it is class `C`. Then it searches in class `C` for a method `f`; suppose none is found. Then it searches the parent class of `C`, class `B`, for a method `f`; then the parent class of `B`; and so on. Suppose in some ancestor class `A` it finds a static method `f`; then it can compile a function call to label `A_f`.

Dynamic methods. But this technique will not work for dynamic methods. If method `f` in `A` is a dynamic method, then it might be overridden in some class `D` which is a subclass of `C` (see Figure 14.3). But there is no way to tell at compile time if the variable `c` is pointing to an object of class `D` (in which case `D_f` should be called) or class `C` (in which case `A_f` should be called).

To solve this problem, the class descriptor must contain a vector with a method instance for each (nonstatic) method name. When class `B` inherits from `A`, the method table *starts with* entries for all method names known to `A`, and then continues with `new` methods declared by `B`. This is very much like the arrangement of fields in objects with inheritance.

Figure 14.3 shows what happens when class `D` overrides method `f`. Al-

```
class A extends Object {var a := 0}
class B extends Object {var b := 0
                        var c := 0}
class C extends A {var d := 0}
class D extends A,B,C {var e := 0}
```

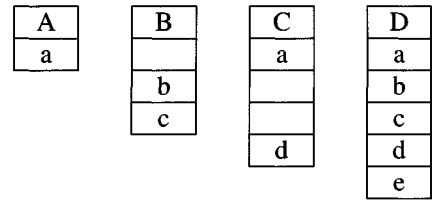


FIGURE 14.4. Multiple inheritance of data fields.

though the entry for `f` is at the beginning of `D`'s method table, as it is also at the beginning of the ancestor class `A`'s method table, it points to a different method-instance label because `f` has been overridden.

To execute `c.f()`, where `f` is a dynamic method, the compiled code must execute these instructions:

1. Fetch the class descriptor d at offset 0 from object `c`.
2. Fetch the method-instance pointer p from the (constant) `f` offset of d.
3. Jump to address p, saving return address (that is, call p).

14.3 MULTIPLE INHERITANCE

In languages that permit a class `D` to extend several parent classes `A,B,C` (that is, where `A` is not a subclass of `B` or vice versa), finding field offsets and method instances is more difficult. It is impossible to put all the `A` fields at the beginning of `D` *and* to put all the `B` fields at the beginning of `D`.

Global graph coloring. One solution to this problem is to statically analyze all classes at once, finding some offset for each field name that can be used in every record containing that field. We can model this as a graph-coloring problem: there is a node for each distinct field name, and an edge for any two fields which coexist (perhaps by inheritance) in the same class.[1] The offsets 0, 1, 2, ... are the colors. Figure 14.4 shows an example.

The problem with this approach is that it leaves empty slots in the middle of objects, since it cannot always color the N fields of each class with colors with the first N colors. To eliminate the empty slots in objects, we pack the fields of each object and have the class descriptor tell where each field is. Figure 14.5 shows an example. We have done graph coloring on all the field

[1] *Distinct field name* does not mean simple equivalence of strings. Each fresh declaration of field or method x (where it is not overriding the x of a parent class) is really a distinct name.

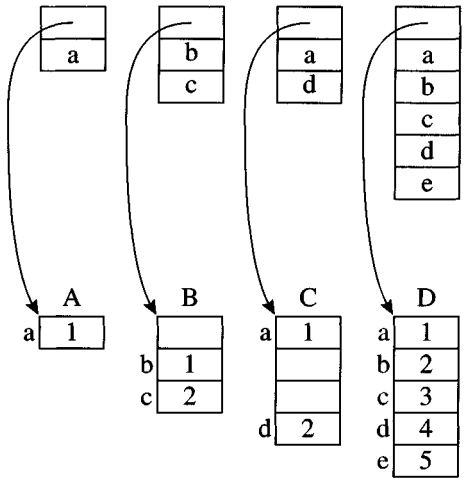


FIGURE 14.5. Field offsets in descriptors for multiple inheritance.

names, as before, but now the "colors" are not the offsets of those fields within the *objects* but within the *descriptors*. To fetch a field a of object x, we fetch the a-word from x's descriptor; this word contains a small integer telling the position of the actual a data within x.

In this scheme, class descriptors have empty slots, but the objects do not; this is acceptable because a system with millions of objects is likely to have only dozens of class descriptors. But each data fetch (or store) requires three instructions instead of one:

1. Fetch the descriptor-pointer from the object.
2. Fetch the field-offset value from the descriptor.
3. Fetch (or store) the data at the appropriate offset in the object.

In practice, it is likely that other operations on the object will have fetched the descriptor-pointer already, and multiple operations on the same field (e.g., fetch then store) won't need to re-fetch the offset from the descriptor; common-subexpression elimination can remove much of this redundant overhead.

Method lookup. Finding method instances in a language with multiple inheritance is just as complicated as finding field offsets. The global graph-coloring approach works well: the method names can be mixed with the field names to form nodes of a large interference graph. Descriptor entries for fields give locations within the objects; descriptor entries for methods give machine-code addresses of method instances.

Problems with dynamic linking. Any global approach suffers from the problem that the coloring (and layout of class descriptors) can be done only at link-time; the job is certainly within the capability of a special-purpose linker.

However, many object-oriented systems have the capability to load new classes into a running system; these classes may be extensions (subclasses) of classes already in use. Link-time graph coloring poses many problems for a system that allows dynamic incremental linking.

Hashing. Instead of global graph coloring, we can put a hash table in each class descriptor, mapping field names to offsets and method names to method instances. This works well with separate compilation and dynamic linking.

The characters of the field names are not hashed at run time. Instead, each field name a is hashed at compile time to an integer hash_a in the range $[0, N-1]$. Also, for each field name a unique run-time record (pointer) ptr_a is made for each field.

Each class descriptor has a field-offset table `Ftab` of size N containing field-offsets and method instances, and (for purposes of collision detection) a parallel *key* table `Ktab` containing field-name pointers. If the class has a field x, then field-offset-table slot number hash_x contains the offset for x, and key-table slot number hash_x contains the pointer ptr_x.

To fetch a field x of object c, the compiler generates code to

1. Fetch the class descriptor d at offset 0 from object `c`.
2. Fetch the field-name f from the address offset $d + \texttt{Ktab} + \text{hash}_x$.
3. Test whether $f = \text{ptr}_x$; if so
4. Fetch the field offset k from $d + \texttt{Ftab} + \text{hash}_x$.
5. Fetch the contents of the field from $c + k$.

This algorithm has four instructions of overhead, which may still be tolerable. A similar algorithm works for dynamic method-instance lookup.

The algorithm as described does not say what to do if the test at line 3 fails. Any hash-table collision-resolution technique can be used.

14.4 TESTING CLASS MEMBERSHIP

Some object-oriented languages allow the program to test membership of an object in a class at run time, as summarized in Table 14.6.

Since each object points to its class descriptor, the address of the class descriptor can serve as a "type-tag." However, if `x` is an instance of `D`, and `D`

	Modula-3	Java
Test whether object x belongs class C, or to any subclass of C.	ISTYPE(x,C)	`x instanceof C`
Given a variable x of class C, where x actually points to an object of class D that extends C, yield an expression whose compile-time type is class D.	NARROW(x,D)	`(D)x`

TABLE 14.6. Facilities for type testing and safe casting.

extends C, then x is also an instance of C. Assuming there is no multiple inheritance, a simple way to implement `x instanceof C` is to generate code that performs the following loop at run time:

$t_1 \leftarrow x$.descriptor
L_1 : **if** $t_1 = C$ **goto** *true*
$t_1 \leftarrow t_1$.super
if $t_1 =$ nil **goto** *false*
goto L_1

where t_1.super is the superclass (parent class) of class t_1.

However, there is a faster approach using a *display* of parent classes. Assume that the class nesting depth is limited to some constant, such as 20. Reserve a 20-word block in each class descriptor. In the descriptor for a class D whose nesting depth is j, put a pointer to descriptor D in the jth slot, a pointer to `D.super` in the $(j-1)$th slot, a pointer to `D.super.super` in slot $j-2$, and so on up to `Object` in slot 0. In all slots numbered greater than j, put nil.

Now, if x is an instance of D, or of any subclass of D, then the jth slot of x's class descriptor will point to the class descriptor D. Otherwise it will not. So `x instanceof D` requires

1. Fetch the class descriptor d at offset 0 from object c.
2. Fetch the jth class-pointer slot from d.
3. Compare with the class descriptor D.

This works because the class-nesting depth of D is known at compile time.

Type coercions. Given a variable c of type C, it is always legal to treat c as any supertype of C – if C extends B, and variable b has type B, then the assignment $b \leftarrow c$ is legal and safe.

But the reverse is not true. The assignment $c \leftarrow b$ is safe only if b is really (at run time) an instance of C, which is not always the case. If we have

$b \leftarrow$ `new` B, $c \leftarrow b$, followed by fetching some field of c that is part of class C but not class B, then this fetch will lead to unpredictable behavior.

Thus, safe object-oriented languages (such as Modula-3 and Java) accompany any coercion from a superclass to a subclass with a run-time type-check that raises an exception unless the run-time value is really an instance of the subclass (e.g. unless `b instanceof C`).

It is a common idiom to write

Modula-3:
```
IF ISTYPE(b,C)
  THEN f(NARROW(b,C))
  ELSE ...
```

Java:
```
if (b instanceof C)
    f((C)b)
else ...
```

Now there are two consecutive, identical type tests: one explicit (ISTYPE or `instanceof`) and one implicit (in NARROW or the cast). A good compiler will do enough flow analysis to notice that the **then**-clause is reached only if `b` is in fact an instance of `C`, so that the type-check in the narrowing operation can be eliminated.

C++ is an unsafe object-oriented language. It has a *static cast* mechanism without run-time checking; careless use of this mechanism can make the program "go wrong" in unpredictable ways. C++ also has `dynamic_cast` with run-time checking, which is like the mechanisms in Modula-3 and Java.

Typecase. Explicit `instanceof` testing, followed by a narrowing cast to a subclass, is not a wholesome "object-oriented" style. Instead of using this idiom, programmers are expected to use dynamic methods that accomplish the right thing in each subclass. Nevertheless, the test-then-narrow idiom is fairly common.

Modula-3 has a **typecase** facility that makes the idiom more beautiful and efficient (but not any more "object-oriented"):

```
TYPECASE expr
OF C1 (v1) => S1
 | C2 (v2) => S2
   ⋮
 | Cn (vn) => Sn
ELSE S0
END
```

If the *expr* evaluates to an instance of class C_i, then a new variable v_i of type C_i points to the result of the *expr*, and statement S_i is executed. The

declaration of v_i is implicit in the TYPECASE, and its scope covers only S_i.

If more than one of the C_i match (which can happen if, for example, one is a superclass of another), then only the first matching clause is taken. If none of the C_i match, then the ELSE clause is taken (statement S_0 is executed).

Typecase can be converted straightforwardly to a chain of **else-if**s, with each **if** doing an instance test, a narrowing, and a local variable declaration. However, if there are very many clauses, then it can take a long time to go through all the **else-if**s. Therefore it is attractive to treat it like a case (switch) statement on integers, using an indexed jump (computed goto).

That is, an ordinary case statement on integers:

ML:

```
case i
 of 0 => s0
  | 1 => s1
  | 2 => s2
  | 3 => s3
  | 4 => s4
  | _ => sd
```

C, Java:

```
switch (i) {
 case 0: s0; break;
 case 1: s1; break;
 case 2: s2; break;
 case 3: s3; break;
 case 4: s4; break;
 default: sd;
}
```

is compiled as follows: first a range-check comparison is made to ensure that i is within the range of case labels (0–4, in this case); then the address of the ith statement is fetched from the ith slot of a table, and control jumps to s_i.

This approach will not work for **typecase**, because of subclassing. That is, even if we could make class descriptors be small integers instead of pointers, we cannot do an indexed jump based on the class of the object, because we will miss clauses that match superclasses of that class. Thus, Modula-3 **typecase** is implemented as a chain of **else-if**s.

Assigning integers to classes is not trivial, because separately compiled modules can each define their own classes, and we do not want the integers to clash. But a sophisticated linker might be able to assign the integers at link time.

If all the classes in the **typecase** were `final` classes (in the sense used by Java, that they cannot be extended), then this problem would not apply. Modula-3 does not have final classes; and Java does not have **typecase**. But a clever Java system might be able to recognize a chain of **else-if**s that do **instanceof** tests for a set of final classes, and generate a indexed jump.

14.5 PRIVATE FIELDS AND METHODS

True object-oriented languages can protect fields of objects from direct manipulation by other objects' methods. A *private* field is one that cannot be fetched or updated from any function or method declared outside the object; a private method is one that cannot be called from outside the object.

Privacy is enforced by the type-checking phase of the compiler. In the symbol table of `c`, along with each field offset and method offset, is a boolean flag indicating whether the field is private. When compiling the expression `c.f()` or `c.x`, it is a simple matter to check that field and reject accesses to private fields from any method outside the object declaration.

There are many varieties of privacy and protection. Different languages allow

- Fields and methods which are accessible only to the class that declares them.
- Fields and methods accessible to the declaring class, and to any subclasses of that class.
- Fields and methods accessible only within the same module (package, namespace) as the declaring class.
- Fields that are read-only from outside the declaring class, but writable by methods of the class.

In general, these varieties of protection can be statically enforced by compile-time type-checking, for class-based languages.

14.6 CLASSLESS LANGUAGES

Some object-oriented languages do not use the notion of **class** at all. In such a language, each object implements whatever methods and has whatever data fields it wants. Type-checking for such languages is usually *dynamic* (done at run time) instead of *static* (done at compile time).

Many objects are created by *cloning*: copying an existing object (or *template* object) and then modifying some of the fields. Thus, even in a classless language there will be groups ("pseudo-classes") of similar objects that can share descriptors. When b is created by cloning a, it can share a descriptor with a. Only if a new field is added or a method field is updated (overridden) does b require a new descriptor.

The techniques used in compiling classless languages are similar to those

for class-based languages with multiple inheritance and dynamic linking: pseudo-class descriptors contain hash tables that yield field offsets and method instances.

The same kinds of global program analysis and optimization that are used for class-based languages – finding which method instance will be called from a (dynamic) method call site – are just as useful for classless languages.

14.7 OPTIMIZING OBJECT-ORIENTED PROGRAMS

An optimization of particular importance to object-oriented languages (which also benefit from most optimizations that apply to programming languages in general) is conversion of dynamic method calls to static method-instance calls.

Compared with an ordinary function call, at each method call site there is a dynamic method lookup to determine the method instance. For single-inheritance languages, method lookup takes only two instructions. This seems like a small cost, but:

- Modern machines can jump to constant addresses more efficiently than to addresses fetched from tables. When the address is manifest in the instruction stream, the processor is able to pre-fetch the instruction cache at the destination and direct the instruction-issue mechanism to fetch at the target of the jump. Unpredictable jumps stall the instruction-issue and -execution pipeline for several cycles.
- An optimizing compiler that does inline expansion or interprocedural analysis will have trouble analyzing the consequences of a call if it doesn't even know which method instance is called at a given site.

For multiple-inheritance and classless languages, the dynamic method-lookup cost is even higher.

Thus, optimizing compilers for object-oriented languages do global program analysis to determine those places where a method call is always calling the same method instance; then the dynamic method call can be replaced by a static function call.

For a method call `c.f()`, where `c` is of class `C`, *type hierarchy analysis* is used to determine which subclasses of `C` contain methods `f` that may override `C_f`. If there is no such method, then the method instance must be `C_f`.

This idea is combined with *type propagation*, a form of static dataflow analysis similar to *reaching definitions* (see Section 17.2). After an assign-

ment $c \leftarrow$ `new` C, the exact class of c is known. This information can be propagated through the assignment $d \leftarrow c$, and so on. When `d.f()` is encountered, the type-propagation information limits the range of the type hierarchy that might contribute method instances to `d`.

Suppose a method `f` defined in class `C` calls method `g` on `self`. But `g` is a dynamic method and may be overridden, so this call requires a dynamic method lookup. An optimizing compiler may make a different copy of a method instance `C_f` for each subclass (e.g. `D, E`) that extends `C`. Then when the (new copy) `D_f` calls `g`, the compiler knows to call the instance `D_g` without a dynamic method lookup.

PROGRAM OBJECT-Tiger

Implement the Object-Tiger object-oriented extensions to your Tiger compiler.

This chapter's description of the Object-Tiger language leaves many things unspecified: if method `f` is declared before method `g`, can `f` call `g`? Can a method access all the class variables, or just the ones declared above it? Can the initializer of a class variable (field) call a method of the class (and can the method therefore see an unitialized field)? You will need to refine and document the definition of the Object-Tiger language.

FURTHER READING

Dahl and Nygaard's Simula-67 language [Birtwistle et al. 1973] introduced the notion of classes, objects, single inheritance, static methods, instance testing, typecase, and the *prefix* technique to implement static single inheritance. In addition it had coroutines and garbage collection.

Cohen [1991] suggested the *display* for constant-time testing of class membership.

Dynamic methods and multiple inheritance appeared in Smalltalk [Goldberg et al. 1983], but the first implementations used slow searches of parent classes to find method instances. Rose [1988] and Connor et al. [1989] discuss fast hash-based field- and method-access algorithms for multiple inheritance. The use of graph coloring in implementing multiple inheritance is due to Dixon et al. [1989]. Lippman [1996] shows how C++-style multiple inheritance is implemented.

Chambers et al. [1991] describe several techniques to make classless, dynamically typed languages perform efficiently: pseudo-class descriptors, multiple versions of method instances, and other optimizations. Diwan et al. [1996] describe optimizations for statically typed languages that can replace dynamic method calls by static function calls.

Conventional object-oriented languages choose a method instance for a call `a.f(x,y)` based only on the class of the method *receiver* (`a`) and not other arguments (`x,y`). Languages with *multimethods* [Bobrow et al. 1989] allow dynamic method lookup based on the types of all arguments. Chambers and Leavens [1995] show how to do static type-checking for multimethods; Amiel et al. [1994] and Chen and Turau [1994] show how to do efficient dynamic multimethod lookup.

Nelson [1991] describes Modula-3, Stroustrup [1997] describes C++, and Arnold and Gosling [1996] describe Java.

EXERCISES

***14.1** A problem with the *display* technique (as explained on page 301) for testing class membership is that the maximum class nesting depth N must be fixed in advance, and every class descriptor needs N words of space even if most classes are not deeply nested. Design a variant of the *display* technique that does not suffer from these problems; it will be a couple of instructions more costly than the one described on page 301.

14.2 The hash-table technique for finding field offsets and method instances in the presence of multiple inheritance is shown incompletely on page 300 – the case of $f \neq \text{ptr}_x$ is not resolved. Choose a collision-resolution technique, explain how it works, and analyze the extra cost (in instructions) in the case that $f = \text{ptr}_x$ (no collision) and $f \neq \text{ptr}_x$ (collision).

***14.3** Consider the following class hierarchy, which contains five method-call sites. The task is to show which of the method-call sites call known method instances, and (in each case) show which method instance. For example, you might say that "method-instance `X_g` always calls `Y_f`; method `Z_g` may call more than one instance of `f`."

```
class A extends Object { method f() = print("1") }
class B extends A      { method g() = (f(); print("2")) }
class C extends B      { method f() = (g(); print("3")) }
class D extends C      { method g() = (f(); print("4")) }
class E extends A      { method g() = (f(); print("5")) }
class F extends E      { method g() = (f(); print("6")) }
```

Do this analysis for each of the following assumptions:

a. This is the entire program, and there are no other subclasses of these modules.

b. This is part of a large program, and any of these classes may be extended elsewhere.

c. Classes `C` and `E` are local to this module, and cannot be extended elsewhere; the other classes may be extended.

***14.4** Use *method replication* to improve your analysis of the program in Exercise 14.3. That is, make *every* class override `f` and `g`. For example, in class `B` (which does not already override `f`), put a copy of method `A_f`, and in `D` put a copy of `C_F`:

```
class B extends A      { ... method f() = (print("1")) }
class D extends C      { ... method f() = (g(); print("3")) }
```

Similarly, add new instances `E_f`, `F_f`, and `C_g`. Now, for each set of assumptions (a), (b), and (c), show which method calls go to known static instances.

****14.5** Devise an efficient implementation mechanism for any **typecase** that only mentions `final` classes. A `final` class is one that cannot be extended. (In Java, there is a `final` keyword; but even in other object-oriented languages, a class that is not exported from a module is effectively `final`, and a link-time whole-program analysis can discover which classes are never extended, whether declared `final` or not.)

You may make any of the following assumptions, but state which assumptions you need to use:

a. The linker has control over the placement of class-descriptor records.

b. Class descriptors are integers managed by the linker that index into a table of descriptor records.

c. The compiler explicitly marks `final` classes (in their descriptors).

d. Code for **typecase** can be generated at link time.

e. After the program is running, no other classes and subclasses are dynamically linked into the program.