4 OBJECT-BASED LANGUAGES

The main features of class-based languages appeared fully formed in Simula. Object-based languages, in contrast, have emerged more gradually. Their main principles were first gathered in the Treaty of Orlando [114]; to a large extent, even their most basic notions are still evolving [118].

Object-based languages are intended to be both simpler and more flexible than traditional class-based languages. Many object-based languages originated in the Lisp, Smalltalk, and artificial intelligence communities, where extreme flexibility is highly valued [11, 31, 32, 78, 80]. As a consequence, little attention has been devoted to designing typed object-based languages, except for simple ones such as Emerald, and for recent ones such as Cecil and Omega. In this chapter we review the main features of object-based languages, considering issues of inheritance and typing.

4.1 Objects without Classes

Object-based languages are refreshing in their simplicity, compared with even the simplest class-based languages. In their minimal form, as in Emerald and in simple closure-based encodings [11], they support only the notions of objects and dynamic dispatch. When typed, they support only object types, subtyping, and subsumption.

The basic characteristics of object-based languages are the absence of classes and the existence of constructs for the creation of individual objects. Since there are no classes it is natural to introduce object types immediately, and to separate object interfaces from object implementations. For example, we may have:

```
ObjectType Cell is
  var contents: Integer;
  method get(): Integer;
  method set(n: Integer);
end;

object cell: Cell is
  var contents: Integer := 0;
  method get(): Integer is return self.contents end;
  method set(n: Integer) is self.contents := n end;
end;
```

This object definition creates a single object of type Cell and names it cell.
Even without classes, we can generate a uniform collection of objects by a procedure that, whenever invoked, returns a new object. Such a procedure can have parameters, for example to initialize fields:

```plaintext
procedure newCell(m: Integer): Cell is
  object cell: Cell is
    var contents: Integer := m;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
  end;
  return cell;
end;

var cellInstance: Cell := newCell(0);
```

Such object-generating procedures are uncannily similar to Simula classes, which also have parameters and an executable body. The role of `new` is taken over by procedure call.

The main insight of the object-based model is that class-based notions need not be assumed, but instead can be emulated by more primitive notions. Moreover, these more primitive notions can be combined in more flexible ways than in a strict class discipline. A similar reduction of classes to objects will appear in our formal treatment of object calculi.

### 4.2 Prototypes and Clones

As we have just seen, procedures may be used for generating objects. However, it may be difficult or inconvenient to anticipate all the possible ways in which objects should be parameterized. The so-called **prototype-based** languages adopt a different approach to object generation. Instead of parameterizing objects beforehand, they generate stock objects from prototypical objects, and customize the stock objects later [32, 79).

In class-based languages, object descriptions provide the templates from which object instances are generated. In prototype-based languages, instead, concrete and fully functional instances are built first. Some of these instances may be later interpreted as canonical representatives (prototypes) of classes of instances. Additional instances are derived from the prototypes. There is no entity that represents a class of instances, other than by convention.

The basic mechanism used for instantiating prototypes is **cloning**. Cloning produces a shallow copy of an object; that is, a copy of the outermost object structure, sharing all the attribute values of the original object, but with independent state.

```plaintext
var cellClone: Cell := clone cellInstance;
```

Cloning is a bit like `new`, but operates on objects instead of classes. Any object can act both as an instance and as a prototype for further cloning. An object in the role of a