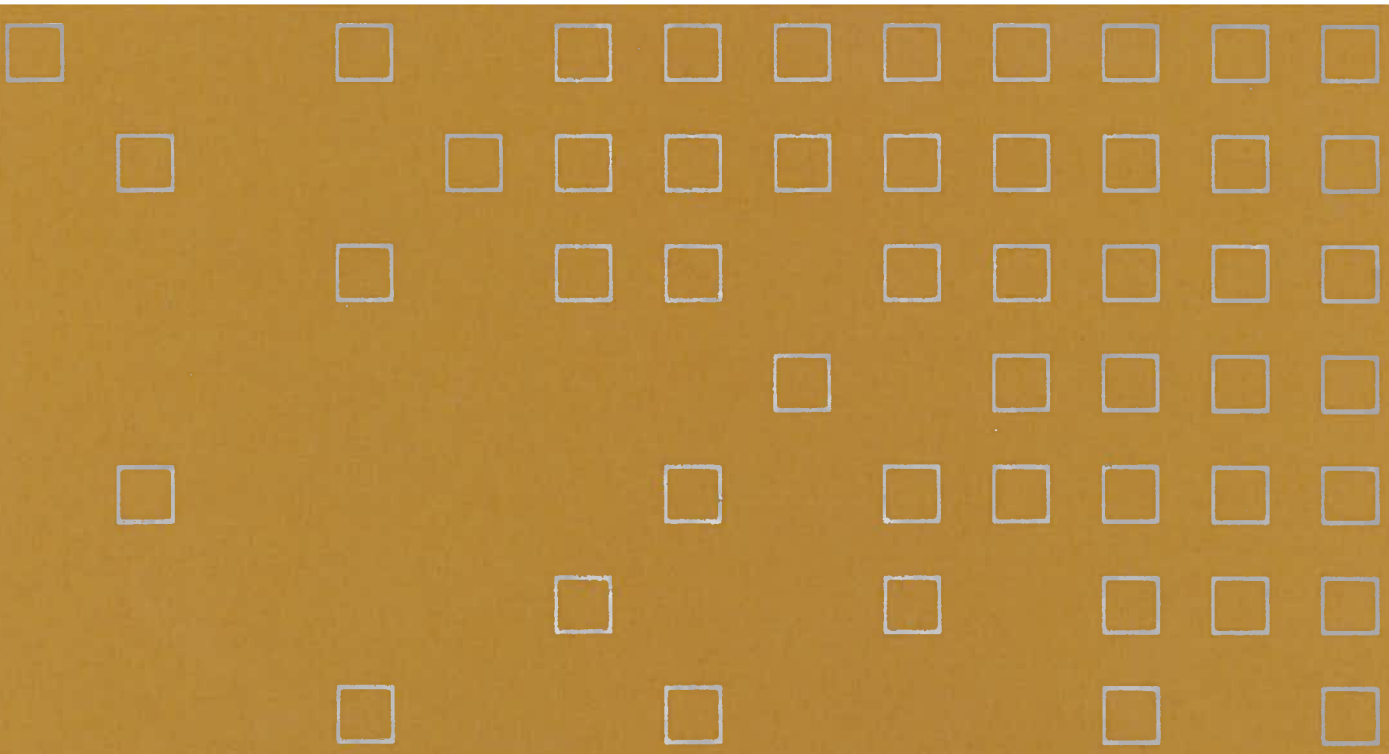

Centrum voor Wiskunde en Informatica

Centre for Mathematics and Computer Science

P.H.M. America, J.J.M.M. Rutten

A parallel object-oriented language: design and semantic foundations

Computer Science/Department of Software Technology Report CS-R8953 December

1989

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

P.H.M. America, J.J.M.M. Rutten

A parallel object-oriented language: design and semantic foundations

Computer Science/Department of Software Technology Report CS-R8953 December

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

A Parallel Object-Oriented Language: Design and Semantic Foundations

P.H.M. America
Philips Research Laboratories,
P.O. Box 80.000, 5600 JA Eindhoven, The Netherlands

J.J.M.M. Rutten
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The language POOL2 integrates the structuring mechanisms of object-oriented programming with facilities for parallel programming. We discuss the most important issues in the design of this language. In particular, we give an overview of the basic principles of object-oriented programming, and we compare several different ways of combining them with parallelism. The language is illustrated by a programming example. Then we define two kinds of formal semantics: an operational semantics based on transition systems and a denotational semantics, which uses the framework of complete metric spaces. These two semantic models are found to be equivalent, in the sense that the operational semantics of a program can be recovered from its denotational semantics by applying an abstraction operation. The semantic techniques are introduced by applying them first to a simplified language.

1980 Mathematics Subject Classification: 68B10, 68C01.
1986 Computing Reviews Categories: D.3.1, F.3.2, F.3.3.
Key Words & Phrases: parallel object-oriented programming, language design, denotational semantics, operational semantics, metric spaces, contractions, semantic equivalence.
Note: This work was carried out in the context of ESPRIT project 415: Parallel Architectures and Languages for Advanced Information Processing — a VLSI-directed approach. This paper was published as a chapter of J.W. DE BAKKER (ed.), *Languages for Parallel Architectures: Design, Semantics, Implementation Models,* Wiley Series in Parallel Computing, 1989, pp. 1-49.

1. Introduction

The use of parallelism offers the theoretical possibility of a significant increase in the performance of computer systems. However, after decades of intensive study, the effective exploitation of parallelism is still a very difficult problem. Whereas in the area of numeric computation impressive advances have been achieved, symbolic applications, with their more irregular and data-dependent structure, have not shown much progress. The most important difficulties seem to lie in programming such complex applications for execution on a parallel machine. What is needed here is a programming notation that serves as an intermediary between human insight and intuition, on the one hand, and the parallel machine architecture, on the other. At the same time it should be the subject of a body of mathematical knowledge, so that formal methods can supplement informal understanding in the design of programs.

In this chapter we present the parallel object-oriented language POOL2 (America, 1988a). This language has been designed for programming symbolic applications such that they can be executed on DOOM, a Decentralized Object-Oriented Machine (Odijk, 1987). It is, however, general enough to be suitable for a large class of machine architectures. The starting point of the language design is the idea of object-oriented programming, as exemplified by Smalltalk-80 (Goldberg and Robson, 1983). Parallelism is integrated in this model by supplying each object with a *body,* an independent parallel

Report CS-R8953
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

process, so that objects are now active entities instead of passive ones. More details on POOL2 are given in section 2.

The rest of the chapter is devoted to a formal semantic study of POOL. This is a syntactically simplified version of POOL2, which retains all the semantic essentials. Since the techniques used are quite complicated, they are first illustrated, in section 3, by applying them to a language $\mathfrak{L}$, which is much simpler than POOL. First we define for this language $\mathfrak{L}$ an *operational semantics* $\mathcal{O}_{Pr}$, which assigns to each program and initial state a set containing all the sequences of states that occur during a possible execution of this program from the initial state. This operational semantics is defined by a *transition system* (in the style of Plotkin (1981, 1983)), which describes the possible transitions that the whole system can make from one configuration to another.

Next we give a *denotational semantics* $\mathcal{D}_{Pr}$ for $\mathfrak{L}$, which assigns to each program an element of the domain $\overline{P}$ of *processes*. This domain $\overline{P}$ is a complete metric space, defined by a domain equation. (The required concepts and properties of complete metric spaces are summarized in the appendix to this chapter.) The processes in $\overline{P}$ are tree-like structures that can describe the execution of a single statement up to a whole system. The main point of such a denotational semantics is its *compositionality:* the meaning of a composite construct can be determined from the semantics of its constituent parts.

Then we prove that the denotational semantics is correct with respect to the operational semantics, or in other words, that the two kinds of semantics are essentially equivalent. We do this by defining an abstraction operation *abstr*, which maps a process $p \in \overline{P}$ to a function from initial states to sets of sequences of states. The relationship between the operational and the denotational semantics can then be described by $\mathcal{O}_{Pr} = abstr \circ \mathcal{D}_{Pr}$.

Finally, in section 4, we do the same for POOL. Using mainly the same techniques as in section 3, we define an operational semantics and a denotational semantics. Again, this denotational semantics is correct with respect to the operational semantics.

2. The design of POOL2

In this section we give an overview of the most important issues that have played a part in the design of the language POOL2. A more extensive discussion can be found in America (1988c). The design of POOL2 is based to a large extent on its predecessor POOL-T, and many of the considerations given in America (1987a) are still valid for POOL2.

The structure of this section is as follows: First, in section 2.1 we give an overview of the principles of object-oriented programming as they have been incorporated into POOL2. Then section 2.2 describes how parallelism is integrated into the language. Section 2.3 presents a programming example and finally section 2.4 discusses some additional issues.

2.1 Object-oriented programming

In the object-oriented programming style a system is described as a collection of *objects*. An object is best defined as an integrated unit of *data* and *procedures* acting on these data. One can think of it as a box that stores some data and has the possibility of acting on these data. The data in an object are stored in *variables*. The contents of a variable can be changed by executing an assignment statement.

A very important principle is that one object's variables are not accessible to other objects: they are strictly private. In other words, the box has a thick wall around it, which separates the inside from the outside. The only way in which objects can interact is by sending *messages* to each other. Such a message is in fact a request from the sender for the receiver to execute a procedure. In POOL these procedures, which are executed in response to messages, are called *methods*. The receiver decides whether and when it executes such a method, and in some cases it even depends on the receiver which method is executed (see section 2.4). In general, the sender of the message can include some

parameters to be passed to the method and the method can return a result, which is passed back to the receiver (see Figure 1). In this way objects can co-operate and communicate. It is important to note that this interaction between objects can only occur according to this precisely determined message interface. Thus every object has the possibility and the responsibility of maintaining its own local data in a consistent state.

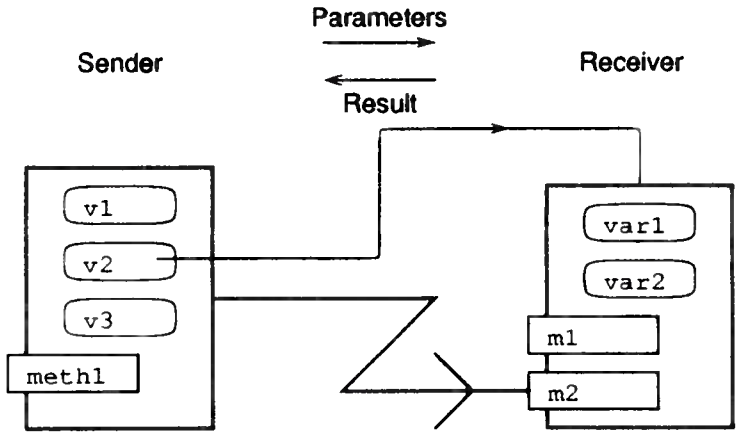

Figure 1: Sending a message

Objects are entities of a dynamic nature. At any point in the execution of a program a new object can be created, so that an arbitrarily large number of objects can come into existence. (Objects are never destroyed explicitly. However, they can be removed by garbage collection if it is certain that this will not influence the correct execution of the program.) In order to describe such systems with many objects, the objects are grouped into *classes.* All the elements (the *instances*) of a class have the same names and types for their variables (although each object has its own set of variables) and they all execute the same code for their methods. In this way, a class can serve as a blueprint for the creation of its instances.

Several object-oriented programming languages use different mechanisms to describe object creation. In general it is agreed that creating new objects is not a natural task for the existing instances of the same class (where would the first instance come from?) but rather for the class itself. In Smalltalk-80 (Goldberg and Robson, 1983) classes are considered to be objects themselves: they can also be created and changed dynamically. Therefore it is natural to describe object creation in *class methods:* a new object can be created by sending an appropriate message to the class. In POOL2 it is not natural to consider classes as objects, because we do not want them to be created or to change during program execution. Therefore in POOL2 the creation of new objects is done by *routines,* a kind of procedure different from methods. Routines are not associated with certain objects and they do not have direct access to any object's variables. Instead, a routine is associated with a *class,* and it can be executed by any object that knows it. By encapsulating the creation of new objects in routines it can be ensured that such a new object is properly initialized before it is used.

Let us now briefly discuss the nature of the data that are stored and manipulated in the objects. In general, a variable contains a *reference* to some object. Also in parameters and results of methods, references are transferred. Some languages, like C++ (Stroustrup, 1986) and Eiffel (Meyer, 1988), also have other built-in data types, like integers and characters, that can be manipulated by the objects. These languages are sometimes called *hybrid* object-oriented. In contrast, in *pure* object-

oriented languages, like Smalltalk-80 and POOL2, every data item is represented by (a reference to) an object. In these languages, even very simple items such as integers are conceptually modelled as objects. For example, the addition 3+4 is performed by sending to object 3 a message mentioning the method add and having (a reference to) object 4 as a parameter (in POOL2, the expression 3+4 is considered as a shorter notation for the message-sending expression 3 ! add (4)). In response to this message, object 3 somehow knows how to add itself to the parameter object and it returns the result, a reference to object 7, to the sender of this message. Of course, this is just a conceptual view: in an actual implementation some optimizations will take place so that these operations can be performed much more efficiently using the hardware facilities for integer addition.

The most important contribution of object-oriented programming in the direction of better software development methods stems from the fact that it is a refinement of programming with abstract data types. It encourages the grouping together of all the information pertinent to a certain kind of entity and it enforces the encapsulation of this information according to an explicit interface with the outside world. For users of a certain class, the set of available methods and routines, together with a description of their behaviour (including at least the types of parameters and results), is all that is relevant. The interior of the objects, the variables and the code of methods and routines, is completely inaccessible to them.

Two important quality aspects of software are addressed by this technique. The first is *adaptability:* If a piece of software must be modified (a frequently occurring phenomenon), it is very often the case that many of the relevant pieces of code are inside one class definition instead of spread out over the whole program. Moreover, if the interface of such a class is unchanged or only extended (new methods are added, but the old ones retain their functionality), it is clear that the rest of the program will not be affected by the change. Another aspect is *re-usability:* A class that is well designed and validated by testing or verification can be used repeatedly in different programs. In order to be able to use a class one need only to consider the external interface; the internal details are irrelevant.

Object-oriented programming also leads to a different way of designing software. The common technique of top-down functional design starts from the required end-to-end functionality of a complete program and divides this iteratively into subfunctions until basic language primitives are obtained. The resulting software is not very adaptable to changing requirements, because in practice the changes mostly pertain exactly to this end-to-end functionality. Moreover, it is very unlikely that the subfunctions into which the program is divided coincide precisely with subfunctions in another program, which would allow re-use of software, because these subfunctions are obtained seperately in an *ad hoc* way for each program. In contrast, object-oriented design initially focuses on the basic entities (objects) manipulated by the program and it grows towards the required end-to-end functionality in a rather bottom-up way. The resulting software is often easier to adapt to changing circumstances, because these basic entities are not very likely to change. Moreover, this way of designing software leads more often to meaningful software components that can be re-used. A more extensive discussion of these issues can be found in (Meyer, 1988).

2.2 Introducing parallelism

Despite the terminology of "message passing", most existing object-oriented languages are sequential in nature. This can be explained by the fact that they observe the following restrictions:

(1) Execution starts with exactly one object being active.
(2) Whenever an object sends a message, it does not do anything before the result of that message has arrived.
(3) An object is only active when it is executing a method in response to an incoming message.

Under these conditions we can see that at any moment there is exactly one active object, although very often control is transferred from one object to another.

Now one can think of several ways of introducing parallelism to object-oriented languages. One possibility is to add processes as an orthogonal concept to the language. In some sense this can be

seen as eliminating restriction (1). Several processes can be active at the same time, each executing an object-oriented program in the way described in section 2.1. These processes act on the same collection of objects; it is even possible that they are executing the same method in the same object at the same time. This way of dealing with parallelism has been adopted by some languages that were initially meant to be purely sequential, such as Smalltalk-80.

While this approach seems appealing theoretically, it is not so attractive in practice. The point is that it does not solve the problems associated with parallelism. The most important problem of parallel programming is dealing with *non-determinism.* The relative execution speed of the several processes in the system is unknown and can even vary from one execution of a program to another. In this way different program executions can lead to different results in a non-reproducible fashion. The number of possible executions increases very quickly with the number of processes and of places where these can interact. Now a certain amount of non-determinism is necessary in order to flexibly exploit the available hardware parallelism, but too much of it makes it almost impossible to ensure the correctness of a program. The main principles of reducing non-determinism are *synchronization* and *mutual exclusion.* Languages that have processes as orthogonal concepts need extra facilities to achieve synchronization and mutual exclusion. To this end, such languages provide some built-in classes (for example, semaphores). However, the programmer must remember to use these, and use them correctly.

More promising approaches can be obtained by relaxing the other above restrictions. By omitting restriction (2), the sender of a message can immediately continue with its own activities without waiting for a result. This is called *asynchronous communication.* In this way the sender can execute in parallel with the receiver of the message. It is possible to obtain a large degree of parallelism after a number of messages have been sent. This scheme has been adopted most notably by the family of *actor languages* (Agha, 1986). One can say that this provides a convenient mechanism for mutual exclusion, because an object only processes one message at a time. However, the facilities for synchronization are not yet satisfactory in this model.

Therefore POOL2 uses yet another approach, which can be characterized by relaxing restriction (3) above. Every object has a *body,* a local independent process, which is started as soon as the object is created and executes in parallel with all the other objects in the system. At arbitrary places in this body it can be indicated explicitly that the object wants to send or receive a message. In this approach, too, a large degree of parallelism can be obtained by creating a sufficient number of objects, whose bodies can execute in parallel.

The basic communication mechanism in POOL2 is *synchronous message passing.* The sender executes a so-called send expression, which has the form

destination ! method (arg1, ..., argn)

We see that it explicitly indicates the receiver, the method name, and a number of arguments to be handed over to the method. The receiver executes an answer statement, which is of the following form:

ANSWER (method1, ..., methodn)

This indicates that exactly one message should be answered, where the method name should be among the ones in the list. Now the actual communication takes place in the form of a rendezvous: the sender and the receiver synchronize (the one that is first willing to communicate waits until the other is ready), and the parameters are passed to the receiver's method, which is then executed. As soon as the method returns a result, this result is passed back to the sender of the message. Now the rendezvous ends and both objects continue independently with their computation. Returning the result is not necessarily done at the end of the method: it is possible that the method still continues after the end of the rendezvous. In any case, the answer statement terminates when the method invocation has ended.

POOL2 also offers an asynchronous communication mechanism. In this case, no result is returned,

and the sender continues immediately after having sent the message on its way to the receiver. This mechanism is considered as being *derived* from the synchronous one: its meaning can be described exactly by creating for each asynchronous message a buffer object to which the message is first sent synchronously and which then passes the message to the end destination, again synchronously.

Together the mechanisms available in POOL2 for dealing with parallelism offer a great deal of flexibility in programming while allowing the construction of very reliable programs. In fact very few classes tend to have an elaborate body: only objects that actively pursue a certain task and objects that wish to impose an explicit ordering on the messages they receive need such a body. The rest of the objects use the default body, which answers all incoming messages in sequence. Even for these objects a significant amount of parallelism can be obtained by letting the methods return their results as soon as possible. In this way the safety of synchronous message passing is retained. Asynchronous communication is mainly used where it is necessary to avoid deadlock. By the encapsulation of data and the associated operations in an object that is strictly sequential internally, mutual exclusion in accessing these data is automatically guaranteed and destructive interference of processes acting on the same data is seldom a problem in POOL2 programming.

2.3 A programming example

We will now illustrate the above principles by a brief example. This also gives us the opportunity of introducing a few additional language constructs. In our example we implement a parallel version of a priority queue that can store integers. These integers can be input in an arbitrary order; when they are retrieved from the queue the largest one is always output first. We first give a *specification unit*, in which the class PQ and its interface with the outside world is described. Such a specification unit gives all the information that is needed to *use* a class or collection of classes.

```
SPEC UNIT Prio_Queue

CLASS PQ
%% Instances of this class are priority queues that store integers.

ROUTINE new () : PQ
%% Creates and returns a new, empty priority queue.

METHOD put (n : Int) : PQ
%% Stores the integer n in the queue; returns SELF.

METHOD get () : Int
%% Deletes the largest integer from the queue and returns it.
%% This method will not be answered if the queue is empty.

END PQ
```

This means that we have a class PQ with a routine new to create fresh priority queues and methods put and get to insert and retrieve integers. (The method put does not yield a meaningful result to return but, being a synchronous method, it should return some result. In this situation we have the convention that the method returns a reference to the receiver itself.) Therefore we can create a new instance of the class PQ by a statement such as

```
q := PQ.new (),
```

we can insert a new element by

```
q ! put (i),
```

and we can extract an element by

```
j := q ! get ().
```

Let us now consider the corresponding *implementation unit,* which gives the actual implementation details of the class PQ:

```
IMPL UNIT Prio_Queue

CLASS PQ
%% Instances of this class are priority queues that store integers.

%% The routine new, which creates and returns an empty priority queue,
%% is defined automatically. An explicit definition is not necessary.


VAR max : Int    %% the largest element in the queue
   rest : PQ    %% a PQ that stores all the other elements
   %% Both variables are automatically initialized to NIL.

%% Invariant: max == NIL  <==> queue is empty
%%            max~== NIL ==> rest~== NIL


METHOD put (n : Int) : PQ
%% Stores the integer n in the queue; returns SELF.
BEGIN
  RESULT SELF;   %% end of rendezvous: sender can continue
  IF max == NIL
  THEN max := n;
      IF rest == NIL THEN rest := PQ.new () FI
  ELSIF max >= n
  THEN rest ! put (n)
  ELSE rest ! put (max);
      max := n
  FI
END put


METHOD get () : Int
%% Deletes the largest integer from the queue and returns it.
%% This method will not be answered if the queue is empty.
BEGIN            %% We know that max~== NIL, so rest~== NIL
  RESULT max;    %% end of rendezvous: sender can continue
  max := rest ! get_largest_or_NIL ()
END get
```

```
METHOD get_largest_or_NIL () : Int
%% Returns NIL if the queue is empty. Otherwise it deletes
%% the largest element and returns it.
BEGIN
  RESULT max;    %% end of rendezvous: sender can continue
  IF max ~== NIL
  THEN max :- rest ! get_largest_or_NIL ()
  FI
END get_largest_or_NIL


BODY
  DO    %% forever
    IF max == NIL
    THEN ANSWER (put, get_largest_or_NIL)
    ELSE ANSWER (put, get, get_largest_or_NIL)
    FI
  OD
YDOB

END PQ
```

Figure 2 shows a number of instances of the class PQ. We see that each instance stores, at most, one integer itself; for the rest of the contents it uses another instance. All the methods in this class return their result immediately at the beginning. In this way the sender can continue its own activities while the method is still processing the message. This is the source of parallelism in the current example: While a newly input integer is propagating throughout the whole queue, the next can already have been entered, and the same holds for requests for output.

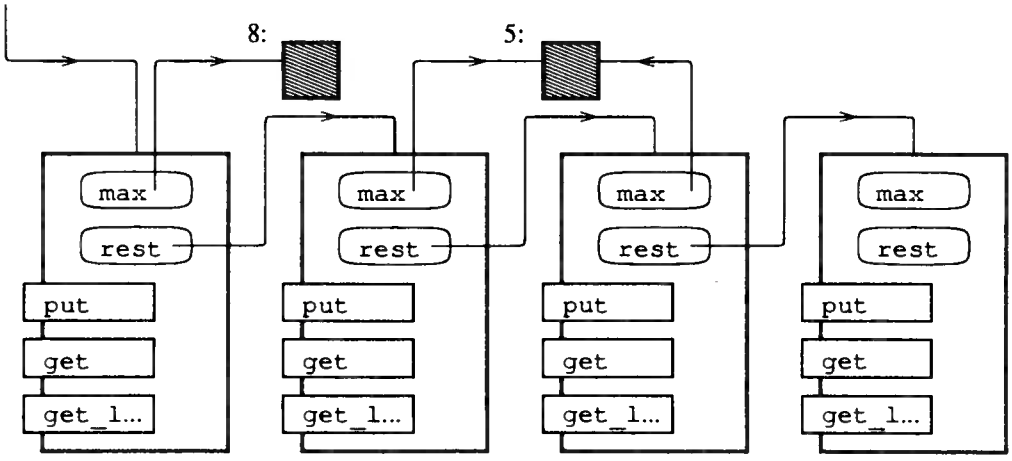

Figure 2: A priority queue containing the number 8 and twice the number 5

The method get_largest_or_NIL does not appear in the specification unit. Therefore it cannot be used by other units. In the current unit we need it because the blocking behaviour of the method get is undesirable when we ask the rest queue for its largest element: this would block the requesting queue even for put messages. Instead, the method get_largest_or_NIL does not block if the queue is

empty but it returns the special value NIL, which is a reference to *no* object. (One may argue that this method get_largest_or_NIL would be quite useful to put in the specification unit for external use as well. We have not done so in order to illustrate the possibility of hiding certain methods.)

The body of the class PQ makes sure that the method get is only answered when the queue is not empty. The loop is never terminated explicitly, but the object may nevertheless be discarded by a garbage collector as soon as no other object no longer has a reference to it.

The example shows quite a few expressions that use operators. Most of these are in fact shorthand notations for send expressions. For example, the expression max >= n is another form of writing max ! greater_or_equal (n). The operator == is an exception, in the sense that it represents a routine call instead of a send expression. For example, the expression max == NIL is equivalent to Int.id (max, NIL). In POOL2, every class automatically gets such a routine id. Without sending a message, it checks whether its two arguments refer to the same object (or both refer to no object, NIL).

The unit Prio_Queue shown above can be used for different purposes. For example, it can sort a sequence of integers, as shown in the following unit:

```
IMPL UNIT Sorting

USE File_IO, Prio_Queue

GLOBAL root := Sorter.new()

CLASS Sorter
%% An instance of this class will read integers from the standard input
%% file until a negative one is encountered. Then it will print the
%% nonnegative ones in descending order.

VAR pq := PQ.new ()
   n : Int := standard_in ! read_Int ()

BODY
  WHILE n >= 0
  DO pq ! put (n);
    n := standard_in ! read_Int ()
  OD;
  DO   %% until deadlock
    standard_out ! write_Int (pq ! get (), 0) ! new_line ()
  OD
YDOB
END Sorter
```

We also see how the execution of a POOL2 program is initiated: The global name root is declared, and in order to initialize it, a new element of the class Sorter is created. (In principle, any object in the system can now refer to this Sorter object under the name root.) This Sorter object then creates other objects, in this case a priority queue, and sets the whole system running. The program ends in a deadlock, where the Sorter object tries to get an integer from an empty queue. This is quite a normal way of termination for a POOL program (it is detected by the run-time system and all the objects are removed). It could be avoided by also inserting the last, negative integer into the queue and terminating when it emerges. Note that this program can, in principle, sort a sequence of integers in linear time, while a sequential program would always need $O(N\log N)$ time for this task.

Another possibility of using the unit Prio_Queue would be to share such a priority queue among several other objects. It could serve as a repository between a number of producers and a number of consumers of information, such that a consumer asking for new input always gets the item with the

highest priority. (Note that items with equal priority are handled in a first-in first-out way, which is not interesting for integers but can be important for generalizations.) The message interface of such a PQ object ensures that requests from consumers and producers are processed in sequence (mutual exclusion is guaranteed) and that consumers are automatically blocked until more input for them is available.

2.4 Typing and inheritance

As can be seen from the above example, POOL2 is a strongly typed language: Every variable, parameter and result has a type associated with it, indicating the class of the object it represents. First, this allows the compiler to check, for example, whether the destination object of a message indeed has the indicated method, so that a certain class of programming errors can be detected even before the program is run. In addition, this typing information constitutes an important kind of documentation of the program, which is now automatically included.

In some cases the strong typing rules might decrease the flexibility of the programmer to construct widely usable classes. More advanced typing schemes are then needed. For example, POOL2 offers the possibility of defining *generic* classes, where some of the types used in the definition are parameters which need only be filled in when the class is actually used. Operations on such types can also be passed as parameters, because in POOL2 routines are considered as objects, so that they can be stored in variables and passed as arguments or results of methods and routines.

In connection with typing there is the concept of *inheritance,* which plays an important role in many object-oriented languages. Its basic principle is that in defining a new class it is often convenient to take over all the variables and methods of an existing class and only add or redefine a few of them. The new class is then said to inherit these variables and methods from the old one. The advantage of this mechanism is the possibility of code sharing: duplication of code is avoided and both the programmer and the implementation can profit from this.

In many cases, the new class is intended to be a specialized version of the old one, in the sense that an instance of the new class can be used whenever an instance of the old one is expected. Therefore the new class is often called a *subclass* of the old one. In typed object-oriented languages this is often reflected by considering the type associated with this new class as a subtype of the old one, which means that, for example, an expression of the new type may be assigned to a variable of the old one.

There are, however, problems with this connection between inheritance and subtyping (America (1987b)). The most important point is that by just taking over the variables and methods of an existing class, even if redefinition is not allowed, it is not automatically the case that the new class is really a specialized version, nor is this the only way in which specialization in behaviour can be obtained. Whereas code sharing is a fairly straightforward mechanism, subtyping should be based on a thorough semantic understanding of the behaviour of objects. In POOL2 the decision has been made not to include inheritance or subtyping until a satisfactory body of fundamental knowledge of these issues has been obtained.

3. Semantic investigation of a simple language

In this section we investigate the semantics of a language £, which, on the one hand, is simple enough to give a formal treatment of its semantics in full detail and, on the other, has enough in common with POOL to be of relevance for the study of the latter's semantics.

After having introduced the language £ and briefly explained its informal semantics, we shall define an *operational* and a *denotational* semantics for £. Next we shall prove that the operational semantics equals the (functional) composition of the denotational semantics with a so-called *abstraction* operation. From this, the semantic *correctness* (a notion we shall shortly introduce formally) of the denotational semantics with respect to the operational semantics follows.

A semantics for a programming language $\mathfrak{L}$ is a mapping $\mathfrak{M}:\mathfrak{L}\to D$, where D is some mathematical domain (a set, a complete partial ordering, a complete metric space), which we call the semantic *universe* of $\mathfrak{M}$. Sometimes $\mathfrak{M}$ is called a model for $\mathfrak{L}$. Traditionally, two main types of semantics are distinguished: *operational* and *denotational*. Without wishing to become involved in a discussion of the precise definitions, we state that the main characteristic of the former is that its definition is based on a *transition system*, or *abstract machine*, in the style of Hennessy and Plotkin (1979) and Plotkin (1981, 1983). A denotational semantics is characterized by the fact that it is defined in a *compositional* manner: the denotational semantics of a composite statement is given in terms of the denotational semantics of its components. (As a second distinctive property one often considers the way in which recursion is treated. The usual view is that an operational semantics treats recursion by means of so-called *syntactic environments* (or body replacement), whereas a denotational semantics uses *semantic environments*, in combination with some fixed-point argument.)

Now consider an operational semantics $\mathcal{O}:\mathfrak{L}\to D$ and a denotational semantics $\mathfrak{D}:\mathfrak{L}\to D'$. Ideally, one would wish both models to coincide; then $\mathfrak{D}$ could be regarded as a compositional reformulation of $\mathcal{O}$. Often, however, this is not the case. The requirement for $\mathfrak{D}$ to be compositional in general implies that it must distinguish more statements than $\mathcal{O}$ does. (Therefore, its semantic universe D' is often more involved than the semantic universe D of $\mathcal{O}$.) A natural question is whether such a denotational semantics $\mathfrak{D}$ distinguishes *at least* the same statements as $\mathcal{O}$; in other words, whether for all statements s and t: if $\mathcal{O}[\![s]\!]\neq\mathcal{O}[\![t]\!]$ then $\mathfrak{D}[\![s]\!]\neq\mathfrak{D}[\![t]\!]$. In that case, we call $\mathfrak{D}$ *correct* with respect to $\mathcal{O}$. If we define for a semantics $\mathfrak{M}:\mathfrak{L}\to D''$ an equivalence relation $\equiv_{\mathfrak{M}}$ by

$$s \equiv_{\mathfrak{M}} t \Leftrightarrow \mathfrak{M}[\![s]\!] = \mathfrak{M}[\![t]\!],$$

for all $s, t \in \mathfrak{L}$, then the correctness of $\mathfrak{D}$ with respect to $\mathcal{O}$ can be expressed formally by the condition

$$\equiv_{\mathfrak{D}} \subset \equiv_{\mathcal{O}}.$$

One way to prove the correctness of $\mathfrak{D}$ is to introduce a so-called *abstraction* operator $\alpha:D'\to D$, which relates the denotational semantic universe with the operational one. If one can prove that

$$\mathcal{O} = \alpha \circ \mathfrak{D}$$

then a precise relationship between $\mathcal{O}$ and $\mathfrak{D}$ has been established, which moreover implies the correctness of $\mathfrak{D}$ with respect to $\mathcal{O}$.

As a mathematical framework for our semantic descriptions we have chosen *complete metric spaces*. (For the basic definitions of metric topology see Dugundji (1966) or Engelking (1977) and the appendix to this chapter.) In this we follow and generalize De Bakker and Zucker (1982). (For other applications of this type of semantic framework see De Bakker *et al* (1986).) We follow Kok and Rutten (1988) in using contractions on complete metric spaces as our main mathematical tool, exploring the fact that contractions have *unique* fixed points (Banach's theorem). We shall define both operators on our semantic universes and the semantic models themselves as fixed points of suitably defined contractions. In this way, we are able to use a general method for proving semantic correctness. Suppose we have defined $\mathcal{O}$ as the fixed point of a contraction

$$\Phi: (\mathfrak{L}\to D)\to(\mathfrak{L}\to D).$$

If we next show that also $\alpha\circ\mathfrak{D}$ is a fixed point of Φ then Banach's theorem implies that $\mathcal{O}=\alpha\circ\mathfrak{D}$.

It will be precisely the scheme outlined above that we shall apply to the language $\mathfrak{L}$, which we introduce next. First some notation: Throughout this chapter we shall write $(x,y\in)X$ when a set X is introduced that has x and y as typical elements. For the definition of $\mathfrak{L}$, we need the following sets: a set of *variables* $(x,y\in)Var$; a set of *expressions* $(e\in)Exp$, which here will have no internal structure (as opposed to the expressions in POOL); a set $(m\in)MName$ of *method names;* and a set $(\alpha,\beta\in)SLabel$ of *statement labels* (which will also be called *object names*). The definition of $\mathfrak{L}$ consists of three parts. First, we introduce the set of *statements* $\mathfrak{L}_S$, next the set of *labelled statements* $\mathfrak{L}_{LS}$, and finally the set of *programs* $\mathfrak{L}_{Pr}$.

Definition 3.1 ($\mathfrak{L}_S$, $\mathfrak{L}_{LS}$, $\mathfrak{L}_{Pr}$)
We defined the set $(s,t \in)\,\mathfrak{L}_S$ of statements by

$$s ::= x := e \mid \alpha!m \mid \textbf{answer} \mid s_1;s_2 \mid s_1 + s_2 \mid \textbf{release}(\alpha,s)$$

As a special element, the set $\mathfrak{L}_S$ contains the *empty* statement E, which stands for termination. The set $((\alpha,s)\in)\mathfrak{L}_{LS}$ of labelled statements is given by

$$\mathfrak{L}_{LS} = SLabel \times \mathfrak{L}_S.$$

Finally, we introduce the set $((X,\delta)\in)\,\mathfrak{L}_{Pr}$ of programs:

$$\mathfrak{L}_{Pr} = \mathcal{P}_{fin}(\mathfrak{L}_{LS}) \times Decl,$$

where *Decl* is the set of declarations, defined below. If a set $X \in \mathcal{P}_{fin}(\mathfrak{L}_{LS})$ contains only pairs with E as a statement then we call it *final*:

$$final(X) \Leftrightarrow \forall\alpha\forall s\ [(\alpha,s)\in X \Rightarrow s = E].$$

Definition 3.2 (Declarations)
The set $(\delta\in)$ *Decl* of declarations is given by

$$\delta ::= <m_1 \Leftarrow s_1, \ldots, m_n \Leftarrow s_n>,$$

for $n \geqslant 0$. If $\delta = < \cdots, m \Leftarrow s, \cdots >$ we sometimes write $\delta(m) = s$.

A program is a pair (X,δ): X is a finite set of labelled statements, which are executed in parallel and can communicate with each other by sending messages; δ is a declaration, giving for each method name a corresponding statement. A labelled statement (α,s) consists of a label α and a statement s. Facilitating an understandable explanation of $\mathfrak{L}$ and its semantics, and following the object-oriented terminology that we use in the description of POOL, we shall say that the *object* α executes the statement s. This statement s is the part of the *body* (the statement with which α starts at the beginning of the program) of the object α that still has to be executed. When describing the communication in $\mathfrak{L}$, this convention is found to be particularly useful.

A statement can be an assignment $x := e$, the execution of which is considered to be atomic; that is, both the evaluation of the expression e and the assignment of the result to the variable x take place in one step.

There are two statements for communication: a send statement $\alpha!m$ and an answer statement **answer**. The form of communication we have here is a simplified version of the POOL rendezvous. A send statement specifies an address to which the message is sent together with the name of the method that should be executed, but no parameters. An answer statement does not specify a method name; it indicates that any method can be executed. Successful communication results in the suspension of the sender until the receiver has executed the requested method. Then both the sender and the receiver continue with their own bodies. Note that we do not have value passing here: The execution of a method consists of the execution of the corresponding statement, given by a declaration; no result is sent back to the sender.

Next we have the sequential composition $s_1;s_2$ of two statements with its usual interpretation, and the non-deterministic choice $s_1 + s_2$ between two statements, also called *global* non-determinism. The execution of $s_1 + s_2$ consists of the execution of either s_1 or s_2; since both s_1 and s_2 can begin with a communication statement this choice may be influenced by the environment, that is, by other labelled statements present in the same program.

Finally, there is the release statement **release**(α,s), which is added only to enable an elegant description of the operational semantics of communication. When it is executed, the object α is put to work again and resumes execution with s. Its use will become clear below.

Obviously, there are many differences between POOL and $\mathfrak{L}$, which is much simpler than POOL.

However, for the semantic description of in particular the communication in $\mathfrak{L}$, we need almost all technical tools that we shall use for the semantics of the full language POOL itself. This fact motivates our choice of $\mathfrak{L}$ as a starting point for the semantic study of POOL. (There is one aspect of the methods used in this section that is not really necessary here: since the number of objects for a given program is always finite, and since we do not have a construct for recursion, we are not able to model infinite behavior. Therefore, we could use sets and structural induction rather than complete metric spaces and contractions. We introduce these notions here because we need them in the next section (since in POOL one *can* model infinite behavior).)

3.1 Operational semantics for $\mathfrak{L}$

We assume given a set of *states* $(\sigma \in)\Sigma$, defined by

$$\Sigma = Var \rightarrow Val,$$

where *Val* is some set of values. We shall use $\sigma\{v/x\}$ to denote the state that is like σ but for its value in x, which is v. For the evaluation of expressions, we stipulate an interpretation function

$$\mathfrak{E}: Exp \rightarrow \Sigma \rightarrow Val.$$

The operational semantics of $\mathfrak{L}$ will be based on the *labelled transition system* $<Conf, \Lambda, \rightarrow>$ defined below. It consists of a set *Conf* of *configurations*, a set Λ of *transition labels*, and a *transition relation* $\rightarrow$. The transition relation specifies transitions between configurations:

$$\rightarrow \subseteq Conf \times \Lambda \times Conf.$$

The label of such a transition gives some information on the kind of step that is is modelled (e.g., a computation step or a communication one). Using a transition system, we can model each of the possible executions of a program as a sequence of transitions, starting in some initial configuration. The transition system is defined as follows.

Definition 3.3 (Transition system)
Let $\delta \in Decl$ be fixed. We define a labelled transition system $<Conf, \Lambda, \rightarrow>$, where the sets $(<X,\sigma> \in)Conf$ of configurations and $(\lambda \in)\Lambda$ of transition labels are given by

$$Conf = \mathcal{P}_{fin}(\mathfrak{L}_{LS}) \times \Sigma,$$

$$\Lambda = \{\tau\} \cup \{(\alpha, \beta!m): \alpha,\beta \in SLabel, m \in MName\} \cup \{\alpha?: \alpha \in SLabel\}.$$

The label τ will be used to denote a computation step; $(\alpha,\beta!m)$ and $\alpha?$ will denote send and answer steps. We define $\rightarrow$ as the smallest relation satisfying the following axioms and rules:

Axioms

(A1) $<\{(\alpha, x:=e)\}, \sigma> \xrightarrow{\tau} <\{(\alpha,E)\}, \sigma\{\mathfrak{E}[\![e]\!](\sigma)/x\}>$

(A2) $<\{(\alpha,\beta!m)\}, \sigma> \xrightarrow{(\alpha,\ \beta!m)} <\{(\alpha,E)\}, \sigma>$

(A3) $<\{(\alpha,$ **answer**$)\}, \sigma> \xrightarrow{\alpha?} <\{(\alpha,E)\}, \sigma>$

Rules

(R1) If $<\{(\alpha,s)\}, \sigma> \xrightarrow{\lambda} <\{(\alpha,s')\}, \sigma'>$

then $<\{(\alpha,s;t)\}, \sigma> \xrightarrow{\lambda} <\{(\alpha,s';t)\}, \sigma'>$

(read t instead of $s';t$ if $s'=E$)

$$<\{(\alpha, s+t)\}, \sigma> \xrightarrow{\lambda} <\{(\alpha, s')\}, \sigma'>$$

$$<\{(\alpha, t+s)\}, \sigma> \xrightarrow{\lambda} <\{(\alpha, s')\}, \sigma'>$$

(R2) If $<\{(\beta, t), (\alpha, s)\}, \sigma> \xrightarrow{\lambda} <X, \sigma'>$

then $<\{(\beta, \textbf{release}(\alpha, s); t)\}, \sigma> \xrightarrow{\lambda} <X, \sigma'>$

(R3) If $<X, \sigma> \xrightarrow{\lambda} <X', \sigma'>$

then $<X \cup Y, \sigma> \xrightarrow{\lambda} <X' \cup Y, \sigma'>$

(R4) If $<X, \sigma> \xrightarrow{(\alpha, \beta!m)} <\{(\alpha, s)\} \cup X', \sigma>$ and

$<Y, \sigma> \xrightarrow{\beta?} <\{(\beta, t)\} \cup Y', \sigma>$

then $<X \cup Y, \sigma> \xrightarrow{\tau} <\{(\beta, \delta(m); \textbf{release}(\alpha, s); t)\} \cup X' \cup Y', \sigma>$

(Note that the fixed declaration δ occurs only in rule R4.)

The label $(\alpha, \beta!m)$ used in axiom A2 indicates that the object α sends a message to object β, requesting the execution of method m. In A3, the label $\alpha?$ indicates that object α is willing to answer any message.

The interpretation of RI is straightforward. In R2, it is expressed that the execution of **release**(α, s) amounts to the extension of the current set of labelled statements with (α, s). Labelled statements are executed in an interleaved way, according to R3: A set of labelled statements is evaluated by repeatedly performing a step of one of its elements.

In rule R4, communication is described. If object α is sending a message to object β, requesting the execution of method m, and if object β is willing to answer a message, then the following happens. Object β starts executing the statement $\delta(m)$, which corresponds to the definition of method m according to the declaration δ. Next, the statement **release**(α, s) is executed, reactivating object α, which will continue with s, the remainder of its body. Finally, object β proceeds with t, the remainder of its own body. Note that during the execution of method m, that is, the statement $\delta(m)$, object α is non-active, as can be seen from the fact that α does not occur as the name of any labelled statement in the configuration resulting from this transition.

Now we are almost ready to define the operational semantics for $\mathfrak{L}$. First, we introduce its *semantic universe.*

Definition 3.4 (Semantic universe *P*)
Let $(w \in)\Sigma_\partial^\infty = \Sigma^* \cup \Sigma^* \cdot \{\partial\} \cup \Sigma^\omega$. We put

$$(p, q \in)\ P = \Sigma \to \mathcal{P}(\Sigma_\partial^\infty),$$

where $\mathcal{P}(\Sigma_\partial^\infty)$ denotes the set of all subsets of Σ_∂^∞, and the symbol ∂ denotes *deadlock.*

The elements of P will be used to represent the operational meanings of statements and units. For a given state $\sigma \in \Sigma$, the set $p(\sigma)$ contains streams $w \in \Sigma_\partial^\infty$, which are sequences of states representing possible computations. They can be of one of three forms: If $w \in \Sigma^*$, it stands for a finite normally terminating computation. If $w \in \Sigma^\omega$, it represents an infinite computation. Finally, if $w \in \Sigma^* \cdot \{\partial\}$, it reflects a finite abnormally terminating computation, which is indicated by the symbol ∂ for deadlock.

Definition 3.5 (Operational semantics for $\mathfrak{L}$)
We define

$$\mathcal{O}_{Pr}: Prog \to P$$

as follows. Let $(X, \delta) \in Prog$ and $\sigma \in \Sigma$. For a word $w \in \Sigma_\partial^\infty$ we put

$$w \in \mathcal{O}_{Pr}[\![(X,\delta)]\!](\sigma)$$

if and only if one of the following conditions is satisfied:

(1) $w = \sigma_0 \cdots \sigma_n$ $(n \geq 0)$ and there exist $X_0, \ldots, X_n$ such that $<X_0,\sigma_0> = <X,\sigma>$ and

$$<X_0,\sigma_0> \xrightarrow{\tau} <X_1,\sigma_1> \xrightarrow{\tau} \cdots \xrightarrow{\tau} <X_n,\sigma_n> \text{ and } \mathit{final}(X_n)$$

(2) $w = \sigma_0\sigma_1 \cdots$ and there exist $X_0, X_1, \ldots$ such that $<X_0,\sigma_0> = <X,\sigma>$ and

$$<X_0,\sigma_0> \xrightarrow{\tau} <X_1,\sigma_1> \xrightarrow{\tau} <X_2,\sigma_2> \xrightarrow{\tau} \cdots$$

(3) $w = \sigma_0 \cdots \sigma_n \cdot \partial$ $(n \geq 0)$ and there exist $X_0, \ldots, X_n$ such that $<X_0,\sigma_0> = <X,\sigma>$ and

$$<X_0,\sigma_0> \xrightarrow{\tau} <X_1,\sigma_1> \xrightarrow{\tau} \cdots \xrightarrow{\tau} <X_n,\sigma_n>$$

with $\neg\, \mathit{final}(X_n)$ and $\neg <X_n,\sigma_n> \xrightarrow{\tau}$.

Here $\neg <X_n,\sigma_n> \xrightarrow{\tau}$ is an abbreviation for $\neg\exists <X',\sigma'> [<X_n,\sigma_n> \xrightarrow{\tau} <X',\sigma'>]$. (Note that in the transition sequences above we consider only computation steps that are labelled by τ.)

Case (1) represents a normally terminating computation and case (2) stands for an infinite computation. In case (3), a deadlocking computation is described: If after a number of computation steps a configuration $<X_n,\sigma_n>$ is reached which is not final and from which no computation steps are possible, this implies that from there only single-sided communication steps are possible, for which there is no matching communication partner present. This we consider as a case of deadlock, which is indicated by ∂.

Anticipating the definition of a denotational semantics for $\mathfrak{L}$ and the proof of its correctness with respect to $\mathcal{O}_{Pr}$, we next give a fixed-point characterization of $\mathcal{O}_{Pr}$. As indicated at the beginning of this section, this will happen in the context of complete metric spaces. Therefore, we first turn P into a complete metric space.

Definition 3.6 (P as a complete metric space)
We redefine the semantic universe P by putting

$$P = \Sigma \to \mathcal{P}_{ncompact}(\Sigma_\partial^\infty),$$

where $\mathcal{P}_{ncompact}(\Sigma_\partial^\infty)$ is the set of all non-empty and compact subsets of Σ_∂^∞. This set is a complete metric space when supplied with the so-called *Hausdorff* metric, induced by the usual metric on Σ_∂^∞ (see Example A.1.1 and Definition A.6(d) of the appendix to this chapter). The metric on P is then defined according to Definition A.6(a).

Definition 3.7 (Φ)
We define a function

$$\Phi : (Prog \to P) \to (Prog \to P).$$

Let $F \in Prog \to P$, $(X,\delta) \in Prog$, and $\sigma \in \Sigma$. We put

$$\Phi(F)((X,\delta))(\sigma) = \begin{cases} \{\epsilon\} & \text{if } \mathit{final}(X) \\ \{\partial\} & \text{if } \neg <X,\sigma> \xrightarrow{\tau} \wedge \neg\, \mathit{final}(X) \\ \bigcup \{\sigma' \cdot F((X',\delta))(\sigma') : <X,\sigma> \xrightarrow{\tau} <X',\sigma'>\} & \text{otherwise} \end{cases}$$

It is straightforward to prove that Φ is contracting and thus has a unique fixed point. In fact, this fixed point is $\mathcal{O}_{Pr}$:

Theorem 3.8: $\Phi(\mathcal{O}_{Pr}) = \mathcal{O}_{Pr}$

Proof

The proof consists of two parts. First, we show that for every program (X,δ) we have that $\mathcal{O}_{Pr}[\![(X,\delta)]\!]$ is in P, that is, for every state σ the set $\mathcal{O}_{Pr}[\![(X,\delta)]\!](\sigma)$ is *compact*. Second, we prove that $\Phi(\mathcal{O}_{Pr})=\mathcal{O}_{Pr}$.

So let $(X,\delta)\in$ *Prog* and $\sigma\in\Sigma$. Let $(w_i)_i$ be a sequence of words in $\mathcal{O}_{Pr}[\![(X,\delta)]\!](\sigma)$ ($\subseteq\Sigma_\partial^\infty$), say,

$$w_i = \sigma_i^1\sigma_i^2\sigma_i^3\cdots .$$

We show that $(w_i)_i$ has a converging subsequence with its limit in $\mathcal{O}_{Pr}[\![(X,\delta)]\!](\sigma)$. Assume for simplicity that all words w_i are infinite. Since $w_i\in\mathcal{O}_{Pr}[\![(X,\delta)]\!](\sigma)$ for every i, there exist infinite transition sequences such that

$$<X,\sigma>\rightarrow<X_i^1,\sigma_i^1>\rightarrow<X_i^2,\sigma_i^2>\rightarrow\cdots$$

(omitting the labels τ). From the definition of $\rightarrow$ it follows that the set

$$\{<X',\sigma'>:\ <X,\sigma>\rightarrow<X',\sigma'>\}$$

is finite. Thus there exists a pair $<X_1,\sigma_1>$ such that for infinitely many i's:

$$<X_i^1,\sigma_i^1> = <X_1,\sigma_1>.$$

Let $f_1:\mathbb{N}\rightarrow\mathbb{N}$ be a monotonic function with, for all i,

$$<X_{f_1(i)}^1,\sigma_{f_1(i)}^1> = <X_1,\sigma_1>.$$

Next we proceed with the subsequence $(w_{f_1(i)})_i$ of $(w_i)_i$ and repeat the above argument, now with respect to the set

$$\{<X',\sigma'>:\ <X_1,\sigma_1>\rightarrow<X',\sigma'>\}.$$

Continuing in this way we find a sequence of monotonic functions $(f_k)_k$, defining a sequence of subsequences of $(w_i)_i$, and a sequence of configurations $(<X_k,\sigma_k>)_k$ such that

$$\forall k\ \forall j\ \forall i\leqslant k\ [\sigma_{f_k(j)}^i = \sigma_i]$$

and

$$<X,\sigma>\rightarrow<X_1,\sigma_1>\rightarrow<X_2,\sigma_2>\rightarrow\cdots$$

and moreover such that the sequence $(w_{f_{k+1}(i)})_i$ is a subsequence of the sequence of $(w_{f_k(i)})_i$. Now we define

$$g(i) = f_i(i).$$

Then we have

$$\lim_{i\rightarrow\infty} w_{g(i)} = \sigma_1\sigma_2\sigma_3\cdots .$$

Thus we have constructed a converging subsequence of $(w_i)_i$ with its limit in $\mathcal{O}_{Pr}[\![(X,\delta)]\!](\sigma)$. (In case not all w_i are infinite a similar argument can be given.)

Second, we show that $\Phi(\mathcal{O}_{Pr})=\mathcal{O}_{Pr}$. Let $(X,\delta)\in$*Prog* with $\neg$ *final*(X), let $\sigma\in\Sigma$ and let $w\in\Sigma_\partial^\infty$. If $w=\partial$ then

$$w\in\Phi(\mathcal{O}_{Pr})((X,\delta))(\sigma) \Leftrightarrow w\in\mathcal{O}_{Pr}[\![(X,\delta)]\!](\sigma).$$

Now suppose $w\neq\partial$. We have

$$\begin{aligned} w\in\mathcal{O}_{Pr}[\![(X,\delta)]\!](\sigma) &\Leftrightarrow \exists\sigma'\in\Sigma\,\exists X'\in\mathcal{P}_{fin}(LStat)\,\exists w'\in\Sigma_\partial^\infty \\ &\qquad [<X,\sigma>\rightarrow<X',\sigma'>\wedge w=\sigma'\cdot w'\wedge w'\in\mathcal{O}_{Pr}[\![(X',\delta)]\!](\sigma)] \\ &\Leftrightarrow [\text{definition }\Phi] \end{aligned}$$

$$w \in \Phi(\mathcal{O}_{Pr})((X,\delta))(\sigma).$$

So we see: $\mathcal{O}_{Pr} = \Phi(\mathcal{O}_{Pr})$. (End of proof.)

3.2 Denotational semantics for $\mathcal{L}$

The domain that we shall take as the semantic universe of our denotational model is defined by a reflexive domain equation. In De Bakker and Zucker (1982) solving these equations in a metric setting was first described. Then, in America and Rutten (1988), this approach was generalized in order to deal with equations in which P occurs at the left-hand side of a function space constructor, as in $P \cong P \to P$; this case that was not covered by De Bakker and Zucker. For a quick overview of the main results of America and Rutten (1988), the reader may wish to read section 2 of America *et al* (1986b).

Further, our model is based on the use of *continuations*. For an extensive introduction to continuations and expression continuations, which we shall also use, we refer to Gordon (1979); see also De Bruin (1986).

We start with the definition of a domain $\bar{P}$, the elements of which we shall call *processes* from now on. Such a process represents the whole or a part of the execution of a program.

Definition 3.9 (Semantic process domain $\bar{P}$)
Let $(p,q \in)\bar{P}$ be a complete ultra-metric space satisfying the following reflexive domain equation:

$$\bar{P} \cong \{p_0\} \cup id_{1/2}(\Sigma \to \mathcal{P}_{compact}(Step_{\bar{P}})),$$

where $(\pi,\rho \in) Step_{\bar{P}}$ is

$$Step_{\bar{P}} = Comp_{\bar{P}} \cup Send_{\bar{P}} \cup Answer_{\bar{P}},$$

with

$$Comp_{\bar{P}} = \Sigma \times \bar{P},$$
$$Send_{\bar{P}} = SLabel \times MName \times \bar{P} \times \bar{P},$$
$$Answer_{\bar{P}} = SLabel \times (MName \to \bar{P} \to {}^{1}\bar{P}).$$

(The sets $\{p_0\}$, Σ, *SLabel*, and *MName* become complete ultra-metric spaces by supplying them with the discrete metric (see Example A.1.1 of the appendix to this chapter); $\bar{P} \to {}^{1}\bar{P}$ denotes the set of all non-expansive functions (A.3.(c)) from $\bar{P}$ to itself.)

America and Rutten (1988) describe how to find for such an equation a solution, which is unique up to isomorphy. Let us try to explain intuitively the intended interpretation of the domain $\bar{P}$. First, we observe that in the equation above the operation $id_{1/2}$ is necessary only to guarantee that the equation is solvable by defining a contracting functor on $\mathcal{C}$, the category of complete metric spaces. For a, say, more operational understanding of the equation this is not important.

A process $p \in \bar{P}$ is either p_0 or a function from Σ to $\mathcal{P}_{compact}(Step_{\bar{P}})$, the set of all *compact* subsets of $Step_{\bar{P}}$. The process p_0 is the terminated process. For $p \neq p_0$, the process p has the choice, depending on the current state σ, among the *steps* in the set $p(\sigma)$. If $p(\sigma) = \varnothing$, then no further action is possible, which is interpreted as abnormal termination (deadlock). For $p(\sigma) \neq \varnothing$, each step $\pi \in p(\sigma)$ consists of some action (for instance, a change of the state σ or an attempt at communication) and a *resumption* of this action, that is, the remaining actions to be taken after this action. There are three different types of steps $\pi \in Step_{\bar{P}}$.

First, a step may be an element of $\Sigma \times \bar{P}$, say

$$\pi = \langle \sigma', p' \rangle.$$

The only action is a change of state: σ' is the new state. Here the process p' is the resumption, indicating the remaining actions process p can do. (When $p'=p_0$ no steps can be taken after this step π.)

Second, π might be a *send step*, $\pi \in Send_{\bar{P}}$. In this case we have, say

$$\pi = <\alpha,m,p_1,p_2>,$$

with $\alpha \in SLabel$, $m \in MName$, and $p_1, p_2 \in \bar{P}$. The action involved here consists of an attempt at communication, in which a message is sent to the object α, specifying the method m. This is the interpretation of the first two components α and m. The third component p_1, called the *dependent* resumption of this send step, indicates the steps that will be taken when the sender becomes active again, that is, after the execution of the method. The last component p_2, called the *independent* resumption of this send step, represents the steps to be taken after this send step that need *not* wait for the result of the method execution. In general, this process p_2 is the result of the parallel composition of this send step with some other process (see Definition 3.10 below).

Finally, π might be an element of $Answer_{\bar{P}}$, say

$$\pi = <\alpha,g>$$

with $\alpha \in SLabel$, and $g \in (MName \to \bar{P} \to {}^1\bar{P})$. It is then called an *answer step*. The first component of π expresses that object α is willing to accept a message. The last component g, the resumption of this answer step, specifies what should happen when an appropriate message actually arrives. The function g is then applied to the method name specified in this message and to the dependent resumption of the sender (specified in its corresponding send step). It then delivers a process which is the resumption of the sender and the receiver *together*, which is to be composed in parallel with the independent resumption of the send step.

We now define a semantic operator for the *parallel composition* (or *merge*) of two processes, for which we shall use the symbol $\|$. It is *auxiliary* in the sense that it does not correspond to a syntactic operator in the language $\mathfrak{L}$.

Definition 3.10 (Parallel composition)
Let $\| : \bar{P} \times \bar{P} \to \bar{P}$ be such that it satisfies the following equation:

$$p\|q = \lambda\sigma\cdot((p(\sigma)\mathbin{\hat{\|}} q) \cup (q(\sigma)\mathbin{\hat{\|}} p) \cup (p(\sigma)\,|_\sigma\, q(\sigma))),$$

for all $p,q \in \bar{P} \setminus \{p_0\}$, and such that $p_0\|q = q\|p_0 = p_0$. Here, $X\mathbin{\hat{\|}} q$ and $X\,|_\sigma\, Y$ are defined by:

$$X\mathbin{\hat{\|}} q = \{\pi\mathbin{\hat{\|}} q : \pi \in X\},$$

$$X\,|_\sigma\, Y = \bigcup\{\pi\,|_\sigma\,\rho : \pi \in X,\ \rho \in Y\},$$

where $\pi\mathbin{\hat{\|}} q$ is given by

$$<\sigma',p'>\mathbin{\hat{\|}} q = <\sigma',p'\|q>,$$

$$<\alpha,m,p_1,p_2>\mathbin{\hat{\|}} q = <\alpha,m,p_1,p_2\|q>, \text{ and}$$

$$<\alpha,g>\mathbin{\hat{\|}} q = <\alpha,\lambda m\cdot\lambda p\cdot(g(m)(p)\|q)>,$$

and $\pi|_\sigma\rho$ by

$$\pi|_\sigma\rho = \begin{cases} \{<\sigma,\ g(m)(p_1)\|p_2>\} & \text{if } \pi = <\alpha,m,p_1,p_2> \text{ and } \rho = <\alpha,g> \\ & \text{or } \rho = <\alpha,m,p_1,p_2> \text{ and } \pi = <\alpha,g> \\ \varnothing & \text{otherwise.} \end{cases}$$

We observe that this definition is self-referential, since the merge operator occurs at the right-hand side of the definition. For a formal justification of this definition see the appendix of America *et al* (1986b), where a similar merge operator is given as the unique fixed point of a contraction on

$\bar{P} \times \bar{P} \to {}^1\bar{P}$.

Since we intend to model parallel composition by interleaving, the merge of two processes p and q consists of three parts. The first part contains all possible first steps of p followed by the parallel composition of their respective resumptions with q. The second part similarly contains the first steps of q. The last part contains the communication steps that result from two matching communication steps taken simultaneously by processes p and q. For $\pi \in Step_{\bar{P}}$ the definition of $\pi \| q$ is straightforward. The definition of $\pi|_\sigma \rho$ is more involved. It is the empty set if π and ρ do not match. Now suppose they do match, say $\pi = <\alpha, m, p_1, p_2>$ and $\rho = <\alpha, g>$. Then π is a *send* step, denoting a request to object α to execute method m, and ρ is an *answer* step, denoting that object α is willing to accept a message. In $\pi|_\sigma \rho$, the state σ remains unaltered. The function g, the second component of ρ, needs as an argument the method name m. Moreover, g depends on the dependent resumption p_1 of the send step π. This explains why both m and p_1 are supplied as arguments to the function g. Now it can be seen that $g(m)(p_1) \| p_2$ represents the resumption of the sender and the receiver together. (In order to obtain more insight into this definition it is advisable to return to it after having seen the definition of the semantics of an answer statement.)

The merge operator is associative, which can easily be proved as follows. Define

$$\epsilon = \sup_{p,q,r \in \bar{P}} \{d_{\bar{P}}((p \| q) \| r, p \| (q \| r))\}$$

Then, using the fact that the operator $\|$ satisfies the equation above, one can show that $\epsilon \leqslant \frac{1}{2} \cdot \epsilon$, hence $\epsilon = 0$.

Now we come to the definition of the semantics of $\mathfrak{L}$. First, we introduce the semantics $\mathfrak{D}_S$ of statements; next, the semantics $\mathfrak{D}_{Pr}$ of programs is defined.

The semantics of statements will be given by a function

$$\mathfrak{D}_S : \mathfrak{L}_S \to SLabel \to Decl \to Cont \to \bar{P},$$

where the set of continuations *Cont* is given by

$$Cont = \bar{P}.$$

Let $s \in \mathfrak{L}_S$, $\alpha \in SLabel$, $\delta \in Decl$, and $p \in \bar{P}$. The semantic value of the statement s is given by

$$\mathfrak{D}_S[\![s]\!](\alpha)(\delta)(p).$$

The statement label α is of importance in case s contains some communication statement. Second, the semantic value of s depends on the declaration δ. Finally, the last argument for $\mathfrak{D}_S$ is the continuation p: the semantic value of everything that will happen after the execution of s. The use of continuations enables us to describe the semantics of $\mathfrak{L}$ compositionally and, moreover, in a concise and elegant way.

Please note the difference between the notions of *resumption* and *continuation*. A resumption is a part of a semantic step $\pi \in Step_{\bar{P}}$, indicating the remaining steps to be taken after the current one. A continuation, on the other hand, is an argument to a semantic function. It may appear as a resumption in the result. A good example of this is the definition of $\mathfrak{D}_S[\![x := e]\!](\alpha)(\delta)(p)$ below.

Definition 3.11 (Denotational semantics of statements)
We define

$$\mathfrak{D}_S : \mathfrak{L}_S \to SLabel \to Decl \to Cont \to {}^1\bar{P},$$

with $Cont = \bar{P}$, as follows:

(0) $\mathfrak{D}_S[\![E]\!](\alpha)(\delta)(p) = p$

(1) $\mathfrak{D}_S[\![x := e]\!](\alpha)(\delta)(p) = \lambda\sigma \cdot \{<\sigma\{\mathfrak{E}[\![e]\!](\sigma)/x\}, p>\}$

(2) $\mathfrak{D}_S[\![\beta ! m]\!](\alpha)(\delta)(p) = \lambda\sigma \cdot \{<\beta, m, p, p_0>\}$

(3) $\mathfrak{D}_S[\![\textbf{answer}]\!](\alpha)(\delta)(p) = \lambda\sigma \cdot \{<\alpha, \lambda m \cdot \lambda q \cdot \mathfrak{D}_S[\![\delta(m)]\!](\alpha)(\delta)(p \| q)>\}$

(4) $\mathfrak{D}_S[\![s_1 ; s_2]\!](\alpha)(\delta)(p) = \mathfrak{D}_S[\![s_1]\!](\alpha)(\delta)(\mathfrak{D}_S[\![s_2]\!](\alpha)(\delta)(p))$

(5) $\mathfrak{D}_S[\![s_1 + s_2]\!](\alpha)(\delta)(p) = \lambda\sigma \cdot (\mathfrak{D}_S[\![s_1]\!](\alpha)(\delta)(p)(\sigma) \cup \mathfrak{D}_S[\![s_2]\!](\alpha)(\delta)(p)(\sigma))$

(6) $\mathfrak{D}_S[\![\textbf{release}(\beta, s)]\!](\alpha)(\delta)(p) = \mathfrak{D}_S[\![s]\!](\beta)(\delta)(p_0) \| p$

This definition needs some justification since it cannot be defined by a simple induction on the complexity of statements, as is apparent from case (3) above. We can give a formally correct definition as follows. If we put

$$S = \mathcal{L}_S \to SLabel \to Decl \to Cont \to {}^1\overline{P}$$

we can define $\mathcal{D}_S$ as the fixed point of a contraction

$$\Psi: S \to S$$

given, for $F \in S$, $\alpha \in SLabel, \delta \in Decl$, and $p \in Cont$, by

(0) $\Psi(F)(E)(\alpha)(\delta)(p) = p$

(1) $\Psi(F)(x:=e)(\alpha)(\delta)(p) = \lambda\sigma\cdot\{<\sigma\{\mathcal{E}[\![e]\!](\sigma)/x\}, p>\}$

(2) $\Psi(F)(\beta!m)(\alpha)(\delta)(p) = \lambda\sigma\cdot\{<\beta,m,p,p_0>\}$

(3) $\Psi(F)$ **(answer)** $(\alpha)(\delta)(p) = \lambda\sigma\cdot\{<\alpha, \lambda m\cdot\lambda q\cdot F(\delta(m))(\alpha)(\delta)(p \| q)>\}$

(4) $\Psi(F)(s_1;s_2)(\alpha)(\delta)(p) = \Psi(F)(s_1)(\alpha)(\delta)(\Psi(F)(s_2)(\alpha)(\delta)(p))$

(5) $\Psi(F)(s_1+s_2)(\alpha)(\delta)(p) = \lambda\sigma\cdot(\Psi(F)(s_1)(\alpha)(\delta)(p)(\sigma) \cup \Psi(F)(s_2)(\alpha)(\delta)(p)(\sigma))$

(6) $\Psi(F)$ **(release**$(\beta,s))(\alpha)(\delta)(p) = \Psi(F)(s)(\beta)(\delta)(p_0) \| p.$

We give some motivation for the definition of $\mathcal{D}_S$. In case (1), the expression e is evaluated and its result is at once assigned to x.

The evaluation of a send statement, in case (2), results in a process containing a send step $<\beta,m,p,p_0>$. Here β refers to the object to which a message is sent, which requests the execution of the method m. The dependent resumption of this send step consists of the continuation p, indicating that the activity of object α, which is executing the send statement, is suspended until the message has been answered and the method has been executed. The independent resumption of this sent step is initialized to p_0.

The function g in the answer step in case (3) represents the execution of an arbitrary method followed by its continuation. It takes as arguments a method name m, indicating the method that will be executed, and a continuation q, both to be received from an object sending a message to α. The execution of method m consists of the execution of the statement $\delta(m)$. Next, both the continuation q of the sending object and the given continuation p are to be executed in parallel. This explains the continuation $p \| q$ in $\mathcal{D}_S[\![\delta(m)]\!]$.

Let us look more closely at the definition of $\pi|_\sigma\rho$ (Definition 3.10) now that we have defined the semantics of send and answer statements. Let $\pi = <\alpha,m,p_1,p_2>$ be the result of the evaluation of a send statement and let $<\alpha,g>$ stem from an answer statement. We have that

$$\pi|_\sigma\rho = \{<\sigma, g(m)(p_1) \| p_2>\}.$$

The execution of the method m proceeds in parallel with the independent resumption p_2 of the sender. From the definition of the semantics of an answer step it follows that

$$g(m)(p_1) = \mathcal{D}_S[\![\delta(m)]\!](\alpha)(p_1 \| q),$$

for some continuation q. The continuation of the execution of m is given by $p_1 \| q$, the parallel composition of continuations p_1 and q, reflecting the fact that after the rendezvous both the sender and the receiver of the message can proceed in parallel again. (Of course, the independent resumption p_2 may still be executing at this point.)

Continuing with the explanation of Definition 3.11 above, we come to case (6), since the semantics of $s_1;s_2$ and s_1+s_2 is defined straightforwardly. In the definition of the semantics of a release statement, the process $\mathcal{D}_S[\![s]\!](\beta)(\delta)(p_0)$ represents the meaning of the execution of s by object β with the empty continuation p_0, indicating that after s nothing remains to be done. This process is put in parallel with p, the continuation of the release statement.

Note that the function $\mathcal{D}_S$ is compositional, which follows from the observation that it is defined in

a compositional manner.

Next, we introduce the denotational semantics of programs.

Definition 3.12 (Denotational semantics of programs)
We define

$$\mathfrak{D}_{Pr}: Prog \to \bar{P}$$

as follows. Let $(X,\delta) \in Prog$, with $X = \{(\alpha_1,s_1), \ldots, (\alpha_n,s_n)\}$. We put

$$\mathfrak{D}_{Pr}[\![(X,\delta)]\!] = \mathfrak{D}_S[\![s_1]\!](\alpha_1)(\delta)(p_0) \| \cdots \| \mathfrak{D}_S[\![s_n]\!](\alpha_n)(\delta)(p_0).$$

Given a set X of labelled statements, the value of $\mathfrak{D}_{Pr}[\![(X,\delta)]\!]$ is obtained by first computing the semantics of every labelled statement $(\alpha_i,s_i) \in X$, which is given by $\mathfrak{D}[\![s_i]\!](\alpha_i)(\delta)(p_0)$; here p_0 indicates that after s_i nothing remains to be done. Next, the resulting processes are put in parallel.

3.3 Semantic correctness of $\mathfrak{D}_{Pr}$

In this section we prove that

$$\mathfrak{O}_{Pr} = abstr \circ \mathfrak{D}_{Pr},$$

where $abstr : \bar{P} \to P$ is an *abstraction* operation (to be defined below), which relates the semantic universes of $\mathfrak{D}_{Pr}$ and $\mathfrak{O}_{Pr}$. To this end, we shall introduce an intermediate semantics

$$\mathfrak{I}: Prog \to \bar{P},$$

which will be related to both $\mathfrak{O}_{Pr}$ and $\mathfrak{D}_{Pr}$. The equality mentioned above then follows.

The intermediate semantics $\mathfrak{I}$ is defined as the fixed point of a contraction Γ, the definition of which is based on the transition relation given in Definition 3.3.

Definition 3.13 (Intermediate semantics $\mathfrak{I}$)
Let

$$\Gamma: (Prog \to \bar{P}) \to (Prog \to \bar{P})$$

be defined as follows. Let $F \in Prog \to \bar{P}$ and $(X,\delta) \in Prog$. If $final(X)$ we put

$$\Gamma(F)((X,\delta)) = p_0.$$

Otherwise,

$$\Gamma(F)((X,\delta)) = \lambda\sigma.(C_F \cup S_F \cup A_F)$$

where

$$C_F = \{<\sigma',\ F((X',\delta))> : <X,\ \sigma> \xrightarrow{\tau} <X',\sigma'>\}$$

$$S_F = \{<\beta,m,F((\{(\alpha,s)\},\ \delta)),\ F((X',\delta))> : <X,\sigma> \xrightarrow{(\alpha,\beta!m)} <\{(\alpha,s)\} \cup X',\sigma>\}$$

$$A_F = \{<\alpha,g> : <X,\sigma> \xrightarrow{\alpha\ ?} <\{(\alpha,s)\} \cup X',\sigma>\},$$

with

$$g = \lambda m.\lambda p.\ (\mathfrak{D}_S[\![\delta(m)]\!](\alpha)(p \| F((\{(\alpha,s)\},\delta))) \| F((X',\delta))).$$

It is easy to show that Γ is a contraction, so we can define a function $\mathfrak{I}: Prog \to \bar{P}$ by

$$\mathfrak{I} = \text{fixed point } (\Gamma).$$

The function Γ closely resembles the function Φ, given in Definition 3.7. The main difference is that Γ is a contraction on $Prog \to \bar{P}$, whereas Φ is a contraction on $Prog \to P$.

Let $(X,\delta) \in Prog$ and $\sigma \in \Sigma$. The set $\mathcal{G}((X,\delta))(\sigma)$ contains not only computation steps but also single-sided communication steps: Corresponding with every send transition of the form

$$<X,\ \sigma> \xrightarrow{(\alpha,\beta!m)} <\{(\alpha,s)\} \cup X',\ \sigma>,$$

it contains a send step

$$<\beta,\ m,\ \mathcal{G}((\{(\alpha,s)\},\delta)),\ \mathcal{G}((X',\delta))>.$$

Here β and m indicate that a message specifying method m is sent to the object β. The dependent resumption of this step is $\mathcal{G}((\{(\alpha,s)\},\delta))$: the meaning of the statement that will be executed by α as soon as method m has been executed. The last component of this send step, the independent resumption, consists of $\mathcal{G}((X',\delta))$, which is the meaning of all the statements executed by objects other than α. Thus it is reflected that these objects need not wait until the message is answered; they may proceed in parallel.

Next, $\mathcal{G}((X,\delta))(\sigma)$ can contain some answer steps. For every answer transition

$$<X,\sigma> \xrightarrow{\alpha\ ?} <\{(\alpha,s)\} \cup X',\sigma>$$

the set $\mathcal{G}((X,\delta))(\sigma)$ includes an answer step

$$<\alpha,g>,$$

with

$$g\ =\ \lambda m{\cdot}\lambda p{\cdot}\ (\mathcal{D}_S[\![\delta(m)]\!](\alpha)(p \| \mathcal{G}((\{(\alpha,s)\},\delta))) \parallel \mathcal{G}((X',\delta))).$$

This indicates that object α is willing to answer a message, while the resumption g indicates what should happen when an appropriate message arrives. This function g, when supplied with a method name m and a dependent resumption p (both to be received from the sending object), consists of the parallel composition of the process $\mathcal{G}((X',\delta))$ together with the process

$$\mathcal{D}_S[\![\delta(m)]\!](\alpha)(p \parallel \mathcal{G}((\{(\alpha,s)\},\delta))).$$

(Note that we have used the function $\mathcal{D}_S$ here; the definition of $\mathcal{G}$ therefore depends on its definition.) The process $\mathcal{G}((X',\delta))$ represents the meaning of all the statements executed by objects other than object α: these objects may proceed in parallel with the execution of method m, the meaning of which is indicated by the second process. Its interpretation is the same as in the definition of $\mathcal{D}_S[\![\ \textbf{answer}\ ]\!](\alpha)(\delta)(p)$ in the previous section but for the fact that here the last resumption of this process consists of $p \| \mathcal{G}((\{(\alpha,s)\},\delta))$: the parallel composition of the dependent resumption of the sender and the meaning of the statement s, with which object α will continue after it has answered the message.

The abstraction operation *abstr*: $\bar{P} \to P$ is given below.

Definition 3.14 (Abstraction operation *abstr*)
Let $p \in \bar{P}$, $\sigma \in \Sigma$, and $w \in \Sigma_\partial^\infty$.

(1) We call w a *finite stream* in $p(\sigma)$ if there exist $<\sigma_1,p_1>, \ldots, <\sigma_n,p_n>$ such that

$$w = \sigma_1 \cdots \sigma_n \wedge <\sigma_1,p_1> \in p(\sigma) \wedge \forall 1 \leqslant i < n\ [<\sigma_{i+1},p_{i+1}> \in p_i(\sigma_i)] \wedge p_n = p_0.$$

(2) We call w an *infinite stream* in $p(\sigma)$ if there exist $<\sigma_1,p_1>, <\sigma_2,p_2>, \ldots$ such that

$$w = \sigma_1\sigma_2 \cdots \wedge <\sigma_1,p_1> \in p(\sigma) \wedge \forall i \geqslant 1\ [<\sigma_{i+1},p_{i+1}> \in p_i(\sigma)].$$

(3) We call w a *deadlocking stream* in $p(\sigma)$ if there exist $<\sigma_1,p_1>, \ldots, <\sigma_n,p_n>$ such that

$$w\ =\ \sigma_1 \cdots \sigma_n{\cdot}\partial \wedge <\sigma_1,p_1> \in p(\sigma) \wedge \forall 1 \leqslant i < n\ [<\sigma_{i+1},p_{i+1}> \in p_i(\sigma_i)] \wedge$$

$$p_n \neq p_0 \wedge p_n(\sigma_n) \cap (\Sigma \times \bar{P}) = \varnothing.$$

Now we define a function *abstr*: $\bar{P} \to P$ by $abstr(p_0) = \lambda\sigma\cdot\{\epsilon\}$ and, for $p \neq p_0$, by

$$abstr(p) = \lambda\sigma\cdot\{w\colon w \text{ is a stream in } p(\sigma)\}.$$

The function *abstr* transforms a process $p \in \bar{P}$ into a function $abstr(p) \in P = \Sigma \to \mathfrak{P}_{compact}(\Sigma_\partial^\infty)$, which yields for every $\sigma \in \Sigma$ a set $abstr(p)(\sigma)$ of streams. (If one regards the process p as a tree-like structure, these streams can be considered the branches of p.) There are three kinds of streams: finite, infinite and deadlocking streams, which all correspond to a similar type of computation. A stream in p for a given state σ is computed as follows. If $p = p_0$, we have finished; we have found a finite stream. If $p(\sigma) \cap Comp_{\bar{P}} = \varnothing$ we cannot proceed because single-sided communication is not possible. In that case, a symbol ∂ is delivered, for deadlock. If $p(\sigma)$ does contain a computation step, say $<\sigma_1, p_1>$, the new state σ_1 is taken as the first element of the stream and is passed on as an argument to the resumption p_1. Next, we look for a second computation step in $p_1(\sigma_1)$. Continuing this way, we can construct all streams in $p(\sigma)$.

Now we want to prove

$$\mathcal{O}_{Pr} = abstr \circ \mathcal{G}.$$

It is found to be convenient to use the fixed-point characterizations $\mathcal{O}_{Pr} = \Phi(\mathcal{O}_{Pr})$ and $\mathcal{G} = \Gamma(\mathcal{G})$ for the proof; moreover, we shall also use a fixed-point property for *abstr*, which we prove next.

Theorem 3.15
We define $\Xi\colon(\bar{P} \to {}^1P) \to (\bar{P} \to {}^1P)$. Let $F \in \bar{P} \to {}^1P$, $P \in \bar{P}$ and $\sigma \in \Sigma$. We put

$$\Xi(F)(p_0)(\sigma) = \{\epsilon\},$$

$$\Xi(F)(p)(\sigma) = \{\partial\}, \quad \text{if } p(\sigma) \cap Comp_{\bar{P}} = \varnothing.$$

(Recall that $Comp_{\bar{P}} = \Sigma \times \bar{P}$.) Otherwise, we set

$$\Xi(F)(p)(\sigma) = \bigcup \{\sigma' \cdot F(p')(\sigma')\colon <\sigma', p'> \in p(\sigma)\}.$$

Now we have:

$$abstr = \Xi(abstr).$$

Proof
First, we have to verify that Ξ is well defined, that is, that for every $F \in \bar{P} \to {}^1P$, $p \in \bar{P}$, and $\sigma \in \Sigma$ the set $\Xi(F)(p)(\sigma)$ is compact. This is proved in Appendix II of Rutten (1988). Second, we have to show, similarly to the proof of Theorem 3.8, that

(1) For every $p \in \bar{P}$, and $\sigma \in \Sigma$: $abstr(p)(\sigma)$ is a compact set.
(2) $\Xi(abstr) = abstr$.

For the proof of part (1), which is not trivial, we refer once again to Appendix II of Rutten (1988). Here we show only part (2). Consider $p \in \bar{P} - \{p_0\}$ and $\sigma \in \Sigma$ such that $p(\sigma) \cap (\Sigma \times \bar{P}) \neq \varnothing$. Then:

$$\begin{aligned} w \in abstr(p)(\sigma) &\Leftrightarrow [\text{definition } abstr] \\ &\quad \exists \sigma' \in \Sigma\, \exists w' \in \Sigma_\partial^\infty\, \exists p' \in \bar{P}\ [w = \sigma' \cdot w' \wedge w' \in abstr(p')(\sigma')] \\ &\Leftrightarrow [\text{definition } \Xi] \\ &\quad w \in \Xi(abstr)(p)(\sigma). \end{aligned}$$

The other cases are easy. We see: $abstr = \Xi(abstr)$.

Now we are ready to prove the following theorem.

Theorem 3.16: $\forall F \in Prog \to \bar{P}\ [\Phi(abstr \circ F) = abstr \circ (\Gamma(F))]$

Proof
Let $F \in Prog \to \bar{P}$, $(X,\delta) \in Prog$, and $\sigma \in \Sigma$. Suppose $\neg final(X)$. If $\neg <X,\sigma> \xrightarrow{\tau}$, then

$$\Phi(abstr \circ F)((X,\delta))(\sigma) = \{\partial\}$$
$$= abstr(\Gamma(F)((X,\delta)))(\sigma)$$

since $\Gamma(F)((X,\delta))(\sigma) \cap Comp_{\bar{P}} = \varnothing$. If $<X,\sigma> \xrightarrow{\tau}$ we have

$$\Phi(abstr \circ F)((X,\delta))(\sigma) = \bigcup \{\sigma' \cdot (abstr \circ F)((X',\delta))(\sigma') : <X,\sigma> \xrightarrow{\tau} <X',\sigma'>\}$$
$$= \bigcup \{\sigma' \cdot (abstr(F((X',\delta)))(\sigma')) : <X,\sigma> \xrightarrow{\tau} <X',\sigma'>\}$$
$$= [\text{Theorem 3.15, Definition 3.13}]$$
$$abstr(\lambda\sigma \cdot C_F)(\sigma)$$
$$= abstr(\lambda\sigma \cdot (C_F \cup S_F \cup A_F))(\sigma)$$
$$= abstr(\Gamma(F)((X,\delta)))(\sigma)$$
$$= (abstr \circ \Gamma(F))((X,\delta))(\sigma).$$

Since Φ and Γ are contractions and thus have unique fixed points, the following corollary is immediate.

Corollary 3.17: $\mathcal{O}_{Pr} = abstr \circ \mathcal{G}$

Finally, we shall show that $\mathcal{D}_{Pr}$ and $\mathcal{G}$ are equal. This follows from the following theorem.

Theorem 3.18: $\Gamma(\mathcal{D}_{Pr}) = \mathcal{D}_{Pr}$

Proof
We prove: For every $(X,\delta) \in Prog$

$$\Gamma(\mathcal{D}_{Pr})((X,\delta)) = \mathcal{D}_{Pr}[\![(X,\delta)]\!],$$

using induction on the number of elements in X.

Case (1): $X = \{(\alpha,s)\}$

(1.1) $s = x := e$

$$\Gamma(\mathcal{D}_{Pr})((\{(\alpha,x:=e)\},\delta)) = \lambda\sigma \cdot \{<\sigma\{\mathcal{E}[\![e]\!](\sigma)/x\}, p_0>\}$$
$$= \mathcal{D}_S[\![x:=e]\!](\alpha)(\delta)(p_0)$$
$$= \mathcal{D}_{Pr}[\![(\{(\alpha,x:=e)\},\delta)]\!]$$

(1.2) $s = \beta!m$

$$\Gamma(\mathcal{D}_{Pr})((\{\alpha,\beta!m)\},\delta)) = \lambda\sigma \cdot \{<\beta,m,p_0,p_0>\}$$
$$= \mathcal{D}_S[\![\beta!m]\!](\alpha)(\delta)(p_0)$$
$$= \mathcal{D}_{Pr}[\![(\{a,\beta!m)\},\delta)]\!]$$

(1.3) $s =$ **answer**

$\Gamma(\mathfrak{D}_{Pr})((\{(\alpha,\ \textbf{answer})\},\ \delta)) = \lambda\sigma\cdot\{<\alpha,g>\}$,

where $g = \lambda m\cdot\lambda p\cdot\ \mathfrak{D}_S[\![\delta(m)]\!](\alpha)(\delta)(p \| \mathfrak{D}_{Pr}(\{(\alpha,E)\},\ \delta))$

$= \lambda m\cdot\lambda p\cdot\mathfrak{D}_S[\![\delta(m)]\!](\alpha)(\delta)(p)$

Now $\lambda\sigma\cdot\{<\alpha,g>\} = \mathfrak{D}_S[\![\textbf{answer}]\!](\alpha)(\delta)(p_0)$

$= \mathfrak{D}_{Pr}[\![(\{(\alpha,\ \textbf{answer})\},\ \delta)]\!]$.

(1.4) $s = s_1;s_2$: case analysis for s_1. We give two examples.

(1.4.1) $s_1 = x:=e$. Let $\sigma' = \sigma\{\mathfrak{E}[\![e]\!](\sigma)/x\}$, then

$\Gamma(\mathfrak{D}_{Pr})((\alpha,\ x:=e;s_2)\},\ \delta)) = \lambda\sigma\cdot\{<\sigma',\mathfrak{D}_{Pr}[\![(\{(\alpha,\ s_2)\},\delta)]\!]>\}$

$= \lambda\sigma\cdot\{<\sigma',\mathfrak{D}_S[\![s_2]\!](\alpha)(\delta)(p_0)>\}$

$= \mathfrak{D}_S[\![x:=e]\!](\alpha)(\delta)(\mathfrak{D}_S[\![s_2]\!](\alpha)(\delta)(p_0))$

$= \mathfrak{D}_S[\![x:=e;s_2]\!](\alpha)(\delta)(p_0)$

$= \mathfrak{D}_{Pr}[\![(\{(\alpha,\ x:=e;s_2)\},\delta)]\!]$

(1.4.2) $s_1 = \textbf{release}(\beta,t)$

$\Gamma(\mathfrak{D}_{Pr})((\{(\alpha,\ \textbf{release}(\beta,t);s_2)\},\delta)) =$ [Definition 3.3(R2)]

$\Gamma(\mathfrak{D}_{Pr}(\{(\alpha,s_2),(\beta,t)\},\delta))$

$=$ [induction, Case (2) below]

$\mathfrak{D}_{Pr}[\![(\{(\alpha,s_2),\ (\beta,t)\},\delta)]\!]$

$= \mathfrak{D}_S[\![s_2]\!](\alpha)(\delta)(p_0) \parallel \mathfrak{D}_s[\![t]\!](\beta)(\delta)(p_0)$

$= \mathfrak{D}_S[\![\textbf{release}(\beta,t)]\!](\alpha)(\delta)(\mathfrak{D}_S[\![s_2]\!](\alpha)(\delta)(p_0))$

$= \mathfrak{D}_s[\![\textbf{release}(\beta,t);s_2]\!](\alpha)(\delta)(p_0)$

$= \mathfrak{D}_{Pr}[\![(\{(\alpha,\ \textbf{release}(\beta,t);s_2)\},\delta)]\!]$

The other subcases of Case (1), $s = s_1 + s_2$ and $s = \textbf{release}(\beta,t)$, are easy.

Case (2): $|X|>1$

Suppose we have two disjoint sets X_1 and X_2 in $\mathcal{P}_{fin}(\mathfrak{L}_{LS})$ with $\neg$ *final*(X_1) and $\neg$ *final*(X_2), and assume that

$\Gamma(\mathfrak{D}_{Pr})((X_1,\delta)) = \mathfrak{D}_{Pr}[\![(X_1,\delta)]\!]$

and

$\Gamma(\mathfrak{D}_{Pr})((X_2,\delta)) = \mathfrak{D}_{Pr}[\![(X_2,\delta)]\!]$.

We show that from this induction hypothesis it follows that

$\Gamma(\mathfrak{D}_{Pr})((X_1 \cup X_2,\delta)) = \mathfrak{D}_{Pr}[\![(X_1 \cup X_2,\delta)]\!]$.

From the definition of $\rightarrow$ (Definition 3.3) we have

$\Gamma(\mathfrak{D}_{Pr})((X_1 \cup X_2,\delta)) = \lambda\sigma\cdot(V_1 \cup V_2 \cup W)$.

Here

$V_1 = \{<\sigma',\mathfrak{D}_{Pr}[\![(X'_1 \cup X_2,\delta)]\!]>:\ <X_1,\sigma> \xrightarrow{\tau} <X_1',\sigma'>\}$

$\cup\ \{<\beta,\ m,\ \mathfrak{D}_{Pr}[\![(\{(\alpha,s)\},\delta)]\!],\ \mathfrak{D}_{Pr}[\![(X_1' \cup X_2,\delta)]\!]>:$

$$<X_1,\sigma> \xrightarrow{(\alpha,\beta!m)} <\{(\alpha,s)\} \cup X_1',\ \sigma>\}$$
$$\cup \{<\alpha,g>:\ <X_1,\sigma> \xrightarrow{\alpha?} <\{(\alpha,s)\} \cup X_1',\sigma>\}$$

with

$$g = \lambda m \cdot \lambda p \cdot (\mathfrak{D}_S[\![\delta(m)]\!](\alpha)(\delta)(p \| \mathfrak{D}_{Pr}[\![(\{\alpha,s)\},\delta)]\!]) \| \mathfrak{D}_{Pr}[\![(X_1' \cup X_2,\delta)]\!]).$$

The set V_2 is like V_1 but with the roles of X_1 and X_2 interchanged. Finally,

$$W = \{<\sigma, \mathfrak{D}_{Pr}[\![(\{(\beta,u)\} \cup X'_1 \cup X_2',\delta)]\!]>:$$
$$<X_i,\sigma> \xrightarrow{(\alpha,\beta!m)} <\{(\alpha,s)\} \cup X_i',\sigma> \text{ and}$$
$$<X_j,\sigma> \xrightarrow{\beta?} <\{(\beta,t)\} \cup X_j',\sigma>,$$
$$\text{for } i=1,\ j=2 \text{ or } i=2,\ j=1\},$$

with

$$u = \delta(m);\ \textbf{release}(\alpha,s);\ t.$$

The steps in V_1 correspond to the transition steps that can be made from $X_1 \cup X_2$ stemming from X_1. Similarly for V_2. The set W contains those steps that correspond with a communication transition from $X_1 \cup X_2$ stemming, by an application of rule (R4) of Definition 3.3, from a send step from X_i and an answer step from X_j (for $i=1,\ j=2$ or $i=2,\ j=1$). Now the following holds:

$$V_1 = \Gamma(\mathfrak{D}_{Pr})((X_1,\delta))(\sigma) \between \mathfrak{D}_{Pr}[\![(X_2,\delta)]\!]$$
$$V_2 = \Gamma(\mathfrak{D}_{Pr})((X_2,\delta))(\sigma) \between \mathfrak{D}_{Pr}[\![(X_1,\delta)]\!]$$
$$W = \Gamma(\mathfrak{D}_{Pr})((X_1,\delta))(\sigma) \mid_\sigma \Gamma(\mathfrak{D}_{Pr})((X_2,\delta))(\sigma).$$

We prove only the first equality, the other two cases being similar:

$$V_1 = \{<\sigma', \mathfrak{D}_{Pr}[\![(X_1',\delta)]\!] \| \mathfrak{D}_{Pr}[\![(X_2,\delta)]\!]>:\ <X_1,\sigma> \xrightarrow{\tau} <X_1',\sigma'>\}$$
$$\cup \{<\beta,\ m,\ \mathfrak{D}_{Pr}[\![(\{(\alpha,s)\},\ \delta)]\!], \mathfrak{D}_{Pr}[\![(X_1',\delta)]\!] \| \mathfrak{D}_{Pr}[\![(X_2,\delta)]\!]>:$$
$$<X_1,\sigma> \xrightarrow{(\alpha,\beta!m)} <\{(\alpha,s)\} \cup X_1',\sigma>\}$$
$$\cup \{<\alpha,g'>:\ <X_1,\sigma> \xrightarrow{\alpha?} <\{(\alpha,s)\} \cup X_1',\sigma>\}$$
$$[\text{with } g' = \lambda m \cdot \lambda p \cdot (\mathfrak{D}_S[\![\delta(m)]\!](\alpha)(\delta)(p \| \mathfrak{D}_{Pr}[\![(\{(\alpha,s)\},\delta)]\!])$$
$$\| \mathfrak{D}_{Pr}[\![(X_1',\delta)]\!] \| \mathfrak{D}_{Pr}[\![(X_2,\delta)]\!]]$$
$$= [\text{according to the definition of } \between \text{ (3.10)}]$$
$$(\{<\sigma', \mathfrak{D}_{Pr}[\![(X_1',\delta)]\!]>:<X_1,\sigma> \xrightarrow{\tau} <X_1',\sigma'>\}$$
$$\cup \{<\beta,m,\mathfrak{D}_{Pr}[\![\{(\alpha,s)\},\delta)]\!], \mathfrak{D}_{Pr}[\![(X_1',\delta)]\!]>:$$
$$<X_1,\sigma> \xrightarrow{(\alpha,\beta!m)} <\{(\alpha,s)\} \cup X_1',\sigma>\}$$
$$\cup <\alpha,g''>:\ <X_1,\sigma> \xrightarrow{\alpha?} <\{(\alpha,s)\} \cup X_1',\sigma>\}) \between \mathfrak{D}_{Pr}[\![(X_2,\delta)]\!]$$
$$[\text{ with } g'' = \lambda m \cdot \lambda p \cdot \mathfrak{D}[\![\delta(m)]\!](\alpha)(\delta)(p \| \mathfrak{D}_{Pr}[\![(\{(\alpha,s)\},\delta)]\!]) \| \mathfrak{D}_{Pr}[\![(X_1',\delta)]\!]\ ]$$
$$= \Gamma(\mathfrak{D}_{Pr})((X_1,\delta))(\sigma) \between \mathfrak{D}_{Pr}[\![(X_2,\delta)]\!].$$

From these equalities we deduce:

$$\Gamma(\mathfrak{D}_{Pr})((X_1 \cup X_2,\delta)) = \lambda\sigma \cdot (V_1 \cup V_2 \cup W)$$
$$= \lambda\sigma \cdot (\Gamma(\mathfrak{D}_{Pr})((X_1,\delta))(\sigma) \between \mathfrak{D}_{Pr}[\![(X_2,\delta)]\!] \cup$$

$$\Gamma(\mathfrak{D}_{Pr})((X_2,\delta))(\sigma)\|_{\perp}\ \mathfrak{D}_{Pr}[\![(X_1,\delta)]\!] \cup$$
$$\Gamma(\mathfrak{D}_{Pr})((X_1,\delta))(\sigma)\ |_{\sigma}\ \Gamma(\mathfrak{D}_{Pr})((X_2,\delta))(\sigma))$$
$$= [\text{ induction hypothesis}]$$
$$\lambda\sigma\cdot(\mathfrak{D}_{Pr}((X_1,\delta))(\sigma)\|_{\perp}\ \mathfrak{D}_{Pr}[\![(X_2,\delta)]\!] \cup$$
$$\mathfrak{D}_{Pr}((X_2,\delta))(\sigma)\|_{\perp}\ \mathfrak{D}_{Pr}[\![(X_1,\delta)]\!] \cup$$
$$\mathfrak{D}_{Pr}((X_1,\delta))(\sigma)\ |_{\sigma}\ \mathfrak{D}_{Pr}((X_2,\delta))(\sigma))$$
$$= [\text{definition } \| \text{ (3.10)}]$$
$$\mathfrak{D}_{Pr}(X_1,\delta)]\!]\|\mathfrak{D}_{Pr}[\![(X_2,\delta)]\!]$$
$$= \mathfrak{D}_{Pr}[\![(X_1 \cup X_2,\delta)]\!].$$

This concludes the proof of Theorem 3.18.

Since $\mathcal{I}$ and $\mathfrak{D}_{Pr}$ are both fixed points of the same contraction Γ, they must be equal:

Corollary 3.19: $\mathcal{I}=\mathfrak{D}_{Pr}$

From Corollaries 3.19 and 3.17, stating that $\mathfrak{O}_{Pr} = abstr \circ \mathcal{I}$, the following theorem, which implies the correctness of $\mathfrak{D}_{Pr}$ with respect to $\mathfrak{O}_{Pr}$, is immediate.

Theorem 3.20: $\mathfrak{O}_{Pr} = abstr \circ \mathfrak{D}_{Pr}$

4. The semantics of POOL

Now we come to the semantic description of POOL, which is a syntactically simplified version of POOL2 that retains all the essential semantic features of the latter. We shall now introduce its abstract syntax and next compare POOL and POOL2 in some detail. We need the following sets of syntactic elements:

$(x \in)IVar$ (instance variables)

$(u \in)TVar$ (temporary variables)

$(C \in)CName$ (class names)

$(m \in)MName$ (method names).

We define the set $(\gamma \in)SObj$ of standard objects by

$$SObj = \mathbf{Z} \cup \{tt,ff\} \cup \{nil\}.$$

($\mathbf{Z}$ is the set of all integers.)

We introduce the sets of POOL expressions (L_E), statements (L_S) and programs ($Unit$).

Definition 4.1 (L_E, L_S, $Unit$)
We define the set $(e \in)L_E$ of expressions:

e ::= x
| u
| e ! m (e_1, . . . , e_n)
| m (e_1, . . . , e_n)
| **new** (C)

| $e_1 \equiv e_2$
| $s\ ;\ e$
| **self**
| γ

The set $(s \in) L_S$ of statements:

s ::= $x \leftarrow e$
| $u \leftarrow e$
| **answer** V $\quad (V \subseteq MName,\ V \neq \emptyset)$
| e
| $s_1\ ;\ s_2$
| **if** e **then** s_1 **else** s_2 **fi**
| **while** e **do** s **od**

The set $(U \in) Unit$ of units:

$$U ::= < C_1 \Leftarrow d_1, \ldots, C_n \Leftarrow d_n > \qquad (n \geq 1).$$

The set $(d \in) ClassDef$ of class definitions:

$$d ::= < (m_1 \Leftarrow \mu_1, \ldots, m_n \Leftarrow \mu_n) , s >$$

and finally the set $(\mu \in) MethDef$ of method definitions:

$$\mu ::= < (u_1, \ldots, u_n) , e >, \text{ with } n \geq 0.$$

Let us briefly explain the intended meaning of these syntactic constructs. An expression of the form x or u delivers the value (a reference to an object) that is currently stored in the variable of that name. The send expression $e\ !\ m(e_1, \ldots, e_n)$ is evaluated by first evaluating the expressions $e, e_1, \ldots, e_n$ in that order and then sending a (synchronous) message to the object resulting from expression e, mentioning method m and carrying as parameters the values resulting from $e_1, \ldots, e_n$. When the method is executed by the receiver of the message and it returns a result, this result will be the value of the whole send expression. The call expression $m(e_1, \ldots, e_n)$ is evaluated by executing method m without sending a message. The expression **new**(C) indicates the creation of a new object of class C, the body of which will execute in parallel with the other objects from now on. The value of this expression is a reference to this newly created object. The expression $e_1 \equiv e_2$ checks whether the expressions e_1 and e_2 result in a reference to the same object (cf. the routine id in section 2). Evaluating the expression $s\ ;\ e$ is done by first executing statement s and then evaluating expression e. Finally the expression **self** results in the name of the object that evaluates this expression, and an expression that has the form of a standard object γ represents this standard object itself.

Next we come to the statements. An assignment statement $x \leftarrow e$ or $u \leftarrow e$ is executed by first evaluating expression e and storing the resulting value in the variables x or u, respectively. The answer statement **answer** V indicates that the object is willing to accept one message mentioning a method name contained in the set V. If no such message has yet been sent to this object, it waits until such a message arrives. Then the appropriate method is executed and the result of this method is sent back to the sender. If a statement consists simply of an expression e, then this statement is executed by evaluating expression e and discarding the result (among others, this is useful for sending a message if one is not interested in the result). The meaning of a sequential composition $s_1\ ;\ s_2$, a conditional **if** e **then** s_1 **else** s_2 **fi**, and a loop **while** e **do** s **od** is as usual.

Next we see that a unit U associates several class names C_i with class definitions d_i. A class definition d associates several method names m_i with method definitions μ_i and it gives the body s to be executed by every instance of the class. A method definition μ gives the names of the temporary variables that will contain the parameters for the method and an expression e that will be evaluated when the method is invoked. The result of this expression will be the result of the method. (Note that

quite complicated expressions are possible here, because an expression can have the form s ; e among others.)

The execution of a unit U starts with the implicit creation of one instance of the last class C_n defined in this unit. This so-called *root object* may create others, which can run in parallel with it, and in this way a whole system can be set to work.

It is clear that the abstract syntax given above is a considerably simplified version of the actual syntax of POOL2. In this way the formal semantic treatment is much easier to understand. Nevertheless, all the essential elements of POOL programs are present and therefore there is a quite straightforward translation from POOL2 to this abstract syntax. This translation comprises the following steps:

- All the units of the POOL2 program are merged into one unit. Clashes between class identifiers can be removed by renaming the appropriate classes.
- Generic class definitions are expanded so that there is a separate class definition for each set of class parameters that was used for the generic class in the original program.
- Each routine definition is replaced by a corresponding method definition in every class that calls the routine. Now every call of such a routine can be replaced by a method call. Calls of the standard routines new and id are replaced by expressions of the form **new**(C) and $e_1 \equiv e_2$, respectively.
- If there is only one global name, and this is not used in the rest of the program, this becomes the new root object.
- All the typing information is discarded. This is of no consequence, since we can assume that the original POOL2 program was correct with respect to typing.

For more complicated (and less frequently used) constructs in POOL2, such as asynchronous communication, routines treated as objects, and multiple globals, it is also possible to find an equivalent in our simplified language POOL above. More details on this can be found in America (1988b).

4.1 An operational semantics for POOL

In this section we give the definition of an operational semantics for POOL, which is a modified version of that given in America *et al* (1986a). It is very similar to the operational semantics of the language ℒ of the previous section, and is again based on a transition system. First, we introduce a number of syntactic and semantic concepts.

Definition 4.2 (Objects)
We assume a set $AObj$ of names for active objects together with a function

$$T: AObj \rightarrow CName,$$

which assigns to each object $\alpha \in AObj$ the class to which it belongs. Furthermore, we assume a function

$$\nu: \mathcal{P}_{fin}(AObj) \times CName \rightarrow AObj,$$

such that $\nu(X,C) \notin X$ and $T(\nu(X,C)) = C$, for finite $X \subseteq AObj$ and $C \in CName$. The function ν gives for a finite set X of object names and a class name C a new name of class C, not in X. (A possible example of such a set $AObj$ and functions T and ν could be obtained by setting:

$$AObj = CName \times \mathbb{N},$$

$$T(<C,n>) = C, \text{ and}$$

$$\nu(X,C) = <C, \max\{n: <C,n> \in X\} + 1>.)$$

The set $AObj$ and the set of standard objects $SObj$ together form the set Obj of *object names*, with typical elements α and β:

$$Obj = AObj \cup SObj$$

$$= AObj \cup \mathbf{Z} \cup \{tt, ff\} \cup \{nil\}.$$

Next, it is convenient to extend the sets L_E of expressions and L_S of statements by adding some auxiliary syntactic constructs.

Definition 4.3 ($L_{E'}$, $L_{S'}$)
Let $(e \in) L_{E'}$ and $(s \in) L_{S'}$ be defined by

e ::= x
| u
| e ! m $(e_1, \ldots, e_n)$
| m $(e_1, \ldots, e_n)$
| **new** (C)
| $e_1 \equiv e_2$
| s ; e
| **self**
| γ
| α
| (e, ϕ)

The set $(s \in) L_{S'}$ of statements:

s ::= $x \leftarrow e$
| $u \leftarrow e$
| **answer** V $\quad (V \subseteq MName,\ V \neq \emptyset)$
| e
| s_1 ; s_2
| **if** e **then** s_1 **else** s_2 **fi**
| **while** e **do** s **od**
| **release**(α, s)
| (e, ψ)

with $\alpha \in AObj$, $\gamma \in SObj$, $\phi \in L_{PE}$, and $\psi \in L_{PS}$. Here the sets of *parameterized expressions* $(\phi \in) L_{PE}$ and *parameterized statements* $(\psi \in) L_{PS}$ are given, taking $e \in L_{E'}$ and $s \in L_{S'}$, by

$$\phi ::= \lambda u \cdot e$$

$$\psi ::= \lambda u \cdot s,$$

with the restriction that u does not occur at the left-hand side of an assignment in e or s. For $\alpha \in Obj$, $\phi = \lambda u \cdot e$, and $\psi = \lambda u \cdot s$ we shall use $\phi(\alpha)$ and $\psi(\alpha)$ to denote the expression and the statement obtained by syntactically substituting α for all free occurrences of u in ϕ and ψ, respectively. The restriction mentioned above ensures that the result of this substitution is again a well-formed expression or statement.

Let us explain the new syntactic constructs. In addition to what we already had in L_E, an expression $e \in L_{E'}$ can be an *active* object α or a pair (e, ϕ) of an expression e and a parameterized expression ϕ. The latter will be executed as follows. First, the expression e is evaluated, then the result β is substituted in ϕ and $\phi(\beta)$ is executed. As new statements we have release statements **release**(β, s) and parameterized statements (e, ψ). If the statement **release**(β, s) is executed, the active object β will start executing the statement s (in parallel to the objects that are already executing). The release statement plays a similar role as in section 3; it will be used in the description of the communication between two objects (see Definition 4.9(R11) below). The interpretation of (e, ψ) is similar to that of (e, ϕ).

Definition 4.4 (Empty statement)
The set $L_{S'}$, as given in the definition above, is extended with a special element E, denoting the *empty statement*. This extended set is again called $L_{S'}$. Note that we do *not* have elements like $s;E$ or **while** e **do** E **od** in $L_{S'}$. (There is, however, one exception: We *do* allow E in **if** e **then** s **else** E **fi**, which is needed in Definition 4.9 below.)

Definition 4.5 (States)
The set of states $(\sigma \in)\Sigma$ is defined by

$$\Sigma = (AObj \rightarrow IVar \rightarrow Obj) \times (AObj \rightarrow TVar \rightarrow Obj) \times \mathcal{P}_{fin}(AObj).$$

The three components of σ are denoted by $<\sigma_1, \sigma_2, \sigma_3>$. The first and second component of a state store the values of the instance variables and the temporary variables of each active object. The third component contains the object names currently in use. We need it in order to give unique names to newly created objects.

We shall use the following variant notation. By $\sigma\{\beta/\alpha, x\}$ (with $x \in IVar$) we shall denote the state σ' that is as σ but for the value of $\sigma_1'(\alpha)(x)$, which is β. Similarly, we denote by $\sigma\{\beta/\alpha, u\}$ (with $u \in TVar$) the state σ' that is as σ but for the value of $\sigma_2'(\alpha)(u)$, which is β.

Definition 4.6 (Labelled statements)
The set of *labelled statements* $((\alpha, s) \in) LStat$ is given by

$$LStat = (AObj \times L_{S'}) \cup \{S_t\}.$$

A labelled statement (α, s) should be interpreted as a statement s which is going to be executed by the active object α. The statement S_t will be used to model the operational behavior of standard objects.

Sometimes we also need labelled parameterized statements. Therefore, we extend *LStat*:

$$LStat' = LStat \cup (AObj \times L_{PS}).$$

A pair (α,ψ) indicates that the active object α will execute the statement ψ as soon as it receives a value which it can supply to ψ as an argument.

We call a set of labelled statements *final* (notation *final*(X)) if

$$X \subset (AObj \times \{E\}) \cup S_t.$$

Before we can give the definition of a transition system for POOL, we first have to explain which *configurations* and *transition labels* we are going to use.

Definition 4.7 (Configurations)
The set of configurations $(\rho \in)Conf$ is given by

$$Conf = \mathcal{P}_{fin}(LStat) \times \Sigma.$$

We also introduce:

$$Conf' = \mathcal{P}_{fin}(LStat') \times \Sigma.$$

Typical elements of *Conf* and *Conf'* will also be indicated by $<X,\sigma>$ and $<Y,\sigma>$.

We shall consider only configurations $<X,\sigma>$ that are *consistent* in the following sense. For

$X=\{(\alpha_1, s_1), \ldots, (\alpha_k, s_k)\}$, we call $<X,\sigma>$ consistent if the following conditions are satisfied:

$$\forall i,j \in \{1, \ldots, k\}\ [i \neq j \Rightarrow \alpha_i \neq \alpha_j], \text{ and}$$

$$\{\alpha_1, \ldots, \alpha_k\} \subseteq \sigma_3.$$

Whenever we introduce a configuration $<X, \sigma>$, it will be tacitly assumed that it is consistent.

A configuration $<X, \sigma>$, consisting of a finite set X of labelled statements and a state σ, represents a "snapshot" of the execution of a POOL program. It shows what objects are active and what statements they are executing. Furthermore, it contains a state σ, in which the values of the variables of the active objects as well as the set of object names currently in use are stored.

Definition 4.8 (Transition labels)
The set of *transition labels* $(\lambda \in)\Lambda$ is given by

$$\Lambda = \{\tau\} \cup \{(\alpha, \beta_1!m(\beta_2)):\ \alpha,\beta_1 \in AObj,\ \beta_2 \in Obj\} \cup \{(\beta?V):\ \beta \in AObj\}.$$

These labels will be used in the definition of the transition relation below and are to be interpreted as follows. The label τ indicates a so-called *computation* step. Next, $(\alpha, \beta_1!m(\beta_2))$ indicates that object α sends a message to object β_1 requesting the execution of the method m with parameter β_2. (We shall only consider send expressions with one parameter expression.) Finally, $(\beta?V)$ indicates that object β is willing to answer a message specifying one of the methods in V.

Now we are ready to define a transition system for POOL (cf. Definition 3.3).

Definition 4.9 (Transition relation for POOL)
Let $U \in Unit$ be fixed. We define a *labelled transition system* $<Conf, \Lambda, \rightarrow>$, consisting of a set $Conf$ of configurations, a set Λ of labels, and a *transition relation*

$$\rightarrow \subseteq Conf \times \Lambda \times Conf.$$

Triples $<\rho_1, \lambda, \rho_2> \in \rightarrow$ will be called *transitions* and are denoted by

$$\rho_1 \xrightarrow{\lambda} \rho_2.$$

The relation $\rightarrow$ is defined as the smallest relation satisfying the following properties:

Axioms

(A1) $<\{(\alpha, (x, \psi))\}, \sigma> \xrightarrow{\tau} <\{(\alpha, (\sigma_1(\alpha)(x), \psi))\}, \sigma>$

(A2) $<\{(\alpha, (u, \psi))\}, \sigma> \xrightarrow{\tau} <\{(\alpha, (\sigma_2(\alpha)(u), \psi))\}, \sigma>$

(A3) $<\{(\alpha, (\beta_1!m(\beta_2), \psi)\}, \sigma> \xrightarrow{(\alpha,\ \beta_1!m(\beta_2))} <\{(\alpha, \psi)\}, \sigma>$

(A4) $<\{(\alpha, (\textbf{new}\ (C), \psi))\}, \sigma> \xrightarrow{\tau} <\{(\alpha, (\beta, \psi)), (\beta, s_C)\}, \sigma'>$, where:
$C \Leftarrow <\ldots, s_C> \in U,\ \beta = \nu(\sigma_3, C),\ \sigma' = <\sigma_1, \sigma_2, \sigma_3 \cup \{\beta\}>.$

(A5) $<\{(\alpha, z \leftarrow \beta)\}, \sigma> \xrightarrow{\tau} <\{(\alpha, E)\}, \sigma\{\beta/\alpha, z\}>$, for $z \in IVar \cup TVar$.

(A6) $<\{(\alpha, \textbf{answer}\ V)\}, \sigma> \xrightarrow{(\alpha?V)} <\{(\alpha, E)\}, \sigma>$

(A7) $<\{(\alpha, \textbf{while}\ e\ \textbf{do}\ s\ \textbf{od}\ )\}, \sigma> \xrightarrow{\tau}$
$<\{(\alpha, \textbf{if}\ e\ \textbf{then}\ (s;\ \textbf{while}\ e\ \textbf{do}\ s\ \textbf{od}\ )\ \textbf{else}\ E\ \textbf{fi}\ )\}, \sigma>$

Rules

(R1) If $<\{(\alpha, (e, \lambda u \cdot z \leftarrow u))\}, \sigma> \xrightarrow{\lambda} \rho$,

then $<\{(\alpha,\ z \leftarrow e)\},\sigma> \xrightarrow{\lambda} \rho$, for $z \in IVar \cup TVar$, and $u \neq z$.

(R2) If $<\{(\alpha,\ s)\},\sigma> \xrightarrow{\lambda} <\{(\alpha,\ s')\} \cup X,\sigma'>$,

then $<\{(\alpha,\ s;t)\},\sigma> \xrightarrow{\lambda} <\{(\alpha,\ s';t)\} \cup X,\sigma'>$

(read t instead of $s';t$ if $s'=E$).

If $<\{(\alpha,\ s)\},\sigma> \xrightarrow{\lambda} <\{(\alpha,\ \psi)\} \cup X,\sigma'>$,

then $<\{(\alpha,\ s;t)\},\sigma> \xrightarrow{\lambda} <\{(\alpha,\ \lambda u \cdot (\psi(u);t))\} \cup X,\sigma'>$.

Here u should not occur in ψ and t (we call u fresh).

(R3) If $<\{(\alpha,\ s_i)\},\sigma> \xrightarrow{\lambda} \rho$, then $<\{(\alpha,$ **if** β **then** s_1 **else** s_2 **fi** $)\},\sigma> \xrightarrow{\lambda} \rho$,

where $s_i = \begin{cases} s_1 & \text{if } \beta = tt \\ s_2 & \text{if } \beta = ff. \end{cases}$

(R4) If $<\{(\alpha,\ t),(\beta,s)\},\sigma> \xrightarrow{\lambda} \rho$, then $<\{(\alpha,$ **release** $(\beta,s);t)\},\sigma> \xrightarrow{\lambda} \rho$

(read **release**(β,s) instead of **release**$(\beta,s);t$ if $t=E$).

(R5) If $<\{(\alpha,\ (e,\lambda u \cdot$ **if** u **then** s_1 **else** s_2 **fi** $))\},\sigma> \xrightarrow{\lambda} \rho$,

then $<\{(\alpha,$ **if** e **then** s_1 **else** s_2 **fi** $)\},\sigma> \xrightarrow{\lambda} \rho$ with u fresh.

(Here s_2 is allowed to be E.)

(R6) If $<\{(\alpha,\ ((e_1,\lambda u_1 \cdot (e_2,\lambda u_2 \cdot u_1!m(u_2))),\psi))\},\sigma> \xrightarrow{\lambda} \rho$,

then $<\{(\alpha,\ (e_1!m(e_2),\psi))\},\sigma> \xrightarrow{\lambda} \rho$, with u_1 and u_2 fresh and $u_1 \neq u_2$.

(R7) If $<\{(\alpha,\ s;(e,\psi))\},\sigma> \xrightarrow{\lambda} \rho$, then $<\{(\alpha,\ (s;e,\psi))\},\sigma> \xrightarrow{\lambda} \rho$.

(R8) If $<\{(\alpha,\ (e,\lambda u \cdot (\phi(u),\psi)))\},\sigma> \xrightarrow{\lambda} \rho$, then $<\{(\alpha,\ ((e,\phi),\psi))\},\sigma> \xrightarrow{\lambda} \rho$,

with u fresh.

(R9) If $<\{(\alpha,\ \psi(\beta))\},\sigma> \xrightarrow{\lambda} \rho$, then $<\{(\alpha,\ (\beta,\psi))\},\sigma> \xrightarrow{\lambda} \rho$, for $\beta \in Obj$.

If $<\{(\alpha,\ \psi(\alpha))\},\sigma> \xrightarrow{\lambda} \rho$, then $<\{(\alpha,$ **self**$,\psi))\},\sigma> \xrightarrow{\lambda} \rho$.

(R10) If $<X,\sigma> \xrightarrow{\lambda} <X',\sigma'>$, then $<X \cup Y,\sigma> \xrightarrow{\lambda} <X' \cup Y,\sigma'>$.

(R11) If $<X,\sigma> \xrightarrow{(\alpha,\ \beta_1!m(\beta_2))} <\{(\alpha,\ \psi)\} \cup X',\sigma>$ and

$<Y,\sigma> \xrightarrow{\beta_1?V} <\{(\beta_1,s)\} \cup Y',\sigma>$, and if $m \in V$

then $<X \cup Y,\sigma> \xrightarrow{\tau}$

$<\{(\beta_1,(e_m,\lambda u \cdot (u_m \leftarrow \sigma_2(\beta_1)(u_m);$ **release**$(\alpha,\ \psi(u));s))))\} \cup X' \cup Y',\sigma'>$,

where

$T(\beta_1) = C$

$C \Leftarrow <\ldots,\ m \Leftarrow \mu,\ \ldots> \in U$

$\mu\ =\ <(u_m),\ e_m>$

$\sigma' = \sigma\{\beta_2/\beta_1, u_m\}$.

Transitions for standard objects. The following transitions are possible from S_t:

$<\{S_t\},\ \sigma> - n?\{\text{add, sub}\} \rightarrow <\{S_t\},\ \sigma>$

$$<\{S_t\}, \sigma> - b?\{\text{and, or, not}\} \rightarrow <\{S_t\}, \sigma>$$

for every $n \in \mathbf{Z}$ and $b \in \{tt, ff\}$. (This list can be extended with transitions for other operations.) Communication with a standard object is now modelled by the following transitions:

$$\text{If } <\{(\alpha,s)\}, \sigma> \xrightarrow{\alpha} <\{(\alpha,\psi)\}, \sigma>$$

$$\text{then } <\{(\alpha,s), S_t\}, \sigma> \xrightarrow{\tau} <\{(\alpha, \psi(n+m)), S_t\}, \sigma>.$$

$$\text{If } <\{(\alpha,s)\}, \sigma> \xrightarrow{(\alpha,\ b_1!\text{and}(b_2))} <\{(\alpha,\psi)\}, \sigma>$$

$$\text{then } <\{(\alpha,s), S_t\}, \sigma> \xrightarrow{\tau} <\{(\alpha,\psi(b_1 \wedge b_2)), S_t\}, \sigma>,$$

and by similar transitions for the other operations. (End of definition.)

Note that we have omitted the axioms and rules for some syntactic constructs (e.g., the method call). Moreover, we have made some simplifications; for example, we assume that a send expression contains only one parameter expression. However, we have omitted only what is either straightforward or similar to a clause that we *have* included in the definition above.

The general scheme for the evaluation of an expression closely resembles the approach taken in America and De Bakker (1988). Expressions always occur in the context of a (possibly parameterized) statement (for example, $x \leftarrow e$). A statement containing e as a subexpression is transformed into a pair (e,ψ) of the expression e and a parameterized statement ψ by application of one of the rules. (In our example, $x \leftarrow e$ becomes $(e, \lambda u \cdot x \leftarrow u)$ by an application of (R1).) Then e is evaluated, using the axioms and rules, and results in some value $\beta' \in Obj$. (In our example, $(x, \lambda u \cdot x \leftarrow u)$ will eventually yield $(\beta', \lambda u \cdot x \leftarrow u)$, for some $\beta' \in Obj$.) Next, an application of (R9) will put the resulting object β' back into the original context ψ (yielding $x \leftarrow \beta'$ in our example). Finally, the statement $\psi(\beta')$ is further evaluated by using the axioms and the rules. (The evaluation of $x \leftarrow \beta'$ results, by using (A6), in a transformation of the state.)

Let us briefly explain some of the axioms and rules above. In (A4) a new object is created. Its name β is obtained by applying the function ν to the set σ_3 of the active object names currently in use and the class name C, and is delivered as the result of the evaluation of **new**(C). The body s_C of class C, defined in the unit U, is going to be evaluated by β. Note that the state σ is changed by extending σ_3 with β.

In (R8), the evaluation of an expression pair (e, ϕ), where ϕ is a parameterized expression, in the context of a parameterized statement ψ is reduced to the evaluation of the expression e in the context of the adapted parameterized statement $\lambda u \cdot (\phi(u), \psi)$.

(R11) describes the communication rendezvous of POOL. If object α is sending a message to object β_1, requesting the execution of method m and if object β_1 is willing to answer such a message, then the following happens. Object β_1 starts executing the expression e_m, which corresponds to the definition of method m in U, while its state $\sigma_2(\beta_1)$ is changed by setting u_m, the formal parameter belonging to m, to β_2, the parameter sent by object α to β_1. After the execution of e_m, object β_1 continues by executing $u_m \leftarrow \sigma_2(\beta_1)(u_m)$, which restores the old value of u_m, followed by the statement **release**$(\alpha,\psi(u))$;s. The execution of **release**$(\alpha,\psi(u))$ will reactivate object α, which starts executing $\psi(u)$, the statement obtained by substituting the result u of the execution of e_m into ψ. Note that during the execution of e_m object α is non-active, as can be seen from the fact that α does not occur as the name of any labelled statement in the configuration resulting from this transition. Finally, object β_1 proceeds with the execution of the statement s which is the remainder of its body.

Now we are ready for the definition of the operational semantics of POOL. It will use the following semantic universe.

Definition 4.10 (Semantic universe P)
Let $(w \in)\Sigma_\partial^\infty = \Sigma^* \cup \Sigma^\omega \cup \Sigma^* \cdot \{\partial\}$, the set of *streams*, with Σ as in Definition 4.5. We define

$$(p,q \in)P = \Sigma \rightarrow \mathcal{P}(\Sigma_\partial^\infty),$$

where $\mathcal{P}(\Sigma_\partial^\infty)$ is the set of all subsets of Σ_∂^∞, and the symbol ∂ denotes deadlock. (The universe P is the same as that given in Definition 3.4 but for the use of a different set of states.)

Next, we introduce the operational semantics for POOL (cf. Definition 3.5).

Definition 4.11 (Operational semantics for POOL)
Let, for a $U \in Unit$, the function

$$\mathcal{O}_U : \mathcal{P}_{fin}(LStat) \to P$$

be given as follows. Let $X \in \mathcal{P}_{fin}(LStat)$ and $\sigma \in \Sigma$. We put for a word $w \in \Sigma_\partial^\infty$:

$$w \in \mathcal{O}_U[\![X]\!](\sigma)$$

if and only if one of the following conditions is satisfied:
(1) $w = \sigma_0 \cdots \sigma_n$ $(n \geq 0)$ and there exist $X_0, \ldots, X_n$ such that $<X_0,\sigma_0> = <X,\sigma>$ and

$$<X,\sigma> \xrightarrow{\tau} <X_1,\sigma_1> \xrightarrow{\tau} \cdots \xrightarrow{\tau} <X_n,\sigma_n> \text{ and } final(X_n)$$

(2) $w = \sigma_0\sigma_1 \cdots$ and there exist $X_0, X_1, \ldots$ such that $<X_0,\sigma_0> = <X,\sigma>$ and

$$<X_0,\sigma_0> \xrightarrow{\tau} <X_1,\sigma_1> \xrightarrow{\tau} <X_2,\sigma_2> \xrightarrow{\tau} \cdots$$

(3) $w = \sigma_0 \cdots \sigma_n \cdot \partial$ $(n \geq 0)$ and there exist $X_0, \ldots, X_n$ such that $<X_0,\sigma_0> = <X,\sigma>$ and

$$<X,\sigma> \xrightarrow{\tau} <X_1,\sigma_1> \xrightarrow{\tau} \cdots \xrightarrow{\tau} <X_n,\sigma_n>$$

and $\neg final(X_n)$ and $\neg <X_n,\sigma_n> \xrightarrow{\tau}$

Finally, we can give the operational semantics of a unit.

Definition 4.12 (Operational semantics of a unit)
Let $[\![\cdots]\!]_\mathcal{O} : Unit \to P$ be given, for a unit $U = <\ldots, C_n \Leftarrow <\ldots s_n>>$, by

$$[\![U]\!]_\mathcal{O} = \mathcal{O}_U[\![\{(\nu(\varnothing, C_n), s_n),\ S_t\}]\!].$$

The execution of the unit U consists of the creation of an object of class C_n and the execution of its body in parallel with the statement S_t, which represents the activity of the standard objects. The name of the new object is given by $\nu(\varnothing, C_n)$.

4.2 A denotational semantics for POOL

We start with the definition of the universe for the denotational semantics, which we shall again call $\bar{P}$ and which will be again a solution of a reflexive domain equation.

Definition 4.13 (Semantic process domain $\bar{P}$)
Let $(p,q \in)\bar{P}$ be a complete ultra-metric space satisfying the following reflexive domain equation:

$$\bar{P} \cong \{p_0\} \cup id_{1/2}(\Sigma \to \mathcal{P}_{compact}(Step_{\bar{P}})),$$

where $(\pi,\rho \in)Step_{\bar{P}}$ is

$$Step_{\bar{P}} = Comp_{\bar{P}} \cup Send_{\bar{P}} \cup Answer_{\bar{P}},$$

with

$$Comp_{\bar{P}} = \Sigma \times \bar{P},$$

$$Send_{\bar{P}} = Obj \times MName \times Obj \times (Obj \to \bar{P}) \times \bar{P},$$
$$Answer_{\bar{P}} = Obj \times MName \times (Obj \to (Obj \to \bar{P}) \to {}^{1}\bar{P}).$$

The interpretation of the domain $\bar{P}$ is almost the same as that of the domain $\bar{P}$ defined in the previous section (Definition 3.9). There are some differences, however, which concern the definition of the set of send and answer steps.

To begin with, the set *SLabel* of statement labels has been replaced by the set *Obj* of object names. Since we have value passing in the communication rendezvous of POOL, a send step consists of an additional component (the third in the definition above), which is used for the parameter that is specified in the send message. The dependent resumption of a send step (the fourth component) is now a function from *Obj* to $\bar{P}$, because it depends on the result of the execution of the requested method, which is returned to the sender after this execution is finished.

An answer step $<\alpha, m, g>$ now expresses that object α is willing to answer any message that specifies method m. The type of the resumption g of this answer step is somewhat different from that in Definition 3.9: It takes as arguments first, the parameter that is specified in the message and second, the dependent resumption of the sender (which here is a function).

Because of these differences, the definition of the operator for parallel composition is a slight variant of that given in Definition 3.10.

Definition 4.14 (Parallel composition)
Let $\| : \bar{P} \times \bar{P} \to \bar{P}$ be such that it satisfies the following equation:

$$p \| q = \lambda\sigma \cdot ((p(\sigma) \lfloor\!\lfloor q) \cup (q(\sigma) \lfloor\!\lfloor p) \cup (p(\sigma) \mid_{\sigma} q(\sigma))),$$

for all $p, q \in \bar{P} \setminus \{p_0\}$, and such that $p_0 \| q = q \| p_0 = p_0$. Here, $X \lfloor\!\lfloor q$ and $X \mid_{\sigma} Y$ are defined by:

$$X \lfloor\!\lfloor q = \{\pi \hat{\|} q : \pi \in X\},$$
$$X \mid_{\sigma} Y = \bigcup \{\pi \mid_{\sigma} \rho : \pi \in X, \rho \in Y\},$$

where $\pi \hat{\|} q$ is given by

$$<\sigma', p'> \hat{\|} q = <\sigma', p' \| q>,$$
$$<\alpha, m, \beta, f, p> \hat{\|} q = <\alpha, m, \beta, f, p \| q>, \text{ and}$$
$$<\alpha, m, g> \hat{\|} q = <\alpha, m, \lambda\beta \cdot \lambda h \cdot (g(\beta)(h) \| q)>,$$

and $\pi|_{\sigma}\rho$ by

$$\pi|_{\sigma}\rho = \begin{cases} \{<\sigma, g(\beta)(f) \| p>\} & \text{if } \pi = <\alpha, m, \beta, f, p> \text{ and } \rho = <\alpha, m, g> \\ & \text{or } \rho = <\alpha, m, \beta, f, p> \text{ and } \pi = <\alpha, m, g> \\ \varnothing & \text{otherwise.} \end{cases}$$

Now we come to the definition of the semantics of expressions and statements. We specify a pair of functions $<\mathfrak{D}_E, \mathfrak{D}_S>$ of the following type:

$$\mathfrak{D}_E : L_E \to AObj \to Cont_E \to {}^{1}\bar{P},$$
$$\mathfrak{D}_S : L_S \to AObj \to Cont_S \to {}^{1}\bar{P}$$

where $Cont_E = Obj \to \bar{P}$ and $Cont_S = \bar{P}$. The semantic value of an expression $e \in L_E$, for an object α and an expression continuation $f \in Cont_E$, is given by

$$\mathfrak{D}_E[\![e]\!](\alpha)(f).$$

The evaluation of an expression e always results in a value (an element of Obj) upon which the continuation of such an expression generally depends. The function f, when applied to the result β of e, will yield a process $f(\beta) \in \bar{P}$ that is to be executed after the evaluation of e.

Definition 4.15 (Semantics of expressions and statements)
Let

$$Q_E = L_E \to AObj \to Cont_E \to {}^1\bar{P}$$

$$Q_S = L_S \to AObj \to Cont_S \to {}^1\bar{P}.$$

For every unit $U \in Unit$ we define a pair of functions $\mathfrak{D}_U = <\mathfrak{D}_E, \mathfrak{D}_S>$ by

$$\mathfrak{D}_U = \text{Fixed Point } (\Psi_U),$$

where

$$\Psi_U : (Q_E \times Q_S) \to (Q_E \times Q_S)$$

is defined by induction on the structure of L_E and L_S by the following clauses. For $F = <F_E, F_S>$ we denote $\Psi_U(F)$ by $\hat{F} = <\hat{F}_E, \hat{F}_S>$. Let $p \in Cont_S = \bar{P}$, $f \in Cont_E = Obj \to \bar{P}$ and $\alpha \in AObj$. Then:

Expressions

(E1, instance variable)

$$\hat{F}_E(x)(\alpha)(f) = \lambda\sigma\cdot\{<\sigma, f(\sigma_1(\alpha)(x))>\}.$$

The value of the instance variable x is searched in the first component of the state σ supplied with the name α of the object that is evaluating the expression. The continuation f is then applied to the resulting value.

(E2, temporary variable)

$$\hat{F}_E(u)(\alpha)(f) = \lambda\sigma\cdot\{<\sigma, f(\sigma_2(\alpha)(u))>\}$$

(E3, send expression)

$$\hat{F}_E(e_1 ! m(e_2))(\alpha)(f) = \hat{F}_E(e_1)(\alpha)(\lambda\beta_1 \cdot \hat{F}_E(e_2)(\alpha)(\lambda\beta_2 \cdot \lambda\sigma\cdot\{<\beta_1, m, \beta_2, f, p_0>\})).$$

The expressions e_1 and e_2 are evaluated successively. Their results correspond to the formal parameters β_1 and β_2 of their respective continuations. Finally, a send step is performed. The object name β_1 refers to the object to which the message is sent; β_2 represents the parameter for the execution of method m. Besides these values and the method name m, the final step $<\beta_1, m, \beta_2, f, p_0>$ also contains the expression continuation f of the send expression as the dependent resumption. If the attempt at communication succeeds, this continuation will be supplied with the result of the method execution. The independent resumption of this send step is initialized to p_0.

(E4, new-expression)

$$\hat{F}_E(\textbf{ new } (C))(\alpha)(f) = \lambda\sigma\cdot\{<\sigma', f(\beta) \| F_S(s_C)(\beta)(p_0)>\},$$

where

$$\beta = \nu(\sigma_3, C),$$

$$\sigma' = <\sigma_1, \sigma_2, \sigma_3 \cup \{\beta\}>, \quad C \Leftarrow <\ldots, s_C> \in U.$$

A new object of class C is created. It is called $\nu(\sigma_3, C)$: the function ν applied to the set of all object names currently in use and the class name C yields a name that is not yet being used. The state σ is changed by expanding the set σ_3 with the new name β. The process $F_S(s_C)(\beta)(p_0)$ is the meaning of the body of the new object β with p_0 as a nil continuation. It is composed in parallel with $f(\beta)$, the

process resulting from the application of the continuation f to β, the result of the evaluation of this new-expression. We are able to perform this parallel composition because we know from f what should happen after the evaluation of this new-expression, so here the use of continuations is essential.

(E5, sequential composition)

$$\hat{F}_E(s;e)(\alpha)(f)=\hat{F}_S(s)(\alpha)(\hat{F}_E(e)(\alpha)(f)).$$

The continuation of s is the execution of e followed by f. Note that a semantic operator for sequential composition is absent: the use of continuations has made it superfluous.

(E6, **self**)

$$\hat{F}_E(\ \textbf{self}\ )(\alpha)(f)=f(\alpha).$$

The continuation of f is supplied with the value of the expression **self**, that is, the name of the object executing this expression. We use $f(\alpha)$ instead of $\lambda\beta\cdot\{<\sigma,f(\alpha)>\}$ in this definition because we wish to express that the value of self is immediately present: it does not take a step to evaluate it.

(E7, objects)

$$\hat{F}_E(\beta)(\alpha)(f)=f(\beta),\quad \text{for}\ \ \beta\in Obj.$$

(E8, parameterized expression)

$$\hat{F}_E((e,\phi))(\alpha)(f)=\hat{F}_E(e)(\alpha)(\lambda\beta\cdot\hat{F}_E(\phi(\beta))(\alpha)(f))$$

The expression e is evaluated and the result will be passed through to the continuation, which consists of the meaning of the parameterized expression ϕ.

Statements

(S1, assignment to an instance variable)

$$\hat{F}_S(x\leftarrow e)(\alpha)(p)=\hat{F}_E(e)(\alpha)(\lambda\beta\cdot\lambda\sigma\cdot\{<\sigma',p>\}),$$

where $\sigma'=\sigma\{\beta/\alpha,x\}$. The expression e is evaluated and the result β is assigned to x.

(S2, assignment to a temporary variable)

$$\hat{F}_S(u\leftarrow e)(\alpha)(p)=\hat{F}_E(e)(\alpha)(\lambda\beta\cdot\lambda\sigma\cdot\{<\sigma',p>\}),$$

where $\sigma'=\sigma\{\beta/\alpha,u\}$.

(S3, answer statement)

$$\hat{F}_S(\ \textbf{answer}\ V)(\alpha)(p)=\lambda\sigma\cdot\{<\alpha,m,g_m>:\ m\in V\},$$

where

$$g_m=\lambda\beta\cdot\lambda f\cdot\lambda\sigma\cdot\{<\sigma',F_E(e_m)(\alpha)(\lambda\beta'\cdot\lambda\bar{\sigma}\cdot\{<\bar{\sigma}',f(\beta')\|p>\})>\},$$

with

$$\sigma'=\sigma\{\beta/\alpha,u_m\},$$

$$\bar{\sigma}'=\bar{\sigma}\{\sigma_2(\alpha)(u_m)/\alpha,u_m\},$$

$$m\Leftarrow<(u_m),e_m>\in U.$$

The function g_m represents the execution of method m followed by its continuation. This function g_m expects a parameter β and an expression continuation f, both to be received from an object sending a message specifying method m. The execution of method m consists of the evaluation of the expression e_m, which is used in the definition of m, preceded by a state transformation in which the temporary variable u_m is initialized at the value β. After the execution of e, this temporary variable is set back to

its old value. Next, both the continuation of the sending object, supplied with the result β' of the execution of the method m, and the given continuation p are to be executed in parallel. This explains the last resumption: $f(\beta')\|p$.

(S4, sequential composition)

$$\hat{F}_S(s_1;s_2)(\alpha)(p) = \hat{F}_S(s_1)(\alpha)(\hat{F}_S(s_2)(\alpha)(p)).$$

(S5, conditional)

$$\begin{aligned}\hat{F}_S(\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}\ )(\alpha)(p) = \\ \hat{F}_E(e)(\alpha)(\lambda\beta\cdot\ &\text{if } \beta = tt \\ &\text{then } \hat{F}_S(s_1)(\alpha)(p) \\ &\text{else } \hat{F}_S(s_2)(\alpha)(p) \\ &\text{fi }).\end{aligned}$$

(S6, loop statement)

$$\begin{aligned}\hat{F}_S(\ \textbf{while}\ e\ \textbf{do}\ s\ \textbf{od}\ )(\alpha)(p) = \\ \lambda\sigma\cdot\{<\sigma,\ \hat{F}_E(e)(\alpha)(\lambda\beta\cdot\ &\text{if } \beta = tt \\ &\text{then } \hat{F}_S(s)(\alpha)(F_S(\ \textbf{while}\ e\ \textbf{do}\ s\ \textbf{od}\ )(\alpha)(p)) \\ &\text{else } p \\ &\text{fi }) >\}.\end{aligned}$$

(S7, parameterized statement)

$$\hat{F}_S((e,\psi))(\alpha)(p) = \hat{F}_E(e)(\alpha)(\lambda\beta\cdot\hat{F}_S(\psi(\beta))(\alpha)(p))$$

(End of definition 4.15.)

(In the above definition, we have applied simplifications similar to those in the definition of the operational semantics for POOL; see the comment following Definition 4.8.)

It is not difficult to prove that Ψ_U is a contraction and hence has a unique fixed point $\mathcal{D}_U$. In fact, we have defined Ψ_U such that it satisfies this property. Note that the original functions F_E and F_S have been used in only three places: in the definition of the semantics of a new-expression, of an answer statement, and of a loop statement. Here the syntactic complexity of the defining part is not necessarily less than that of what is being defined. At those places, we have ensured that the definition is "guarded" by some step $\lambda\sigma\cdot\{<\sigma', \ldots >\}$. It is easily verified that in this way the contractiveness of Ψ_U is indeed implied.

Before we can define the denotational semantics of a unit we first have to give a denotational interpretation of standard objects. We introduce a *standard* process p_{St}, which represents the activity of the standard objects. In order to let this standard process p_{St} fit into our semantic domain, we are forced to use closed subsets of steps rather than compact ones. Let us indicate the process domain given in Definition 4.13 by $\overline{P}_{co}$. We introduce here $\overline{P}_{cl}$, which satisfies:

$$\overline{P}_{cl} \cong \{p_0\} \cup id_{1/2}(\Sigma \to \mathcal{P}_{closed}(Step_{\overline{P}_{cl}})).$$

We have, via an obvious embedding, that $\overline{P}_{co} \subseteq \overline{P}_{cl}$.

Next we introduce $p_{St} \in \overline{P}_{cl}$, which represents the meaning of all standard objects. It satisfies the following equation:

$$\begin{aligned}p_{St} = \lambda\sigma\cdot\ &(\{<n, \text{add}, g_n^+>: n \in \mathbf{Z}\} \cup \\ &\{<n, \text{sub}, g_n^->: n \in \mathbf{Z}\} \cup \\ &\{<b, \text{and}, g_b^\wedge>: b \in \{tt, ff\}\} \cup \\ &\{<b, \text{or}, g_b^\vee>: b \in \{tt, ff\}\} \cup \\ &\{<b, \text{not}, g_b^\neg>: b \in \{tt, ff\}\}),\end{aligned}$$

where

$$g_n^+ = \lambda\beta{\in}Obj{\cdot}\lambda f{\in}Obj{\to}\overline{P}{\cdot}\ (\text{if } \beta{\in}\mathbf{Z} \text{ then } f(n+\beta)\|p_{St} \text{ else } p_{St} \text{ fi}),$$

$$g_n^- = \lambda\beta{\in}Obj{\cdot}\lambda f{\in}Obj{\to}\overline{P}{\cdot}\ (\text{if } \beta{\in}\mathbf{Z} \text{ then } f(n-\beta)\|p_{St} \text{ else } p_{St} \text{ fi}),$$

$$g_b^\wedge = \lambda\beta{\in}Obj{\cdot}\lambda f{\in}Obj{\to}\overline{P}{\cdot}\ (\text{if } \beta{\in}\{tt,ff\} \text{ then } f(b\wedge\beta)\|p_{St} \text{ else } p_{St} \text{ fi})$$

$$g_b^\vee = \lambda\beta{\in}Obj{\cdot}\lambda f{\in}Obj{\to}\overline{P}{\cdot}\ (\text{if } \beta{\in}\{tt,ff\} \text{ then } f(b\vee\beta)\|p_{St} \text{ else } p_{St} \text{ fi})$$

$$g_b^\neg = \lambda\beta{\in}Obj{\cdot}\lambda f{\in}Obj{\to}\overline{P}{\cdot}\ f(\neg b)\|p_{St}.$$

This definition is self-referential since p_{St} occurs at the right-hand side of the definition. Formally, p_{St} can be given as the fixed point of a suitably defined contraction on $\overline{P}_{cl}$.

We observe that p_{St} is an infinitely branching process, which is an element of $\overline{P}_{cl}$ but not of $\overline{P}_{co}$. This explains the introduction of $\overline{P}_{cl}$.

The operational intuition behind the definition of p_{St} is the following. For every $n{\in}\mathbf{Z}$ the set $p_{St}(\sigma)$ contains, among others, two elements, namely $<n, \text{add}, g_n^+>$ and $<n, \text{sub}, g_n^->$. These steps indicate that the integer object n is willing to execute its methods add and sub. If, for example by evaluating n !add(n'), a certain active object sends a request to integer object n to execute method add with parameter n', then g_n^+, supplied with n' and the continuation f of the active object, is executed. We have that $g_n^+(n')(f)$ is, by definition, the parallel composition of f supplied with the immediate result of the execution of method add, namely $n+n'$, and the process p_{St}, which remains unaltered: $g_n^+(n')(f) = f(n+n')\|p_{St}$. (A similar explanation applies to the presence in $p_{St}(\sigma)$ of the triples representing the booleans.)

The standard objects are assumed to be present at the execution of every unit U. Therefore, the definition of the denotational semantics is given as follows.

Definition 4.16 (Denotational semantics of a unit)
We define $[\![\ \cdots\ ]\!]_{\mathfrak{D}}$: $Unit{\to}\overline{P}$. For a unit $U{\in}Unit$, with $U = <\ldots,\ C_n \Leftarrow <\ldots, s_n>>$, we set

$$[\![U]\!]_{\mathfrak{D}} = \mathfrak{D}_S[\![s_n]\!](\nu(\varnothing, C_n))(p_0)\ \|p_{St}.$$

The execution of a unit always starts with the creation of an object of class C_n and the execution of its body. Therefore, the meaning of a unit U is given by the parallel composition of the denotational meaning of the body of this first object together with the standard process p_{St}.

4.3 Semantic correctness of $[\![\ \cdots\ ]\!]_{\mathfrak{D}}$

Analogously to section 3.3, we can establish a similar correctness result for the denotational semantics of POOL with respect to the operational model. In other words, we can again define a suitable abstraction operator *abstr* and prove that

$$[\![U]\!]_{\mathfrak{O}} = abstr([\![U]\!]_{\mathfrak{D}}),$$

for every $U{\in}Unit$. We refer the reader to Rutten (1988), where this is proved in full.

5. References

G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems,* MIT Press, 1986.

P. America, *POOL-T: a parallel object-oriented language,* in "Object-Oriented Concurrent Programming" (A. Yonezawa and M. Tokoro, eds), MIT Press, 1987a, pp. 199-220.

P. America, *Inheritance and subtyping in a parallel object-oriented language,* in "ECOOP '87: European

Conference on Object-Oriented Programming", Paris, June 15-17, 1987b, Springer Lecture Notes in Computer Science 276, pp. 234-242.

P. America, *Definition of POOL2, a parallel object-oriented language,* ESPRIT Project 415 Document 364, Philips Research Laboratories, Eindhoven, 1988a.

P. America, *Rationale for the design of POOL2,* ESPRIT Project 415 Document 393, Philips Research Laboratories, Eindhoven, 1988b.

P. America, *Issues in the design of a parallel object-oriented language,* ESPRIT Project 415 Document 452, Philips Research Laboratories, Eindhoven, 1988c.

P. America and J.W. de Bakker, *Designing equivalent semantic models for process creation,* Theoretical Computer Science 60, 1988, pp. 109-176.

P. America, J.W. de Bakker, J.N. Kok and J.J.M.M. Rutten, *Operational semantics of a parallel object-oriented language,* in: "Conference Record of the 13th Symposium on Principles of Programming Languages, St Petersburg, Florida," 1986a, pp. 194-208.

P. America, J.W. de Bakker, J.N. Kok and J.J.M.M. Rutten, *A denotational semantics of a parallel object-oriented language,* Technical Report (CS-R8626), Centre for Mathematics and Computer Science, Amsterdam, 1986b. (To appear in: Information and Computation.)

P. America and J.J.M.M. Rutten, *Solving reflexive domain equations in a category of complete metric spaces,* in: Proceedings of the Third Workshop on Mathematical Foundations of Programming Language Semantics (M. Main, A. Melton, M. Mislove, D. Schmidt, eds), Lecture Notes in Computer Science 298, Springer-Verlag, 1988, pp. 254-288. (To appear in the Journal of Computer and System Sciences.)

ANSI, *Reference manual for the Ada programming language,* ANSI/MIL-STD 1815 A, United States Department of Defense, Washington D. C., 1983.

J.W. de Bakker, J.A. Bergstra, J.W. Klop and J.-J.Ch. Meyer, *Linear time and branching time semantics for recursion with merge,* Theoretical Computer Science 34, 1984, pp. 135-156.

J.W. de Bakker, J.N. Kok, J.-J.Ch. Meyer, E.-R. Olderog and J.I. Zucker, *Contrasting themes in the semantics of imperative concurrency,* in: Current Trends in Concurrency (J.W. de Bakker, W.P. de Roever and G. Rozenberg, eds), Lecture Notes in Computer Science 224, Springer-Verlag, 1986, pp. 51-121.

J.W. de Bakker and J.I. Zucker, *Processes and the denotational semantics of concurrency,* Information and Control 54, 1982, pp. 70-120.

A. de Bruin, *Experiments with continuation semantics: Jumps, backtracking, dynamic networks,* Ph. D. thesis, Free University of Amsterdam, 1986.

J. Dugundji, *Topology,* Allyn and Bacon, Boston, 1966.

E. Engelking, *General topology,* Polish Scientific Publishers, 1977.

M.J.C. Gordon, *The denotational description of programming languages,* Springer-Verlag, 1979.

A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, 1983.

M. Hennessy and G.D. Plotkin, *Full abstraction for a simple parallel programming language,* in: Proceedings 8th MFCS (J. Bečvář ed.), Lecture Notes in Computer Science 74, Springer-Verlag, 1979, pp. 108-120.

J.N. Kok and J.J.M.M. Rutten, *Contractions in comparing concurrency semantics,* in: Proceedings 15th

ICALP, Tampere, Lecture Notes in Computer Science 317, Springer-Verlag, 1988, pp. 317-332. (To appear in Theoretical Computer Science.)

B. Meyer, *Object-Oriented Software Construction,* Prentice-Hall, 1988.

E. Michael, *Topologies on spaces of subsets,* Trans. AMS 71, 1951, pp.152-182.

E.A.M. Odijk, *The* DOOM *system and its applications: a survey of ESPRIT 415 subproject A,* in: "Parallel Architectures and Languages Europe, Volume I" (J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, eds), Lecture Notes in Computer Science 258, Springer-Verlag, 1987, pp. 461-479.

G.D. Plotkin, *A structural approach to operational semantics,* Report DAIMI FN-19, Comp. Sci. Dept., Aarhus Univ. 1981.

G.D. Plotkin, *An operational semantics for CSP,* in: Formal Description of Programming Concepts II (D. Bjørner ed.), North-Holland, Amsterdam, 1983, pp. 199-223.

J.J.M.M. Rutten, *Semantic correctness for a parallel object-oriented language,* Technical Report CS-R8843, Centre for Mathematics and Computer Science, Amsterdam, 1988. (To appear in SIAM Journal of Computation.)

B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, 1986.

Appendix: Mathematical definitions

Definition A.1 (Metric space)
A *metric space* is a pair (M,d) with a non-empty set M and a mapping $d:M\times M\to[0,1]$ (a *metric* or *distance*) that satisfies the following properties:
(a) $\forall x,y\in M\,[d(x,y)=0 \Leftrightarrow x=y]$
(b) $\forall x,y\in M\,[d(x,y)=d(y,x)]$
(c) $\forall x,y,z\in M\,[d(x,y)\leqslant d(x,z)+d(z,y)]$.
We call (M,d) an *ultra-metric space* if the following stronger version of property (c) is satisfied:
(c') $\forall x,y,z\in M\,[d(x,y)\leqslant \max\{d(x,z),d(z,y)\}]$.
Note that we consider only metric spaces with bounded diameters: the distance between two points never exceeds 1.

EXAMPLES A.1.1
(a) Let A be an arbitrary set. The *discrete* metric d_A on A is defined as follows. Let $x,y\in A$, then

$$d_A(x,y)=\begin{cases}0 & \text{if } x=y\\ 1 & \text{if } x\neq y.\end{cases}$$

(b) Let A be an alphabet, and let $A^\infty=A^*\cup A^\omega$ denote the set of all finite and infinite words over A. Let, for $x\in A^\infty$, $x[n]$ denote the prefix of x of length n, in case $length(x)\geqslant n$, and x otherwise. We put

$$d(x,y)=2^{-sup\{n\,:\,x[n]=y[n]\}},$$

with the convention that $2^{-\infty}=0$. Then (A^∞,d) is a metric space.

Definition A.2
Let (M,d) be a metric space, let $(x_i)_i$ be a sequence in M.
(a) We say that $(x_i)_i$ is a *Cauchy sequence* whenever we have
$\forall\epsilon>0\ \exists N\in\mathbb{N}\ \forall n,m>N\ [d(x_n,x_m)<\epsilon]$.
(b) Let $x\in M$. We say that $(x_i)_i$ *converges to* x and call x the *limit* of $(x_i)_i$ whenever we have

$\forall \epsilon > 0\ \exists N \in \mathbb{N}\ \forall n > N\ [d(x,x_n) < \epsilon]$.
Such a sequence we call *convergent*. Notation: $\lim_{i\to\infty} x_i = x$.

(c) The metric space (M,d) is called *complete* whenever each Cauchy sequence converges to an element of M.

Definition A.3
Let $(M_1,d_1),(M_2,d_2)$ be metric spaces.

(a) We say that (M_1,d_1) and (M_2,d_2) are *isometric* if there exists a bijection $f:M_1\to M_2$ such that $\forall x,y \in M_1\ [d_2(f(x),f(y)) = d_1(x,y)]$. We then write $M_1 \cong M_2$. When f is not a bijection (but only an injection), we call it an *isometric embedding*.

(b) Let $f:M_1\to M_2$ be a function. We call f *continuous* whenever for each sequence $(x_i)_i$ with limit x in M_1 we have that $\lim_{i\to\infty} f(x_i) = f(x)$.

(c) Let $A \geqslant 0$. With $M_1 \to^A M_2$ we denote the set of functions f from M_1 to M_2 that satisfy the following property:
$\forall x,y \in M_1\ [d_2(f(x),f(y)) \leqslant A \cdot d_1(x,y)]$.
Functions f in $M_1 \to^1 M_2$ we call *non-expansive*, functions f in $M_1 \to^\epsilon M_2$ with $0 \leqslant \epsilon < 1$ we call *contracting*. (For every $A \geqslant 0$ and $f \in M_1 \to^A M_2$ we have: f is continuous.)

Proposition A.4 (Banach's fixed-point theorem)
Let (M,d) be a complete metric space and $f:M\to M$ a contracting function. Then there exists an $x \in M$ such that the following holds:
(1) $f(x) = x$ *(x is a fixed point of f),*
(2) $\forall y \in M\ [f(y) = y \Rightarrow y = x]$ *(x is unique),*
(3) $\forall x_0 \in M\ [\lim_{n\to\infty} f^{(n)}(x_0) = x]$, *where* $f^{(0)}(x_0) = x_0$ *and* $f^{(n+1)}(x_0) = f(f^{(n)}(x_0))$.

Definition A.5 (Closed and compact subsets)
A subset X of a complete metric space (M,d) is called *closed* whenever each Cauchy sequence in X has a limit in X and is called *compact* whenever each sequence in X has a subsequence that converges to an element of X.

Definition A.6
Let $(M,d),(M_1,d_1),\ldots,(M_n,d_n)$ be metric spaces.

(a) With $M_1 \to M_2$ we denote the set of all continuous functions from M_1 to M_2. We define a metric d_F on $M_1 \to M_2$ as follows. For every $f_1,f_2 \in M_1 \to M_2$

$$d_F(f_1,f_2) = \sup_{x\in M_1}\{d_2(f_1(x),f_2(x))\}.$$

For $A \geqslant 0$ the set $M_1 \to^A M_2$ is a subset of $M_1 \to M_2$, and a metric on $M_1 \to^A M_2$ can be obtained by taking the restriction of the corresponding d_F.

(b) With $M_1 \cup \cdots \cup M_n$ we denote the *disjoint union* of $M_1,\ldots,M_n$, which can be defined as $\{1\}\times M_1 \cup \cdots \cup \{n\}\times M_n$. We define a metric d_U on $M_1 \cup \cdots \cup M_n$ as follows. For every $x,y \in M_1 \cup \cdots \cup M_n$

$$d_U(x,y) = \begin{cases} d_j(x,y) & \text{if } x,y \in \{j\}\times M_j,\ 1 \leqslant j \leqslant n \\ 1 & \text{otherwise.} \end{cases}$$

(c) We define a metric d_P on $M_1 \times \cdots \times M_n$ by putting for every $(x_1,\ldots,x_n)$, $(y_1,\ldots,y_n) \in M_1 \times \cdots \times M_n$

$$d_P((x_1,\ldots,x_n),(y_1,\ldots,y_n)) = \max_i\{d_i(x_i,y_i)\}.$$

(d) Let $\mathfrak{P}_{closed}(M) = \{X: X \subseteq M \wedge X \text{ is closed}\}$. We define a metric d_H on $\mathfrak{P}_{closed}(M)$, called the *Hausdorff distance*, as follows. For every $X,Y \in \mathfrak{P}_{closed}(M)$ with $X,Y \neq \emptyset$

$$d_H(X,Y)=\max\{\sup_{x\in X}\{d(x,Y)\},\sup_{y\in Y}\{d(y,X)\}\},$$

where $d(x,Z)=^{def}\inf_{z\in Z}\{d(x,z)\}$ for every $Z\subseteq M$, $x\in M$. For $X\neq\varnothing$ we put

$$d_H(\varnothing,X)=d_H(X,\varnothing)=1.$$

The following spaces

$$\mathfrak{P}_{compact}(M) = \{X\colon X\subseteq M\wedge X \text{ is compact}\}$$

$$\mathfrak{P}_{ncompact}(M) = \{X\colon X\subseteq M\wedge X \text{ is non-empty and compact}\}$$

are supplied with a metric by taking the respective restrictions of d_H.

(e) Let $c\in(0,1]$. We define: $id_c(M,d)=(M,c\cdot d)$.

Proposition A.7

Let (M,d), $(M_1,d_1),\ldots,(M_n,d_n)$, d_F, d_U, d_P and d_H be as in definition A.6 and suppose that (M,d), $(M_1,d_1),\ldots,(M_n,d_n)$ are complete. We have that

(a) $(M_1\underline{\rightarrow}M_2,d_{\underline{F}})$, $(M_1\rightarrow^A M_2,d_F)$,

(b) $(M_1\cup\cdots\cup M_n,d_U)$,

(c) $(M_1\times\cdots\times M_n,d_P)$,

(d) $(\mathfrak{P}_{closed}(M),d_H)$, $(\mathfrak{P}_{compact}(M),d_H)$ and $(\mathfrak{P}_{ncompact}(M),d_H)$

are complete metric spaces. If (M,d) and (M_i,d_i) are all ultra-metric spaces these composed spaces are again ultra-metric. (Strictly speaking, for the completeness of $M_1\rightarrow M_2$ and $M_1\rightarrow^A M_2$ we do not need the completeness of M_1. The same holds for the ultra-metric property.)

The proofs of proposition A.7 (a), (b) and (c) are straightforward. Part (d) is more involved. It can be proved with the help of the following characterization of the completeness of the Hausdorff metric.

Proposition A.8

Let $(\mathfrak{P}_{closed}(M),d_H)$ be as in definition A.6. Let $(X_i)_i$ be a Cauchy sequence in $\mathfrak{P}_{closed}(M)$. We have:

$$\lim_{i\rightarrow\infty}X_i = \{\lim_{i\rightarrow\infty}x_i \mid x_i\in X_i,\ (x_i)_i \text{ a Cauchy sequence in } M\}.$$

The proof of proposition A.8 can be found in Dugundji (1966) and Engelking (1977). The completeness of the Hausdorff space containing compact sets is proved in Michael (1951).