Messages will still ping-pong indefinitely, but how? Which messages will be sent and how will the object executions be interleaved? Will the interleaving be in lock-step (alternating between objects strictly), looser (subject to fluctuations due to thread scheduling), or something in between?

2. **Lift control system**. Section 5.4.4 gives the design of a simple lift control system. Let us explore it:

   - The current design has one controller object per lift. To economize on costs, the developer decides to change this to keep just one controller for the whole system. Each lift then communicates with this controller. The controller's internal definition stays the same. Is this a good idea? How does it change the behavior of the lift control system?
   - In the current design, the controller steps up or down one floor at a time. It stops at all floors that it passes, even if the floor was not requested. Change the lift and controller objects to avoid this jumpy behavior by stopping only at requested floors.

3. **Fault tolerance for the lift control system**. There are two kinds of faults that can happen: components can be blocked temporarily or they can be permanently out of order. Let us see how to handle each case:

   - A lift is blocked. Extend the system to continue working when a lift is temporarily blocked at a floor by a malicious user. First extend the floor to reset the door timer when the floor is called while the doors are open. Then the lift's schedule should be given to other lifts and the floors should no longer call that particular lift. When the lift works again, floors should again be able to call the lift. This can be done with time-outs.
   - A lift is out of order. The first step is to add generic primitives for failure detection. We might need both synchronous and asynchronous detection. In synchronous detection, when a component goes down, we assume that any message sent to it gets the immediate reply `down(Id)`, where `Id` identifies the component. In asynchronous detection, we "link" a component to another when they are both still working. Then, when the second component crashes, the `down` message is sent to the first one immediately. Now extend the system to continue working when a lift is out of order. The system should reconfigure itself to continue working for a building with one less lift.
   - A floor is out of order. Extend the system to continue working when a floor is out of order. The system should reconfigure itself to continue working for a building with one less floor.
   - Lift maintenance. Extend the system so that a lift can be brought down for maintenance and brought back up again later.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

- Interactions. What happens if several floors and lifts become out of order simultaneously? Does your system handle this properly?

4. **Termination detection**. Replace definition of `SubThread` in Section 5.5.3 by:

```
proc {SubThread P}
   thread
      {Send Pt 1} {P} {Send Pt ~1}
   end
end
```

   Explain why the result is not correct. Give an execution such that there exists a point where the sum of the elements on the port's stream is zero, yet all threads have not terminated.

5. **Concurrent filter**. Section 5.5.4 defines a concurrent version of `Filter`, called `ConcFilter`, that calculates each output element independently, i.e., without waiting for the previous ones to be calculated.

   (a) What happens when the following is executed:

```
declare Out
{ConcFilter [5 1 2 4 0] fun {$ X} X>2 end Out}
{Show Out}
```

   How many elements are displayed by the `Show` What is the order of the displayed elements? If several displays are possible, give all of them. Is the execution of `ConcFilter` deterministic? Why or why not?

   (b) What happens when the following is executed:

```
declare Out
{ConcFilter [5 1 2 4 0] fun {$ X} X>2 end Out}
{Delay 1000}
{Show Out}
```

   What is displayed now by `Show`? If several displays are possible, give all of them.

   (c) What happens when the following is executed:

```
declare Out A
{ConcFilter [5 1 A 4 0] fun {$ X} X>2 end Out}
{Delay 1000}
{Show Out}
```

   What is displayed now? What is the order of the displayed elements? If, after the above, `A` is bound to 3, then what happens to the list `Out`?

   (d) If the input list has n elements, what is the complexity (in "big-Oh" notation) of the number of operations of `ConcFilter`? Discuss the difference in execution time between `Filter` and `ConcFilter`.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

6. **Semantics of Erlang's receive**. Section 5.6.3 shows how to translate Erlang's `receive` operation. The second form of this operation, with time out, is the most general one. Let us take a closer look.

   (a) Verify that the second form reduces to the third form, when the time out delay is zero.

   (b) Verify that the second form reduces to the first form, when the time out delay approaches infinity.

   (c) Another way to translate the second form would be to insert a unique message (using a name) after n milliseconds. This requires some care to keep the unique message from appearing in the output stream. Write another translation of the third form that uses this technique. What are the advantages and disadvantages of this translation with respect to the one in the book?

7. **Erlang's receive as a control abstraction**. For this exercise, implement the Erlang `receive` operation, which is defined in Section 5.6.3, as the following control abstraction:

   - `C={Mailbox.new}` creates a new mailbox `C`.
   - `{Mailbox.send C M}` sends message `M` to mailbox `C`.
   - `{Mailbox.receive C [P1#E1 P2#E2 ... Pn#En] D}` performs a receive on mailbox `C`. `Pi` is a one-argument boolean function **`fun`** `{$ M}` ⟨expr⟩ **`end`** that represents a pattern and its guard. The function returns **`true`** if and only if the pattern and guard succeed for message `M`. `Ei` is a one-argument function **`fun`** `{$ M}` ⟨expr⟩ **`end`** that represents a body. It is executed when message `M` is received and the receive returns its result. `D` represents the delay. It is either a nonnegative integer giving the delay in milliseconds or the atom `infinity`, which represents an infinite delay.

8. **Limitations of stream communication**. In this exercise, we explore the limits of stream communication in the nondeterministic concurrent model. Section 5.7.1 claims that we can partially solve the problem of putting server references in a data structure. How far can we go? Consider the following active object:

```
declare NS
thread {NameServer NS nil} end
```

   where `NameServer` is defined as follows:

```
proc {NameServer NS L}
   case NS
   of register(A S)|NS1 then
```

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

```
      {NameServer NS1 A#S|L}
   [] getstream(A S)|NS1 then L1 OldS NewS in
      L1={Replace L A OldS NewS}
      thread {StreamMerger S NewS OldS} end
      {NameServer NS1 L1}
   [] nil then
      skip
   end
end

fun {Replace InL A OldS NewS}
   case InL
   of B#S|L1 andthen A=B then
      OldS=S
      A#NewS|L1
   [] E|L1 then
      E|{Replace L1 A OldS NewS}
   end
end
```

The `NameServer` object understands two commands. Assume that `S` is a server's input stream and `foo` is the name we wish to give the server. Given a reference `NS` to the name server's input stream, doing `NS=register(foo S)|NS1` will add the pair `foo#S` to its internal list `L`. Doing `NS=getstream(foo S1)|NS1` will create a fresh input stream, `S1`, for the server whose name is `foo`, which the name server has stored on its internal list `L`. Since `foo` is a constant, we can put it in a data structure. Therefore, it seems that we *can* put server references in a data structure, by defining a name server. Is this a practical solution? Why or why not? Think before reading the answer in the footnote.[10]

[10] It's not possible to name the name server! It has to be added as an extra argument to all procedures. Eliminating this argument needs explicit state.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

Chapter 6

Explicit State

"L'état c'est moi."
"*I am the state.*"
– *Louis XIV* (1638–1715)

"If declarative programming is like a crystal, immutable and practically eternal, then stateful programming is organic: it grows and evolves as we watch."
– Inspired by On Growth and Form, *D'Arcy Wentworth Thompson* (1860–1948)

At first glance, explicit state is just a minor extension to declarative programming: in addition to depending on its arguments, the component's result also depends on an internal parameter, which is called its "state". This parameter gives the component a long-term memory, a "sense of history" if you will.[1] Without state, a component has only short-term memory, one that exists during a particular invocation of the component. State adds a potentially infinite branch to a finitely running program. By this we mean the following. A component that runs for a finite time can only have gathered a finite amount of information. If the component has state, then to this finite information can be added the information stored by the state. This "history" can be indefinitely long, since the component can have a memory that reaches far into the past.

Oliver Sacks has described the case of people with brain damage who only have a short-term memory [161]. They live in a continuous "present" with no memory beyond a few seconds into the past. The mechanism to "fix" short-term memories into the brain's long-term storage is broken. Strange it must be to live in this way. Perhaps these people use the external world as a kind of long-term memory? This analogy gives some idea of how important state can be for people. We will see that state is just as important for programming.

[1] Chapter 5 also introduced a form of long-term memory, the port. It was used to define port objects, active entities with an internal memory. The main emphasis there was on concurrency. The emphasis of this chapter is on the expressiveness of state without concurrency.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

Structure of the chapter

This chapter gives the basic ideas and techniques of using state in program design. The chapter is structured as follows:

- We first introduce and define the concept of explicit state in the first three sections.
  - Section 6.1 introduces explicit state: it defines the general notion of "state", which is independent of any computation model, and shows the different ways that the declarative and stateful models implement this notion.
  - Section 6.2 explains the basic principles of system design and why state is an essential part of system design. It also gives first definitions of component-based programming and object-oriented programming.
  - Section 6.3 precisely defines the stateful computation model.
- We then introduce ADTs with state in the next two sections.
  - Section 6.4 explains how to build abstract data types both with and without explicit state. It shows the effect of explicit state on building secure abstract data types.
  - Section 6.5 gives an overview of some useful stateful ADTs, namely collections of items. It explains the trade-offs of expressiveness and efficiency in these ADTs.
- Section 6.6 shows how to reason with state. We present a technique, the method of invariants, that can make this reasoning almost as simple as reasoning about declarative programs, when it can be applied.
- Section 6.7 explains component-based programming. This is a basic program structuring technique that is important both for very small and very large programs. It is also used in object-oriented programming.
- Section 6.8 gives some case studies of programs that use state, to show more clearly the differences with declarative programs.
- Section 6.9 introduces some more advanced topics: the limitations of stateful programming and how to extend memory management for external references.

Chapter 7 continues the discussion of state by developing a particularly rich programming style, namely object-oriented programming. Because of the wide applicability of object-oriented programming, we devote a full chapter to it.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

A problem of terminology

Stateless and stateful programming are often called declarative and imperative programming, respectively. The latter terms are not quite right, but tradition has kept their use. Declarative programming, taken literally, means programming with *declarations*, i.e., saying *what* is required and letting the system determine *how* to achieve it. Imperative programming, taken literally, means to give *commands*, i.e., to say *how* to do something. In this sense, the declarative model of Chapter 2 is imperative too, because it defines sequences of commands.

The real problem is that "declarative" is not an absolute property, but a matter of degree. The language Fortran, developed in the late 1950's, was the first mainstream language that allowed writing arithmetic expressions in a syntax that resembles mathematical notation [13]. Compared to assembly language this is definitely declarative! One could tell the computer that `I+J` is required without specifying where in memory to store `I` and `J` and what machine instructions are needed to retrieve and add them. In this relative sense, languages have been getting more declarative over the years. Fortran led to Algol-60 and structured programming [46, 45, 130], which led to Simula-67 and object-oriented programming [137, 152].[2]

This book sticks to the traditional usage of declarative as stateless and imperative as stateful. We call the computation model of Chapter 2 "declarative", even though later models are arguably more declarative, since they are more expressive. We stick to the traditional usage because there is an important sense in which the declarative model really is declarative according to the literal meaning. This sense appears when we look at the declarative model from the viewpoint of logic and functional programming:

- A logic program can be "read" in two ways: either as a set of logical axioms (the *what*) or as a set of commands (the *how*). This is summarized by Kowalski's famous equation Program = Logic + Control [106]. The logical axioms, when supplemented by control flow information (either implicit or explicitly given by the programmer), give a program that can be run on a computer. Section 9.3.3 explains how this works for the declarative model.

- A functional program can also be "read" in two ways: either as a definition of a set of functions in the mathematical sense (the *what*) or as a set of commands for evaluating those functions (the *how*). As a set of commands, the definition is executed in a particular order. The two most popular orders are eager and lazy evaluation. When the order is known, the mathematical definition can be run on a computer. Section 4.9.2 explains how this works for the declarative model.

[2] It is a remarkable fact that all three languages were designed in one ten-year period, from approximately 1957 to 1967. Considering that Lisp and Absys, among other languages, also date from this period and that Prolog is from 1972, we can speak of a veritable golden age in programming language design.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

However, in practice, the declarative reading of a logic or functional program can lose much of its "what" aspect because it has to go into a lot of detail on the "how" (see the O'Keefe quote for Chapter 3). For example, a declarative definition of tree search has to give almost as many orders as an imperative definition. Nevertheless, declarative programming still has three crucial advantages. First, it is easier to build abstractions in a declarative setting, since declarative operations are by nature compositional. Second, declarative programs are easier to test, since it is enough to test single calls (give arguments and check the results). Testing stateful programs is harder because it involves testing *sequences* of calls (due to the internal history). Third, reasoning with declarative programming is simpler than with imperative programming (e.g., algebraic reasoning is possible).

6.1 What is state?

We have already programmed with state in the declarative model of Chapter 3. For example, the accumulators of Section 3.4.3 are state. So why do we need a whole chapter devoted to state? To see why, let us look closely at what state really is. In its simplest form, we can define state as follows:

> A *state* is a sequence of values in time that contains the intermediate results of a desired computation.

Let us examine the different ways that state can be present in a program.

6.1.1 Implicit (declarative) state

The sequence need only exist in the mind of the programmer. It does not need any support at all from the computation model. This kind of state is called *implicit state* or *declarative state*. As an example, look at the declarative function `SumList`:

```
fun {SumList Xs S}
   case Xs
   of nil then S
   [] X|Xr then {SumList Xr X+S}
   end
end
```

It is recursive. Each call has two arguments: `Xs`, the unexamined rest of the input list, and `S`, the sum of the examined part of the input list. While calculating the sum of a list, `SumList` calls itself many times. Let us take the pair `(Xs#S)` at each call, since it gives us all the information we need to know to characterize the call. For the call `{SumList [1 2 3 4] 0}` this gives the following sequence:

```
[1 2 3 4] # 0
[2 3 4] # 1
[3 4] # 3
```

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

```
[4] # 6
nil # 10
```

This sequence is a state. When looked at in this way, `SumList` calculates with state. Yet neither the program nor the computation model "knows" this. The state is completely in the mind of the programmer.

6.1.2 Explicit state

It can be useful for a function to have a state that lives across function calls and that is hidden from the callers. For example, we can extend `SumList` to count how many times it is called. There is no reason why the function's callers need to know about this extension. Even stronger: for modularity reasons the callers *should not* know about the extension. This cannot be programmed in the declarative model. The closest we can come is to add two arguments to `SumList` (an input and output count) and thread them across all the callers. To do it without additional arguments we need an explicit state:

> An *explicit state* in a procedure is a state whose lifetime extends over more than one procedure call without being present in the procedure's arguments.

Explicit state cannot be expressed in the declarative model. To have it, we extend the model with a kind of container that we call a *cell*. A cell has a name, an indefinite lifetime, and a content that can be changed. If the procedure knows the name, it can change the content. The declarative model extended with cells is called the *stateful model*. Unlike declarative state, explicit state is not just in the mind of the programmer. It is visible in both the program and the computation model. We can use a cell to add a long-term memory to `SumList`. For example, let us keep track of how many times it is called:

```
local
   C={NewCell 0}
in
   fun {SumList Xs S}
      C:=@C+1
      case Xs
      of nil then S
      [] X|Xr then {SumList Xr X+S}
      end
   end
   fun {SumCount} @C end
end
```

This is the same definition as before, except that we define a cell and update its content in `SumList`. We also add the function `SumCount` to make the state observable. Let us explain the new operations that act on the explicit state. `NewCell` creates a new cell with initial content 0. `@` gets the content and `:=` puts

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

in a new content. If `SumCount` is not used, then this version of `SumList` cannot be distinguished from the previous version: it is called in the same way and gives the same results.[3]

The ability to have explicit state is very important. It removes the limits of declarative programming (see Section 4.7). With explicit state, abstract data types gain tremendously in modularity since it is possible to *encapsulate* an explicit state inside them. The access to the state is limited according to the operations of the abstract data type. This idea is at the heart of object-oriented programming, a powerful programming style that is elaborated in Chapter 7. The present chapter and Chapter 7 both explore the ramifications of explicit state.

6.2 State and system building

The principle of abstraction

As far as we know, the most successful system-building principle for intelligent beings with finite thinking abilities, such as human beings, is the *principle of abstraction*. Consider any system. It can be thought of as having two parts: a *specification* and an *implementation*. The specification is a *contract*, in a mathematical sense that is stronger than the legal sense. The contract defines how the rest of the world interacts with the system, as seen from the outside. The implementation is how the system is constructed, as seen from the inside. The miraculous property of the distinction specification/implementation is that the specification is usually much simpler to understand than the implementation. One does not have to know how to build a watch in order to read time on it. To paraphrase evolutionist Richard Dawkins, it does not matter whether the watchmaker is blind or not, as long as the watch works.

This means that it is possible to build a system as a concentric series of layers. One can proceed step by step, building layer upon layer. At each layer, build an implementation that takes the next lower specification and provides the next higher one. It is not necessary to understand everything at once.

Systems that grow

How is this approach supported by declarative programming? With the declarative model of Chapter 2, all that the system "knows" is on the *outside*, except for the fixed set of knowledge that it was born with. To be precise, because a procedure is stateless, all its knowledge, its "smarts," are in its arguments. The smarter the procedure gets, the "heavier" and more numerous the arguments get. Declarative programming is like an organism that keeps all its knowledge outside of itself, in its environment. Despite his claim to the contrary (see the chapter quote), this was exactly the situation of Louis XIV: the state was not in his person

[3] The only differences are a minor slowdown and a minor increase in memory use. In almost all cases, these differences are irrelevant in practice.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

but all around him, in 17th century France.[4] We conclude that the principle of abstraction is not well supported by declarative programming, because we cannot put new knowledge inside a component.

Chapter 4 partly alleviated this problem by adding concurrency. Stream objects can accumulate internal knowledge in their internal arguments. Chapter 5 enhanced the expressive power dramatically by adding ports, which makes possible port objects. A port object has an identity and can be viewed from the outside as a stateful entity. But this requires concurrency. In the present chapter, we add explicit state without concurrency. We shall see that this promotes a very different programming style than the concurrent component style of Chapter 5. There is a total order among all operations in the system. This cements a strong dependency between all parts of the system. Later, in Chapter 8, we will add concurrency to remove this dependency. The model of that chapter is difficult to program in. Let us first see what we can do with state without concurrency.

6.2.1 System properties

What properties should a system have to best support the principle of abstraction? Here are three:

- **Encapsulation**. It should be possible to *hide the internals* of a part.
- **Compositionality**. It should be possible to *combine parts* to make a new part.
- **Instantiation/invocation**. It should be possible to *create many instances* of a part based on a single definition. These instances "plug" themselves into their environment (the rest of the system in which they will live) when they are created.

These properties need support from the programming language, e.g., lexical scoping supports encapsulation and higher-order programming supports instantiation. The properties do not require state; they can be used in declarative programming as well. For example, encapsulation is orthogonal to state. On the one hand, it is possible to use encapsulation in declarative programs without state. We have already used it many times, for example in higher-order programming and stream objects. On the other hand, it is also possible to use state without encapsulation, by defining the state globally so all components have free access to it.

Invariants

Encapsulation and explicit state are most useful when used together. Adding state to declarative programming makes reasoning about the program much hard-

[4] To be fair to Louis, what he meant was that the decision-making power of the state was vested in his person.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

er, because the program's behavior depends on the state. For example, a procedure can do a *side effect*, i.e., it modifies state that is visible to the rest of the program. Side effects make reasoning about the program extremely difficult. Bringing in encapsulation does much to make reasoning tractable again. This is because stateful systems can be designed so that a well-defined property, called an *invariant*, is always true when viewed from the outside. This makes reasoning about the system independent of reasoning about its environment. This partly gives us back one of the properties that makes declarative programming so attractive.

Invariants are only part of the story. An invariant just says that the component is not behaving incorrectly; it does not guarantee that the component is making progress towards some goal. For that, a second property is needed to mark the progress. This means that even with invariants, programming with state is not quite as simple as declarative programming. We find that a good rule of thumb for complex systems is to keep as many components as possible declarative. State should not be "smeared out" over many components. It should be concentrated in just a few carefully-selected components.

6.2.2 Component-based programming

The three properties of encapsulation, compositionality, and instantiation define *component-based programming* (see Section 6.7). A component specifies a program fragment with an inside and an outside, i.e., with a well-defined interface. The inside is hidden from the outside, except for what the interface permits. Components can be combined to make new components. Components can be instantiated, making a new instance that is linked into its environment. Components are a ubiquitous concept. We have already seen them in several guises:

- **Procedural abstraction**. We have seen a first example of components in the declarative computation model. The component is called a *procedure definition* and its instance is called a *procedure invocation*. Procedural abstraction underlies the more advanced component models that came later.
- **Functors** (compilation units). A particularly useful kind of component is a compilation unit, i.e., it can be compiled independently of other components. In this book, we call such components *functors* and their instances *modules*.
- **Concurrent components**. A system with independent, interacting entities can be seen as a graph of concurrent components that send each other messages.

In component-based programming, the natural way to extend a component is by using *composition*: build a new component that *contains* the original one. The new component offers a new functionality and uses the old component to implement the functionality.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

We give a concrete example from our experience to show the usefulness of components. Component-based programming was an essential part of the Information Cities project, which did extensive multi-agent simulations using the Mozart system [155, 162]. The simulations were intended to model evolution and information flow in parts of the Internet. Different simulation engines (in a single process or distributed, with different forms of synchronization) were defined as reusable components with identical interfaces. Different agent behaviors were defined in the same way. This allowed rapidly setting up many different simulations and extending the simulator without having to recompile the system. The setup was done by a program, using the module manager provided by the System module `Module`. This is possible because components are values in the Oz language (see Section 3.9.3).

6.2.3 Object-oriented programming

A popular set of techniques for stateful programming is called *object-oriented programming*. We devote the whole of Chapter 7 to these techniques. Object-oriented programming adds a fourth property to component-based programming:

- **Inheritance**. It is possible to build the system in incremental fashion, as a small extension or modification of another system.

Incrementally-built components are called *classes* and their instances are called *objects*. Inheritance is a way of structuring programs so that a new implementation extends an existing one.

The advantage of inheritance is that it factors the implementation to avoid redundancy. But inheritance is not an unmixed blessing. It implies that a component strongly depends on the components it inherits from. This dependency can be difficult to manage. Much of the literature on object-oriented design, e.g., on design patterns [58], focuses on the correct use of inheritance. Although component composition is less flexible than inheritance, it is much simpler to use. We recommend to use it whenever possible and to use inheritance only when composition is insufficient (see Chapter 7).

6.3 The declarative model with explicit state

One way to introduce state is to have concurrent components that run indefinitely and that can communicate with other components, like the stream objects of Chapter 4 or the port objects of Chapter 5. In the present chapter we directly add explicit state to the declarative model. Unlike in the two previous chapters, the resulting model is still sequential. We will call it the *stateful model*.

Explicit state is a *pair* of two language entities. The first entity is the state's identity and the second is the state's current content. There exists an operation that when given the state's identity returns the current content. This operation

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

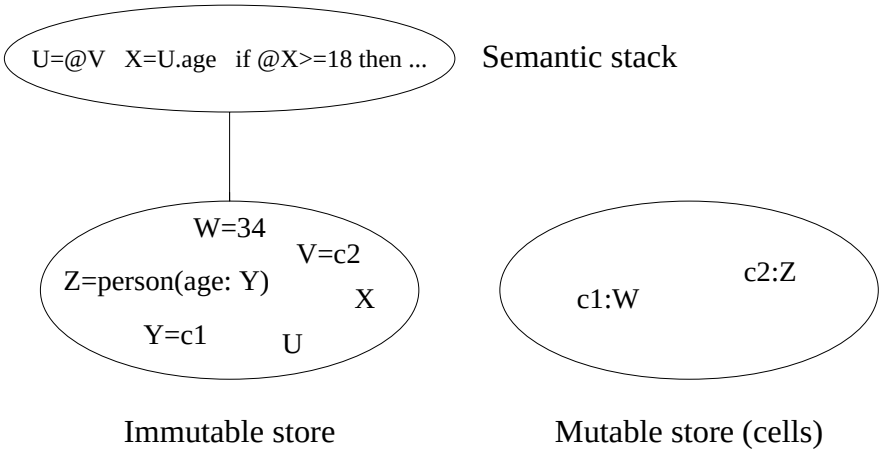

Figure 6.1: The declarative model with explicit state

defines a system-wide *mapping* between state identities and all language entities. What makes it stateful is that the mapping can be modified. Interestingly, neither of the two language entities themselves is modified. It is only the mapping that changes.

6.3.1 Cells

We add explicit state as one new basic type to the computation model. We call the type a *cell*. A cell is a pair of a constant, which is a name value, and a reference into the single-assignment store. Because names are unforgeable, cells are a true abstract data type. The set of all cells lives in the *mutable store*. Figure 6.1 shows the resulting computation model. There are two stores: the immutable (single-assignment) store, which contains dataflow variables that can be bound to one value, and the mutable store, which contains pairs of names and references. Table 6.1 shows its kernel language. Compared to the declarative model, it adds just two new statements, the cell operations `NewCell` and `Exchange`. These operations are defined informally in Table 6.2. For convenience, this table adds two more operations, `@` (access) and `:=` (assignment). These do not provide any new functionality since they can be defined in terms of `Exchange`. Using `C:=Y` as an expression has the effect of an `Exchange`: it gives the old value as the result.

Amazingly, adding cells with their two operations is enough to build all the wonderful concepts that state can provide. All the sophisticated concepts of objects, classes, and other abstract data types can be built with the declarative model extended with cells. Section 7.6.2 explains how to build classes and Section 7.6.3 explains how to build objects. In practice, their semantics are defined in this way, but the language has syntactic support to make them easy to use and the implementation has support to make them more efficient [75].

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

⟨s⟩ ::=	
skip	Empty statement
⟨s⟩$_1$ ⟨s⟩$_2$	Statement sequence
local ⟨x⟩ **in** ⟨s⟩ **end**	Variable creation
⟨x⟩$_1$=⟨x⟩$_2$	Variable-variable binding
⟨x⟩=⟨v⟩	Value creation
if ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Conditional
case ⟨x⟩ **of** ⟨pattern⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Pattern matching
{⟨x⟩ ⟨y⟩$_1$ ... ⟨y⟩$_n$}	Procedure application
{NewName ⟨x⟩}	Name creation
⟨y⟩=!!⟨x⟩	Read-only view
try ⟨s⟩$_1$ **catch** ⟨x⟩ **then** ⟨s⟩$_2$ **end**	Exception context
raise ⟨x⟩ **end**	Raise exception
{NewCell ⟨x⟩ ⟨y⟩}	**Cell creation**
{Exchange ⟨x⟩ ⟨y⟩ ⟨z⟩}	**Cell exchange**

Table 6.1: The kernel language with explicit state

Operation	Description
{NewCell X C}	Create a new cell C with initial content X.
{Exchange C X Y}	Atomically bind X with the old content of cell C and set Y to be the new content.
X=@C	Bind X to the current content of cell C.
C:=X	Set X to be the new content of cell C.
X=C:=Y	Another syntax for {Exchange C X Y}.

Table 6.2: Cell operations

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

6.3.2 Semantics of cells

The semantics of cells is quite similar to the semantics of ports given in Section 5.1.2. It is instructive to compare them. In similar manner to ports, we first add a mutable store. The same mutable store can hold both ports and cells. Then we define the operations `NewCell` and `Exchange` in terms of the mutable store.

Extension of execution state

Next to the single-assignment store σ and the trigger store τ, we add a new store μ called the *mutable store*. This store contains cells, which are pairs of the form $x : y$, where x and y are variables of the single-assignment store. The mutable store is initially empty. The semantics guarantees that x is always bound to a name value that represents a cell. On the other hand, y can be any partial value. The execution state becomes a triple (MST, σ, μ) (or a quadruple (MST, σ, μ, τ) if the trigger store is considered).

The `NewCell` operation

The semantic statement (`{NewCell` $\langle$x$\rangle$ $\langle$y$\rangle$`}`$, E$) does the following:

- Create a fresh cell name n.
- Bind $E(\langle$y$\rangle)$ and n in the store.
- If the binding is successful, then add the pair $E(\langle$y$\rangle) : E(\langle$x$\rangle)$ to the mutable store μ.
- If the binding fails, then raise an error condition.

Observant readers will notice that this semantics is almost identical to that of ports. The principal difference is the type. Ports are identified by a port name and cells by a cell name. Because of the type, we can enforce that cells can only be used with `Exchange` and ports can only be used with `Send`.

The `Exchange` operation

The semantic statement (`{Exchange` $\langle$x$\rangle$ $\langle$y$\rangle$ $\langle$z$\rangle$`}`$, E$) does the following:

- If the activation condition is true ($E(\langle$x$\rangle)$ is determined), then do the following actions:
  - If $E(\langle$x$\rangle)$ is not bound to the name of a cell, then raise an error condition.
  - If the mutable store contains $E(\langle$x$\rangle) : w$ then do the following actions:
    - Update the mutable store to be $E(\langle$x$\rangle) : E(\langle$z$\rangle)$.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

* Bind $E(\langle \texttt{y} \rangle)$ and w in the store.

- If the activation condition is false, then suspend execution.

Memory management

Two modifications to memory management are needed because of the mutable store:

- Extending the definition of reachability: A variable y is reachable if the mutable store contains $x : y$ and x is reachable.
- Reclaiming cells: If a variable x becomes unreachable, and the mutable store contains the pair $x : y$, then remove this pair.

The same modifications are needed independent of whether the mutable store holds cells or ports.

6.3.3 Relation to declarative programming

In general, a stateful program is no longer declarative, since running the program several times with the same inputs can give different outputs depending on the internal state. It is possible, though, to write stateful programs that *behave* as if they were declarative, i.e., to write them so they satisfy the definition of a declarative operation. It is a good design principle to write stateful components so that they behave declaratively.

A simple example of a stateful program that behaves declaratively is the `SumList` function we gave earlier. Let us show a more interesting example, in which the state is used as an intimate part of the function's calculation. We define a list reversal function by using a cell:

```
fun {Reverse Xs}
   Rs={NewCell nil}
in
   for X in Xs do Rs := X|@Rs end
   @Rs
end
```

Since the cell is encapsulated inside the `Reverse`, there is no way to tell the difference between this implementation and a declarative implementation. It is often possible to take a declarative program and convert it to a stateful program with the same behavior by replacing the declarative state with an explicit state. The reverse direction is often possible as well. We leave it as an exercise for the reader to take a declarative implementation of `Reverse` and to convert it to a stateful implementation.

Another interesting example is *memoization*, in which a function remembers the results of previous calls so that future calls can be handled quicker. Chapter 10 gives an example using a simple graphical calendar display. It uses memoization to avoid redrawing the display unless it has changed.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

6.3.4 Sharing and equality

By introducing cells we have extended the concept of equality. We have to distinguish the equality of cells from the equality of their contents. This leads to the concepts of sharing and token equality.

Sharing

Sharing, also known as *aliasing*, happens when two identifiers `X` and `Y` refer to the same cell. We say that the two identifiers are *aliases* of each other. Changing the content of `X` also changes the content of `Y`. For example, let us create a cell:

```
X={NewCell 0}
```

We can create a second reference `Y` to this cell:

```
declare Y in
Y=X
```

Changing the content of `Y` will change the content of `X`:

```
Y:=10
{Browse @X}
```

This displays 10. In general, when a cell's content is changed, then all the cell's aliases see the changed content. When reasoning about a program, the programmer has to be careful to keep track of aliases. This can be difficult, since they can easily be spread out through the whole program. This problem can be made manageable by *encapsulating* the state, i.e., using it in just a small part of a program and guaranteeing that it cannot escape from there. This is one of the key reasons why abstract data types are an especially good idea when used together with explicit state.

Token equality and structure equality

Two values are equal if they have the same structure. For example:

```
X=person(age:25 name:"George")
Y=person(age:25 name:"George")
{Browse X==Y}
```

This displays **`true`**. We call this *structure equality*. It is the equality we have used up to now. With cells, though, we introduce a new notion of equality called *token equality*. Two cells are not equal if they have the same content, rather they are equal if they are the same cell! For example, let us create two cells:

```
X={NewCell 10}
Y={NewCell 10}
```

These are different cells with different identities. The following comparison:

```
{Browse X==Y}
```

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

displays **false**. It is logical that the cells are not equal, since changing the content of one cell will not change the content of the other. However, our two cells happen to have the same content:

```
{Browse @X==@Y}
```

This displays **true**. This is a pure coincidence; it does not have to stay true throughout the program. We conclude by remarking that aliases *do* have the same identities. The following example:

```
X={NewCell 10}
Y=X
{Browse X==Y}
```

displays **true** because `X` and `Y` are aliases, i.e., they refer to the same cell.

6.4 Abstract data types

As we saw in Section 3.7, an abstract data type is a set of values together with a set of operations on those values. Now that we have added explicit state to the model, we can complete the discussion started in Section 3.7. That section shows the difference between secure and open ADTs in the case of declarative programming. State adds an extra dimension to the possibilities.

6.4.1 Eight ways to organize ADTs

An ADT with the same functionality can be organized in many different ways. For example, in Section 3.7 we saw that a simple ADT like a stack can be either open or secure. Here we will introduce two more axes, state and bundling, each with two choices. Because these axes are orthogonal, this gives eight ways in all to organize an ADT! Some are rarely used. Others are common. But each has its advantages and disadvantages. We briefly explain each axis and give some examples. In the examples later on in the book, we choose whichever of the eight ways that is appropriate in each case.

Openness and security

An *open* ADT is one in which the internal representation is completely visible to the whole program. Its implementation can be spread out over the whole program. Different parts of the program can extend the implementation independently of each other. This is most useful for small programs in which expressiveness is more important than security.

A *secure* ADT is one in which the implementation is concentrated in one part of the program and is inaccessible to the rest of the program. This is usually what is desired for larger programs. It allows the ADT to be implemented and tested independently of the rest of the program. We will see the different ways to define a secure ADT. Perhaps surprisingly, we will see that a secure ADT can

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

be defined completely in the declarative model with higher-order programming. No additional concepts (such as names) are needed.

An ADT can be partially secure, e.g., the rights to look at its internal representation can be given out in a controlled way. In the stack example of Section 3.7, the `Wrap` and `Unwrap` functions can be given out to certain parts of the program, for example to extend the implementation of stacks in a controlled way. This is an example of programming with capabilities.

State

A *stateless* ADT, also known as a *declarative* ADT, is written in the declarative model. Chapter 3 gives examples: a declarative stack, queue, and dictionary. With this approach, ADT instances cannot be modified, but new ones must be created. When passing an ADT instance to a procedure, you can be sure about exactly what value is being passed. Once created, the instance never changes. On the other hand, this leads to a proliferation of instances that can be difficult to manage. The program is also less modular, since instances must be explicitly passed around, even through parts that may not need the instance themselves.

A *stateful* ADT internally uses explicit state. Examples of stateful ADTs are components and objects, which are usually stateful. With this approach, ADT instances can change as a function of time. One cannot be sure about what value is encapsulated inside the instance without knowing the history of all procedure calls at the interface since its creation. In contrast to declarative ADTs, there is only one instance. Furthermore, this one instance often does not have to be passed as a parameter; it can be accessed inside procedures by lexical scoping. This makes the program more concise. The program is also potentially more modular, since parts that do not need the instance do not need to mention it.

Bundling

Next to security and state, a third choice to make is whether the data is kept separate from the operations (unbundled) or whether they are kept together (bundled). Of course, an unbundled ADT can always be bundled in a trivial way by putting the data and operations in a record. But a bundled ADT cannot be unbundled; the language semantics guarantees that it always stays bundled.

An *unbundled* ADT is one that can separate its data from its operations. It is a remarkable fact that an unbundled ADT can still be secure. To achieve security, each instance is created together with a "key". The key is an authorization to access the internal data of the instance (and update it, if the instance is stateful). All operations of the ADT know the key. The rest of the program does not know the key. Usually the key is a *name*, which is an unforgeable constant (see Section B.2).

An unbundled ADT can be more efficient than a bundled one. For example, a file that stores instances of an ADT can contain just the data, without any operations. If the set of operations is very large, then this can take much less

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

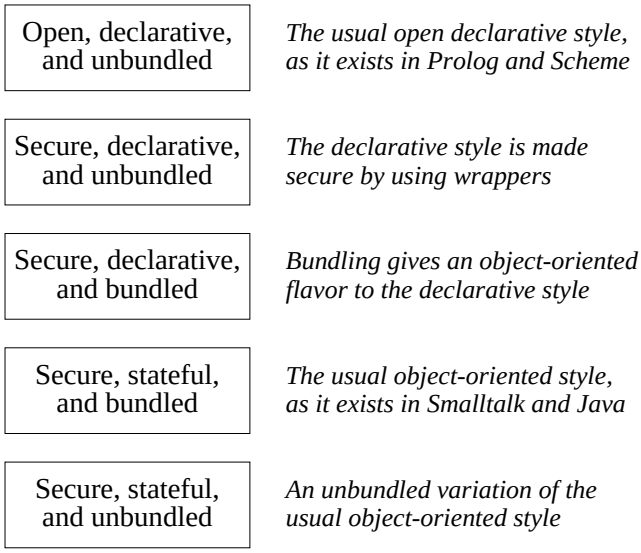

Figure 6.2: Five ways to package a stack

space than storing both the data and the operations. When the data is reloaded, then it can be used as before as long as the key is available.

A *bundled* ADT is one that keeps together its data and its operations in such a way that they cannot be separated by the user. As we will see in Chapter 7, this is what object-oriented programming does. Each object instance is bundled together with its operations, which are called "methods".

6.4.2 Variations on a stack

Let us take the ⟨Stack T⟩ type from Section 3.7 and see how to adapt it to some of the eight possibilities. We give five useful possibilities. We start from the simplest one, the open declarative version, and then use it to build four different secure versions. Figure 6.2 summarizes them. Figure 6.3 gives a graphic illustration of the four secure versions and their differences. In this figure, the boxes labeled "`Pop`" are procedures that can be invoked. Incoming arrows are inputs and outgoing arrows are outputs. The boxes with keyholes are wrapped data structures that are the inputs and outputs of the `Pop` procedures. The wrapped data structures can only be unwrapped inside the `Pop` procedures. Two of the `Pop` procedures (the second and third) themselves wrap data structures.

Open declarative stack

We set the stage for these secure versions by first giving the basic stack functionality in the simplest way:

```
fun {NewStack} nil end
fun {Push S E} E|S end
```

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

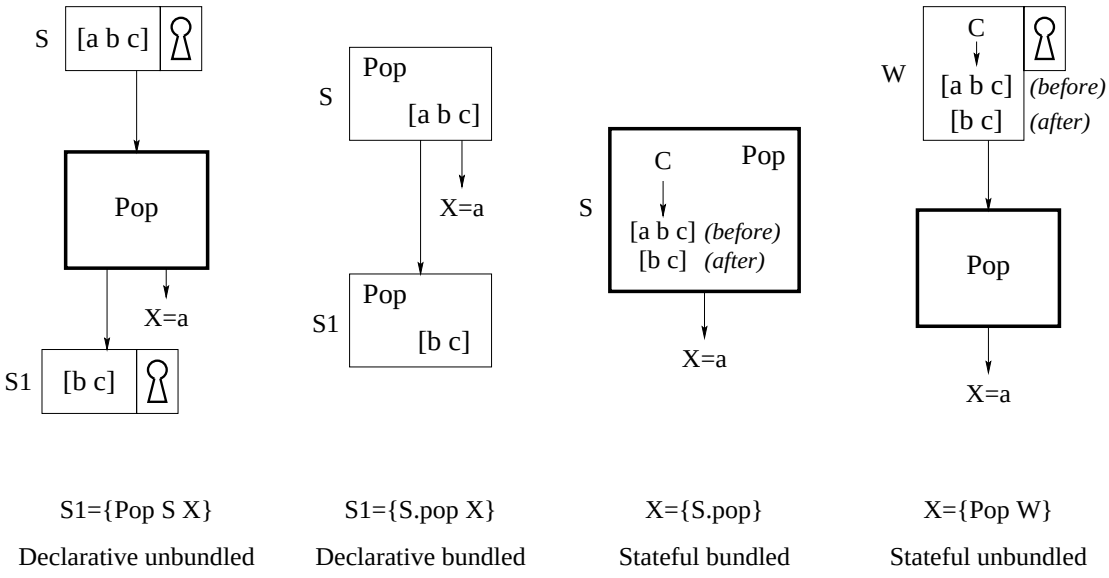

Figure 6.3: Four versions of a secure stack

```
fun {Pop S ?E}
   case S of X|S1 then E=X S1 end
end
fun {IsEmpty S} S==nil end
```

This version is open, declarative, and unbundled.

Secure declarative unbundled stack

We make this version secure by using a wrapper/unwrapper pair, as seen in Section 3.7:

```
local Wrap Unwrap in
   {NewWrapper Wrap Unwrap}
   fun {NewStack} {Wrap nil} end
   fun {Push S E} {Wrap E|{Unwrap S}} end
   fun {Pop S ?E}
      case {Unwrap S} of X|S1 then E=X {Wrap S1} end
   end
   fun {IsEmpty S} {Unwrap S}==nil end
end
```

This version is secure, declarative, and unbundled. The stack is unwrapped when entering the ADT and wrapped when exiting. Outside the ADT, the stack is always wrapped.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

Secure declarative bundled stack

Let us now make a bundled version of the declarative stack. The idea is to hide the stack inside the operations, so that it cannot be separated from them. Here is how it is programmed:

```
local
   fun {StackOps S}
      fun {Push X} {StackOps X|S} end
      fun {Pop ?E}
         case S of X|S1 then E=X {StackOps S1} end
      end
      fun {IsEmpty} S==nil end
   in ops(push:Push pop:Pop isEmpty:IsEmpty) end
in
   fun {NewStack} {StackOps nil} end
end
```

This version is secure, declarative, and bundled. Note that it does not use wrapping, since wrapping is only needed for unbundled ADTs. The function `StackOps` takes a list `S` and returns a record of procedure values, `ops(pop:Pop push:Push isEmpty:IsEmpty)`, in which `S` is hidden by lexical scoping. Using a record lets us group the operations in a nice way. Here is an example use:

```
declare S1 S2 S3 X in
S1={NewStack}
{Browse {S1.isEmpty}}
S2={S1.push 23}
S3={S2.pop X}
{Browse X}
```

It is a remarkable fact that making an ADT secure needs neither explicit state nor names. It can be done with higher-order programming alone. Because this version is both bundled and secure, we can consider it as a declarative form of object-oriented programming. The stack `S1` is a *declarative object*.

Secure stateful bundled stack

Now let us construct a stateful version of the stack. Calling `NewStack` creates a new stack with three operations `Push`, `Pop`, and `IsEmpty`:

```
fun {NewStack}
   C={NewCell nil}
   proc {Push X} C:=X|@C end
   fun {Pop}
      case @C of X|S1 then C:=S1 X end
   end
   fun {IsEmpty} @C==nil end
in
   ops(push:Push pop:Pop isEmpty:IsEmpty)
```

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

```
end
```

This version is secure, stateful, and bundled. In like manner to the declarative bundled version, we use a record to group the operations. This version provides the basic functionality of object-oriented programming, namely a group of operations ("methods") with a hidden internal state. The result of calling `NewStack` is an object instance with three methods `Push`, `Pop`, and `IsEmpty`. Since the stack value is always kept hidden inside the implementation, this version is already secure even without names.

Comparing two popular versions

Let us compare the simplest secure versions in the declarative and stateful models, namely the declarative unbundled and the stateful bundled versions. Each of these two versions is appropriate for secure ADTs in its respective model. It pays to compare them carefully and think about the different styles they represent:

- In the declarative unbundled version, each operation that changes the stack has *two arguments more* than the stateful version: an input stack and an output stack.
- The implementations of both versions have to do actions when entering and exiting an operation. The calls of `Unwrap` and `Wrap` correspond to calls of `@` and `:=`, respectively.
- The declarative unbundled version needs no higher-order techniques to work with many stacks, since all stacks work with all operations. On the other hand, the stateful bundled version needs instantiation to create new versions of `Push`, `Pop` and `IsEmpty` for each instance of the stack ADT.

Here is the interface of the declarative unbundled version:

```
⟨fun {NewStack}: ⟨Stack T⟩⟩
⟨fun {Push ⟨Stack T⟩ T}: ⟨Stack T⟩⟩
⟨fun {Pop ⟨Stack T⟩ T}: ⟨Stack T⟩⟩
⟨fun {IsEmpty ⟨Stack T⟩}: ⟨Bool⟩⟩
```

Because it is declarative, the stack type ⟨Stack T⟩ appears in every operation. Here is the interface of the stateful bundled version:

```
⟨fun {NewStack}: ⟨Stack T⟩⟩
⟨proc {Push T}⟩
⟨fun {Pop}: T⟩
⟨fun {IsEmpty}: ⟨bool⟩⟩
```

In the stateful bundled version, we define the stack type ⟨Stack T⟩ as ⟨op(push:⟨proc {$ T}⟩ pop:⟨fun {$}: T⟩ isEmpty:⟨fun {$}: ⟨Bool⟩⟩)⟩.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

Secure stateful unbundled stack

It is possible to combine wrapping with cells to make a version that is secure, stateful, and unbundled. This style is little used in object-oriented programming, but deserves to be more widely known. It does not need higher-order programming. Each operation has one stack argument instead of two for the declarative version:

```
local Wrap Unwrap in
   {NewWrapper Wrap Unwrap}
   fun {NewStack} {Wrap {NewCell nil}} end
   proc {Push S X} C={Unwrap S} in C:=X|@C end
   fun {Pop S}
   C={Unwrap S} in
      case @C of X|S1 then C:=S1 X end
   end
   fun {IsEmpty S} @{Unwrap S}==nil end
end
```

In this version, `NewStack` only needs `Wrap` and the other functions only need `Unwrap`.

6.4.3 Revocable capabilities

Using explicit state, it is possible to build secure ADTs that have controllable security. As an example of this, let us show how to build revocable capabilities. Chapter 3 introduced the concept of a capability, which gives its owner an irrevocable right to do something. Sometimes we would like to give a *revocable* right instead, i.e., a right that can be removed. We can implement this with explicit state. Without loss of generality, we assume the capability is represented as a one-argument procedure.[5] Here is a generic procedure that takes any capability and returns a revocable version of that capability:

```
proc {Revocable Obj ?R ?RObj}
   C={NewCell Obj}
in
   proc {R}
      C:=proc {$ M} raise revokedError end end
   end
   proc {RObj M}
      {@C M}
   end
end
```

Given any one-argument procedure `Obj`, the procedure returns a revoker `R` and a revocable version `RObj`. At first, `RObj` forwards all its messages to `Obj`. After

[5] This is an important case because it covers the object system of Chapter 7.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

executing `{R}`, calling `RObj` invariably raises a `revokedError` exception. Here is an example:

```
fun {NewCollector}
   Lst={NewCell nil}
in
   proc {$ M}
      case M
      of add(X) then T in {Exchange Lst T X|T}
      [] get(L) then L={Reverse @Lst}
      end
   end
end

declare C R in
C={Revocable {NewCollector} R}
```

The function `NewCollector` creates an instance of an ADT that we call a *collector*. It has two operations, `add` and `get`. With `add`, it can collect items into a list in the order that they are collected. With `get`, the current value of the list can be retrieved at any time. We make the collector revocable. When it has finished its job, the collector can be made inoperable by calling `R`.

6.4.4 Parameter passing

Now that we have introduced explicit state, we are at a good point to investigate the different ways that languages do parameter passing. This book almost always uses call by reference. But many other ways have been devised to pass information to and from a called procedure. Let us briefly define the most prominent ones. For each mechanism, we give an example in a Pascal-like syntax and we code the example in the stateful model of this chapter. This coding can be seen as a semantic definition of the mechanism. We use Pascal because of its simplicity. Java is a more popular language, but explaining its more elaborate syntax is not appropriate for this section. Section 7.7 gives an example of Java syntax.

Call by reference

The identity of a language entity is passed to the procedure. The procedure can then use this language entity freely. This is the primitive mechanism used by the computation models of this book, for all language entities including dataflow variables and cells.

Imperative languages often mean something slightly different by call by reference. They assume that the reference is stored in a cell local to the procedure. In our terminology, this is a call by value where the reference is considered as a value (see below). When studying a language that has call by reference, we recommend looking carefully at the language definition to see exactly what is meant.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

Call by variable

This is a special case of call by reference. The identity of a cell is passed to the procedure. Here is an example:

```
procedure sqr(var a:integer);
begin
   a:=a*a
end

var c:integer;
c:=25;
sqr(c);
browse(c);
```

We code this example as follows:

```
proc {Sqr A}
   A:=@A*@A
end

local
   C={NewCell 0}
in
   C:=25
   {Sqr C}
   {Browse @C}
end
```

For the call `{Sqr C}`, the `A` inside `Sqr` is a synonym of the `C` outside.

Call by value

A value is passed to the procedure and put into a cell local to the procedure. The implementation is free either to copy the value or to pass a reference, as long as the procedure cannot change the value in the calling environment. Here is an example:

```
procedure sqr(a:integer);
begin
   a:=a+1;
   browse(a*a)
end;

sqr(25);
```

We code this example as follows:

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

```
proc {Sqr D}
   A={NewCell D}
in
   A:=@A+1
   {Browse @A*@A}
end

{Sqr 25}
```

The cell `A` is initialized with the argument of `Sqr`. The Java language uses call by value for both values and object references. This is explained in Section 7.7.

Call by value-result

This is a modification of call by variable. When the procedure is called, the content of a cell (i.e., a mutable variable) is put into another mutable variable local to the procedure. When the procedure returns, the content of the latter is put into the former. Here is an example:

```
procedure sqr(inout a:integer);
begin
   a:=a*a
end

var c:integer;
c:=25;
sqr(c);
browse(c);
```

This uses the keyword "`inout`" to indicate call by value-result, as is used in the Ada language. We code this example as follows:

```
proc {Sqr A}
   D={NewCell @A}
in
   D:=@D*@D
   A:=@D
end

local
   C={NewCell 0}
in
   C:=25
   {Sqr C}
   {Browse @C}
end
```

There are two mutable variables: one inside `Sqr` (namely `D`) and one outside (namely `C`). Upon entering `Sqr`, `D` is assigned the content of `C`. Upon exiting, `C`

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

is assigned the content of `D`. During the execution of `Sqr`, modifications to `D` are invisible from the outside.

Call by name

This mechanism is the most complex. It creates a procedure value for each argument. A procedure value used in this way is called a *thunk*. Each time the argument is needed, the procedure value is evaluated. It returns the name of a cell, i.e., the address of a mutable variable. Here is an example:

```
procedure sqr(callbyname a:integer);
begin
   a:=a*a
end;

var c:integer;
c:=25;
sqr(c);
browse(c);
```

This uses the keyword "`callbyname`" to indicate call by name. We code this example as follows:

```
proc {Sqr A}
   {A}:=@{A}*@{A}
end

local C={NewCell 0} in
   C:=25
   {Sqr fun {$} C end}
   {Browse @C}
end
```

The argument `A` is a function that when evaluated returns the name of a mutable variable. The function is evaluated each time the argument is needed. Call by name can give unintuitive results if array indices are used in the argument (see Exercise).

Call by need

This is a modification of call by name in which the procedure value is evaluated only once. Its result is stored and used for subsequent evaluations. Here is one way to code call by need for the call by name example:

```
proc {Sqr A}
   B={A}
in
   B:=@B*@B
```

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.