18 Case Study: Imperative Objects

In this chapter we come to our first substantial programming example. We will use most of the features we have defined—functions, records, general recursion, mutable references, and subtyping—to build up a collection of programming idioms supporting objects and classes similar to those found in object-oriented languages like Smalltalk and Java. We will not introduce any new concrete syntax for objects or classes in this chapter: what we're after here is to try to *understand* these rather complex language features by showing how to approximate their behavior using lower-level constructs.

For most of the chapter, the approximation is actually quite accurate: we can obtain a satisfactory implementation of most features of objects and classes by regarding them as derived forms that are desugared into simple combinations of features we have already seen. When we get to virtual methods and `self` in §18.9, however, we will encounter some difficulties with evaluation order that make the desugaring a little unrealistic. A more satisfactory account of these features can be obtained by axiomatizing their syntax, operational semantics, and typing rules directly, as we do in Chapter 19.

18.1 What Is Object-Oriented Programming?

Most arguments about "What is the essence of...?" do more to reveal the prejudices of the participants than to uncover any objective truth about the topic of discussion. Attempts to define the term "object-oriented" precisely are no exception. Nonetheless, we can identify a few fundamental features that are found in most object-oriented languages and that, in concert, support a distinctive programming style with well-understood advantages and disadvantages.

The examples in this chapter are terms of the simply typed lambda-calculus with subtyping (Figure 15-1), records (15-3), and references (13-1). The associated OCaml implementation is `fullref`.

1. **Multiple representations.** Perhaps the most basic characteristic of the object-oriented style is that, when an operation is invoked on an object, the object itself determines what code gets executed. Two objects responding to the same set of operations (i.e., with the same *interface*) may use entirely different representations, as long as each carries with it an implementation of the operations that works with its particular representation. These implementations are called the object's *methods*. Invoking an operation on an object—called *method invocation* or, more colorfully, sending it a *message*—involves looking up the operation's name at run time in a method table associated with the object, a process called *dynamic dispatch*.

   By contrast, a conventional *abstract data type* (*ADT*) consists of a set of values plus a *single* implementation of the operations on these values. (This static definition of implementations has both advantages and disadvantages over objects; we explore these further in §24.2.)

2. **Encapsulation.** The internal representation of an object is generally hidden from view outside of the object's definition: only the object's own methods can directly inspect or manipulate its fields.[1] This means that changes to the internal representation of an object can affect only a small, easily identifiable region of the program; this constraint greatly improves the readability and maintainability of large systems.

   Abstract data types offer a similar form of encapsulation, ensuring that the concrete representation of their values is visible only within a certain scope (e.g., a module, or an ADT definition), and that code outside of this

---

1. In some object-oriented languages, such as Smalltalk, this encapsulation is mandatory—the internal fields of an object simply cannot be *named* outside of its definition. Other languages, such as C++ and Java, allow fields to be marked either `public` or `private`. Conversely, all the methods of an object are publicly accessible in Smalltalk, while Java and C++ allow *methods* to be marked `private`, restricting their call sites to other methods in the same object. We will ignore such refinements here, but they have been considered in detail in the research literature (Pierce and Turner, 1993; Fisher and Mitchell, 1998; Fisher, 1996a; Fisher and Mitchell, 1996; Fisher, 1996b; Fisher and Reppy, 1999).

   Although most object-oriented languages take encapsulation as an essential notion, there are several that do not. The *multi-methods* found in CLOS (Bobrow, DeMichiel, Gabriel, Keene, Kiczales, and Moon, 1988; Kiczales, des Rivières, and Bobrow, 1991), Cecil (Chambers, 1992, 1993), Dylan (Feinberg, Keene, Mathews, and Withington., 1997; Shalit), and KEA (Mugridge, Hamer, and Hosking, 1991) and in the lambda-& calculus of Castagna, Ghelli, and Longo (1995; Castagna, 1997) keep object states separate from methods, using special type-tags to select appropriate alternatives from overloaded method bodies at method invocation time. The underlying mechanisms for object creation, method invocation, class definition, etc., in these languages are fundamentally different from the ones we describe in this chapter, although the high-level programming idioms that they lead to are quite similar.

scope can manipulate these values only by invoking operations defined within this privileged scope.

3. **Subtyping.** The type of an object—its *interface*—is just the set of names and types of its operations. The object's internal representation does *not* appear in its type, since it does not affect the set of things that we can directly do with the object.

   Object interfaces fit naturally into the subtype relation. If an object satisfies an interface `I`, then it clearly also satisfies any interface `J` that lists fewer operations than `I`, since any context that expects a `J`-object can invoke only `J`-operations on it and so providing an `I`-object should always be safe. (Thus, object subtyping is similar to record subtyping. Indeed, for the model of objects developed in this chapter, they will be the same thing.) The ability to ignore parts of an object's interface allows us to write a single piece of code that manipulates many different sorts of objects in a uniform way, demanding only a certain common set of operations.

4. **Inheritance.** Objects that share parts of their interfaces will also often share some behaviors, and we would like to implement these common behaviors just once. Most object-oriented languages achieve this reuse of behaviors via structures called *classes*—templates from which objects can be instantiated—and a mechanism of *subclassing* that allows new classes to be derived from old ones by adding implementations for new methods and, when necessary, selectively overriding implementations of old methods. (Instead of classes, some object-oriented languages use a mechanism called *delegation*, which combines the features of objects and classes.)

5. **Open recursion.** Another handy feature offered by most languages with objects and classes is the ability for one method body to invoke another method of the same object via a special variable called `self` or, in some languages, `this`. The special behavior of `self` is that it is *late-bound*, allowing a method defined in one class to invoke another method that is defined later, in some subclass of the first.

The remaining sections of this chapter develop these features in succession, beginning with very simple "stand-alone" objects and then considering increasingly powerful forms of classes.

Later chapters examine different accounts of objects and classes. Chapter 19 presents a direct treatment (not an encoding) of objects and classes in the style of Java. Chapter 27 returns to the encoding developed in the present chapter, improving the run-time efficiency of class construction using bounded quantification. Chapter 32 develops a more ambitious version of the encoding that works in a purely functional setting.