Formal Aspects of Computing (1992) 4: 376–408
© 1992 BCS

Formal Aspects of Computing

A Layered Semantics for a Parallel Object-Oriented Language*

Pierre America[1] and Jan Rutten[2]

[1]Philips Research Laboratories, Eindhoven, The Netherlands
[2]Centre for Mathematics and Computer Science, Amsterdam, The Netherlands

Keywords: Object-oriented programming; Denotational semantics; Parallelism; Complete metric spaces

Abstract. We develop a denotational semantics for POOL, a parallel object-oriented programming language. The main contribution of this semantics is an accurate mathematical model of the most important concept in object-oriented programming: the object. This is achieved by structuring the semantics in layers working at three different levels: for statements, objects and programs. For each of these levels we define a specialized mathematical domain of processes, which we use to assign a meaning to each language construct. This is done in the mathematical framework of complete metric spaces. We also define operators that translate between these domains. At the program level we give a precise definition of the observable input/output behaviour of a particular program, which could be used at a later stage to decide the issue of full abstractness. We illustrate our semantic techniques by first applying them to a toy language similar to CSP.

1. Introduction

In the design of a programming language, a formal study of its semantics can be of considerable advantage [Ame89c]. First of all, the conciseness and mathematical elegance of the formal semantic definition of a language is a very good measure of its conceptual integrity. If the basic concepts of a language or the way in which they are combined are not well chosen, then an attempt to describe the meaning

*This paper describes work done in ESPRIT Basic Research Action 3020, *Integration.*
Correspondence and offprint requests to: Jan Rutten, Centre for Mathematics and Computer Science, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands.

of programs written in that language by formal, i.e., mathematical, means will certainly run into problems. Second, a formal description of the semantics of a language may form a basis for proving the correctness of a certain implementation. Sometimes this may apply to a complete implementation, but more often it will only apply to specific techniques used in such an implementation. Last but not least, formal semantics for a language can function as a gauge for an equally formal theory of reasoning about the correctness of programs written in the language. Since reasoning about a program can be done at several levels of abstraction, it is important that for the formal description of the semantics the right abstraction level is chosen.

In this paper we shall study the semantics of POOL, a parallel object-oriented language [Ame89b]. This language has been designed to support the development of symbolic (i.e., not only numerical) programs that can be run efficiently on a parallel computer without shared memory. Up to now, the formal semantics of POOL has been described in several different ways. First an operational semantics was defined [ABK86], using the technique of transition systems and Structural Operational Semantics [Plo81]. After that we developed a denotational semantic description of POOL [ABK89]. This took place in the mathematical framework of complete metric spaces and used mathematical structures called processes [BaZ82] to represent the behaviour of a program and its parts. In [Rut90] it was proved that these operational and denotational semantics, which were developed more or less independently, are in a certain sense equivalent. The semantics of POOL has also been described using other formalisms, for example process algebra [Vaa86].

Here we want to concentrate on denotational semantics. The main characteristic of denotational semantics is that it assigns a meaning (a value out of some mathematical domain) to each language construct in a *compositional* way. This means that the meaning of a composite construct only depends on the meanings of its constituents, not on their actual syntactic form. In general, this is the best way of describing each concept in the language accurately and individually. The denotational semantics developed so far for POOL [ABK89] had two flaws. Firstly, it did not give a description of the semantics of a single object, clearly a very important concept in the language. Secondly, the denotational semantics was not sufficiently abstract, and certainly not *fully abstract*. This principle of full abstractness can be defined as follows: In denotational semantics, the meaning of a program fragment must contain sufficient information to be able to determine the meaning of any larger fragment that contains the first one as a constituent. However, if we look at a complete program, it is in general clear which aspects of its behaviour can be actually observed, for example, its output as a function of its input. A semantic description is called fully abstract if the meaning of any program fragment contains only that information that is necessary to fix the observable behaviour of any complete program that contains it. More precisely, whenever two program fragments have different meanings then there should be a context (a program with a 'hole') that gives different observable behaviours when it is filled with these fragments.

This paper develops a semantics for POOL that works at three different levels: the statement level, the object level, and the program level. For each level there is a specialized domain where the values reside that represent the meaning of the individual language constructs. The relationship between the levels is given by translation operators that map meanings at one level to meanings at the next higher level, forgetting whenever possible about details that are no longer relevant at the higher level. The semantics at the level of programs will define the behaviour that we can ultimately observe, and the statement level is of course necessary to get off the ground. The object level is most interesting, because it centres by definition around the most important concept of object-oriented programming. Getting a clear, formal idea of what constitutes the meaning of an object is not just an intellectual challenge. An object is the basic unit of encapsulation and reuse in object-oriented programming. As was argued in [Ame89a], it is important to abstract away from the internal details of an object, since these cannot be observed anyway. Therefore reasoning about the correctness of programs is best done at the level of the observable behaviour of the objects. This can also shed some light on the nature of inheritance and subtyping, two of the most interesting issues in object-oriented programming (see also [Ame91]).

A particular aspect that the reader might be less familiar with is the use of complete metric spaces instead of the more common complete partial orders. We use them mainly because of two advantages: Firstly, (guarded) recursive definitions have *unique* solutions (by Banach's theorem, see the appendix). Secondly, the metric power set construction is simpler than its order-theoretic counterparts. Even then, the techniques that we use in this paper are relatively complex. The most important reason for this is that the language under consideration is (an abstract version of) a real programming language which has many different features. In order to introduce the reader to all this, in Section 2 a language called Toy is treated, which is semantically much simpler than POOL. Section 3 then applies these techniques to POOL. Both Section 2 and Section 3 first introduce the language and its syntax and then describe the semantics at the level of statements, objects, and programs. In Section 4 we draw some conclusions from our work and sketch some possibilities for further work. Appendix A sketches the mathematical preliminaries necessary to understand the technicalities in the rest of the paper.

2. A Toy Language

In this section a simple language, called Toy, is introduced and supplied with a denotational semantics. Toy is very similar to CSP [Hoa78], but a little simpler. A program consists of a fixed, finite number of *objects* (the CSP terminology would be 'processes'), which can only communicate with each other by exchanging messages. In order to communicate, the sender and receiver of a message synchronize (the first one that is ready to communicate waits for the other) and then they exchange a single value.

A denotational semantics is given to this language in three stages: first for statements, then for objects, and finally for programs. At each stage a different *domain* (a kind of mathematical structure) will be used to describe the meaning of the language constructs and operations to translate these structures into each other will be defined.

2.1. Syntax of Toy

The basic building blocks for the syntax of Toy are a set $(x \in) Var$ of *variables* (by this notation we mean that the set is called Var and that symbols like $x, x', x_1, x_2, \ldots$ denote elements of this set), a set $(e \in) Exp$ of *expressions*, and a set $(O \in) OLab$ of *object labels*. The symbol $OLab^+$ is used as a shorthand for $OLab \cup \{*\}$, where $*$ indicates that the object is left unspecified (see below for examples of its use). The expressions in the set *Exp* are considered to be simple, in the sense that they do not have side-effects.

Now we can define the set $(s \in) Stat$ of Toy statements as follows:

$$
\begin{array}{lcl}
s & ::= & x := e \\
 & | & O!e \mid *!e \\
 & | & O?x \mid *?x \\
 & | & s_1 ; s_2 \\
 & | & \textsf{if } e \textsf{ then } s_1 \textsf{ else } s_2 \textsf{ fi} \\
 & | & \textsf{while } e \textsf{ do } s \textsf{ od}
\end{array}
$$

The intended interpretation of the statements is as usual: The assignment statement $x := e$ stores the value of the expression e in the variable x. The output statement $O!e$ sends the value of the expression e to the object with label O and the input statement $O?x$ stores the value it receives from object O in the variable x. These communication actions take place *synchronously:* the object that reaches its communication statement first must wait for its partner. When this partner also reaches a communication statement and moreover the two statements *match* (one is an output statement, the other is an input statement, and they mention each other's object labels), the transfer of the value is performed. After this communication both partners can continue their execution in parallel. In one of the partners (but not in both), the label of the other side can be replaced by an asterisk $*$, so that the statement takes the forms $*!e$ or $*?x$. Such a statement is willing to communicate with an arbitrary partner object, as long as that partner explicitly mentions the name of the object in which the statement occurs. The standard control structures, sequential composition, conditional, and loop, are also present in the language.

A *program* $P \in Prog$ in Toy is a finite sequence of *objects*, where an object is simply a statement labelled by an object name (in CSP terminology [Hoa78], an object would be called 'process', but we reserve the word 'process' for certain semantic entities to be introduced below):

$$P ::= \langle O_1 :: s_1 \parallel \cdots \parallel O_n :: s_n \rangle \qquad \text{where } n \geq 1.$$

These objects are executed in parallel and they can communicate with each other by the communication statements described above. Each object has its own set of variables; it cannot access the variables of another object. Therefore the same variable name, used in different objects, refers to different variables.

2.2. Semantics of Toy Statements

In order to give a semantics to our language, we first have to give an interpretation to its simplest elements, the variables. We assume that our variables can store values that are elements of a set $(v \in) Val$, and that at the beginning of the program execution all variables are initialized to the special undefined value $nil \in Val$.

Now we define the set $(\sigma \in) \Sigma$ of *states* by

$$\Sigma = Var \to Val.$$

Note that states are *local:* A state σ can store the values of all the variables of a *single* object. Each object has its own set of variables and therefore its own state.

For the evaluation of expressions, we just assume the presence of an evaluation function

$$[\![\ ]\!] : Exp \to \Sigma \to Val.$$

(The function space operator $\to$ always brackets to the right, so that this means $Exp \to (\Sigma \to Val)$.) Since expressions do not have side effects and cannot refer to the variables of other objects, a state σ contains enough information to determine the value of an expression instantly.

For describing the semantics of the larger constructs in our language, we use *processes.* These are mathematical structures that describe exactly the execution of the language constructs in question (see also [BaZ82]). We use different kinds of processes for statements, objects and programs. The processes that describe the semantics of statements are called *statement processes* and are elements of the domain $(p \in) SProc$. This domain is a complete metric space defined by the following reflexive domain equation:

$$\begin{aligned} SProc \cong \{p_0\} &\cup (\Sigma \times SProc) \\ &\cup (OLab^+ \times Val \times SProc) \\ &\cup (OLab^+ \times (Val \to SProc)) \end{aligned}$$

In Appendix A we give an overview of the techniques that can be used to prove that this domain equation has exactly one solution up to isomorphism, provided we (implicitly) apply the functor $\mathrm{id}_{1/2}$ to all occurrences of *SProc* at the right-hand side.

Let us now look at the structure of statement processes: The process p_0 is the (successfully) terminated process, which does not perform any action. A statement process of the form $[\sigma, p]$ represents an internal computation step. The first component σ registers the new state after this step (which might be an assignment) and the second component p, called the *resumption* of this step, represents the activity that follows after this first step. A process of the form $[O, v, p]$ represents a send step. The object label O (possibly equal to $*$, the unspecified object label) indicates the receiving object, the second component v is the value to be sent, and the third component, the process p is the resumption of this send step: it describes what happens after this step. Finally, a statement process can have the form $[O, f]$, in which case it models a receive step. The object label O (possibly $*$) indicates from which process a value is expected. The resumption f of this step is a function from values to processes, since the behaviour of the statements after this step in general depends on the value that is received: if this value is v then $f(v)$ is the process that describes what happens after this receive step.

The semantics of statements is now given by a function $\mathcal{M}_S$ of type

$$\mathcal{M}_S : Stat \to Cont \to \Sigma \to SProc.$$

The meaning $\mathcal{M}_S[\![s]\!]$ of a statement s depends on two arguments: a *continuation* $g \in Cont$ and a state σ. The state σ simply represent the values of the variables before the statement s is executed. The set *Cont* of continuations is given by

$$Cont = \Sigma \to SProc.$$

Such a continuation g represents the meaning of everything that will happen after the statement s. Generally it depends on the state resulting from the execution of s. Using continuations can drastically reduce the complexity of the equations that define the semantics of a language. For a simple language like Toy this technique is not really necessary, but we present it here to prepare for Section 3, where it is used to define the semantics of POOL. For a good introduction to continuation semantics, see [Gor79].

The function $\mathcal{M}_S$ is defined by the following clauses:

- Assignment:

$$\mathcal{M}_S[\![x := e]\!](g)(\sigma) = [\sigma', g(\sigma')]$$

where $\sigma' = \sigma\{[\![e]\!](\sigma)/x\}$. Here we have made use of the *variant notation:* If $f : X \to Y$ is a function, $x \in X$, and $y \in Y$, then $f\{y/x\}$ is again a function in $X \to Y$, defined by

$$f\{y/x\}(z) = \begin{cases} y & \text{if } z = x \\ f(z) & \text{otherwise.} \end{cases}$$

The statement process describing the execution of an assignment first performs an internal computation step. The first component of this step describes the new state σ', which differs from the original state σ in that the variable x has got the value $[\![e]\!](\sigma)$ of the expression e in the original state σ. The second component, the resumption of this step, which is the process describing everything that happens after the first step, can be obtained by applying the continuation g to the new state σ'.

- Output statement:

$$\begin{aligned} \mathcal{M}_S[\![O\,!e]\!](g)(\sigma) &= [O, [\![e]\!](\sigma), g(\sigma)] \\ \mathcal{M}_S[\![*\,!e]\!](g)(\sigma) &= [*, [\![e]\!](\sigma), g(\sigma)] \end{aligned}$$

Here the first step is a send step. It contains the label O of the receiving object (or $*$, if the receiver is not specified), the value $[\![e]\!](\sigma)$ to be transmitted, and the resumption, which is obtained by applying the continuation g to the (unchanged) state σ.

- Input statement:

$$\begin{aligned} \mathcal{M}_S[\![O\,?x]\!](g)(\sigma) &= [O, \lambda v.g(\sigma\{v/x\})] \\ \mathcal{M}_S[\![*\,?x]\!](g)(\sigma) &= [*, \lambda v.g(\sigma\{v/x\})] \end{aligned}$$

The first step executed by an input statement is a receive step of the form $[O, f]$. The first component O is the label of the sending object (or $*$). The second component f is the resumption, which depends on the value v that is received. The function f is defined in such a way that for a given value v the resumption $f(v)$ is equal to $g(\sigma\{v/x\})$. This means that first a new state $\sigma\{v/x\}$ is determined, where v is stored in the variable x, and then the continuation g is applied to this new state, yielding the process $g(\sigma\{v/x\}) = f(v)$ that describes the actions of the current object after this receive step.

- Sequential composition:

$$\mathcal{M}_S[\![s_1 ; s_2]\!](g)(\sigma) = \mathcal{M}_S[\![s_1]\!]\Big(\mathcal{M}_S[\![s_2]\!](g)\Big)(\sigma)$$

Here we see most clearly the kind of simplification in the semantic equations that can result from the use of continuations. The sequential composition of two statements can be described by using the semantics of the second statement as the continuation for the semantics of the first statement. In more detail: g is a function in $\Sigma \to SProc$ describing everything that happens *after* the two statements; $\mathcal{M}_S[\![s_2]\!](g)$ is also a function in $\Sigma \to SProc$ (so it can also be used as a continuation) and it describes the execution of s_2 plus everything that happens afterwards, so $\mathcal{M}_S[\![s_1]\!](\mathcal{M}_S[\![s_2]\!](g))$ is also a function in $\Sigma \to SProc$ that, when applied to a state σ, delivers a process that describes the execution of first the statement s_1, then the statement s_2, and then the rest.

- Conditional statement:

$$\mathcal{M}_S[\![\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}]\!](g)(\sigma) = \begin{cases} \mathcal{M}_S[\![s_1]\!](g)(\sigma) & \text{if } [\![e]\!](\sigma) \neq nil \\ \mathcal{M}_S[\![s_2]\!](g)(\sigma) & \text{otherwise} \end{cases}$$

Since there is no special data type for Booleans in the language Toy, we base the decision in a conditional statement on whether the value of the expression e is *nil* or not, where *nil* stands for 'false'.

- Loop statement:

$$\mathcal{M}_S[\![\mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}]\!](g)(\sigma) = \begin{cases} \mathcal{M}_S[\![s]\!]\left(\mathcal{M}_S[\![\mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}]\!](g)\right)(\sigma) \\ \qquad\qquad \text{if } [\![e]\!](\sigma) \neq nil \\ [\sigma, g(\sigma)] \quad \text{otherwise} \end{cases}$$

If the condition is not *nil*, then executing the loop is equivalent to first executing the statement s and then executing the loop again. If the condition is *nil*, then the loop immediately terminates and control passes to the statements following it, which are represented by the continuation g.

The definition of $\mathcal{M}_S$ needs some formal justification, since it cannot be justified by a simple induction on the syntactic complexity of the statements (in the clause for the while statement, the function value to be defined occurs also at the right-hand side). Rather than treating the while statement separately, we give the definition of $\mathcal{M}_S$ as a whole as a fixed point of a higher-order contracting function, as follows. Define the domain D by

$$(F \in) D = Stat \to Cont \overset{1/2}{\to} \Sigma \to SProc.$$

(Here $X \overset{1/2}{\to} Y$ is the space of all functions $f : X \to Y$ such that $d(f(x_1), f(x_2)) \leq 1/2 \cdot d(x_1, x_2)$ for any $x_1, x_2 \in X$.) Now we define the operator $\Psi : D \to D$ by the following clauses:

$$\begin{aligned} \Psi(F)[\![x := e]\!](g)(\sigma) &= [\sigma', g(\sigma')] \qquad \text{where } \sigma' = \sigma\{[\![e]\!](\sigma)/x\} \\ &\vdots \\ \Psi(F)[\![s_1 ; s_2]\!](g)(\sigma) &= \Psi(F)[\![s_1]\!](\Psi(F)[\![s_2]\!](g))(\sigma) \\ &\vdots \\ \Psi(F)[\![\mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}]\!](g)(\sigma) &= \begin{cases} \Psi(F)[\![s]\!](F[\![\mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}]\!](g))(\sigma) \\ \qquad\qquad \text{if } [\![e]\!](\sigma) \neq nil \\ [\sigma, g(\sigma)] \quad \text{otherwise} \end{cases} \end{aligned}$$

It is clear that the above definition of Ψ *can* be justified by induction on the syntactic complexity. By induction on the complexity of a statement s we can prove that for any $F \in D$ the result $\Psi(F)[\![s]\!]$ is indeed an element of $Cont \overset{1/2}{\to} \Sigma \to SProc$, i.e., that it reduces distances by a factor $1/2$. Here we use the fact that the functor

$\mathrm{id}_{1/2}$ is applied to all occurrences of *SProc* in its defining domain equation, and that in the basic clauses for $\Psi(F)$ the continuation g is always applied to a state to yield a process that serves as a resumption. Now we note that the only place where the function F occurs at the right-hand side without Ψ being applied to it is in the clause for the while statement, where it occurs in the continuation for $\Psi(F)[\![s]\!]$. Therefore Ψ is indeed a contracting function (see Appendix A), so by Banach's Theorem it has a unique fixed point. This fixed point satisfies exactly the equations that we have given above for $\mathcal{M}_S$, so we can define $\mathcal{M}_S$ to be this fixed point.

2.3. Semantics of Objects

The semantics of an object is obtained by taking the statement semantics ($\mathcal{M}_S$) of the statement executed by the object and forgetting about the local computation steps. To this end we introduce a domain $(q \in)OProc$ of *object processes.* This domain is defined by

$$\begin{aligned} OProc \cong \{q_0\} &\cup (OLab^+ \times Val \times OProc) \\ &\cup (OLab^+ \times (Val \to OProc)). \end{aligned}$$

The domain *OProc* can be viewed as being (isomorphic to) the subset of *SProc* consisting of those processes that do not contain internal computation steps.

Next we define an abstraction operator $\alpha : SProc \to OProc$, which makes all the internal computation steps invisible, so that their effects only become apparent through the send and receive steps that the process performs. Note that this corresponds to the intuitive fact that we cannot observe the state of an object directly, but only indirectly through the messages that it sends and receives. We want the operator α to satisfy the following equations:

$$\begin{aligned} \alpha(p_0) &= q_0 \\ \alpha([\sigma, p]) &= \alpha(p) \\ \alpha([O, v, p]) &= [O, v, \alpha(p)] \\ \alpha([O, f]) &= [O, \lambda v.\alpha(f(v))] \\ \alpha([\sigma_1, [\sigma_2, [\sigma_3, \ldots]]]) &= q_0 \end{aligned}$$

(Note that the last clause is really necessary, since the first four clauses do not fix the value of α for an infinite sequence of internal steps.) We can obtain such an operator α as the unique fixed point of the higher-order contracting operator $\Phi : (SProc \to OProc) \to (SProc \to OProc)$ defined by

$$\begin{aligned} \Phi(\phi)([\sigma_1, \cdots [\sigma_n, p_0] \cdots]) &= q_0 & (n \geq 0) \\ \Phi(\phi)([\sigma_1, \cdots [\sigma_n, [O, v, p]] \cdots]) &= [O, v, \phi(p)] & (n \geq 0) \\ \Phi(\phi)([\sigma_1, \cdots [\sigma_n, [O, f]] \cdots]) &= [O, \lambda v.\phi(f(v))] & (n \geq 0) \\ \Phi(\phi)([\sigma_1, [\sigma_2, [\sigma_3, \ldots]]]) &= q_0 & \end{aligned}$$

It is not difficult to see that Φ is indeed a contraction (at the right-hand side, ϕ occurs only inside a resumption, where the functor $\mathrm{id}_{1/2}$ applies) and that its unique fixed point satisfies the equations given above for α. As is usually the case with operators that hide (internal computation) steps, α is not continuous: If we define the sequence $p_1, p_2, \ldots$ by $p_1 = [O, v, p_0]$ and $p_{n+1} = [\sigma, p_n]$ for some arbitrary O, v, and σ, then $\lim_n p_n = p_\infty = [\sigma, [\sigma, [\sigma \ldots]]]$. Applying α we get that $\alpha(p_n) = [O, v, q_0]$ for all n, but $\alpha(p_\infty) = q_0$. It is somewhat surprising that α can be defined as the fixed point of a higher-order contracting operator, although it is not continuous itself.

Now we can introduce the second semantic mapping $\mathscr{M}_O : Stat \to OProc$, given by

$$\mathscr{M}_O[\![s]\!] = \alpha(\mathscr{M}_S[\![s]\!](\lambda\sigma.p_0)(\lambda x.nil)).$$

It is obtained by applying the abstraction operator α to the meaning of s as a statement (given by $\mathscr{M}_S$), supplied with the empty continuation $\lambda\sigma.p_0$ (indicating that after s nothing has to be done any more) and the nowhere defined state $\lambda x.nil$ (indicating that at the beginning of the execution of s all variables have been initialized to nil).

The semantics of objects, given by the function $\mathscr{M}_O$, contains all the details that are necessary to describe how objects interact with each other (by communication), but the information describing how an object works internally (e.g., how it accesses and changes its own state) has been removed.

2.4. Semantics of Programs

The meaning of a program (the parallel composition of a number of objects) will consist of the communications between this program and the outside world. Therefore let us start by defining the latter.

We assume the presence of two special elements O_{in} and O_{out} in $OLab$, representing the input and the output half of the outside world. These object labels may occur in the communication statements of a program, and in this way the program can communicate with the outside world. For instance, the statement $O_{out}!3$ will output the value 3 to the outside world. Conversely, $O_{in}?x$ will input a value and store it in the variable x.

Formally, the outside world is modelled by a pair of object processes, q_{in} and q_{out} in $OProc$. More precisely, the process q_{in} depends on a finite or infinite sequence $w \in Val^{\infty}$, consisting of the values that are offered as input to the program. We define

$$\begin{array}{lcl} q_{in}(\langle\rangle) & = & q_0 \\ q_{in}(v \cdot w) & = & [*, v, q_{in}(w)] \end{array}$$

The latter triple indicates that the value v is sent to any process that is willing to accept it (by a statement of the form $O_{in}?x$), after which the remaining values in w will be sent. (In order to define q_{in} rigorously on infinite sequences, it can be taken as the fixed point of a contracting operator in the usual way.)

The output half of the world, q_{out}, is given by

$$q_{out} = [*, \lambda v.q_{out}].$$

It represents a continuous willingness to accept values from any process wishing to send a value to the outside world (by a statement of the form $O_{out}!e$). The process q_{out} itself does nothing with the values it receives; we shall see below how they are extracted to arrive at the output of the program.

In order to describe the global behaviour of programs, a third kind of semantic domain is introduced: the set $(r \in)GProc$ of global processes, defined by

$$\begin{array}{rcl}
GProc & = & \{r_0\} \cup \mathcal{P}_{co}(GStep) \\
(\pi \in)GStep & = & (OLab \times OLab^+ \times Val \times GProc) \\
& \cup & (OLab^+ \times OLab \times (Val \to GProc)) \\
& \cup & (Comm \times GProc) \\
(c \in)Comm & = & OLab \times Val \times OLab
\end{array}$$

The terminated process is indicated by r_0. All other kinds of global processes consist of a *set* of possible steps. This is the way in which nondeterminism (which comes from the fact that parallelism is modelled by nondeterministic interleaving, as we shall see below) is modelled in our semantics: If such a process is executed, it will nondeterministically choose one step from among the members of the set. A step can have one of three possible forms: a send step, a receive step, or a communication step. The interpretation of send steps (of the form $[O_1, O_2, v, r]$) and receive steps (of the form $[O_1, O_2, f]$) is similar to their counterparts in *OProc*. The only difference is that now the labels of both the sending and the receiving objects (in that order) are registered. (Note that in a send step $[O_1, O_2, v, r]$ the receiver O_2 might be unspecified ($*$) and symmetrically, in a receive step $[O_1, O_2, f]$, the sender O_1 may be $*$.) Finally, a step of type $[c, r]$ represents a successful communication c with resumption r. Communications are of the form $[O_1, v, O_2]$, indicating that object O_1 has sent the value v to object O_2.

We shall need to be able to compose global processes in parallel. For this purpose we define the operator $\parallel : GProc \times GProc \to GProc$ by

$$\begin{array}{rcl}
r \parallel r_0 & = & r_0 \parallel r = r \\
r_1 \parallel r_2 & = & \{\pi \mathbin{\|\!\_}\, r_2 : \pi \in r_1\} \cup \{\pi \mathbin{\|\!\_}\, r_1 : \pi \in r_2\} \\
& \cup & \bigcup\{\pi_1 \mid \pi_2 : \pi_1 \in r_1, \pi_2 \in r_2 \text{ or } \pi_1 \in r_2, \pi_2 \in r_1\} \\
{[O_1, O_2, v, r]} \mathbin{\|\!\_}\, r_2 & = & [O_1, O_2, v, r \parallel r_2] \\
{[O_1, O_2, f]} \mathbin{\|\!\_}\, r_2 & = & [O_1, O_2, \lambda v.(f(v) \parallel r_2)] \\
{[c, r]} \mathbin{\|\!\_}\, r_2 & = & [c, r \parallel r_2] \\
\pi_1 \mid \pi_2 & = & \begin{cases} \{[(O_1, v, O_2), f(v) \parallel r]\} & \text{if } \pi_1 = [O_1, O_2^+, v, r] \\ & \text{and } \pi_2 = [O_1^+, O_2, f] \\ \emptyset & \text{otherwise} \end{cases}
\end{array}$$

(Here r_1 and r_2 are supposed to be unequal to r_0, and the notation O_i^+ has been used as a shorthand for O_i or $*$, where at most one of O_1^+ and O_2^+ may be $*$.)

A brief explanation: As already announced above, we model two processes executing in parallel by taking all the possible ways in which their individual steps can be combined or interleaved. Composing a process r in parallel with the terminated process yields r itself. The result of composing in parallel two processes r_1 and r_2, both of which are not r_0, is a set union of three parts: in the first part, the first step is performed by r_1 (indicated by the left merge operator $\|\!\_$); in the second part, the first step is performed by r_2; and in the last part, the first step is a communication of a step from r_1 with a step from r_2 (indicated by the communication merge $|$). The left merge $\|\!\_$ operator effectively composes its second argument with the resumption of the first. The communication merge of two steps yields a singleton if the steps match, and the empty set otherwise.

Before we can define the global semantics of programs, one more definition is needed. It is an operator $\omega : OProc \to OLab \to GProc$ that translates an object process, together with the label of the object that executes it, into a global process, as follows:

$$\begin{array}{lcl} \omega(q_0)(O') & = & r_0 \\ \omega([O,v,q])(O') & = & \{[O',O,v,\omega(q)(O')]\} \\ \omega([O,f])(O') & = & \{[O,O',\lambda v.\omega(f(v))(O')]\} \end{array}$$

Finally, we can define the meaning function for programs $\mathcal{M}_G : Prog \to Val^\infty \to GProc$:

$$\begin{array}{l} \mathcal{M}_G[\![\langle O_1 :: s_1 \parallel \cdots \parallel O_n :: s_n\rangle]\!](w) \\ \quad = \ \omega(\mathcal{M}_O[\![s_1]\!])(O_1) \parallel \cdots \parallel \omega(\mathcal{M}_O[\![s_n]\!])(O_n) \\ \qquad \parallel \omega(q_{in}(w))(O_{in}) \parallel \omega(q_{out})(O_{out}) \end{array}$$

We see that the semantics of a program consists of the parallel composition of the object processes of all the objects plus the input and output object, after they have been translated to global processes.

However, processes in *GProc* contain more information than we consider relevant for the observable behaviour of a program. In particular, only the values sent by the program to the outside world are of importance. These can be extracted from a global process by means of the operator *output* defined below. First the operator *path* : $GProc \to \mathcal{P}(Comm \times GProc)^\infty$ is introduced, which computes all the finite and infinite sequences of succesful communication steps of a process:

$$\begin{array}{l} path(r) = \Big\{\langle [c_1,r_1],\ldots,[c_n,r_n]\rangle : \\ \qquad [c_1,r_1] \in r \wedge \forall 1 \le i < n\ [c_{i+1},r_{i+1}] \in r_i \wedge \neg\exists c,r'\ [c,r'] \in r_n\Big\} \\ \quad \cup \Big\{\langle [c_1,r_1],\ldots\rangle : [c_1,r_1] \in r \wedge \forall i \ge 1\ [c_{i+1},r_{i+1}] \in r_i\Big\} \end{array}$$

Now we can define the function *output* : $GProc \to \mathcal{P}(Val^\infty)$ by

$$output(r) = \{\, \mathcal{V}(c_1)\cdot\mathcal{V}(c_2)\cdot\cdots : \langle [c_i,r_i]\rangle_i \in path(r)\,\}$$

where

$$\mathcal{V}(c) = \begin{cases} \langle v\rangle & \text{if } c = [O,v,O_{out}] \\ \langle\rangle & \text{otherwise} \end{cases}$$

Finally, the observable behaviour of a program can be given as follows:

$$\begin{array}{l} obs : Prog \to Val^\infty \to \mathcal{P}(Val^\infty) \\ obs[\![P]\!](w) = output(\mathcal{M}_G[\![P]\!](w)) \end{array}$$

For a given program and a (finite or infinite) sequence of input values, this function *obs* delivers the set of all possible sequences of output values.

3. The Language POOL and its Semantics

In this section we shall introduce the language POOL, a parallel object-oriented programming language, and give a semantics for it at three levels, following the same basic scheme as that in Section 2.

3.1. Informal Introduction to the Language

The language POOL [Ame87, Ame89b] makes use of the principles of object-oriented programming in order to give structure to parallel systems. A POOL program describes the behaviour of a whole system in terms of its constituents, *objects*. Objects contain some internal data and some procedures that act on these data (these are called *methods* in the object-oriented jargon). Objects are entities of a dynamic nature: they can be created dynamically, their internal data can be modified, and they even have an internal activity of their own. At the same time they are units of protection: the internal data of one object are not directly accessible to other objects.

An object uses *variables* (more specifically: instance variables) to store its internal data. Each variable can contain a *reference* to an object (another object or, possibly, itself). An assignment to a variable can make it refer to a different object. The variables of one object cannot be accessed directly by other objects. They can only be read and changed by the object itself.

Objects can only interact by sending *messages* to each other. A message is a request for the receiver to execute a certain method. Messages are sent and received explicitly. In sending a message, the sender mentions the destination object, the method to be executed, and possibly some parameters (which are again references to objects) to be passed to this method. After this its activity is suspended. The receiver can specify the set of methods that will be accepted, but it can place no restrictions on the identity of the sender or on the parameters of messages. If a message arrives specifying an appropriate method, the method is executed with the parameters contained in the message. Upon termination, this method delivers a result (a reference to an object), which is returned to the sender of the message. The latter then resumes its own execution.

A method can access the variables of the object that executes it (the receiver of a message). Furthermore it can have some temporary variables, which exist only during the execution of the method. In addition to answering a message, an object can execute a method of its own simply by calling it. Because of this, and because answering a message within a method is also allowed, recursive invocations of methods are possible. Each of these invocations has its own set of parameters and temporary variables.

When an object is created, a local activity is started: the object's *body*. When several objects have been created, their bodies may execute in parallel. This is the way parallelism is introduced into the language. Synchronization and communication take places by sending messages, as described above.

Objects are grouped into *classes*. All objects in one class (the *instances* of that class) have the same number and kind of variables, the same methods for answering messages, and the same body. In creating an object, only its desired class must be specified. In this way a class serves as a blueprint for the creation of its instances.

There is a special value, *nil*, which refers to no object at all. If a message is sent with *nil* as destination, an error occurs. Upon the creation of a new object, its instance variables are initialized to *nil* and when a method is invoked its temporary variables are also initialized to *nil*.

There are a few standard classes predefined in the language. In this semantic description we shall only incorporate the classes Bool and Int. The usual operations can be performed on these objects, but they must be formulated by sending messages. For example, the addition $2+4$ is indicated by the expression 2!add(4), sending a message with method name add and parameter 4 to the object 2.

3.2. Syntax of POOL

In this section we describe the syntax of the language POOL as we study it in this paper. The concrete syntax of the language that is used for actual programming is relatively complex, since it offers many convenient short-hand notations for programmers. In order to avoid this complexity in this paper, we shall define an *abstract syntax*, which is much simpler. Nevertheless, all the essential semantic ingredients of the language have been maintained, so that every concrete POOL program can be translated straightforwardly into our abstract syntax.

As a starting point for the definition of the POOL syntax, we assume the existence of the set $(x \in) IVar$ of instance variables, the set $(u \in) TVar$ of temporary variables, the set $(C \in) CName$ of class names, and the set $(m \in) MName$ of method names. We define the set $(\phi \in) SObj$ of standard objects as follows:

$$SObj = \mathbf{Z} \cup \{\mathbf{t}, \mathbf{f}\} \cup \{nil\}$$

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

Now we can define the set $(e \in) Exp$ of expressions by the following clauses:

$$\begin{array}{rcll} e & ::= & x & \\ & | & u & \\ & | & m(e_1, \ldots, e_n) & (n \geq 0) \\ & | & e!m(e_1, \ldots, e_n) & (n \geq 0) \\ & | & \mathsf{condans}\{m_1, \ldots, m_n\} & (n \geq 1) \\ & | & \mathsf{new}(C) & \\ & | & e_1 \equiv e_2 & \\ & | & s; e & \\ & | & \mathsf{self} & \\ & | & \phi & \end{array}$$

The set $(s \in) Stat$ of statements is defined by

$$\begin{array}{rcll} s & ::= & x \leftarrow e & \\ & | & u \leftarrow e & \\ & | & \mathsf{answer}\{m_1, \ldots, m_n\} & (n \geq 1) \\ & | & e & \\ & | & s_1; s_2 & \\ & | & \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi} & \\ & | & \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od} & \end{array}$$

The set $(\mu \in) MethDef$ of method definitions is given by

$$\mu ::= [\langle u_1, \ldots, u_n \rangle, e] \qquad (n \geq 0),$$

the set $(d \in) ClassDef$ of class definitions by

$$d ::= [\langle m_1 \Leftarrow \mu_1, \ldots, m_n \Leftarrow \mu_n \rangle, s] \qquad (n \geq 0),$$

and finally the set $(P \in) Prog$ of programs is defined by

$$P ::= \langle C_1 \Leftarrow d_1, \ldots, C_n \Leftarrow d_n \rangle \qquad (n \geq 1).$$

3.2.1. *Informal Explanation*

First of all, it may be important to note that the difference between expressions and statements in POOL is only that expressions yield a value whereas statements do not. In particular, expressions can have quite drastic side-effects (but these are always defined exactly by the language).

Expressions: An instance variable or a temporary variable used as an expression will yield as its value the object name that is currently stored in that variable.

A method call simply means that the corresponding method is executed by the object itself. This is done as follows: First the argument expressions $e_1, \ldots, e_n$ are evaluated from left to right. Then a new set of temporary variables is taken, in the sense that their current values are remembered and they are given new values as follows: The argument values are assigned to the corresponding parameters, i.e., the temporary variables listed in the method definition, and the other temporary variables are initialized to *nil*. Then the expression in the method definition is evaluated; the result of this evaluation will be the value of the method call. Before the method call terminates, the original values of the temporary variables are restored.

The next kind of expression is a send expression. Here e is the destination object to which the message will be sent, m is the method to be invoked, and $e_1, \ldots, e_n$ are the arguments. When a send expression is evaluated, first the destination expression is evaluated, then the arguments are evaluated from left to right and then the message is sent to the destination object and the sending object does nothing while it awaits the result. When the destination object answers the message (which might, however, never happen), the corresponding method is executed; that is, the parameters are initialized to the argument values contained in the message, the other temporary variables are initialized to *nil*, and the expression in the method definition is evaluated. The value which results from this evaluation is sent back to the sender of the message and this will be the value of the send expression.

The conditional answer expression is a variant of the answer statement described below. This expression can answer a message that mentions a method name from the set $\{m_1, \ldots, m_n\}$, if such a message is present. In this case its value will be **t** (true). Otherwise it terminates without answering a message, yielding the value **f** (false).

A new-expression indicates that a new object is to be created, an instance of the class C. The instance variables of this object are initialized to *nil* and its body starts executing in parallel with all other objects in the system. The result of the new-expression is a reference to this newly created object.

The next type of expression checks whether e_1 and e_2 result in a reference to the same object. If so, **t** is returned, otherwise **f**. An expression may also be preceded by a statement. In this case the statement is executed before the expression is evaluated. The expression self always results in a reference to the object that is executing this expression. Finally, the evaluation of a standard object ϕ results in that object itself. For instance, the value of the expression 23 will be the natural number 23.

Statements: The first two kinds of statements are assignments to an instance variable and to a temporary variable. An assignment is executed by first evaluating the expression on the right and then making the variable on the left refer to the resulting object.

The next kind of statement is an answer statement. This indicates that a message is to be answered. The object executing the answer statement waits until a message arrives with a method name that is contained in the set $\{m_1, \ldots, m_n\}$. Then it executes the method (after initializing the parameters and temporary variables). The result of the method is sent back to the sender of the message and the answer statement terminates. The difference with a conditional answer expression is that an answer statement always answers exactly one message before terminating, whereas a conditional answer expression answers at most one message.

Any expression may also occur as a statement. Upon execution, the expression is evaluated and the result is discarded. So only the side effects of the expression evaluation (e.g., the sending of a message) are important. Sequential composition, conditionals and loops have the usual meaning.

Method definitions: A method definition describes a method. Here $u_1, \ldots, u_n$ are the parameters and e is the expression to be evaluated when the method is invoked. Upon execution of this method, the parameters are initialized to the corresponding argument values, the other temporary variables are initialized to *nil*, and the expression e is evaluated. Not only is the value of this expression important, but in general also its side-effects.

Class definitions: A class definition describes how instances of the specified class behave. It indicates the methods and the body each instance of the class will have. The set of instance variables is implicit here: it consists of all the elements of *IVar* that occur in the methods or in the body.

Programs: A program consists of a number of bindings of class names to class definitions. If a program is to be executed, a single new instance of the *last* class defined in the program is created and execution of its body is started. This object, which we call the *root object*, has the task of starting the whole system by creating new objects and putting them to work.

3.2.2. Context Conditions

For a POOL program to be valid a few more conditions need to be satisfied. We assume in the semantic treatment that the underlying program is valid. These context conditions are the following:

- All class names in a program are different.
- All method names in a class definition are different.
- All parameters in a method definition are different.
- Every method name that is used in a method call, conditional answer expression, or answer statement within a certain class definition is bound to a method definition in that class definition.

Any POOL program that is a translation of a valid POOL-T [Ame87] or POOL2 [Ame89b] program will automatically satisfy these conditions. POOL-T and POOL2 are even more restrictive. For example, they require that the type of every expression conforms with its use and they forbid assignments to formal parameters. However, the conditions above are sufficient to ensure that the program will have a well-defined semantics.

3.3. Semantics of POOL Expressions and Statements

Before the domain of statement (and expression) processes for POOL can be defined, we first need to introduce a few more sets. We define the set *AObj* of *active object names* by $AObj = \mathbf{N}^*$. That is, we use finite sequences of integers to name objects. The intention is that the empty sequence $\langle\rangle$ denotes the root object, and for any active object name α and integer n, the object name $\alpha \cdot \langle n \rangle$ denotes the nth object created by the object α.

The set $AObj$ of active object names and the set $SObj$ of standard objects together form the set $(\alpha, \beta, \gamma \in) Obj$ of object names: $Obj = AObj \cup SObj$. Now we define the set $(\sigma \in)\Sigma$ of *states* by

$$\Sigma = (IVar \rightarrow Obj) \times (TVar \rightarrow Obj).$$

Every state σ consists of two components that register, for a particular object, the values of the instance variables and the values of the temporary variables. For readability we also introduce the following sets:

$$\begin{array}{lcl} New & = & CName \\ NewName & = & AObj \\ Result & = & Obj \\ Send & = & Obj \times MName \times Obj^* \end{array}$$

(For any set A, we denote by A^* the set of finite sequences of elements of A.)

Now we can define the domain $(p \in) SProc$ of *statement processes* to be the unique fixed point of the following domain equation:

$$\begin{aligned} SProc \cong \{p_0\} & \cup (\Sigma \times SProc) \\ & \cup (New \times (NewName \rightarrow SProc)) \\ & \cup (Send \times (Result \rightarrow SProc)) \\ & \cup (MName \xrightarrow{fin} (Obj^* \rightarrow SProc)) \\ & \cup (MName \xrightarrow{fin} (Obj^* \rightarrow SProc)) \times SProc \\ & \cup (Result \times SProc) \end{aligned}$$

(With $A \xrightarrow{fin} B$ we denote the set of finite partial maps from A to B.)

We see that a statement process can have one of seven possible forms:

1. The terminated process p_0.
2. An internal computation step $[\sigma, p]$. The first component indicates the new state immediately after this step and the second component is the resumption, which describes everything that will happen after the first step.
3. A creation step $[C, f]$. This describes the creation of an object of class C. The creation itself is done by a mechanism outside the object. The resumption of this step is given by $f(\beta)$, where β is the name of the new object.
4. A send step $[(\beta, m, \bar{\beta}), f]$. The first component describes the contents of the message that is sent: β is the destination, m is the method name, and $\bar{\beta}$ is the sequence of argument values. The resumption of this send step is given by applying the function f to the result of the message.
5. An answer step g. This step indicates that the object is ready to answer any message that mentions a method name m that is in the (finite) domain of g. If the argument values in the message are given by $\bar{\beta}$, then the resumption of this step is $g(m)(\bar{\beta})$.
6. A conditional answer step $[g, p]$. This process is similar to the previous one but it has an extra component. If a message of the form $[\beta, m, \bar{\beta}]$ with $m \in \operatorname{dom} g$ has arrived, it can be answered, in which case the resumption is $g(m)(\bar{\beta})$. Otherwise, no message is answered and the resumption is just p.

7. A result step $[\gamma, p]$. This step returns γ as a result of a message that has been sent earlier to this object (an external mechanism will deliver this result to the sending object). The resumption of this step is given by p.

Next the semantics of expressions and statements in a class definition d is given by means of two meaning functions

$$\begin{array}{lll} \mathcal{M}_E^d & : & Exp \to AObj \to ECont \to \Sigma \to SProc \\ \mathcal{M}_S^d & : & Stat \to AObj \to SCont \to \Sigma \to SProc \end{array}$$

where

$$\begin{array}{lll} (h \in)ECont & = & Obj \to \Sigma \to SProc \\ (c \in)SCont & = & \Sigma \to SProc \end{array}$$

are the sets of *expression continuations* and *statement continuations.*

We see that the types of the meaning functions for expressions and for statements are very similar. The reason why we cannot use a very simple meaning function for expressions such as the one in Section 2.2 is that in POOL an expression can have side-effects: the evaluation of an expression may involve creating new objects and sending or answering messages. Therefore the only difference between expressions and statements in POOL is that expressions yield a value whereas statements do not. This difference is reflected in their respective continuations: the continuation of a statement depends only on the state after this statement, but the continuation of an expression also depends on its value.

If we compare the types of these semantic functions to the one in Section 2.2, we see that they need one extra argument: the name of the object that executes the expression or statement. Since it does not change during the computation, it does not belong in the state. In fact, it is only needed to evaluate the expression self.

We define the functions $\mathcal{M}_E^d$ and $\mathcal{M}_S^d$ by the following clauses:

Expressions:

- Instance variable:

  $$\mathcal{M}_E^d[\![x]\!](\alpha)(h)(\sigma) = [\sigma, h(\sigma_{(1)}(x))(\sigma)]$$

  We deliver an internal computation step where the state is unchanged and the resumption is obtained by feeding the continuation h with the current value of the variable x, which can be found in the first component $\sigma_{(1)}$ of the state.
- Temporary variable:

  $$\mathcal{M}_E^d[\![u]\!](\alpha)(h)(\sigma) = [\sigma, h(\sigma_{(2)}(u))(\sigma)]$$

  This is similar to an instance variable, but now the value is found in the second component $\sigma_{(2)}$.
- Method call:

  $$\begin{aligned} &\mathcal{M}_E^d[\![m(e_1, \ldots, e_n)]\!](\alpha)(h) \\ &\quad = \mathcal{M}_E^d[\![e_1]\!](\alpha)\Big(\\ &\qquad\quad \lambda\beta_1.\mathcal{M}_E^d[\![e_2]\!](\alpha)\Big(\ldots \\ &\qquad\qquad \lambda\beta_n.\lambda\sigma.\Big[\check{\sigma}, \mathcal{M}_E^d[\![e]\!](\alpha)(h')(\check{\sigma})\Big]\ldots\Big)\Big) \end{aligned}$$

  where

$$\begin{array}{rcl} \check{\sigma} & = & [\sigma_{(1)}, (\lambda u.nil)\{\beta_i/u_i\}_{i=1}^n] \\ h' & = & \lambda\gamma.\lambda\sigma'.h(\gamma)(\widehat{\sigma'}) \\ \widehat{\sigma'} & = & [\sigma'_{(1)}, \sigma_{(2)}] \end{array}$$

and $m \Leftarrow [\langle u_1, \ldots, u_n\rangle, e]$ occurs in the class definition d.
The first action to be taken here is the evaluation of the first argument expression e_1. The corresponding meaning function $\mathcal{M}_E^d[\![e_1]\!]$ is provided with a continuation that takes the value β_1 of e_1 and starts to evaluate the second argument expression e_2. This continues until all the arguments have been evaluated. The last continuation takes the last value β_n of e_n and a state σ and performs an internal computation step where the state is changed to $\check{\sigma}$, having new values for the temporary variables (in implementation terms, one could say that a fresh set of temporary variables is pushed onto the execution stack). Most of these temporary variables are initialized to *nil*, but the parameters $u_1, \ldots, u_n$ of the method m are set to the corresponding argument values $\beta_1, \ldots, \beta_n$. After that (in the resumption of this computation step) the expression e in the method definition is evaluated. The meaning function $\mathcal{M}_E^d[\![e]\!]$ that does this is fed with a continuation h' that takes the value γ of e and the resulting state σ' and feeds these into the original continuation h, but only after restoring the original values of the temporary variables from $\sigma_{(2)}$ in $\widehat{\sigma'}$ (the execution stack is popped).
It might be instructive for the reader to write out explicitly the cases where the number of argument expressions is 0 or 1.

- Send expression:

$$\begin{array}{l} \mathcal{M}_E^d[\![e!m(e_1, \ldots, e_n)]\!](\alpha)(h) \\ \quad = \mathcal{M}_E^d[\![e]\!](\alpha)\Big(\\ \qquad \lambda\beta.\mathcal{M}_E^d[\![e_1]\!](\alpha)\Big(\ldots \\ \qquad\quad \lambda\beta_n.\lambda\sigma.[(\beta, m, \langle\beta_1, \ldots, \beta_n\rangle), \lambda\gamma.h(\gamma)(\sigma)]\ldots\Big)\Big) \end{array}$$

This is similar to a method call, except that after evaluating the destination expression e and the argument expressions $e_1, \ldots, e_n$, a send step is performed. The first component of this send step contains the destination object β, the method name m, and the argument values $\beta_1, \ldots, \beta_n$. The resumption is obtained by applying the continuation h to the result value γ of the message and the state σ just before the send step.

- Conditional answer expression:

$$\mathcal{M}_E^d[\![\mathsf{condans}\{m_1, \ldots, m_n\}]\!](\alpha)(h)(\sigma) = [g, h(\mathbf{f})(\sigma)]$$

where

$$\begin{array}{rcl} g(m)(\langle\beta_1, \ldots, \beta_k\rangle) & = & \begin{cases} \left[\check{\sigma}, \mathcal{M}_E^d[\![e]\!](\alpha)\left(\lambda\gamma.\lambda\sigma'.[\gamma, h(\mathbf{t})(\widehat{\sigma'})]\right)(\check{\sigma})\right] & \text{if } m \in \{m_1, \ldots, m_n\} \\ \text{undefined} & \text{otherwise} \end{cases} \\ \check{\sigma} & = & [\sigma_{(1)}, (\lambda u.nil)\{\beta_i/u_i\}_{i=1}^k] \\ \widehat{\sigma'} & = & [\sigma'_{(1)}, \sigma_{(2)}] \end{array}$$

and $m \Leftarrow [\langle u_1, \ldots, u_k\rangle, e]$ occurs in the class definition d.
Here a conditional answer step is performed. The second component reflects the fact that such a step can be taken if no suitable messages are present,

in which case the value of the conditional answer expression is **f** (false). The first component is a function g that is only defined on the method names $m_1, \ldots, m_n$ mentioned in the conditional answer expression. When applied to such a method name m and a sequence $\langle \beta_1, \ldots, \beta_k \rangle$ of argument values, it delivers a process, which starts with an internal step. In this first step a new set of temporary variables is prepared (cf. $\check{\sigma}$) and in the resumption the expression e from the method definition is evaluated. The meaning function $\mathscr{M}^d_E[\![e]\!]$ that describes this is given a continuation that begins with a result step, in which the value γ of e is returned as a result to the sender of the method. The resumption is obtained by applying the continuation h to the value **t** of the conditional answer expression and the state $\widehat{\sigma'}$ in which the temporary variables have been restored to their original values.

- New-expression:

$$\mathscr{M}^d_E[\![\mathsf{new}(C)]\!](\alpha)(h)(\sigma) = [C, \lambda\beta.h(\beta)(\sigma)]$$

  The meaning of a new-expression is represented by a creation step, which consists of the class name C of the object to be created and a resumption that depends on the name β of the resulting object.

- Identity test:

$$\begin{aligned}&\mathscr{M}^d_E[\![e_1 \equiv e_2]\!](\alpha)(h)\\ &\quad = \mathscr{M}^d_E[\![e_1]\!](\alpha)\Big(\lambda\beta_1.\mathscr{M}^d_E[\![e_2]\!](\alpha)\Big(\lambda\beta_2.\text{if } \beta_1 = \beta_2 \text{ then } h(\mathbf{t}) \text{ else } h(\mathbf{f})\Big)\Big)\end{aligned}$$

  Here the expressions e_1 and e_2 are evaluated (in that order) and if they result in identical object names, **t** is returned; otherwise **f** is returned.

- Statement before expression:

$$\mathscr{M}^d_E[\![s;e]\!](\alpha)(h) = \mathscr{M}^d_S[\![s]\!](\alpha)\Big(\mathscr{M}^d_E[\![e]\!](\alpha)(h)\Big)$$

- The expression self:

$$\mathscr{M}^d_E[\![\mathsf{self}]\!](\alpha)(h) = h(\alpha)$$

- Standard object:

$$\mathscr{M}^d_E[\![\phi]\!](\alpha)(h) = h(\phi)$$

Statements:

- Assignment to instance variable:

$$\mathscr{M}^d_S[\![x \leftarrow e]\!](\alpha)(c) = \mathscr{M}^d_E[\![e]\!](\alpha)\Big(\lambda\beta.\lambda\sigma.[\sigma', c(\sigma')]\Big)$$

  where $\sigma' = [\sigma_{(1)}\{\beta/x\}, \sigma_{(2)}]$.
  The last action to be taken in an assignment statement is an internal step in which the state is modified: The variable x is given the value β, which is the result of the expression e. The resumption is the result of applying the continuation c to the new state σ'.

- Assignment to temporary variable:

$$\mathscr{M}^d_S[\![u \leftarrow e]\!](\alpha)(c) = \mathscr{M}^d_E[\![e]\!](\alpha)\Big(\lambda\beta.\lambda\sigma.[\sigma'', c(\sigma'')]\Big)$$

  where $\sigma'' = [\sigma_{(1)}, \sigma_{(2)}\{\beta/u\}]$.

- Answer statement:

$$\mathcal{M}^d_S[\![\mathsf{answer}\{m_1,\ldots,m_n\}]\!](\alpha)(c)(\sigma) = g$$

where

$$\begin{array}{rcl} g(m)(\langle\beta_1,\ldots,\beta_k\rangle) & = & \begin{cases} \left[\check{\sigma}, \mathcal{M}^d_E[\![e]\!](\alpha)\left(\lambda\gamma.\lambda\sigma'.[\gamma, c(\widehat{\sigma'})]\right)(\check{\sigma})\right] \\ \qquad\qquad \text{if } m \in \{m_1,\ldots,m_n\} \\ \text{undefined} \quad \text{otherwise} \end{cases} \\ \check{\sigma} & = & [\sigma_{(1)}, (\lambda u.nil)\{\beta_i/u_i\}_{i=1}^k] \\ \widehat{\sigma'} & = & [\sigma'_{(1)}, \sigma_{(2)}] \end{array}$$

and $m \Leftarrow [\langle u_1,\ldots,u_k\rangle, e]$ occurs in the class definition d.
Here an answer step is performed. It is described by a function g that is defined only on the method names $m_1,\ldots,m_n$ that are mentioned in the answer statement. When given such a method name and a sequence of argument values, the function yields a process that first changes the state, thereby introducing a new set of temporary variables, evaluates the expression e in the method definition, and finally performs a result step, in which the value γ of the expression e is returned and the resumption consists of the continuation c applied to the state $\widehat{\sigma'}$, in which the original values of the temporary variables have been restored.

- Expression as statement:

$$\mathcal{M}^d_S[\![e]\!](\alpha)(c) = \mathcal{M}^d_E[\![e]\!](\alpha)(\lambda\beta.c)$$

Here we fill in a continuation $\lambda\beta.c$ that simply ignores the value β of the expression.

- Sequential composition:

$$\mathcal{M}^d_S[\![s_1;s_2]\!](\alpha)(c) = \mathcal{M}^d_S[\![s_1]\!](\alpha)\left(\mathcal{M}^d_S[\![s_2]\!](\alpha)(c)\right)$$

- Conditional statement:

$$\begin{array}{l} \mathcal{M}^d_S[\![\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}]\!](\alpha)(c) \\ \quad = \mathcal{M}^d_E[\![e]\!](\alpha)\left(\lambda\beta.\text{if } \beta = \mathbf{t} \text{ then } \mathcal{M}^d_S[\![s_1]\!](\alpha)(c) \text{ else } \mathcal{M}^d_S[\![s_2]\!](\alpha)(c)\right) \end{array}$$

- While loop:

$$\begin{array}{l} \mathcal{M}^d_S[\![\mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}]\!](\alpha)(c) \\ \quad = \mathcal{M}^d_E[\![e]\!](\alpha)\Big(\lambda\beta.\lambda\sigma. \\ \qquad\qquad \Big[\sigma,\ \text{if } \beta = \mathbf{t} \\ \qquad\qquad\quad \text{then } \mathcal{M}^d_S[\![s]\!](\alpha)\left(\mathcal{M}^d_S[\![\mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}]\!](\alpha)(c)\right)(\sigma) \\ \qquad\qquad\quad \text{else } c(\sigma)\Big]\Big) \end{array}$$

As in Section 2.2, induction on the syntactic complexity of expressions and statements is not enough to justify the above definition of $\mathcal{M}^d_E$ and $\mathcal{M}^d_S$. This time the while statement is not the only offending case: in the clauses for method calls, conditional answer expressions, and answer statements an expression is evaluated that comes from a method definition and therefore need not be smaller than the original statement/expression. Again we can define a higher-order contracting function Φ in such a way that the pair $[\mathcal{M}_E^d, \mathcal{M}_S^d]$ is its unique fixed point. Note that the 'extra' internal computation steps that have been introduced precisely in the four above-mentioned cases are necessary to make sure that this function Φ is indeed contracting. One could also consider these internal steps as representing the overhead of the corresponding language construct.

3.4. Semantics of POOL Objects

The domain $(q \in) OProc$ of *object processes* is defined analogously to that of statement processes except for the fact that object processes do not contain internal computation steps. It is given by

$$\begin{aligned} OProc \cong\ & \{q_0\} \cup (New \times (NewName \to OProc)) \\ & \cup (Send \times (Result \to OProc)) \\ & \cup (MName \overset{fin}{\to} (Obj^* \to OProc)) \\ & \cup (MName \overset{fin}{\to} (Obj^* \to OProc)) \times OProc \\ & \cup (Result \times OProc) \end{aligned}$$

The semantics of an object is obtained by applying an abstraction operator $abstr : SProc \to OProc$ to the semantics of the body of this object. This operator $abstr$ is characterized by the following equations:

$$\begin{array}{lcl} abstr(p_0) & = & q_0 \\ abstr([\sigma, p]) & = & abstr(p) \\ abstr([C, f]) & = & [C, \lambda\beta.abstr(f(\beta))] \\ abstr([(\beta, m, \bar{\beta}), f]) & = & [(\beta, m, \bar{\beta}), \lambda\gamma.abstr(f(\gamma))] \\ abstr(g) & = & \lambda m.\lambda\bar{\beta}.abstr(g(m)(\bar{\beta})) \\ abstr([g, p]) & = & [\lambda m.\lambda\bar{\beta}.abstr(g(m)(\bar{\beta})), abstr(p)] \\ abstr([\gamma, p]) & = & [\gamma, abstr(p)] \\ abstr([\sigma_1, [\sigma_2, [\sigma_3, \cdots]]]) & = & q_0 \end{array}$$

(The last clause is needed because the previous clauses do not define the value of *abstr* for infinite sequences of internal steps.) As in Section 2.3, a unique (non-continuous) operator satisfying these equations can be obtained as the fixed point of a higher-order contraction.

Now we can define the semantics of objects, or rather of class definitions, by giving a meaning function $\mathcal{M}_O : ClassDef \to AObj \to OProc$. This function $\mathcal{M}_O$ is defined by

$$\mathcal{M}_O[\![d]\!](\alpha) = abstr\left(\mathcal{M}_S^d[\![s]\!](\alpha)(c_0)(\sigma_0)\right)$$

where

$$\begin{array}{lcl} d & = & [\langle\ldots\rangle, s] \\ c_0 & = & \lambda\sigma.p_0 \\ \sigma_0 & = & [\lambda x.nil, \lambda u.nil] \end{array}$$

3.5. Semantics of POOL Programs

So far we have only described the behaviour of objects in isolation. Next we want to see how several objects in parallel behave and interact. The object processes that describe the individual objects do not describe how to select a message to be answered, how to return a result to the sender, or how to create a new object. Therefore, a mechanism is needed that takes care of this. Such a mechanism is implemented by the operator ω defined below, which translates an object process into a global process. Such a global process can describe one or more objects running in parallel.

First we introduce the domain $(r \in)GProc$ of *global processes*, determined by the following domain equation:

$$\begin{array}{rcl} GProc & = & \{r_0\} \cup \mathcal{P}_{cl}(GStep) \\ (\pi \in)GStep & = & Send \times Obj \times GProc \\ & \cup & Obj \times (Result \to GProc) \\ & \cup & MName \times Obj \times (Obj \to Obj^* \to GProc) \\ & \cup & Obj \times Result \times Obj \times GProc \\ & \cup & Comm^+ \times GProc \end{array}$$

where

$$\begin{array}{rcl} (c \in)Comm^+ & = & Comm \cup \{*\} \\ Comm & = & Obj \times MName^+ \times Obj^* \times Obj \\ MName^+ & = & MName \cup \{*\} \end{array}$$

Again the terminated process is indicated by r_0. Otherwise a global process r is a set of possible steps, among which a choice is made nondeterministically during execution (we see here that an object in itself is deterministic, but a collection of objects running in parallel is not). The reason that in this domain equation we use the constructor $\mathcal{P}_{cl}$ (delivering a power set consisting of all the *closed* subsets of its argument set) instead of $\mathcal{P}_{co}$ (using only *compact* subsets) is that below we want to define a process that describes the behaviour of all the standard objects. In turns out to be impossible to describe an infinite number of integers with a compact process.

The steps resemble the various possibilities that we had for statement and object processes but there are important differences. One of these is the fact that a global step always contains the names of all objects involved. This is necessary because a global process can describe more than one object. Let us review all the possibilities:

1. A *send step* $[(\beta, m, \bar{\beta}), \alpha, r]$ indicates that the object α sends a message to the object β, mentioning the method name m and the sequence $\bar{\beta}$ of argument values. After that, execution continues with the resumption r. Note that this step does not describe directly what should happen when the result of the message arrives. This is done by a separate receive step:
2. A *receive step* $[\beta, f]$ indicates that the object β is waiting for the result of a message. When this result, let us call it γ, arrives, the object β will continue by executing the process $f(\gamma)$. The reason for separating the send and receive steps here is that a global process, unlike an object process, can in general perform an arbitrary number of actions between sending of a message and receiving the result, because it can describe a collection of objects running in parallel.
3. An *answer step* $[m, \alpha, g]$ indicates that the object α is willing to answer a message mentioning the method name m. If this step is performed, the function g is applied to the name of the sender object and to the sequence of arguments to yield the resumption process. Since a global process can consist of more

than one step, we can describe an answer statement by a set of several of these answer steps, so that the individual answer steps are simpler than the ones in statement and object processes.

4. A *result step* $[\beta, \gamma, \alpha, r]$ indicates that the object α wants to return the value γ to the object β as a result of a message that β might have sent to α before. The process r is the resumption of this step.
5. A *completed step* $[c, r]$ indicates a step that the process can take without communication with other processes. It may either indicate an internal step within one of the objects described by the global process, in which case c simply has the value $*$. Alternatively, such a step can indicate a complete communication that takes place between two objects both described by the present global process. A communication c of the form $[\beta, m, \bar{\beta}, \alpha]$ indicates that object α sends a message to object β, requesting execution of method m with arguments $\bar{\beta}$. A communication of the form $[\alpha, *, \bar{\beta}, \beta]$ indicates that β returns $\bar{\beta}$ to α as the result of a message (in this case $\bar{\beta}$ is always a singleton $\langle\gamma\rangle$). In all cases, r is the resumption of this step.

The operator $\|: GProc \times GProc \to GProc$ for parallel composition is defined as follows:

$$\begin{array}{rcl}
r \parallel r_0 & = & r_0 \parallel r = r \\
r_1 \parallel r_2 & = & \{\pi \lfloor\!\lfloor r_2 : \pi \in r_1\} \cup \{\pi \lfloor\!\lfloor r_1 : \pi \in r_2\} \\
& \cup & \bigcup\{\pi_1 \mid \pi_2 : \pi_1 \in r_1, \pi_2 \in r_2 \text{ or } \pi_1 \in r_2, \pi_2 \in r_1\} \\
[(\beta, m, \bar{\beta}), \alpha, r_1] \lfloor\!\lfloor r_2 & = & [(\beta, m, \bar{\beta}), \alpha, r_1 \parallel r_2] \\
[\beta, f] \lfloor\!\lfloor r & = & [\beta, \lambda\gamma . f(\gamma) \parallel r] \\
[m, \alpha, g] \lfloor\!\lfloor r & = & [m, \alpha, \lambda\beta . \lambda\bar{\beta} . g(\beta)(\bar{\beta}) \parallel r] \\
[\beta, \gamma, \alpha, r_1] \lfloor\!\lfloor r_2 & = & [\beta, \gamma, \alpha, r_1 \parallel r_2] \\
[c, r_1] \lfloor\!\lfloor r_2 & = & [c, r_1 \parallel r_2] \\
\pi_1 \mid \pi_2 & = & \{[(\beta, m, \bar{\beta}, \alpha), r_1 \parallel g(\alpha)(\bar{\beta})] : \\
& & \quad \pi_1 = [(\beta, m, \bar{\beta}), \alpha, r_1], \pi_2 = [m, \beta, g]\} \\
& \cup & \{[(\beta, *, \langle\gamma\rangle, \alpha), r_1 \parallel f(\gamma)] : \\
& & \quad \pi_1 = [\beta, \gamma, \alpha, r_1], \pi_2 = [\beta, f]\}
\end{array}$$

(Here r_1 and r_2 are supposed to be unequal to r_0.)

Now for each program P we introduce a translation operator

$$\omega^P : OProc \to Obj \to RetStack \to CrCount \to GProc,$$

where

$$\begin{array}{rcl}
(\rho \in) RetStack & = & Obj^* \\
(n \in) CrCount & = & \mathbf{N}
\end{array}$$

The operator ω^P translates an object process q to a global process r when given the name α of the object that executes the object process, a return stack ρ, and a creation counter n. Here the return stack remembers the names of the objects that are waiting for a result to be returned by the current object (note that answer statements can occur in methods, so that nested rendezvous are possible). The creation counter remembers how many objects have already been created by the current object, so that the next one can be given a unique name. In general, when the program P is clear, we shall just write ω for ω^P. Our operator ω is then defined by the following clauses:

- Terminated process:

$$\omega(q_0)(\alpha)(\rho)(n) = r_0$$

- Creation step:

$$\begin{aligned}&\omega([C,f])(\alpha)(\rho)(n)\\ &\quad = \{[*, \omega(f(\beta))(\alpha)(\rho)(n+1) \parallel \omega(\mathcal{M}_O[\![d]\!](\beta))(\beta)(\langle\rangle)(0)]\}\end{aligned}$$

where $\beta = \alpha \cdot \langle n \rangle$ and $C \Leftarrow d$ occurs in the program P.
An object of class C is to be created, so we construct a new name β for it, look up the corresponding class definition d in the program, and thus we get an object process $\mathcal{M}_O[\![d]\!](\beta)$ representing its execution. After translating this into a global process, using an empty return stack and a creation counter of zero, it is put in parallel with the resumption $f(\beta)$ of its creator, again translated into a global process, incrementing the creation counter.

- Send step:

$$\omega([(\beta, m, \bar{\beta}), f])(\alpha)(\rho)(n) = \{[(\beta, m, \bar{\beta}), \alpha, r]\}$$

where

$$r = \{[\alpha, \lambda\gamma.\omega(f(\gamma))(\alpha)(\rho)(n)]\}.$$

The resumption r of the resulting send step consists of a receive step that is obtained by applying the function f to the result γ of the message and translating the resulting object process again into a global process.

- Answer step:

$$\begin{aligned}&\omega(g)(\alpha)(\rho)(n)\\ &\quad = \{[m, \alpha, \lambda\beta.\lambda\bar{\beta}.\omega(g(m)(\bar{\beta}))(\alpha)(\rho \cdot \langle\beta\rangle)(n)] \; : m \in \mathrm{dom}\,(g)\}.\end{aligned}$$

An answer step in an object process is translated into a set of answer steps in the global process, one for each method that can be answered ($m \in \mathrm{dom}\,(g)$). For each answer step in the set, the resumption is obtained by applying the original resumption g to the method name m and the argument list $\bar{\beta}$ and translating the resulting object process into a global process, using a return stack to which the sender β of the message is appended.

- Conditional answer step:

$$\begin{aligned}&\omega([g,q])(\alpha)(\rho)(n)\\ &\quad = \{[m, \alpha, \lambda\beta.\lambda\bar{\beta}.\omega(g(m)(\bar{\beta}))(\alpha)(\rho \cdot \langle\beta\rangle)(n)] \; : m \in \mathrm{dom}\,(g)\}\\ &\qquad \cup \{[*, \omega(q)(\alpha)(\rho)(n)]\}.\end{aligned}$$

Also in this case a set of answer steps is generated, but here there is an additional completed step, which can be taken even if no message is present.

- Result step:

$$\omega([\gamma, q])(\alpha)(\rho)(n) = \{[\beta, \gamma, \alpha, \omega(q)(\alpha)(\rho')(n)] \; : \rho = \rho' \cdot \langle\beta\rangle\}.$$

The result γ is sent to the destination β, which is taken from the return stack ρ; the resumption is translated using the shorter return stack ρ'.

As in Section 2.4, the outside world is represented by objects, but here we need only one object, since we can distinguish between input and output by using different method names. So let *world* be a special element in *AObj* and let input, output $\in$ *MName*. Now we define a function $q_{world} : SObj^{\infty} \to OProc$

that gives us for any (finite or infinite) sequence w of input values (which are standard objects) a process $q_{world}(w)$, which always starts with an answer step, so that $q_{world}(w) \in MName \xrightarrow{fin} (Obj^* \to OProc)$:

$$q_{world}(\langle\rangle)(m) = \begin{cases} \lambda\bar{\beta}.[world, q_{world}(\langle\rangle)] & \text{if } m = \mathsf{output} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$q_{world}(\phi \cdot w)(m) = \begin{cases} \lambda\bar{\beta}.[\phi, q_{world}(w)] & \text{if } m = \mathsf{input} \\ \lambda\bar{\beta}.[world, q_{world}(\phi \cdot w)] & \text{if } m = \mathsf{output} \\ \text{undefined} & \text{otherwise} \end{cases}$$

(This function q_{world} can again be obtained as the unique fixed point of a suitable higher-order operator.) For a non-empty w, the process $q_{world}(w)$ is willing to answer either an input message, in which case it returns the first element of w and continues with the rest of the elements, or an output message, to which it replies with the name of the world process itself and continues with w unchanged. In both cases the actual argument values of the messages are ignored, but we shall see later how the output values are recovered.

We shall also define processes that deal with messages sent to standard objects. Messages sent to *nil* are never answered, so we do not need any process for this. The Boolean **t** can be modelled by an object process $q_{\mathbf{t}}$ defined by

$$\begin{aligned} q_{\mathbf{t}} &= g_{\mathbf{t}} : MName \xrightarrow{fin} (Obj^* \to OProc) \\ g_{\mathbf{t}}(\mathsf{and}) &= \lambda\beta.\begin{cases} [\mathbf{t}, q_{\mathbf{t}}] & \text{if } \bar{\beta} = \langle\mathbf{t}\rangle \\ [\mathbf{f}, q_{\mathbf{t}}] & \text{if } \bar{\beta} = \langle\mathbf{f}\rangle \\ q_0 & \text{otherwise} \end{cases} \\ g_{\mathbf{t}}(\mathsf{or}) &= \lambda\beta.\begin{cases} [\mathbf{t}, q_{\mathbf{t}}] & \text{if } \bar{\beta} = \langle\mathbf{t}\rangle \text{ or } \bar{\beta} = \langle\mathbf{f}\rangle \\ q_0 & \text{otherwise} \end{cases} \\ g_{\mathbf{t}}(\mathsf{not}) &= \lambda\beta.\begin{cases} [\mathbf{f}, q_{\mathbf{t}}] & \text{if } \bar{\beta} = \langle\rangle \\ q_0 & \text{otherwise} \end{cases} \\ & g_{\mathbf{t}} \text{ is undefined if } m \notin \{\mathsf{and}, \mathsf{or}, \mathsf{not}\} \end{aligned}$$

An object process $q_{\mathbf{f}}$ modelling the Boolean **f** can be defined analogously. Now the global process r_{Bool} modelling all Booleans is given by

$$r_{\mathsf{Bool}} = \omega(q_{\mathbf{t}})(\mathbf{t})(\langle\rangle)(0) \parallel \omega(q_{\mathbf{f}})(\mathbf{f})(\langle\rangle)(0).$$

In modelling the integers we run into a complication: It is not difficult to define for each integer k an object process q_k that models k's behaviour, but composing this infinite number of processes in parallel is difficult, since

$$\lim_{n\to\infty} \omega(q_{-n})(-n)(\langle\rangle)(0) \parallel \cdots \parallel \omega(q_n)(n)(\langle\rangle)(0)$$

does not exist. To overcome this problem, we define 'by hand' a process r_{Int} that performs exactly the steps that we would expect intuitively from the above limit:

$$r_{\mathsf{Int}} = \{[m, n, g_{m,n}] : n \in \mathbf{Z} \wedge m = \mathsf{add}, \ldots\}$$

where

$$g_{\mathsf{add},n}(\alpha)(\bar{\beta}) = \{[\alpha, n+k, n, r_{\mathsf{Int}}] : k \in \mathbf{Z} \wedge \bar{\beta} = \langle k\rangle\}.$$

The process r_{Int} consists of an infinite number of answer steps, one for each integer n and for each method name m applicable for integers (here we have written only the method add). The resumption $g_{m,n}$ of such a step, when applied to a sending object α and an argument list $\bar{\beta}$, yields a process that immediately does a result step, where the resulting value ($n+k$ in the case of the method add) is returned to the sender α. Note that no step is generated if the method is called with an erroneous argument list; in this case the process becomes blocked. (As a mathematical detail, note that r_{Int} is certainly a closed set, because all its elements have a fixed minimum distance to each other. But since this set is infinite, it is certainly not compact.)

Now we can give the semantics of programs by the function $\mathcal{M}_G : Prog \to SObj^\infty \to GProc$, defined by

$$\begin{aligned}&\mathcal{M}_G[\![P]\!](w)\\&\quad = \omega^P(\mathcal{M}_O[\![d_n]\!](\alpha))(\langle\rangle)(0) \parallel \omega(q_{world})(world)(\langle\rangle)(0) \parallel r_{\mathsf{Bool}} \parallel r_{\mathsf{Int}}\end{aligned}$$

where

$$\begin{aligned}P &= \langle C_1 \Leftarrow d_1, \ldots, C_n \Leftarrow d_n\rangle\\ \alpha &= \langle\rangle\end{aligned}$$

Finally we define the operators needed to extract the observable behaviour from a global process. The operator $path : GProc \to \mathcal{P}\Big((Comm^+ \times GProc)^\infty\Big)$ extracts all the possible computation paths out of a process:

$$\begin{aligned}path(r) \;=\; &\Big\{ \langle[c_1,r_1],\ldots,[c_n,r_n]\rangle :\\ &\qquad [c_1,r_1] \in r \wedge \forall 1 \le i < n\, [c_{i+1},r_{i+1}] \in r_i\\ &\qquad \wedge (r_n = r_0 \vee r_n \cap Comm^+ \times GProc = \emptyset) \Big\}\\ \cup\; &\Big\{ \langle[c_1,r_1],\ldots\rangle :\\ &\qquad [c_1,r_1] \in r \wedge \forall i \ge 1\, [c_{i+1},r_{i+1}] \in r_i \Big\}\end{aligned}$$

Next we have the operator $output : GProc \to \mathcal{P}(SObj^\infty)$ defined by

$$output(r) = \{\, \mathcal{V}(c_1) \cdot \mathcal{V}(c_2) \cdot \cdots : \langle[c_i,r_i]\rangle_i \in path(r) \,\}$$

where

$$\mathcal{V}(c) = \begin{cases} \langle v\rangle & \text{if } c = [\alpha, \mathsf{output}, \langle v\rangle, world] \text{ and } v \in SObj\\ \langle\rangle & \text{otherwise}\end{cases}$$

At last we can define the observable behaviour of a program by the function $obs : Prog \to SObj^\infty \to \mathcal{P}(SObj^\infty)$, which returns the set of all possible sequences of output values for a given sequence of input values:

$$obs[\![P]\!](w) = output(\mathcal{M}_G[\![P]\!](w)).$$

4. Conclusions

In the preceding sections we have given a layered denotational semantics for the languages Toy and POOL, where 'layered' means that the semantics is defined at three different levels: for statements, objects, and programs. For each of these levels we have defined a specialized domain of processes and we have defined operators that translate between these domains. In both languages we allow programs to interact with the outside world by communicating with special objects. In this way we can define the overall observable behaviour of a program by specifying the set of possible sequences of output values for a given sequence of input values. However, the most important contribution of this work is that it provides an explicit model of the behaviour of a single object in isolation.

There are several questions still to be answered. It might be interesting to see whether this new semantics for POOL can in some sense be related to the operational and denotational semantics developed previously [ABK86, ABK89]. Despite the fact that these operational and denotational semantics use completely different formalisms, they have been proved to be equivalent to each other. Although this proof is quite complex [Rut90], their precise relationship can be described relatively easily by an operator that extracts all possible paths from a tree-like structure (very much like our operator *path* in Sections 2.4 and 3.5). This is only possible because the two semantics can be fine-tuned to each other, so that the operational semantics performs a step precisely when the denotational does so. With the present layered semantics such fine-tuning is clearly impossible, particularly because the abstraction operator that translates statement processes into object processes deletes all the internal computation steps. Establishing a precise relationship between the layered semantics and the older two is therefore a challenge that calls for the development of new semantic techniques.

Another open question is the issue of full abstractness. At the level of programs we have defined a clear notion of observable behaviour by the operator *obs*, which can serve as a gauge for defining the notion of full abstractness. Note that this notion itself now makes sense for the semantics at the statement level $\mathcal{M}_S$ as well as at the object level $\mathcal{M}_O$ (at the program level the semantics given by *obs* is trivially fully abstract; the semantics $\mathcal{M}_G$ is certainly not fully abstract and it was not intended to be). Intuitively, we have the impression that our semantics for Toy might well be fully abstract at the statement level and at the object level. Proving this, however, is another matter. For the statement level semantics of POOL, the question is completely open, but the object level is certainly not fully abstract: It is possible that the object creates another object that remains completely invisible to the rest of the system, but nevertheless a creation step will appear in its semantics. At this moment it is not at all clear how this problem could be solved. For our investigation on full abstractness we propose to tackle the issue for the Toy language first and then to concentrate on POOL again.

Acknowledgements

We wish to thank the following people for their valuable contributions in many discussions about the contents of this paper: Jaco de Bakker, Ton Kalker, Joost Kok, and Frank van der Linden.

References

[ABK86] Pierre America, Jaco de Bakker, Joost N. Kok, and Jan Rutten. Operational semantics of a parallel object-oriented language. In *Conference Record of the 13th Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 13–15, 1986, 194–208.

[ABK89] Pierre America, Jaco de Bakker, Joost N. Kok, and Jan Rutten. Denotational semantics of a parallel object-oriented language. *Information and Computation*, 83(2):152–205, November 1989.

[Ame87] Pierre America. POOL-T: A parallel object-oriented language. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, 199–220. MIT Press, 1987.

[Ame89a] Pierre America. A behavioural approach to subtyping in object-oriented programming languages. In *Workshop on Inheritance Hierarchies in Knowledge Representation and Programming Languages*, Viareggio, Italy, February 6–8, 1989. Wiley. Also appeared in *Philips Journal of Research*, 44(2/3):365–383, July 1989.

[Ame89b] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[Ame89c] Pierre America. The practical importance of formal semantics. In J. W. Klop, J.-J. Ch. Meyer, and J. J. M. M. Rutten, editors, *J. W. de Bakker, 25 Jaar Semantiek, Liber Amicorum*, 31–40. Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, April 1989.

[Ame91] Pierre America. Designing an Object-Oriented Programming Language with Behavioural Subtyping. In J.W. de Bakker, W.P. de Roever and G. Rozenberg, editors, *Proceedings REX/FOOL Workshop on the Foundations of Object-Oriented Languages*, 60–90. Springer LNCS 489, 1991.

[AmR89] Pierre America and Jan Rutten. Solving reflexive domain equations in a category of complete metric spaces. *Journal of Computer and System Sciences*, 39(3):343–375, December 1989.

[BaZ82] J. W. de Bakker and J. I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54:70–120, 1982.

[Dug66] J. Dugundji. *Topology*. Allyn and Bacon, Boston, Massachusetts, 1966.

[Eng89] R. Engelking. *General Topology*. Revised and completed version. Sigma Series in Pure Mathematics, Vol. 6, Heldermann, Berlin, 1989.

[Gor79] Michael J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction*. Springer, 1979.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Mic51] E. Michael. Topologies on spaces of subsets. *Transactions of the AMS*, 71:152–182, 1951.

[MLa71] Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics, Vol. 5, Springer, 1971.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Aarhus University, Computer Science Department, Aarhus, Denmark, September 1981.

[Rut90] Jan Rutten. Semantic correctness for a parallel object-oriented language. *SIAM Journal on Computing*, 19(3):341–383, 1990.

[Vaa86] Frits W. Vaandrager. Process algebra semantics for POOL. In J.C.M. Baeten, editor, *Applications of process algebra*, 173–236. Cambridge Tracts in Theoretical Computer Science 17, Cambridge University Press, 1990.

Appendix A. Mathematical Preliminaries

As mathematical domains for our semantics we use complete metric spaces satisfying a so-called *reflexive domain equation* of the following form:

$$P \cong F(P)$$

(The symbol $\cong$ is defined below; it says that there is a bijection from P to $F(P)$ that respects the metric defined on the spaces.) Here $F(P)$ is an expression built from P and a number of standard constructions on metric spaces (also to be formally introduced shortly). A few examples are

$$P \cong A \cup (B \times P) \tag{A.1}$$

$$P \cong A \cup \mathcal{P}_{co}(B \times P) \tag{A.2}$$

$$P \cong A \cup (B \to P) \tag{A.3}$$

where A and B are given fixed complete metric spaces. De Bakker and Zucker have first described how to solve these equations in a metric setting [BaZ82]. Roughly, their approach amounts to the following: In order to solve $P \cong F(P)$ they define a sequence of complete metric spaces $(P_n)_n$ by: $P_0 = A$ and $P_{n+1} = F(P_n)$, for $n > 0$, such that $P_0 \subseteq P_1 \subseteq \cdots$. Then they take the *metric completion* of the union of these spaces P_n, say $\bar{P}$, and show: $\bar{P} \cong F(\bar{P})$. In this way they are able to solve equations (A.1), (A.2) and (A.3) above.

There is one type of equation for which this approach does not work, namely,

$$P \quad \cong \quad A \cup (P \xrightarrow{1} G(P)) \tag{A.4}$$

in which P occurs at the *left* side of a function space arrow, and $G(P)$ is an expression possibly containing P. This is due to the fact that it is not always the case that $P_n \subseteq F(P_n)$.

In [AmR89] the above approach is generalized in order to overcome this problem. The family of complete metric spaces is made into a *category* $\mathscr{C}$ by providing some additional structure. (For an extensive introduction to category theory we refer the reader to [MLa71].) Then the expression F is interpreted as a *functor* $F : \mathscr{C} \to \mathscr{C}$ which is (in a sense) *contracting*. It is proved that a generalized version of Banach's theorem (see below) holds, i.e., that contracting functors have a fixed point (up to isometry). Such a fixed point, satisfying $P \cong F(P)$, is a solution of the domain equation.

We shall now give a quick overview of these results, omitting many details and all proofs. For a full treatment we refer the reader to [AmR89]. We start by listing the basic definitions and facts of metric topology that we shall need.

We assume the following notions to be known (the reader might consult [Dug66] or [Eng89]): metric space, ultra-metric space, complete (ultra-)metric space, continuous function, closed set, compact set. In our definition the distance between two elements of a metric space is always between 0 and 1, inclusive.

An arbitrary set A can be supplied with a metric d_A, called the *discrete* metric, defined by

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

Now (A, d_A) is a metric space (it is even an ultra-metric space).

Let (M_1, d_1) and (M_2, d_2) be two complete metric spaces. A function $f : M_1 \to M_2$ is called *non-expansive* if for all $x, y \in M_1$

$$d_2(f(x), f(y)) \leq d_1(x, y)$$

The set of all non-expansive functions from M_1 to M_2 is denoted by $M_1 \xrightarrow{1} M_2$. A function $f : M_1 \to M_2$ is called *contracting* (or a *contraction*) if there exists an $\epsilon < 1$ such that for all $x, y \in M_1$

$$d_2(f(x), f(y)) \leq \epsilon \cdot d_1(x, y)$$

(Non-expansive functions and contractions are always continuous.)

The following fact is known as Banach's theorem: Let (M, d) be a complete metric space and $f : M \to M$ a contraction. Then f has a unique fixed point, that is, there exists a unique $x \in M$ such that $f(x) = x$. This x can be obtained by taking the limit of $f^n(x_0)$ for any arbitrary $x_0 \in M$ (where $f^0(y) = y$ and $f^{n+1}(y) = f(f^n(y))$).

We call M_1 and M_2 *isometric* (notation: $M_1 \cong M_2$) if there exists a bijective mapping $f : M_1 \to M_2$ such that for all $x, y \in M_1$

$$d_2(f(x), f(y)) = d_1(x, y)$$

Definition A.1
Let (M, d), $(M_1, d_1), \ldots, (M_n, d_n)$ be metric spaces.

1. We define a metric d_F on the set $M_1 \to M_2$ of all functions from M_1 to M_2 as follows: For every $f_1, f_2 \in M_1 \to M_2$ we put

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

   This supremum always exists since the values taken by our metrics are always between 0 and 1. The set $M_1 \xrightarrow{1} M_2$ is a subset of $M_1 \to M_2$, and a metric on $M_1 \xrightarrow{1} M_2$ can be obtained by taking the restriction of the corresponding d_F.
2. With $M_1 \bar{\cup} \cdots \bar{\cup} M_n$ 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 $M_1 \bar{\cup} \cdots \bar{\cup} M_n$ as follows: For every $x, y \in M_1 \bar{\cup} \cdots \bar{\cup} M_n$,

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

   If no confusion is possible we often write $\cup$ rather than $\bar{\cup}$.
3. We define a metric d_P on the Cartesian product $M_1 \times \cdots \times M_n$ by the following clause: For every $(x_1, \ldots, x_n), (y_1, \ldots, y_n) \in M_1 \times \cdots \times M_n$,

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

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

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

   where $d(x, Z) = \inf_{z \in Z}\{d(x, z)\}$ for every $Z \subseteq M$, $x \in M$. (We use the convention that $\sup \emptyset = 0$ and $\inf \emptyset = 1$.) The spaces $\mathscr{P}_{co}(M) = \{X : X \subseteq M \wedge X \text{ is compact}\}$ and $\mathscr{P}_{nc}(M) = \{X : X \subseteq M \wedge X \text{ is non-empty and compact}\}$ are supplied with a metric by taking the restriction of d_H.
5. For any real number ϵ with $0 < \epsilon \le 1$ we define

$$\mathrm{id}_\epsilon((M, d)) = (M, d')$$

   where $d'(x, y) = \epsilon \cdot d(x, y)$, for every x and y in M.

Proposition A.2
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.1 and suppose that (M, d), $(M_1, d_1), \ldots, (M_n, d_n)$ are complete. Then

$$(M_1 \to M_2, d_F) \quad (M_1 \xrightarrow{1} M_2, d_F) \tag{a}$$
$$(M_1 \bar{\cup} \cdots \bar{\cup} M_n, d_U) \tag{b}$$
$$(M_1 \times \cdots \times M_n, d_P) \tag{c}$$
$$(\mathscr{P}_{cl}(M), d_H) \quad (\mathscr{P}_{co}(M), d_H) \quad (\mathscr{P}_{nc}(M), d_H) \tag{d}$$
$$\mathrm{id}_\epsilon((M, d)) \tag{e}$$

are complete metric spaces. If (M,d) and (M_i,d_i) are all ultra-metric spaces, then so are these composed spaces. (Strictly speaking, for the completeness of $M_1 \to M_2$ and $M_1 \xrightarrow{1} M_2$ we do not need the completeness of M_1. The same holds for the ultra-metric property.)

Whenever in the sequel we write $M_1 \to M_2$, $M_1 \xrightarrow{1} M_2$, $M_1 \bar{\cup} \cdots \bar{\cup} M_n$, $M_1 \times \cdots \times M_n$, $\mathcal{P}_{cl}(M)$, $\mathcal{P}_{co}(M)$, $\mathcal{P}_{nc}(M)$, or $\mathrm{id}_\epsilon(M)$, we mean the metric space with the metric defined above.

The proofs of Proposition A.2(a), (b), (c), and (e) are straightforward. Part (d) is more complex. It can be proved with the help of the following characterization of the completeness of $(\mathcal{P}_{cl}(M), d_H)$.

Proposition A.3
Let $(\mathcal{P}_{cl}(M), d_H)$ be as in Definition A.1. Let $(X_i)_i$ be a Cauchy sequence in $\mathcal{P}_{cl}(M)$. We have

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

Proofs of Propositions A.2(d) and A.3 can be found in, for instance, [Dug66] and [Eng89]. The proofs are also repeated in [BaZ82]. The completeness of $\mathcal{P}_{co}(M)$ is proved in [Mic51].

We proceed by introducing a category of complete metric spaces and some basic definitions, after which a categorical fixed point theorem will be formulated.

Definition A.4
Let $\mathcal{C}$ denote the category that has complete metric spaces for its objects. The arrows ι in $\mathcal{C}$ are defined as follows: Let M_1, M_2 be complete metric spaces. Then $M_1 \to^{\iota} M_2$ denotes a pair of maps $M_1 \rightleftarrows^{i}_{j} M_2$, satisfying the following properties:

1. i is an isometric embedding,
2. j is non-expansive,
3. $j \circ i = \mathrm{id}_{M_1}$.

(We sometimes write $[i,j]$ for ι.) Composition of the arrows is defined in the obvious way.

We can consider M_1 as an approximation to M_2: In a sense, the set M_2 contains more information than M_1, because M_1 can be isometrically embedded into M_2. Elements in M_2 are approximated by elements in M_1. For an element $m_2 \in M_2$ its (best) approximation in M_1 is given by $j(m_2)$. Clause 3. states that M_2 is a consistent extension of M_1.

Definition A.5
For every arrow $M_1 \to^{\iota} M_2$ in $\mathcal{C}$ with $\iota = [i,j]$ we define

$$\delta(\iota) = d_{M_2 \to M_1}(i \circ j, \mathrm{id}_{M_2}) \quad (= \sup_{m_2 \in M_2} \{d_{M_2}(i \circ j(m_2), m_2)\})$$

This number can be regarded as a measure of the quality with which M_2 is approximated by M_1: the smaller $\delta(\iota)$, the better M_2 is approximated by M_1.

Increasing sequences of metric spaces are generalized as follows:

Definition A.6

1. We call a sequence $(D_n, \iota_n)_n$ of complete metric spaces and arrows a *tower* whenever we have that $\forall n \in \mathbf{N}\, D_n \to^{\iota_n} D_{n+1} \in \mathcal{C}$.

2. The sequence $(D_n, \iota_n)_n$ is called a *converging tower* when the following condition is also satisfied:

   $$\forall \epsilon > 0 \exists N \in \mathbf{N} \forall m > n \geq N \, \delta(\iota_{nm}) < \epsilon$$

   where $\iota_{nm} = \iota_{m-1} \circ \cdots \circ \iota_n : D_n \to D_m$.

A special case of a converging tower is a tower $(D_n, \iota_n)_n$ satisfying, for some ϵ with $0 \leq \epsilon < 1$,

$$\forall n \in \mathbf{N} \, \delta(\iota_{n+1}) \leq \epsilon \cdot \delta(\iota_n)$$

Note that

$$\begin{aligned} \delta(\iota_{nm}) &\leq \delta(\iota_n) + \cdots + \delta(\iota_{m-1}) \\ &\leq \epsilon^n \cdot \delta(\iota_0) + \cdots + \epsilon^{m-1} \cdot \delta(\iota_0) \\ &\leq \frac{\epsilon^n}{1-\epsilon} \cdot \delta(\iota_0) \end{aligned}$$

We shall now generalize the technique of forming the metric *completion* of the union of an increasing sequence of metric spaces by proving that, in $\mathscr{C}$, every converging tower has an *initial cone.* The construction of such an initial cone for a given tower is called the *direct limit* construction. Before we treat this direct limit construction, we first give the definition of a cone and an initial cone.

Definition A.7
Let $(D_n, \iota_n)_n$ be a tower. Let D be a complete metric space and $(\gamma_n)_n$ a sequence of arrows. We call $(D, (\gamma_n)_n)$ a *cone* for $(D_n, \iota_n)_n$ whenever the following condition holds:

$$\forall n \in \mathbf{N} \, D_n \xrightarrow{\gamma_n} D \in \mathscr{C} \wedge \gamma_n = \gamma_{n+1} \circ \iota_n$$

Definition A.8
A cone $(D, (\gamma_n)_n)$ for a tower $(D_n, \iota_n)_n$ is called *initial* whenever for every other cone $(D', (\gamma'_n)_n)$ for $(D_n, \iota_n)_n$ there exists a unique arrow $\iota : D \to D'$ in $\mathscr{C}$ such that:

$$\forall n \in \mathbf{N} \, \iota \circ \gamma_n = \gamma'_n$$

Definition A.9
Let $(D_n, \iota_n)_n$, with $\iota_n = [i_n, j_n]$, be a converging tower. The *direct limit* of $(D_n, \iota_n)_n$ is a cone $(D, (\gamma_n)_n)$, with $\gamma_n = [g_n, h_n]$, that is defined as follows:

$$D = \{ (x_n)_n : \forall n \geq 0 \, x_n \in D_n \wedge j_n(x_{n+1}) = x_n \}$$

is equipped with a metric d_D defined by

$$d_D((x_n)_n, (y_n)_n) = \sup\{d_{D_n}(x_n, y_n)\}$$

for all $(x_n)_n$ and $(y_n)_n \in D$. The mapping $g_n : D_n \to D$ is defined by $g_n(x) = (x_k)_k$, where

$$x_k = \begin{cases} j_{kn}(x) & \text{if } k < n \\ x & \text{if } k = n \\ i_{nk}(x) & \text{if } k > n \end{cases}$$

and $h_n : D \to D_n$ is defined by $h_n((x_k)_k) = x_n$.

Lemma A.10
The direct limit of a converging tower (as defined in Definition A.9) is an initial cone for that tower.

As a category-theoretic equivalent of a contracting function on a metric space, we have the following notion of a *contracting functor* on $\mathscr{C}$.

Definition A.11
We call a functor $F : \mathscr{C} \to \mathscr{C}$ contracting whenever the following holds: There exists an ϵ, with $0 \leq \epsilon < 1$, such that, for all $D \to^{\iota} E \in \mathscr{C}$,

$$\delta(F(\iota)) \leq \epsilon \cdot \delta(\iota)$$

A contracting function on a complete metric space is continuous, so it preserves Cauchy sequences and their limits. Similarly, a contracting functor preserves converging towers and their initial cones:

Lemma A.12
Let $F : \mathscr{C} \to \mathscr{C}$ be a contracting functor, let $(D_n, \iota_n)_n$ be a converging tower with an initial cone $(D, (\gamma_n)_n)$. Then $(F(D_n), F(\iota_n))_n$ is again a converging tower with $(F(D), (F(\gamma_n))_n)$ as an initial cone.

Theorem A.13
Let F be a contracting functor $F : \mathscr{C} \to \mathscr{C}$ and let $D_0 \to^{\iota_0} F(D_0) \in \mathscr{C}$. Let the tower $(D_n, \iota_n)_n$ be defined by $D_{n+1} = F(D_n)$ and $\iota_{n+1} = F(\iota_n)$ for all $n \geq 0$. This tower is converging, so it has a direct limit $(D, (\gamma_n)_n)$. We have $D \cong F(D)$.

In [AmR89] it is shown that contracting functors that are moreover contracting on all *hom-sets* (the sets of arrows in $\mathscr{C}$ between any two given complete metric spaces) have *unique* fixed points (up to isometry). It is also possible to impose certain restrictions upon the category $\mathscr{C}$ such that every contracting functor on $\mathscr{C}$ has a unique fixed point.

Let us now indicate how this theorem can be used to solve Equations (A.1) to (A.4) above. We define

$$F_1(P) = A \cup \mathrm{id}_{1/2}(B \times P) \tag{A.5}$$
$$F_2(P) = A \cup \mathscr{P}_{co}(B \times \mathrm{id}_{1/2}(P)) \tag{A.6}$$
$$F_3(P) = A \cup (B \to \mathrm{id}_{1/2}(P)) \tag{A.7}$$

If the expression $G(P)$ in Equation (A.4) is, for example, equal to P, then we define F_4 by

$$F_4(P) = A \cup \mathrm{id}_{1/2}(P \xrightarrow{1} P) \tag{A.8}$$

Note that the definitions of these functors specify, for each metric space (P, d_P), the metric on $F(P)$ *implicitly* (see Definition A.1).

Now it is easily verified that F_1, F_2, F_3, and F_4 are contracting functors on $\mathscr{C}$. Intuitively, this is a consequence of the fact that in the definitions above each occurrence of P is preceded by a factor $\mathrm{id}_{1/2}$. Thus these functors have a fixed point, according to Theorem A.13, which is a solution for the corresponding equation. (We often omit the factor $\mathrm{id}_{1/2}$ in the reflexive domain equations, assuming that the reader will be able to fill in the details.)

In [AmR89] it is shown that functors like F_1 to F_4 are also contracting on hom-sets, which guarantees that they have *unique* fixed points (up to isometry).

The results above hold for complete *ultra-metric* spaces too, which can be easily verified.

Received February 1991
Accepted in revised form May 1991 by B.T. Denvir