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 and , 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.

18.2 Objects

In its simplest form, an *object* is just a data structure encapsulating some internal *state* and offering access to this state to clients via a collection of *methods*. The internal state is typically organized as a number of mutable *instance variables* (or *fields*) that are shared among the methods and inaccessible to the rest of the program.

Our running example throughout the chapter will be objects representing simple counters. Each counter object holds a single number and provides two methods (i.e., responds to two messages)—`get`, which causes it to return its current value; and `inc`, which increments the value.

A straightforward way of obtaining this behavior using the features we have discussed in previous chapters is to use a reference cell for the object's internal state and a record of functions for the methods. A counter object whose current state is `1` looks like this:

```
c = let x = ref 1 in
       {get = λ_:Unit. !x,
        inc = λ_:Unit. x:=succ(!x)};
▸ c : {get:Unit→Nat, inc:Unit→Unit}
```

The method bodies are both written as functions with trivial parameters (written `_` because we don't need to refer to them in the bodies). The abstractions block evaluation of the method bodies when the object is created, allowing the bodies to be evaluated repeatedly, later, by applying them over and over to the `unit` argument. Also, note how the state of this object is shared among the methods and inaccessible to the rest of the program: the encapsulation of the state arises directly from the lexical scope of the variable `x`.

To invoke a method of the object `c`, we just extract a field of the record and apply it to an appropriate argument. For example:

```
c.inc unit;
▸ unit : Unit

c.get unit;
▸ 2 : Nat

(c.inc unit; c.inc unit; c.get unit);
▸ 4 : Nat
```

The fact that the `inc` method returns `unit` allows us to use the ;-notation (§11.3) for sequences of increments. We could equivalently have written the last line above as:

```
let _ = c.inc unit in let _ = c.inc unit in c.get unit;
```

Since we may want to create and manipulate many counters, it is convenient to introduce an abbreviation for their type:

```
Counter = {get:Unit→Nat, inc:Unit→Unit};
```

Our attention in this chapter is focused on how objects are *built,* rather than on how they are *used* in organizing larger programs. However, we do want to see at least one function that uses objects, so that we can verify that it works on objects with different internal representations. Here is a trivial one—a function that takes a counter object and invokes its `inc` method three times:

```
inc3 = λc:Counter. (c.inc unit; c.inc unit; c.inc unit);
▸ inc3 : Counter → Unit

(inc3 c; c.get unit);
▸ 7 : Nat
```

18.3 Object Generators

We have seen how to build individual counter objects, one at a time. It is equally easy to write a *counter generator*—a function that creates and returns a new counter every time it is called.

```
newCounter =
  λ_:Unit. let x = ref 1 in
              {get = λ_:Unit. !x,
               inc = λ_:Unit. x:=succ(!x)};
▸ newCounter : Unit → Counter
```

18.4 Subtyping

One of the reasons for the popularity of object-oriented programming styles is that they permit objects of many shapes to be manipulated by the same client code. For example, suppose that, in addition to the `Counter` objects defined above, we also create some objects with an additional method that allows them to be reset to their initial state (say, 1) at any time.

```
ResetCounter = {get:Unit→Nat, inc:Unit→Unit, reset:Unit→Unit};
```

```
newResetCounter =
  λ_:Unit. let x = ref 1 in
              {get   = λ_:Unit. !x,
               inc   = λ_:Unit. x:=succ(!x),
               reset = λ_:Unit. x:=1};

▸ newResetCounter : Unit → ResetCounter
```

Since `ResetCounter` has all the fields of `Counter` (plus one more), the record subtyping rule tells us that `ResetCounter <: Counter`. This means that client functions like `inc3` that take counters as arguments can also safely be used with reset counters:

```
rc = newResetCounter unit;

▸ rc : ResetCounter

(inc3 rc; rc.reset unit; inc3 rc; rc.get unit);

▸ 4 : Nat
```

18.5 Grouping Instance Variables

So far, the states of our objects have consisted of just a single reference cell. Obviously, more interesting objects will often have several instance variables. In the sections that follow, it will be useful to be able to manipulate all of these instance variables as a single unit. To allow for this, let's change the internal representation of our counters to be a *record* of reference cells, and refer to instance variables in method bodies by projecting fields from this record.

```
c = let r = {x=ref 1} in
      {get = λ_:Unit. !(r.x),
       inc = λ_:Unit. r.x:=succ(!(r.x))};

▸ c : Counter
```

The type of this record of instance variables is called the *representation type* of the object.

```
CounterRep = {x: Ref Nat};
```

18.6 Simple Classes

The definitions of `newCounter` and `newResetCounter` are identical except for the `reset` method in the latter. Of course, both of these definitions are so short anyway that this makes little difference, but if we imagine them stretching over many pages, as can easily happen in practice, it is clear that we would prefer to have some means for describing the common functionality in one place. The mechanism by which this is achieved in most object-oriented languages is called *classes*.

The class mechanisms in real-world object-oriented languages tend to be complex and loaded with features—`self`, `super`, visibility annotations, static fields and methods, inner classes, friend classes, annotations such as `final` and `Serializable`, etc., etc.[2] We'll ignore most of these here and focus our attention on the most basic aspects of classes: code reuse via inheritance, and the late binding of `self`. For the moment, let's consider just the former.

In its most primitive form, a class is simply a data structure holding a collection of methods that can either be *instantiated* to yield a fresh object or *extended* to yield another class.

Why can't we just reuse the methods from some counter *object* to build a reset counter? Simply because, in any particular counter object, the method bodies contain references to some particular record of instance variables. Clearly, if we want to be able to reuse the same code with a different record of instance variables, what we need to do is to *abstract* the methods with respect to the instance variables. This amounts to breaking our `newCounter` function above into two pieces, one that defines the method bodies with respect to an *arbitrary* record of instance variables,

```
counterClass =
  λr:CounterRep.
    {get = λ_:Unit. !(r.x),
     inc = λ_:Unit. r.x:=succ(!(r.x))};
▸ counterClass : CounterRep → Counter
```

and one that allocates a record of instance variables and supplies it to the method bodies to create an object:

```
newCounter =
  λ_:Unit. let r = {x=ref 1} in
             counterClass r;
```

2. The main reason for all this complexity is that, in most of these languages, classes are the *only* large-scale structuring mechanism. Indeed, there is just one widely used language—OCaml—that provides both classes and a sophisticated module system. So classes in most languages tend to become the dumping ground for all language features that have anything to do with large-scale program structure.

```
▸ newCounter : Unit → Counter
```

The method bodies from `counterClass` can be reused to define new classes, called *subclasses*. For example, we can define a class of reset counters:

```
resetCounterClass =
  λr:CounterRep.
    let super = counterClass r in
      {get   = super.get,
       inc   = super.inc,
       reset = λ_:Unit. r.x:=1};
▸ resetCounterClass : CounterRep → ResetCounter
```

Like `counterClass`, this function takes a record of instance variables and returns an object. Internally, it works by first using `counterClass` to create a counter object with the *same* record of instance variables `r`; this "parent object" is bound to the variable `super`. It then builds a new object by copying the `get` and `inc` fields from `super` and supplying a new function as the value for the `reset` field. Since `super` was built on `r`, all three methods share the same instance variables.

To build a reset counter object, we again just allocate memory for its instance variables and call `resetCounterClass`, where the real work happens.

```
newResetCounter =
  λ_:Unit. let r = {x=ref 1} in resetCounterClass r;
▸ newResetCounter : Unit → ResetCounter
```

18.6.1 EXERCISE [RECOMMENDED, ⋆⋆]: Write a subclass of `resetCounterClass` with an additional method `dec` that subtracts one from the current value stored in the counter. Use the `fullref` checker to test your new class. □

18.6.2 EXERCISE [⋆⋆ ↛]: The explicit copying of most of the superclass fields into the record of subclass methods is notationally rather clumsy—it avoids explicitly re-entering the code of superclass methods in subclasses, but it still involves a lot of typing. If we were going to develop larger object-oriented programs in this style, we would soon wish for a more compact notation like "`super with {reset = λ_:Unit. r.x:=1}`," meaning "a record just like `super` but with a field `reset` containing the function `λ_:Unit. r.x:=1`." Write out syntax, operational semantics, and typing rules for this construct. □

We should emphasize that these classes are *values,* not *types*. Also we can, if we like, create many classes that generate objects of exactly the same type. In mainstream object-oriented languages like C++ and , classes have a more complex status—they are used both as compile-time types and as run-time data structures. This point is discussed further in §19.3.

18.7 Adding Instance Variables

It happens that our counter and reset counter objects use exactly the same internal representation. However, in general a subclass may need to extend not only the methods but also the instance variables of the superclass from which it is derived. For example, suppose we want to define a class of "backup counters" whose `reset` method resets their state to whatever value it has when we last called the method `backup`, instead of resetting it to a constant value:

```
BackupCounter = {get:Unit→Nat, inc:Unit→Unit,
                 reset:Unit→Unit, backup: Unit→Unit};
```

To implement backup counters, we need an extra instance variable to store the backed-up value of the state.

```
BackupCounterRep = {x: Ref Nat, b: Ref Nat};
```

Just as we derived `resetCounterClass` from `counterClass` by copying the `get` and `inc` methods and adding `reset`, we derive `backupCounterClass` from `resetCounterClass` by copying `get` and `inc` and providing `reset` and `backup`.

```
backupCounterClass =
  λr:BackupCounterRep.
    let super = resetCounterClass r in
       {get    = super.get,
        inc    = super.inc,
        reset  = λ_:Unit. r.x:=!(r.b),
        backup = λ_:Unit. r.b:=!(r.x)};

▸ backupCounterClass : BackupCounterRep → BackupCounter
```

Two things are interesting about this definition. First, although the parent object `super` includes a method `reset`, we chose to write a fresh implementation because we wanted a different behavior. The new class *overrides* the `reset` method of the superclass. Second, subtyping is used in an essential way here in typing the expression that builds `super`: `resetCounterClass` expects an argument of type `CounterRep`, which is a supertype of the actual type, `BackupCounterRep`, of the argument `r`. In other words, we are actually providing the parent object with a larger record of instance variables than its methods require.

18.7.1 EXERCISE [RECOMMENDED, ⋆⋆]: Define a subclass of `backupCounterClass` with two new methods, `reset2` and `backup2`, controlling a second "backup

register." This register should be completely separate from the one added by `backupCounterClass`: calling `reset` should restore the counter to its value at the time of the last call to `backup` (as it does now) and calling `reset2` should restore the counter to its value at the time of the last call to `backup2`. Use the `fullref` checker to test your class. □

18.8 Calling Superclass Methods

The variable `super` has been used to copy functionality from superclasses into new subclasses. We can also use `super` in the bodies of method definitions to *extend* the superclass's behavior with something extra. Suppose, for instance, that we want a variant of our `backupCounter` class in which every call to the `inc` method is automatically preceded by a call to `backup`. (Goodness knows why such a class would be useful—it's just an example.)

```
funnyBackupCounterClass =
  λr:BackupCounterRep.
    let super = backupCounterClass r in
       {get = super.get,
        inc = λ_:Unit. (super.backup unit; super.inc unit),
        reset = super.reset,
        backup = super.backup};

▸ funnyBackupCounterClass : BackupCounterRep → BackupCounter
```

Note how the calls to `super.inc` and `super.backup` in the new definition of `inc` avoid repeating the superclass's code for `inc` or `backup` here. In larger examples, the savings of duplicated functionality in such situations can be substantial.

18.9 Classes with Self

Our final extension is allowing the methods of classes to refer to each other via `self`. To motivate this extension, suppose that we want to implement a class of counters with a `set` method that can be used from the outside to set the current count to a particular number.

```
SetCounter = {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit};
```

Moreover, suppose that we want the `inc` method in terms of `set` and `get`, rather than directly reading and assigning the instance variable `x`. (Imagine a much larger example in which the definitions of `set` and `get` each take many

pages.) Since `get`, `set`, and `inc` are all defined in the same class, what we are asking for is essentially to make the methods of this class mutually recursive.

We saw how to build mutually recursive records of functions using the `fix` operator in §11.11. We simply abstract the record of methods on a parameter that is itself a record of functions (which we call `self`), and then use the `fix` operator to "tie the knot," arranging that the very record we are constructing is the one passed as `self`.

```
setCounterClass =
  λr:CounterRep.
    fix
      (λself: SetCounter.
        {get = λ_:Unit. !(r.x),
         set = λi:Nat.  r.x:=i,
         inc = λ_:Unit. self.set (succ (self.get unit))});

▸ setCounterClass : CounterRep → SetCounter
```

This class has no parent class, so there is no need for a `super` variable. Instead, the body of the `inc` method invokes `get` and then `set` from the record of methods passed to it as `self`. This use of `fix` is entirely internal to `setCounterClass`. We then create set-counters in exactly the same way as usual.

```
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
              setCounterClass r;

▸ newSetCounter : Unit → SetCounter
```

18.10 Open Recursion through Self

Most object-oriented languages actually support a more general form of recursive call between methods, known as *open recursion,* or *late-binding* of `self`. We can achieve this more general behavior by removing the use of `fix` from the class definition,

```
setCounterClass =
  λr:CounterRep.
     λself: SetCounter.
       {get = λ_:Unit. !(r.x),
        set = λi:Nat.  r.x:=i,
        inc = λ_:Unit. self.set (succ(self.get unit))};

▸ setCounterClass : CounterRep → SetCounter → SetCounter
```

and instead placing it in the object creation function.

```
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
              fix (setCounterClass r);
▸ newSetCounter : Unit → SetCounter
```

Notice that moving the use of `fix` changes the type of `setCounterClass`: instead of being abstracted just on a record of instance variables, it is also abstracted on a "`self`-object"; both are supplied at instantiation time.

The reason why open recursion through `self` is interesting is that it allows the methods of a *superclass* to call the methods of a *subclass,* even though the subclass does not exist when the superclass is being defined. In effect, we have changed the interpretation of `self` so that, instead of "the methods of this class," it provides access to "the methods of the class from which the current object was instantiated [which may be a subclass of this one]."

For example, suppose we want to build a subclass of our set-counters that keeps track of how many times the `set` method has been called. The interface of this class includes one extra operation for extracting the access count,

```
InstrCounter = {get:Unit→Nat, set:Nat→Unit,
                inc:Unit→Unit, accesses:Unit→Nat};
```

and the representation includes an instance variable for the access count:

```
InstrCounterRep = {x: Ref Nat, a: Ref Nat};
```

In the definition of the instrumented counter class, the `inc` and `get` methods are copied from the `setCounterClass` that we defined above. The `accesses` method is written out in the ordinary way. In the `set` method, we first increment the access count and then use `super` to invoke the superclass's `set`.

```
instrCounterClass =
  λr:InstrCounterRep.
     λself: InstrCounter.
       let super = setCounterClass r self in
         {get = super.get,
          set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
          inc = super.inc,
          accesses = λ_:Unit. !(r.a)};
▸ instrCounterClass : InstrCounterRep →
                      InstrCounter → InstrCounter
```

Because of the open recursion through `self`, the call to `set` from the body of `inc` will result in the instance variable `a` being incremented, even though the incrementing behavior of `set` is defined in the subclass and the definition of `inc` appears in the superclass.

18.11 Open Recursion and Evaluation Order

There is one problem with our definition of `instrCounterClass`—we cannot use it to build instances! If we write `newInstrCounter` in the usual way

```
newInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0} in
              fix (instrCounterClass r);

▸ newInstrCounter : Unit → InstrCounter
```

and then attempt to create an instrumented counter by applying it to `unit`,

```
ic = newInstrCounter unit;
```

the evaluator will diverge. To see how this happens, consider the sequence of evaluation steps that ensue when we start from this term.

1. We first apply `newInstrCounter` to `unit`, yielding

   ```
   let r = {x=ref 1, a=ref 0} in fix (instrCounterClass r)
   ```

2. We next allocate two `ref` cells, package them into a record—let's call it `<ivars>`—and substitute `<ivars>` for `r` in the rest.

   ```
   fix (instrCounterClass <ivars>)
   ```

3. We pass `<ivars>` to `instrCounterClass`. Since `instrCounterClass` begins with two lambda-abstractions, we immediately get back a function that is waiting for `self`,

   ```
   fix (λself:InstrCounter.
          let super = setCounterClass <ivars> self in <imethods>)
   ```

   where `<imethods>` is the record of instrumented-counter methods. Let's call this function `<f>` and write the current state `(fix <f>)`.

4. We apply the evaluation rule for `fix` (E-FIX in Figure 11-12, page 144), which "unfolds" `fix <f>` by substituting `(fix <f>)` for `self` in the body of `<f>`, yielding

   ```
   let super = setCounterClass <ivars> (fix <f>) in <imethods>
   ```

5. We now reduce the application of `setCounterClass` to `<ivars>`, yielding:

   ```
   let super = (λself:SetCounter. <smethods>) (fix <f>)
               in <imethods>
   ```

where <smethods> is the record of set-counter methods.

6. By the evaluation rules for applications, we cannot reduce the application of (λself:SetCounter. <smethods>) to (fix <f>) until the latter has been reduced to a value. So the next step of evaluation again unfolds fix <f>, yielding:

```
let super = (λself:SetCounter. <smethods>)
               (let super = setCounterClass <ivars> (fix <f>)
                in <imethods>)
in <imethods>
```

7. Since the argument to the outer lambda-abstraction is still not a value, we must continue to work on evaluating the inner one. We perform the application of setCounterClass to <ivars>, yielding

```
let super = (λself:SetCounter. <smethods>)
               (let super = (λself:SetCounter. <smethods>)
                            (fix <f>)
                in <imethods>)
in <imethods>
```

8. Now we have created an inner application similar in form to the outer one. Just as before, this inner application cannot be reduced until its argument, fix <f>, has been fully evaluated. So our next step is again to unfold fix <f>, yielding a yet more deeply nested expression of the same form as in step 6. It should be clear at this point that we are *never* going to get around to evaluating the outer application.

Intuitively, the problem here is that the argument to the fix operator is using its own argument, self, too early. The operational semantics of fix is defined with the expectation that, when we apply fix to some function λx.t, the body t should refer to x only in "protected" positions, such as the bodies of inner lambda-abstractions. For example, we defined iseven on page 143 by applying fix to a function of the form λie. λx. ..., where the recursive reference to ie in the body was protected by the abstraction on x. By contrast, the definition of instrCounterClass tries to use self right away in calculating the value of super.

At this point, we can proceed in several ways:

- We can protect the reference to self in instrCounterClass to prevent it from being evaluated too early, for example by inserting dummy lambda-abstractions. We develop this solution below. We will see that it is not completely satisfactory, but it is straightforward to describe and understand

using the mechanisms we have already seen. We will also find it useful later, when we consider purely functional object encodings in Chapter 32.

- We can look for different ways of using low-level language features to model the semantics of classes. For example, instead of using `fix` to build the method table of a class, we could build it more explicitly using references. We develop this idea in §18.12 and further refine it in Chapter 27.
- We can forget about encoding objects and classes in terms of lambda-abstraction, records, and `fix`, and instead take them as language primitives, with their own evaluation (and typing) rules. Then we can simply choose evaluation rules that match our intentions about how objects and classes should behave, rather than trying to work around the problems with the given rules for application and `fix`. This approach will be developed in Chapter 19.

Using dummy lambda-abstractions to control evaluation order is a well-known trick in the functional programming community. The idea is that an arbitrary expression `t` can be turned into a function `λ_:Unit.t`, called a *thunk*. This "thunked form" of `t` is a syntactic value; all the computation involved in evaluating `t` is postponed until the thunk is applied to `unit`. This gives us a way to pass `t` around in unevaluated form and, later, ask for its result.

What we want to do at the moment is to delay the evaluation of `self`. We can do this by changing its type from an object (e.g. `SetCounter`) to an object thunk (`Unit→SetCounter`). This involves (1) changing the type of the `self` parameter to the class, (2) adding a dummy abstraction before we construct the result object, and (3) changing every occurrence of `self` in the method bodies to `(self unit)`.

```
setCounterClass =
  λr:CounterRep.
  λself: Unit→SetCounter.
    λ_:Unit.
      {get = λ_:Unit. !(r.x),
       set = λi:Nat.  r.x:=i,
       inc = λ_:Unit. (self unit).set(succ((self unit).get unit))};
▸ setCounterClass : CounterRep →
                    (Unit→SetCounter) → Unit → SetCounter
```

Since we do not want the type of `newSetCounter` to change (it should still return an object), we also need to modify its definition slightly so that it passes a `unit` argument to the thunk that results when we form the fixed point of `setCounterClass`.

```
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
              fix (setCounterClass r) unit;
```

```
▸ newSetCounter : Unit → SetCounter
```

Similar changes are needed in the definition of `instrCounterClass`. Note that none of these modifications actually require any thinking: once we have changed the type of `self`, every other change is dictated by the typing rules.

```
instrCounterClass =
  λr:InstrCounterRep.
  λself: Unit→InstrCounter.
    λ_:Unit.
      let super = setCounterClass r self unit in
         {get = super.get,
          set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
          inc = super.inc,
          accesses = λ_:Unit. !(r.a)};
```

```
▸ instrCounterClass : InstrCounterRep →
                      (Unit→InstrCounter) → Unit → InstrCounter
```

Finally, we change `newInstrCounter` so that it supplies a dummy argument to the thunk constructed by `fix`.

```
newInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0} in
              fix (instrCounterClass r) unit;
```

```
▸ newInstrCounter : Unit → InstrCounter
```

We can now use `newInstrCounter` to actually build an object.

```
ic = newInstrCounter unit;
```

```
▸ ic : InstrCounter
```

Recall that this was the step that diverged before we added thunks.

The following tests demonstrate how the `accesses` method counts calls to both `set` and `inc`, as we intended.

```
(ic.set 5; ic.accesses unit);
```

```
▸ 1 : Nat
```

```
(ic.inc unit; ic.get unit);
```

```
▸ 6 : Nat
```

```
ic.accesses unit;
▸ 2 : Nat
```

18.11.1 EXERCISE [RECOMMENDED, ⋆⋆⋆]: Use the `fullref` checker to implement the following extensions to the classes above:

1. Rewrite `instrCounterClass` so that it also counts calls to `get`.
2. Extend your modified `instrCounterClass` with a subclass that adds a `reset` method, as in §18.4.
3. Add another subclass that also supports backups, as in §18.7. □

18.12 A More Efficient Implementation

The tests above demonstrate that our implementation of classes matches the "open recursion" behavior of method calls through `self` in languages like Smalltalk, C++, and Java. However, we should note that the implementation is not entirely satisfactory from the point of view of efficiency. All the thunks we have inserted to make the `fix` calculation converge have the effect of postponing the calculation of the method tables of classes. In particular, note that all the calls to `self` inside method bodies have become `(self unit)`—that is, the methods of `self` are being recalculated on the fly every time we make a recursive call to one of them!

We can avoid all this recalculation by using reference cells instead of fixed points to "tie the knot" in the class hierarchy when we build objects.[3] Instead of abstracting classes on a record of methods called `self` that will be constructed later using `fix`, we abstract on a *reference* to a record of methods, and allocate this record *first*. That is, we instantiate a class by first allocating a heap cell for its methods (initialized with a dummy value), then constructing the real methods (passing them a pointer to this heap cell, which they can use to make recursive calls), and finally back-patching the cell to contain the real methods. For example, here is `setCounterClass` again.

```
setCounterClass =
  λr:CounterRep. λself: Ref SetCounter.
    {get = λ_:Unit. !(r.x),
     set = λi:Nat. r.x:=i,
     inc = λ_:Unit. (!self).set (succ ((!self).get unit))};
```

3. This is essentially the same idea as we used in the solution to Exercise 13.5.8. I am grateful to James Riely for the insight that it can be applied to class construction by exploiting the covariance of `Source` types.

```
▸ setCounterClass : CounterRep → (Ref SetCounter) → SetCounter
```

The `self` parameter is a pointer to the cell that contains the methods of the current object. When `setCounterClass` is called, this cell is initialized with a dummy value:

```
dummySetCounter =
  {get = λ_:Unit. 0,
   set = λi:Nat.  unit,
   inc = λ_:Unit. unit};

▸ dummySetCounter : SetCounter

newSetCounter =
  λ_:Unit.
    let r = {x=ref 1} in
    let cAux = ref dummySetCounter in
    (cAux := (setCounterClass r cAux); !cAux);

▸ newSetCounter : Unit → SetCounter
```

However, since all of the dereference operations (`!self`) are protected by lambda-abstractions, the cell will not actually be dereferenced until after it has been back-patched by `newSetCounter`.

To support building subclasses of `setCounterClass`, we need to make one further refinement in its type. Each class expects its `self` parameter to have the same type as the record of methods that it constructs. That is, if we define a subclass of instrumented counters, then the `self` parameter of this class will be a pointer to a record of instrumented counter methods. But, as we saw in §15.5, the types `Ref SetCounter` and `Ref InstrCounter` are incompatible—it is unsound to promote the latter to the former. This will lead to trouble (i.e., a parameter type mismatch) when we try to create `super` in the definition of `instrCounterClass`.

```
instrCounterClass =
  λr:InstrCounterRep. λself: Ref InstrCounter.
  let super = setCounterClass r self in
  {get = super.get,
   set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
   inc = super.inc,
   accesses = λ_:Unit. !(r.a)};

▸ Error: parameter type mismatch
```

The way out of this difficulty is to replace the `Ref` constructor in the type of `self` by `Source`—i.e., to pass to the class just the capability to read from

the method pointer, not the capability to write to it (which it does not need anyway). As we saw in §15.5, `Source` permits covariant subtyping—i.e., we have `Ref InstrCounter <: Ref SetCounter`—so the creation of `super` in `instrCounterClass` becomes well typed.

```
setCounterClass =
  λr:CounterRep. λself: Source SetCounter.
    {get = λ_:Unit. !(r.x),
     set = λi:Nat. r.x:=i,
     inc = λ_:Unit. (!self).set (succ ((!self).get unit))};

▸ setCounterClass : CounterRep → (Source SetCounter) → SetCounter

instrCounterClass =
  λr:InstrCounterRep. λself: Source InstrCounter.
  let super = setCounterClass r self in
  {get = super.get,
   set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
   inc = super.inc,
   accesses = λ_:Unit. !(r.a)};

▸ instrCounterClass : InstrCounterRep →
                      (Source InstrCounter) → InstrCounter
```

To build an instrumented counter object, we first define a dummy collection of instrumented counter methods, as before, to serve as the initial value of the `self` pointer.

```
dummyInstrCounter =
  {get = λ_:Unit. 0,
   set = λi:Nat.  unit,
   inc = λ_:Unit. unit,
   accesses = λ_:Unit. 0};

▸ dummyInstrCounter : InstrCounter
```

We then create an object by allocating heap space for the instance variables and methods, calling `instrCounterClass` to construct the actual methods, and back-patching the reference cell.

```
newInstrCounter =
  λ_:Unit.
    let r = {x=ref 1, a=ref 0} in
    let cAux = ref dummyInstrCounter in
    (cAux := (instrCounterClass r cAux); !cAux);

▸ newInstrCounter : Unit → InstrCounter
```

The code for constructing the method table (in `instrCounterClass` and `counterClass`) is now called once per object creation, rather than once per method invocation. This achieves what we set out to do, but it is still not quite as efficient as we might wish: after all the method table that we construct for each instrumented counter object is always exactly the same, so it would seem we should be able to compute this method table just *once,* when the class is defined, and never again. We will see in Chapter 27 how this can be accomplished using the *bounded quantification* introduced in Chapter 26.

18.13 Recap

The first section of this chapter listed several characteristic features of the object-oriented programming style. Let us recall these features and briefly discuss how they relate to the examples developed in the chapter.

1. **Multiple representations.** All of the objects that we have seen in this chapter are counters—i.e., they belong to the type `Counter`. But their representations vary widely, from single reference cells in §18.2 to records containing several references in §18.9. Each object is a record of functions, providing implementations of the `Counter` methods (and perhaps others) appropriate to its own internal representation.

2. **Encapsulation.** The fact that the instance variables of an object are accessible only from its methods follows directly from the way we build objects, building the methods by passing the record of instance variables to a constructor function. It is obvious that the instance variables can be *named* only from inside the methods.

3. **Subtyping.** In this setting, subtyping between object types is just ordinary subtyping between types of records of functions.

4. **Inheritance.** We modeled inheritance by copying implementations of methods from an existing superclass to a newly defined subclass. There were a few interesting technicalities here: strictly speaking, both the superclass and the new subclass are *functions* from instance variables to records of methods. The subclass waits for its instance variables, then instantiates the superclass with the instance variables it is given, forming a record of superclass methods operating on the same variables.

5. **Open recursion.** The open recursion provided by `self` (or `this`) in real-world object-oriented languages is modeled here by abstracting classes not only on instance variables but also on a `self` parameter, which can

be used in methods to refer to other methods of the same object. This parameter is resolved at object-creation time by using `fix` to "tie the knot."

18.13.1 EXERCISE [⋆⋆⋆]: Another feature of objects that is useful for some purposes is a notion of *object identity*—an operation `sameObject` that yields `true` if its two arguments evaluate to the very same object, and `false` if they evaluate to objects that were created at different times (by different calls to `new` functions). How might the model of objects in this chapter be extended to support object identity? □

18.14 Notes

Object encodings are a staple source of examples and problems for the programming languages research community. An early encoding was given by Reynolds (1978); general interest in the area was sparked by an article by Cardelli (1984). The understanding of `self` in terms of fixed points was developed by Cook (1989), Cook and Palsberg (1989), Kamin (1988), and Reddy (1988); relations between these models were explored by Kamin and Reddy (1994) and Bruce (1991).

A number of important early papers in the area are collected in Gunter and Mitchell (1994). Later developments are surveyed by Bruce, Cardelli, and Pierce (1999) and by Abadi and Cardelli (1996). Bruce (2002) gives an up-to-date picture of progress in the area. Alternative foundational approaches to objects and their type systems can be found in Palsberg and Schwartzbach (1994) and Castagna (1997).

Some additional historical notes can be found at the end of Chapter 32.

Inheritance is highly overrated. —*Grady Booch*