5 MODELING OBJECT-ORIENTED LANGUAGES

In the previous chapters we have presented a relatively unbiased description of a large (although not exhaustive) collection of object-oriented features. We now begin to discuss our way of modeling those features.

5.1 Reduction to Basic Mechanisms

In our review of object-oriented languages we have examined differences and similarities between various object-oriented features. It should be apparent that, despite all the variations, many characteristics of class-based and object-based languages are just different presentations of a few general ideas. A natural question is whether these different presentations can be unified into a single conceptual and formal framework.

We can imagine two possible ways of encompassing the variety of object-oriented languages within a single framework. One approach is to create an object model so general that it can express directly all the conceivable variations of object classification and inheritance. Although such a model could probably be built, it would not be very useful because of either its complexity or its excessive generality. A different approach is to create a simple model that is flexible enough to represent more complex notions, but that does not directly embody any particular one. This is the approach that has resulted in a better understanding of procedural languages, via $\lambda$-calculi. This is also the approach that we adopt for object-oriented languages in this book. As we discuss later in Section 6.7 and Chapter 18, $\lambda$-calculi are not completely satisfactory as a foundation for object-oriented languages. By substituting object calculi for $\lambda$-calculi, we obtain the desired modeling power.

The object calculi that we develop are formalisms at the same level of abstraction as $\lambda$-calculi, but based exclusively on objects rather than functions. Much as in the development of $\lambda$-calculi, we start with a minimum untyped kernel and enrich it with derived constructs and with increasingly sophisticated type systems until language features can be realistically modeled. In the course of this book we investigate untyped, simply-typed, and polymorphic calculi, considering both functional and imperative execution models. A common kernel of object primitives provides conceptual unity to the whole family of calculi.

Object calculi can be seen as the result of reducing class-based and object-based languages to basic mechanisms. Class-based languages integrate many ideas into a single construct: the class. Much of the effort in understanding and analyzing class-based languages goes into distinguishing the contributions of those ideas. Object-based languages decompose class-based features in order to simplify them, and reconstruct
them in different ways. The reconstruction is not always as comfortable as the original. Still, this decomposition provides an explanation for class-based mechanisms and suggests possible variations. In our technical development we decompose object-based languages further into object calculi. The emulation of original object-based or class-based features may not be straightforward and convenient. The raw expressive power is, however, the same. By reduction to a few powerful primitives we gain a clearer semantics and we simplify the task of reasoning formally.

5.2 The Role of Method Update

In our search for minimal object calculi, it seems natural to take objects to be collections of methods. Fields are important too, but they can be seen as a derived concept; for example, a field can be viewed as a method that does not use its self parameter. The hiding of fields from public view has been advocated as a means of concealing representation choices, and thereby allowing flexibility in implementation; identifying fields with methods confers much of the same flexibility. When fields and methods are identified it is trivial to convert one into the other, conceptually turning passive data into active computation, and vice versa.

The unification of fields with methods has the advantage of simplicity: both objects and object operations assume a uniform structure. In contrast, the separation of fields from methods induces a corresponding separation of object operations, and leads to the implicit or explicit splitting of self into two components. Unifying fields with methods gives more compact and therefore more elegant calculi.

This unification, however, has one debatable consequence. The natural operation on methods is method invocation, and the natural operations on fields are field selection and field update. By unifying fields with methods, we can collapse field selection and method invocation into a single operation. To complete the unification, though, we are led to generalize field update to method update.28 At the lowest level of formalization, the choice is therefore between distinguishing methods from fields or adopting method update. We choose the latter.

The reliance on method update is one of the most unusual aspects of our formal treatment, since this operation is not so common in programming languages. However, method update can be seen as a form of dynamic inheritance: it supports the dynamic modification of object behavior, allowing objects, in a sense, to change their class dynamically. Thus method update gives us an edge in modeling object-based constructs, in addition to allowing us to model the more traditional class-based constructs where fields and methods are sharply separated.

A further justification for method update can be found in the desire to tame dynamic inheritance. Dynamic inheritance is a unique feature of object-based lan-

28 The language Self unifies field update with method invocation; clients perform only method invocations. But in objects there is still a distinction between so-called data slots and assignment slots.