INFORMATION AND COMPUTATION **83**, 152–205 (1989)

Denotational Semantics of a Parallel Object-Oriented Language*

PIERRE AMERICA

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

JACO DE BAKKER, JOOST N. KOK, AND JAN RUTTEN

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

A denotational model is presented for the language POOL, a parallel object-oriented language. It is a syntactically simplified version of POOL-T, a language that is actually used to write programs for a parallel machine. The most important aspect of this language is that it describes a system as a collection of communicating objects that all have internal activities which are executed in parallel. To describe the semantics of this language we construct a mathematical domain of *processes*. This domain is obtained as a solution of a reflexive domain equation over a category of complete metric spaces. A new technique is developed to solve a wide class of such equations, including function space constructions. The desired domain is obtained as the fixed point of a contracting functor implicit in the equation. The domain is sufficiently rich to allow a fully compositional definition of the language constructs in POOL, including concepts such as object creation and method invocation by messages. The semantic equations give a meaning to each syntactic construct depending on the POOL object executing the construct, the environment constituted by the declarations, and a continuation, representing the actions to be performed after the execution of the current construct. After the process representing the execution of an entire program is constructed, a yield function can extract the set of possible execution sequences from it. A preliminary discussion is provided on how to deal with fairness. Full mathematical details are supplied, with the exception of the general domain construction, which is described elsewhere.

1. INTRODUCTION

In this paper we give a formal semantics of a language called POOL (parallel object-oriented language). It is a syntactically simplified version of the language POOL-T, which is defined in (America, 1985) and for which (America, 1986, 1987) give an account of the design considerations.

* This work was carried out in the context of ESPRIT Project 415: Parallel Architectures and Languages for AIP—a VLSI-directed approach.

0890-5401/89 $3.00

POOL-T was designed in subproject A of ESPRIT project 415 with the purpose of programming a highly parallel machine which is also being developed in this project (see Odijk, 1987 for an overview). The language provides all the facilities needed to program reasonably large parallel systems and many small and several large applications have been written in it.

The language POOL for which we shall give a formal semantics is described in detail in Section 3. In this language, a system is viewed as a collection of *objects*. These are dynamic entities containing *data* (stored in *variables*) and *methods* (kinds of procedures). Objects can be created dynamically during the execution of a program and each of them has an internal activity (its *body*) in which it can execute expressions and statements. While inside an object everything proceeds sequentially, the concurrent execution of the bodies of all the objects can give rise to a large amount of parallelism. Objects can interact by sending *messages* to each other. Acceptance of a message gives rise to a rendezvous between sender and receiver, during which an appropriate method is executed.

The relationship between POOL (as described in Section 3) and POOL-T is such that there is a straightforward translation from valid POOL-T programs to valid POOL programs. This translation merely performs some syntactic simplifications and it omits some context information (POOL-T is a statically typed language, POOL is not). At no point does this translation replace any semantic primitive by another one. The sole reason for using two languages and translating between them is that POOL-T is a practical programming language, where readability, among others, is much more important than syntactic simplicity. In order not to overload the present paper, we shall not describe POOL-T and the above translation, but take as a starting point the language POOL as described in Section 3.

After having defined an operational semantics for POOL in (America *et al.*, 1986), in this paper we set out to develop a *denotational* semantics. In general, denotational semantics assigns to every construct in the language a *meaning*, which is a value from a suitably chosen mathematical domain. The most important principle in denotational semantics is *compositionality*: The meaning of a composite construct is determined solely on the basis of the *meanings* of its components, which means that the actual form of these components is irrelevant.

An important choice we have made is to use the mathematical framework of *complete metric spaces* for our semantic description. In this we follow and generalize the approach of (De Bakker and Zucker, 1982). (For other applications of this type of semantic framework see De Bakker *et al.*, 1986.) First, we construct a suitable domain P of *processes*, which is a set of mathematical objects that will be used as meanings. It will satisfy a reflexive domain equation, which will be solved by deriving from it a

functor on a certain category of complete metric spaces and then constructing a fixed point for this functor. The mathematical techniques to do this are sketched in Section 2 and presented in detail in (America and Rutten, 1988). They are not necessary for an understanding of the rest of the paper.

After having constructed the domain P, we want to define a mapping from the set of POOL programs (also called *units*) to P. Before we assign a semantic value to the unit as a whole, we first define the semantics of statements and expressions. This semantics will be given by functions of the type:

$$[\![\cdots]\!]_E : Exp \to Env \to Obj \to Cont_E \to P$$

$$[\![\cdots]\!]_S : Stat \to Env \to Obj \to Cont_S \to P,$$

where

$$Cont_E = Obj \to P,$$

$$Cont_S = P.$$

We give the formal description of the type of these semantic functions here because we want to stress three of their characteristics: the use of *environments*, *objects*, and *continuations*.

The environments (elements of the set *Env*) are used to store the meanings of declarations (of classes and methods). With the help of $[\![\cdots]\!]_E$ and $[\![\cdots]\!]_S$ we can define for each unit U a suitable environment γ_U, which contains the meanings of the classes and methods as they are defined in U. It will be constructed as the unique fixed point of a contracting operator on the complete metric space of environments. The semantic domain *Obj* stands for the set of object names. Its appearance in the defining equations reflects the fact that in POOL each expression or statement is evaluated *by a certain object*. Finally, a continuation will be given as an argument to the semantic functions. This describes what will happen *after* the execution of the current expression or statement. As the continuation of an expression generally depends upon the result of this expression (an object name), its type is $Obj \to P$, whereas the type of continuations of statements is simply P. This use of continuations makes it possible to define the semantics, especially of object creation, in a convenient and concise way. (For more examples of the use of continuations in semantics, see (De Bruin, 1986) and (Gordon, 1979).)

The denotational semantics proper for POOL is presented in Section 4. It first discusses the details of the process domain P. Next, it defines an auxiliary operator for parallel composition, which is used, e.g., in the equation for the creation of a new object. (POOL itself does not have a syntactic operator for parallel execution: parallelism occurs implicitly as a consequence of object creation.) Then the core of the semantic definitions, in terms of the various semantic equations for the respective classes of expres-

sions and statements, is displayed. Once the reader has understood (or taken for granted) the underlying mathematical foundations he will appreciate, we hope, that the framework allows a concise, rigorous, and compositional (the touchstone of a denotational model) definition of intricate notions such as the creation of a new object or the passing of messages leading to the invocation of the appropriate method. Section 4 then continues with the discussion of the standard process p_{ST}, which describes the standard objects (integers, booleans, and *nil*) of the language. Next, the definition of the environment γ_U corresponding to a unit U is given and used to define a process p_U. In a last step we show how the set of all possible sequences of computation steps can be obtained from the process resulting from the parallel composition of p_U and p_{ST}.

In Section 5 the semantic model is adapted to provide the possibility to formulate requirements that distinguish between *fair* and *unfair* executions of the program. The ideas in this section are not in their final form and will probably be developed further in subsequent work. Section 6 presents some conclusions and gives some directions for further research.

As related work concerning the semantics of POOL, we first refer to (America *et al.*, 1986), where we describe the semantics in an *operational* way, using a transition system in the style of (Hennessy and Plotkin, 1979). In (Vaandrager, 1986), the semantics of the language is described by translating it into process algebra and using the several kinds of semantics that had already been developed for the latter (see, e.g., Bergstra and Klop, 1984). In order to do this, some extra process algebra operators are introduced. The advantage of this approach is that it uses an existing framework which admits algebraic calculations with meanings of programs, and furthermore that it can deal with fairness in a natural way. However, due to the extra translation step, the meaning of an individual construct is quite hard to understand.

Semantic treatments of parallel object-oriented languages in general are scarce; we only know (Clinger, 1981), which gives a detailed mathematical model for an actor language. This is done by defining a set of so-called augmented actor event diagrams, each of which is a fairly complicated structure representing (the beginning of) a single computation. In order to deal with nondeterminism, a novel power domain construction is used. This technique deals very well with fairness, but the event diagrams seem a rather ad hoc construction.

As to the material in Section 2, there is a vast amount of literature on order-theoretic domain theory (see, for instance, (Gierz *et al.*, 1980)). Our approach, in which a category of metric spaces and (generalizations of) Banach's theorem are central, may be an attractive alternative that can be used in a situation where the contractivity of the various functions encountered is a natural phenomenon.

2. Metric Spaces and Domain Equations

In this section we first collect some definitions and properties concerning metric spaces. Then we show how the well-known direct limit construction can be used as a means to produce a solution of a recursive domain equation in a category of complete metric spaces.

It is not absolutely necessary to read this section in order to understand the rest of this paper. It mainly gives a mathematical justification for the constructions used in Sections 4 and 5.

2.1. *Metric Spaces*

Definition 2.1 (Metric space). A *metric space* is a pair (M, d) with M a non-empty set and d a mapping $d: M \times M \to [0, 1]$ (a *metric* or *distance*), which satisfies the properties:

(a) $\forall x, y \in M[d(x, y) = 0 \Leftrightarrow x = y]$

(b) $\forall x, y \in M[d(x, y) = d(y, x)]$

(c) $\forall x, y, z \in M[d(x, y) \leqslant d(x, z) + d(z, y)]$.

We call (M, d) an *ultra-metric space* if the following stronger version of property (c) is satisfied:

(c′) $\forall x, y, z \in M[d(x, y) \leqslant \max\{d(x, z), d(z, y)\}]$.

Remark. In our definition the distance between two elements of a metric space is always bounded by 1.

Example. Let A be an arbitrary set. The *discrete* metric d_A on A is defined as follows: Let $x, y \in A$, then

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

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

Definition 2.2. Let (M, d) be a metric space, let $(x_i)_i$ be a sequence in M.

(a) We say that $(x_i)_i$ is a *Cauchy sequence* whenever we have:

$$\forall \varepsilon > 0 \exists N \in \mathbb{N} \forall n, m > N\ [d(x_n, x_m) < \varepsilon].$$

(b) Let $x \in M$. We say that $(x_i)_i$ *converges to* x (denoted by $x = \lim_{i \to \infty} x_i$) and call x the *limit* of $(x_i)_i$ whenever we have

$$\forall \varepsilon > 0\ \exists N \in \mathbb{N}\ \forall n > N\ [d(x, x_n) < \varepsilon].$$

Such a sequence we call *convergent.*

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

(d) A subset X of a complete metric space (M, d) is called *closed* whenever each Cauchy sequence in X converges to an element of X.

DEFINITION 2.3. Let (M_1, d_1), (M_2, d_2) be metric spaces.

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

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

(c) Let $\varepsilon \geqslant 0$. With $M_1 \to^{\varepsilon} M_2$ we denote the set of functions f from M_1 to M_2, that satisfy the following property: $\forall x, y \in M_1[d_2(f(x), f(y)) \leqslant \varepsilon \cdot d_1(x, y)]$. Functions f in $M_1 \to^1 M_2$ we call *non-distance-increasing* (NDI), functions f in $M_1 \to^{\varepsilon} M_2$ with $0 \leqslant \varepsilon < 1$, we call *contracting.*

PROPOSITION 2.4. *Let (M_1, d_1), (M_2, d_2) be metric spaces. For every $\varepsilon \geqslant 0$ and $f \in M_1 \to^{\varepsilon} M_2$ we have: f is continuous.*

THEOREM 2.5 (Banach's fixed point theorem). *Let (M, d) be a complete metric space and $f: M \to M$ a contracting function. Then there exists an $x \in M$ such that the following holds:*

(1) $f(x) = x$ *(x is a fixed point of f),*

(2) $\forall y \in M\ [f(y) = y \Rightarrow y = x]$ *(x is unique),*

(3) $\forall x_0 \in M[\lim_{n \to \infty} f^n(x_0) = x]$ *where* $f^{n+1}(x_0) = f(f^n(x_0))$ *and* $f^0(x_0) = x_0$.

Remark. This theorem will be the main mathematical tool that we shall use: Contracting functions and their unique fixed points play an important role throughout this paper.

DEFINITION 2.6. Let (M, d), (M_1, d_1),..., (M_n, d_n) be metric spaces.

(a) With $M_1 \to M_2$ we denote the set of all functions from M_1 to M_2. We define a metric d_F on $M_1 \to M_2$ as follows: For every $f_1, f_2 \in M_1 \to M_2$ 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 codomain of our metrics is always $[0, 1]$. For $\varepsilon \geqslant 0$ the set $M_1 \to^{\varepsilon} M_2$ is a subset of $M_1 \to M_2$, and a metric

on $M_1 \to^{\varepsilon} M_2$ can be obtained by taking the restriction of the corresponding d_F.

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

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

If no confusion is possible we shall often write $\cup$ rather than $\cup$.

(c) We define a metric d_p on the Cartesian product $M_1 \times \cdots \times M_n$ by the clause: For every $(x_1, ..., x_n), (y_1, ..., y_n) \in M_1 \times \cdots \times M_n$,

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

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

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

where $d(x, Z) = \inf_{z \in Z} \{d(x, z)\}$ for every $Z \subseteq M$, $x \in M$. (We use the convention that $\sup \varnothing = 0$ and $\inf \varnothing = 1$.)

(e) For any real number ε with $\varepsilon \in [0, 1]$ we define

$$id_\varepsilon((M, d)) = (M, d'),$$

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

PROPOSITION 2.7. *Let* (M, d), (M_1, d_1),..., (M_n, d_n), d_F, d_U, d_P, *and* d_H *be as in Definition* 2.6 *and suppose that* (M, d), (M_1, d_1),..., (M_n, d_n) *are complete. We have that*

(a) $(M_1 \to M_2, d_F)$, $(M_1 \to^{\varepsilon} M_2, d_F)$,

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

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

(d) $(\mathscr{P}_{\mathrm{cl}}(M), d_H)$,

(e) $id_\varepsilon((M, d))$,

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* $M_1 \to^{\varepsilon} M_2$ *we do not need the completeness of* M_1. *The same holds for the ultra-metric property.)*

Whenever in the sequel we write $M_1 \to M_2$, $M_1 \to^{\varepsilon} M_2$, $M_1 \cup \cdots \cup M_n$, $M_1 \times \cdots \times M_n$, $\mathscr{P}_{\mathrm{cl}}(M)$, or $id_\varepsilon(M)$, we mean the metric space with the

metric defined above. The proofs of Proposition 2.7(a), (b), (c), and (e) are straightforward. Part (d) is more involved. It can be proved with the help of the following characterization of the completeness of $(\mathscr{P}_{\mathrm{cl}}(M), d_H)$.

PROPOSITION 2.8. *Let* $(\mathscr{P}_{\mathrm{cl}}(M), d_H)$ *be as in Definition* 2.6. *Let* $(X_i)_i$ *be a Cauchy sequence in* $\mathscr{P}_{\mathrm{cl}}(M)$. *We have*

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

Proofs of Propositions 2.7(d) and 2.8 can be found in (for instance) (Dugundji, 1966; Engelking, 1977). Proposition 2.8 is due to Hahn (1948). The proofs are also repeated in (De Bakker and Zucker, 1982).

2.2. *Solving Reflexive Domain Equations*

As a mathematical domain for our denotational semantics we shall use a complete metric space satisfying a so-called *reflexive domain equation* of the form

$$P \cong F(P).$$

Here $F(P)$ is an expression composed of P and some given fixed spaces by applying one or more of the constructions introduced in Definition 2.6. A few examples are:

$$P \cong A \cup id_{1/2}(B \times P) \tag{1}$$

$$P \cong A \cup \mathscr{P}_{\mathrm{cl}}(B \times id_{1/2}(P)) \tag{2}$$

$$P \cong A \cup (B \to id_{1/2}(P)), \tag{3}$$

where A and B are given fixed complete metric spaces. De Bakker and Zucker (1982) have first described how to solve these equations in a metric setting (see also De Bakker *et al.*, 1986 for many examples).

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 Eqs. (1)–(3) above.

For our denotational semantics we shall have to solve a domain equation of yet another type, namely,

$$P \cong A \cup id_{1/2}(P \to^{!} G(P)), \tag{4}$$

in which P occurs at the *left* side of a function space arrow, and $G(P)$ is an expression possibly containing P. Here, the method of (De Bakker and Zucker, 1982) fails, since, with F as in (4), it is not always the case that $P_n \subseteq F(P_n)$.

In (America and Rutten, 1988) the 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 (Mac Lane, 1971).) 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 holds, i.e., that contracting functors have a unique 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 (America and Rutten, 1988).

DEFINITION 2.9 (Category of complete metric spaces). Let $\mathscr{C}$ denote the category that has complete metric spaces for its objects. The arrows ι in $\mathscr{C}$ 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:

(a) i is an isometric embedding,

(b) j is non-distance-increasing (NDI),

(c) $j \circ i = id_{M_1}$.

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

We can consider M_1 as an approximation of 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 (c) states that M_2 is a consistent extension of M_1.

DEFINITION 2.10. For every arrow $M_1 \to^{\iota} M_2$ in $\mathscr{C}$ with $\iota = \langle i, j \rangle$ we define

$$\delta(\iota) = d_{M_2 \to M_2}(i \circ j, id_{M_2}) \left(= \sup_{m_2 \in M_2} \{ d_{M_2}(i \circ j(m_2), m_2) \}\right).$$

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 in the following

DEFINITION 2.11 (Converging tower). (a) We call a sequence $(D_n, \iota_n)_n$ of complete metric spaces and arrows a *tower* whenever we have that $\forall n \in \mathbb{N}\ [D_n \to^{\iota_n} D_{n+1} \in \mathscr{C}]$.

(b) The sequence $(D_n, \iota_n)_n$ is called a *converging tower* when further-

more the following condition is satisfied: $\forall \varepsilon > 0 \exists N \in \mathbb{N} \forall m > n \geqslant N$ $[\delta(\iota_{nm}) < \varepsilon]$, where $\iota_{nm} = \iota_{m-1} \circ \cdots \circ \iota_n : D_n \to D_m$.

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

$$\forall n \in \mathbb{N}\ [\delta(\iota_{n+1}) \leqslant \varepsilon \cdot \delta(\iota_n)].$$

(Please note that

$$\delta(\iota_{nm}) \leqslant \delta(\iota_n) + \cdots + \delta(\iota_{m-1}) \leqslant \varepsilon^n \cdot \delta(\iota_0) + \cdots + \varepsilon^{m-1} \cdot \delta(\iota_0) \leqslant \frac{\varepsilon^n}{1-\varepsilon} \cdot \delta(\iota_0).)$$

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 2.12 (Cone). 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:

$$\forall n \in \mathbb{N}\ [D_n \to^{\gamma_n} D \in \mathscr{C} \wedge \gamma_n = \gamma_{n+1} \circ \iota_n].$$

DEFINITION 2.13 (Initial cone). 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:

$$\forall n \in \mathbb{N}\ [\iota \circ \gamma_n = \gamma'_n].$$

DEFINITION 2.14 (Direct limit construction). Let $(D_n, \iota_n)_n$, with $\iota_n = \langle i_n, j_n \rangle$, be a converging tower. The *direct limit* of $(D_n, \iota_n)_n$ is a cone $(D, (\gamma_n)_n)$, with $\gamma_n = \langle g_n, h_n \rangle$, that is defined as follows:

$$D = \{(x_n)_n \mid \forall n \geqslant 0 [x_n \in D_n \wedge j_n(x_{n+1}) = x_n]\}$$

is equipped with a metric $d : D \times D \to [0, 1]$ defined by: $d((x_n)_n, (y_n)_n) = \sup\{d_{D_n}(x_n, y_n)\}$, for all $(x_n)_n$ and $(y_n)_n \in D$. $g_n : D_n \to D$ is defined by $g_n(x) = (x_k)_k$, where

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

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

LEMMA 2.15. *The direct limit of a converging tower (as in Definition* 2.14) *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 2.16 (Contracting functor). We call a functor $F: \mathscr{C} \to \mathscr{C}$ contracting whenever the following holds: There exists an ε, with $0 \leqslant \varepsilon < 1$, such that, for all $D \to^{\iota} E \in \mathscr{C}$,

$$\delta(F(\iota)) \leqslant \varepsilon \cdot \delta(\iota).$$

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 2.17. *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 2.18 (Fixed-point theorem). *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 $n \geqslant 0$. This tower is converging, so it has a direct limit $(D, (\gamma_n)_n)$. We have: $D \cong F(D)$.*

Remark. In (America and Rutten, 1988) 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 Eqs. (1)–(4) above. We define

$$F_1(P) = A \cup id_{1/2}(B \times P) \tag{1}$$

$$F_2(P) = A \cup \mathscr{P}_{\mathrm{cl}}(B \times id_{1/2}(P)) \tag{2}$$

$$F_3(P) = A \cup (B \to id_{1/2}(P)). \tag{3}$$

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

$$F_4(P) = A \cup id_{1/2}(P \to^{1} P). \tag{4}$$

(Please note that the definitions of these functors specify, for each metric space (P, d_P), the metric on $F(P)$ *implicitly* (see Definition 2.6).) 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 $id_{1/2}$. Thus these functors have a fixed point, according to Theorem 2.18, which is a solution for the corresponding equation.

Remarks. (1) In (America and Rutten, 1988) it is shown that functors like F_1 through F_4 are also contracting on hom-sets, which guarantees that they have *unique* fixed points (up to isometry).

(2) The results above hold for complete *ultra-metric* spaces too, which can be easily verified. The domain we shall use for our denotational semantics is an ultra-metric space.

3. The Language POOL

3.1. *An Informal Introduction to the Language*

The language POOL makes use of the principles of object-oriented programming in order to give structure to parallel systems. Object-oriented programming (of which the language Smalltalk-80, Goldberg and Robson, 1983, is a representative example) offers a way to structure large systems. Originally it was only used in sequential systems, but it offers excellent possibilities for a very advantageous integration with parallelism. (This was already proposed in Hewitt, 1977, using an approach quite different from ours.)

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 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 for other objects.

An object uses *variables* (more specifically: instance variables) to store its internal data. Each variable can contain the *name* of an object (another object, or, possibly, the object under consideration itself). An assignment to a variable can make it refer to a different object than before. 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 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 object names) 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 (an object name), which is returned to the sender of the message. The latter then resumes its own execution. Note that this form of communication strongly resembles the rendezvous mechanism of Ada (ANSI, 1983).

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 execute in parallel. This is the way parallelism is introduced into the language. Synchronization and communication takes place by sending messages, as described above.

Objects are grouped into *classes*. All objects in one class (the *instances* of that class) use the same names for their variables, they have the same methods for answering messages, and they execute 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 are a few standard classes predefined in the language. In this semantic description we will only incorporate the classes *Boolean* and *Integer*. On these objects the usual operations can be performed, 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.

There is a special standard object, *nil*, which can be considered to be an element of every class. 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*. If a message is sent to this object, an error occurs. In general, whenever a run-time error occurs, the whole system will halt immediately.

At this point it is useful to emphasize the distinction between an object and its name. Objects are intuitive entities as described above. In this paper there will appear no mathematical construction that directly models a single object with all its dynamic properties (although it would be interesting to see a semantics which does this). Object names, on the other hand, are modeled explicitly as elements of some abstract set *Obj*. Object names are only *references* to objects. On its own, an object name gives little information about the object it refers to. In fact, object names are just sufficient to distinguish the individual objects from each other. Note that variables

and parameters contain object names, and that expressions result in object names, not objects. Only for standard objects: integers, booleans, and *nil*, it does not seem to make sense to distinguish between an object and its name. However, even for these objects a separate description of their behaviour is necessary (see Section 4.4). If in the sequel we speak, for example, of "the object α," we hope that the reader understands that the object with name α is meant.

3.2. *Syntax of POOL*

In this section the (abstract) syntax of the language POOL is described. We assume that the following sets of syntactic elements are given:

IVar	(instance variables)	with typical element x,
TVar	(temporary variables)	with typical element u,
CName	(class names)	with typical element C,
MName	(method names)	with typical element m.

We define the set *SObj* of standard objects, with typical element ϕ, by

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

($\mathbb{Z}$ is the set of all integers.) Note that for standard objects, we do not distinguish between object names and the objects themselves.

We now define the set *Exp* of expressions, with typical element e:

$$\begin{aligned} e ::= {} & x \\ & \mid u \\ & \mid e!m(e_1, ..., e_n) \\ & \mid m(e_1, ..., e_n) \\ & \mid \textbf{new}(C) \\ & \mid e_1 \equiv e_2 \\ & \mid s; e \\ & \mid \textbf{self} \\ & \mid \phi \end{aligned}$$

The set *Stat* of statements, with typical elements s, ...:

$$\begin{aligned} s ::= {} & x \leftarrow e \\ & \mid u \leftarrow e \\ & \mid \textbf{answer } V \qquad (V \subseteq MName,\ V \neq \varnothing) \\ & \mid e \\ & \mid s_1 ; s_2 \\ & \mid \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} \\ & \mid \textbf{do } e \textbf{ then } s \textbf{ od} \\ & \mid \textbf{sel } g_1 \textbf{ or } \cdots \textbf{ or } g_n \textbf{ les} \end{aligned}$$

The set *GCom* of guarded commands, with typical elements g, ...:

$$g ::= e \textbf{ answer } V \textbf{ then } s \qquad (V \subseteq MName).$$

(Note that $V = \varnothing$ is allowed.)
The set *Unit* of units, with typical elements U, ...:

$$U ::= \langle C_1 \Leftarrow d_1, ..., C_n \Leftarrow d_n \rangle \qquad (n \geqslant 1).$$

The set *ClassDef* of class definitions, with typical elements d, ...:

$$d ::= \langle (m_1 \Leftarrow \mu_1, ..., m_n \Leftarrow \mu_n), s \rangle$$

And finally the set *MethDef* of method definitions, with typical elements μ, ...:

$$\mu ::= \langle (u_1, ..., u_n), e \rangle.$$

3.2.1. *Informal Explanation*

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.

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 through e_n are the parameters. When a send expression is evaluated, first the destination expression is evaluated, then the parameters are evaluated from left to right and then the message is sent to the destination object. When this object answers the message, the corresponding method is executed, that is, the formal parameters are initialized to the objects names in the message, the 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.

A method call simply means that the corresponding method is executed (after the evaluation of the parameters from left to right). The result of this execution will be the value of the method call.

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

The next type of expression checks whether e_1 and e_2 result in the same object. If so, *tt* is returned, otherwise *ff*.

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 the name of the object that is executing this expression.

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, respectively. 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 V. Then it executes the method (after initializing the formal parameters and temporary variables). The result of the method is sent back to the sender of the message, and the answer statement terminates.

Next it is indicated that 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.

A select statement is executed as follows: First, all the expressions (called guards) in the guarded commands are evaluated from left to right. They must all result in an object of class *Boolean*, otherwise an error occurs and the system is halted immediately. The guarded commands of which the guards have resulted in *ff* are discarded (they do not play a role in the further execution of the statement). Now one of the remaining guarded commands is chosen. For this there are two possibilities: One possibility is that the (textually) *first* guarded command is chosen in which the answer statement contains no method names (if there is such a guarded command). In this case the statement after **then** is executed and the select statement terminates. The second possibility is that a guarded command with a non-empty answer set is chosen. For this the following requirements must be satisfied:

— A message has arrived specifying a method in this answer set.

— This guarded command must be the (textually) *first* one that contains this method in its answer set and for which the guard resulted in *tt*.

— There must be no guarded command with an empty answer set and a true guard occurring before this one.

If this case applies, the above message is answered (by executing the specified method and returning the result), the statement after **then** is executed, and then the select statement terminates.

Guarded commands. These are sufficiently described in the treatment of the select statement.

Units. These are the programs of POOL. A unit consists of a number of bindings of class names to class definitions. If a unit is to be executed, a single new instance of the *last* class defined in the unit is created and execution of its body is started. This object has the task to start the whole system, by creating new objects and putting them to work.

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.

Method definitions. A method definition describes a method. Here u_1 through u_n are the formal parameters and e is the expression to be evaluated when the method is invoked. The set of temporary variables is again implicit: it consists of all the elements of *TVar* that occur in the expression e, with the exception of the formal parameters.

3.2.2. *Context conditions*

For a POOL program to be valid there are a few more conditions 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 unit are different.
- All method names in a class definition are different.
- All parameters in a method definition are different.

Any POOL program that is a translation of a valid POOL-T program will automatically satisfy these conditions. POOL-T is even more restrictive. For example, it requires that the type of every expression conforms with its use, and it forbids assignments to formal parameters. However, the conditions above are sufficient to ensure that the program will have a well-defined semantics.

3.3. *An Example Program*

As an illustration of programs that can be written in POOL, we present an example. In the following program (unit) *U*, a parallel implementation of Eratosthenes' sieve for generating prime numbers is given. An object of the class *Primes* (the "root" object) generates an infinite ascending stream of integers, which it feeds into a chain of instances of the class *Sieve*. Each of those remembers in its variable p the first number it gets (always a prime), and from the rest passes on only those numbers that are not divisible by p. The computation proceeds in a pipelined way:

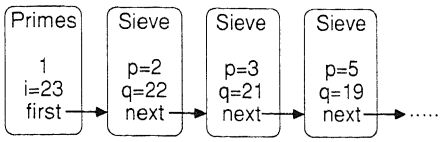

FIGURE 1

$$U = \langle \text{Sieve} \Leftarrow d_{\text{Sieve}}, \text{Primes} \Leftarrow d_{\text{Primes}} \rangle$$

where

$$d_{\text{Sieve}} = \langle (\text{input} \Leftarrow \mu_{\text{input}}, \text{create} \Leftarrow \mu_{\text{create}}), s_{\text{Sieve}} \rangle,$$

with

$$\mu_{\text{input}} = \langle (n), q \leftarrow n; \textbf{self} \rangle,$$

$$\mu_{\text{create}} = \langle (), \textbf{new}(\text{Sieve}) \rangle,$$

$$s_{\text{Sieve}} = \textbf{answer}(\text{input});$$
p←q;
next←create();
do *tt*
then **answer**(input);
if q!mod(p)!equal(0)!not()
then next!input(q)
fi
od,

and

$$d_{\text{Primes}} = \langle (), s_{\text{Primes}} \rangle,$$

with

$$s_{\text{Primes}} = \text{first} \leftarrow \textbf{new}(\text{Sieve});$$
i←2;
do *tt*
then first!input(i); i←i!add(1)
od.

(It is assumed that $\{p, q, \text{next}, i, \text{first}\} \subset IVar$ and $n \in TVar$.)

4. DENOTATIONAL SEMANTICS

This section constitutes the heart of our paper. First, the sets of objects and states are introduced and the mathematical domain P of processes is defined which we use for our denotational semantics. Second, an auxiliary semantic operator for parallel composition is defined, followed by the

definition of environments. Then the semantics of expressions and statements is defined, with the use of the notion of *continuations*, some familiarity with which may be helpful for the reader. (For an extensive treatment of continuations and so-called expression continuations, which we shall also use, we refer to Gordon, 1979.) Next, the semantics for the standard objects (integers and booleans) of POOL is given. The section culminates in the definition of the semantics of a unit (a POOL program). This involves in particular the definition of the environment corresponding to it. Finally, the notions of *paths* and *yield* of a process are introduced.

4.1. *Domain Definitions*

Before we can give the definition of our process domain we have to define the sets of objects and the set of states.

DEFINITION 4.1. (Objects). We assume given a set $AObj$ of names for active objects together with a function

$$\tau: AObj \to CName,$$

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

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

such that $\nu(X, C) \notin X$ and $\tau(\nu(X, C)) = C$, for finite $X \subseteq AObj$ and $C \in CName$. The function ν gives for a finite set X of object names and a class name C a new name of class C, not in X. The set $AObj$ and the set of standard objects $SObj$ together form the set Obj of *object names*, with typical elements α and β:

$$\begin{aligned} Obj &= AObj \cup SObj \\ &= AObj \cup \mathbb{Z} \cup \{tt, ff\} \cup \{nil\}. \end{aligned}$$

Remark. A possible example of such a set $AObj$ and functions τ and ν could be obtained by setting:

$$\begin{aligned} AObj &= CName \times \mathbb{N}, \\ \tau(\langle C, n \rangle) &= C, \\ \nu(X, C) &= \langle C, \max\{n: \langle C, n \rangle \in X\} + 1 \rangle. \end{aligned}$$

DEFINITION 4.2. (States). The set of states Σ, with typical element σ, is defined by

$$\begin{aligned}\Sigma = &(AObj \to IVar \to Obj)\\ &\times (AObj \to TVar \to Obj)\\ &\times \mathscr{P}_{fin}(AObj).\end{aligned}$$

Remarks. (1) We denote the three components of $\sigma \in \Sigma$ by $\sigma = \langle \sigma_1, \sigma_2, \sigma_3 \rangle$.

(2) The first and the second component of a state store the values of the instance variables and the temporary variables of each active object. The third component contains the object names currently in use. We need it in order to give unique names to newly created objects.

In order to give a meaning to expressions, statements, and units we shall define a mathematical domain P, the elements of which we shall from now on call *processes.*

DEFINITION 4.3 (Semantic process domain P). Let P, with typical elements p and q, be a complete ultra-metric space satisfying the following reflexive domain equation:

$$P \cong \{p_0\} \cup id_{1/2}(\Sigma \to \mathscr{P}_{cl}(Step_P)),$$

where $Step_P$, with typical elements π and ρ, is

$$Step_P = (\Sigma \times P) \cup Send_P \cup Answer_P,$$

with

$$Send_P = Obj \times MName \times Obj^* \times (Obj \to P) \times P,$$

$$Answer_P = Obj \times MName \times (Obj^* \to (Obj \to P) \to^1 P).$$

Here Obj^*, with typical elements $\bar{\alpha}$ and $\bar{\beta}$, is the set of finite sequences of object names. (The sets $\{p_0\}$, Σ, Obj, $MName$, and Obj^* become complete ultra-metric spaces by supplying them with the discrete metric (see the example preceding Definition 2.2).)

In Section 2 it is described how to solve such an equation. Let us try to explain intuitively the intended interpretation of the domain P. First, we observe that in the equation above the subexpression $id_{1/2}$ is necessary only to guarantee that the equation is solvable by defining a contracting functor on the category $\mathscr{C}$ (see Section 2). For a, say, more operational understanding of the equation it does not matter.

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

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

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

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

Second, π might be a *send step*, $\pi \in Send_P$. In this case we have, say

$$\pi = \langle \alpha, m, \bar{\beta}, f, p \rangle,$$

with $\alpha \in Obj$, $m \in MName$, $\bar{\beta} \in Obj^*$, $f \in (Obj \to P)$, and $p \in P$. The action involved here consists of the registration of an attempt at communication, in which a message is sent to the object α, specifying the method m, together with the parameters $\bar{\beta}$. This is the interpretation of the first three components α, m, and $\bar{\beta}$. The fourth component f, called the *dependent* resumption of this send step, indicates the steps that will be taken after the sender has received the result of the message. These actions will depend on the result, which is modeled by f being a function that yields a process when it is applied to an object name (the result of the message). The last component p, called the *independent* resumption of this send step, represents the steps to be taken after this send step that need *not* wait for the result of the method execution.

Finally, π might be an element of $Answer_P$, say

$$\pi = \langle \alpha, m, g \rangle$$

with $\alpha \in Obj$, $m \in MName$, and $g \in (Obj^* \to (Obj \to P) \to^1 P)$. It is then called an *answer step*. The first two components of π express that the object α is willing to accept a message that specifies the method m. The last component g, the resumption of this answer step, specifies what should happen when an appropriate message actually arrives. The function g is then applied to the parameters in this message and to the dependent resumption of the sender (specified in its corresponding send step). It then delivers a process which is the resumption of the sender and the receiver *together*,

which is to be composed in parallel with the independent resumption of the send step.

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

DEFINITION 4.4 (Parallel composition). Let

$$\| : P \times P \to P$$

be such that it satisfies the equation

$$p \| q = \lambda\sigma \cdot (\{\pi \,\hat{\|}\, q : \pi \in p(\sigma) \wedge q(\sigma) \neq \varnothing\} \cup \{\pi \,\hat{\|}\, p : \pi \in q(\sigma) \wedge p(\sigma) \neq \varnothing\} \cup \bigcup \{\pi|_\sigma \rho : \pi \in p(\sigma),\ \rho \in q(\sigma)\})$$

for all $p, q \in P \setminus \{p_0\}$, and such that $p_0 \| q = q \| p_0 = p_0$. Here, $\pi \,\hat{\|}\, q$ is defined by

$$\langle \sigma', p' \rangle \,\hat{\|}\, q = \langle \sigma', p' \| q \rangle,$$
$$\langle \alpha, m, \beta, f, p \rangle \,\hat{\|}\, q = \langle \alpha, m, \beta, f, p \| q \rangle,$$
$$\langle \alpha, m, g \rangle \,\hat{\|}\, q = \langle \alpha, m, \lambda\beta \cdot \lambda h \cdot (g(\beta)(h) \| q) \rangle,$$

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

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

Remarks. (1) We observe that this definition is self-referential, since the merge operator occurs at the righthand side of the definition. For a formal justification of this definition see the Appendix (A.1), where the merge operator is given as the unique fixed point of a contraction $\Phi_{PC} : (P \times P \to^1 P) \to (P \times P \to^1 P)$.

(2) Since we intend to model parallel composition by interleaving, the merge of two processes p and q consists of three parts. The first part contains all possible first steps of p followed by the parallel composition of their respective resumptions with q. The second part similarly contains the first steps of q. The last part contains the communication steps that result from two matching communication steps taken simultaneously by processes p and q. For $\pi \in Step_P$ the definition of $\pi \| q$ is straightforward. The definition of $\pi|_\sigma \rho$ is more involved. It is the empty set if π and ρ do not

match. Now suppose they do match, say $\pi = \langle \alpha, m, \vec{\beta}, f, p \rangle$ and $\rho = \langle \alpha, m, g \rangle$. Then π is a *send* step, denoting a request to object α to execute the method m, and ρ is an *answer* step, denoting that the object α is willing to accept a message that requests the execution of the method m. In $\pi|_\sigma \rho$, the state σ remains unaltered. Since g, the third component of ρ, represents the meaning of the execution of the method m, it needs the parameters $\vec{\beta}$ that are specified by α. Moreover, g depends on the dependent resumption f of the send step π. This explains why both $\vec{\beta}$ and f are supplied as arguments to the function g. Now it can be seen that $g(\vec{\beta})(f) \parallel p$ represents the resumption of the sender and the receiver together. In order to get more insight in this definition it is advisable to return to it after having seen the definition of the semantics of an answer statement.

(3) If, for a given state σ, either $p(\sigma)$ or $q(\sigma)$ is empty, then $(p \parallel q)(\sigma)$ is the empty set. Since the empty set is used to model abnormal termination, this can be understood as follows: If abnormal termination occurs in one of the two components of a parallel composition, then the entire composition is considered to terminate abnormally.

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

$$\varepsilon = \sup_{p,q,r \in P} \{d_P((p \parallel q) \parallel r, p \parallel (q \parallel r))\}.$$

Then, using the fact that the operator $\parallel$ satisfies the equation above, one can show that $\varepsilon \leqslant \frac{1}{2}\varepsilon$. Therefore $\varepsilon = 0$, and $\parallel$ is associative.

Next, *environments* are introduced.

DEFINITION 4.5 (Environments). The set of environments is defined as

$$Env = (AObj \to P) \times (MName \to AObj \to Obj^* \to (Obj \to P) \to^1 P).$$

Remarks. (1) We denote the first and the second component of γ by γ_1 and γ_2.

(2) When we are going to compute the semantics of a certain unit U, we shall define an environment γ_U such that it contains all information about the definitions that are present in U. It will be needed in the computation of the semantics of U. The first component γ_1 of an environment γ is a function that, supplied with an object name α, gives the process representing the execution of α's *body*. Note that this body depends on the class of α, which can, however, be determined from the object name by applying the function τ. We shall need this first component when we want to define the semantics of a new-expression.

The second component γ_2 gives the meaning of method executions and is used to define the semantics of an answer statement, a method call, and a select statement. When we supply γ_2 with arguments m and α we get the meaning of the execution of the method m by the object α. It depends on the parameters that are passed to the method, so $\bar{\beta}$ is a third argument. The final argument is the expression continuation f, which, applied to the object resulting from the execution of the method, yields a process that represents the steps to be taken next. The result $\gamma_2(m)(\alpha)(\bar{\beta})(f) \in P$ is a process expressing the meaning of the execution of the method m by the object α with parameters $\bar{\beta}$ and expression continuation f.

4.2. *Semantics of Statements and Expressions*

In this section we define the semantics of statements by specifying a function $[\![\cdots]\!]_S$ of the type

$$[\![\cdots]\!]_S : Stat \to Env \to AObj \to Cont_S \to^1 P,$$

where $Cont_S = P$, the set of *continuations* of statements. Let $s \in Stat$, $\gamma \in Env$, $\alpha \in AObj$, and $p \in P$. The semantic value of s is the process given by

$$[\![s]\!]_S(\gamma)(\alpha)(p).$$

The environment γ contains information about class definitions (needed to evaluate new-expressions) and method definitions (needed to evaluate answer statements, select statements, and method calls). The second parameter of $[\![s]\!]_S$, the object name α, represents the object that executes the statement s. The semantic value of s finally depends on its so-called *continuation*: the semantic value of everything that will happen after the execution of s. The main advantage of the use of continuations is that it enables us to describe the semantics of expressions, in particular the new-expression, in a concise and elegant way. For that purpose, we shall specify a function

$$[\![\cdots]\!]_E : Exp \to Env \to AObj \to Cont_E \to^1 P,$$

where $Cont_E = Obj \to P$, the set of expression continuations. Let $e \in Exp$, $\gamma \in Env$, $\alpha \in AObj$, and $f \in Obj \to P$. The semantic value of e is the process given by

$$[\![e]\!]_E(\gamma)(\alpha)(f).$$

The environment γ, the object α, and the continuation f serve the same purpose as in the semantics of a statement s. However, there is one important difference: the type of the continuation. The evaluation of expressions

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

Remark. Please note the difference between the notions of *resumption* and *continuation.* A resumption is a part of a semantic step $\pi \in Step_P$, indicating the remaining steps to be taken after the current one (see the explanation following Definition 4.3 above). A continuation is one of the arguments that we give to our semantic functions. Such a continuation, when supplied as an argument to $[\![s]\!]_S(\gamma)(\alpha)$, for a statement s, an environment γ, and an object α, indicates the actions that should be taken after the statement s has been executed. It may appear as a resumption in the result. A good example of this is the definition of $[\![x \leftarrow e]\!]_S$ (in Definition 4.7, S1) below.

DEFINITION 4.6 (Semantics of expressions). We define a function

$$[\![\cdots]\!]_E : Exp \to Env \to AObj \to Cont_E \to^! P,$$

where

$$Cont_E = Obj \to P,$$

by the following clauses. Let $\gamma \in Env$, $\alpha \in AObj$, $f \in Obj \to P$.

(E1, instance variable)

$$[\![x]\!]_E(\gamma)(\alpha)(f) = \lambda\sigma \cdot \{\langle \sigma, f(\sigma_1(\alpha)(x))\rangle\}.$$

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

(E2, temporary variable)

$$[\![u]\!]_E(\gamma)(\alpha)(f) = \lambda\sigma \cdot \{\langle \sigma, f(\sigma_2(\alpha)(u))\rangle\}.$$

(E3, send expression)

$$\begin{aligned}
&[\![e!m(e_1, \ldots, e_n)]\!]_E(\gamma)(\alpha)(f) = \\
&[\![e]\!]_E(\gamma)(\alpha)(\\
&\quad \lambda\beta \cdot ([\![e_1]\!]_E(\gamma)(\alpha)(\\
&\qquad \lambda\beta_1 \cdot ([\![e_2]\!]_E(\gamma)(\alpha)(\\
&\qquad\quad \cdots \\
&\qquad\quad \lambda\beta_{n-1} \cdot ([\![e_n]\!]_E(\gamma)(\alpha)(\\
&\qquad\qquad \lambda\beta_n \cdot \lambda\sigma \cdot \{\langle \beta, m, \bar{\beta}, f, p_0\rangle\}))\cdots)))))
\end{aligned}$$

where

$$\bar{\beta} = \langle \beta_1, ..., \beta_n \rangle.$$

The expressions $e, e_1, ..., e_n$ are evaluated from left to right. Their results correspond to the formal parameters $\beta, \beta_1, ..., \beta_n$ of their respective continuations. Finally a send step is performed. The object name β refers to the object to which the message is sent. The sequence $\langle \beta_1, ..., \beta_n \rangle$ represents the parameters for the execution of the method m. Besides these values and the method name m the final step $\langle \beta, m, \bar{\beta}, f, p_0 \rangle$ also contains the expression continuation f of the send expression as the dependent resumption. If the attempt at communication succeeds, this continuation will be supplied with the result of the method execution (see Section 4.1). The independent resumption of this send step is initialized at p_0.

(E4, method call)

$$\begin{aligned}
&[\![m(e_1, ..., e_n)]\!]_E(\gamma)(\alpha)(f) = \\
&[\![e_1]\!]_E(\gamma)(\alpha)(\\
&\lambda\beta_1 \cdot ([\![e_2]\!]_E(\gamma)(\alpha)(\\
&\qquad \cdots \\
&\qquad \lambda\beta_{n-1} \cdot ([\![e_n]\!]_E(\gamma)(\alpha)(\\
&\qquad\qquad \lambda\beta_n \cdot \lambda\sigma \cdot \{\langle \sigma, \gamma_2(m)(\alpha)(\bar{\beta})(f) \rangle\})) \cdots)))
\end{aligned}$$

where

$$\bar{\beta} = \langle \beta_1, ..., \beta_n \rangle.$$

Here the final step is not a communication step. It represents the execution of the method m by the object α with the parameters $\bar{\beta}$ and the continuation f.

(E5, new-expression)

$$[\![\textbf{new}(C)]\!]_E(\gamma)(\alpha)(f) = \lambda\sigma \cdot \{\langle \sigma', \gamma_1(\beta) \parallel f(\beta) \rangle\}$$

where

$$\begin{aligned}
\sigma' &= \langle \sigma_1\{\lambda x \cdot nil/\beta\}, \sigma_2, \sigma_3 \cup \{\beta\} \rangle, \\
\beta &= \nu(\sigma_3, C).
\end{aligned}$$

A new object of class C is created. It is called $\nu(\sigma_3, C)$: the function ν, supplied with the set of all object names currently in use and the class name C as an argument yields a name of class C that is not yet being used. The state σ is changed by initializing the values of the instance variables of the new object to *nil* and by expanding the set σ_3 with the new name β. The

process $\gamma_1(\beta)$, representing the body of the new object, is composed in parallel with the process resulting from the application of the continuation f to β, which is the value of the evaluation of this new expression. We are able to perform this parallel composition because we know from f what should happen after the evaluation of this new-expression, so here the use of continuations is essential.

(E6, identity checking)

$$\begin{aligned}[\![e_1 \equiv e_2]\!]_E(\gamma)(\alpha)(f) = [\![e_1]\!]_E(\gamma)(\alpha)(&\\ \lambda\beta_1 \cdot [\![e_2]\!]_E(\gamma)(\alpha)(&\\ \lambda\beta_2 \cdot \text{if } \beta_1 = \beta_2 &\\ \text{then } f(tt) &\\ \text{else } f(ff) &\\ \text{fi})).&\end{aligned}$$

(E7, sequential composition)

$$[\![s; e]\!]_E(\gamma)(\alpha)(f) = [\![s]\!]_S(\gamma)(\alpha)([\![e]\!]_E(\gamma)(\alpha)(f)).$$

The definition of $[\![\cdots]\!]_S$ is given below in Definition 4.7. Lemma 4.8 states that $[\![\cdots]\!]_E$ and $[\![\cdots]\!]_S$ are well defined, although their definitions refer to each other.

(E8, **self**)

$$[\![\textbf{self}]\!]_E(\gamma)(\alpha)(f) = f(\alpha).$$

The continuation f is supplied with the value of the expression **self**, that is the name of the object executing this expression. We use $f(\alpha)$ instead of $\lambda\sigma \cdot \{\langle \sigma, f(\alpha)\rangle\}$ in this definition, wishing to express that the value of **self** is immediately present: it does not take a step to evaluate it. A similar remark applies to Definition E9:

(E9, standard objects)

$$[\![\phi]\!]_E(\gamma)(\alpha)(f) = f(\phi).$$

DEFINITION 4.7 (Semantics of statements). The function

$$[\![\cdots]\!]_S : Stat \to Env \to AObj \to Cont_S \to^1 P,$$

where

$$Cont_S = P,$$

is defined by the following clauses. Let $\gamma \in Env$, $\alpha \in AObj$, $p \in P$.

(S1, assignment to an instance variable)

$$[\![x \leftarrow e]\!]_S(\gamma)(\alpha)(p) = [\![e]\!]_E(\gamma)(\alpha)(\lambda\beta \cdot \lambda\sigma \cdot \{\langle \sigma', p\rangle\},$$

where

$$\sigma' = \langle \sigma_1\{(\sigma_1(\alpha)\{\beta/x\})/\alpha\}, \sigma_2, \sigma_3\rangle.$$

The expression e is evaluated and the result β is assigned to x.

(S2, assignment to a temporary variable)

$$[\![u \leftarrow e]\!]_S(\gamma)(\alpha)(p) = [\![e]\!]_E(\gamma)(\alpha)(\lambda\beta \cdot \lambda\sigma \cdot \{\langle \sigma', p\rangle\}),$$

where

$$\sigma' = \langle \sigma_1, \sigma_2\{(\sigma_2(\alpha)\{\beta/u\})/\alpha\}, \sigma_3\rangle.$$

(S3, answer statement)

$$[\![\textbf{answer } V]\!]_S(\gamma)(\alpha)(p) = \lambda\sigma \cdot \{\langle \alpha, m, g_m\rangle : m \in V\},$$

where for $m \in V$

$$g_m = \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to P \cdot \gamma_2(m)(\alpha)(\bar{\beta})(\lambda\beta \cdot (f(\beta) \parallel p)).$$

For each method m the function g_m represents its execution followed by its continuation. In the definition of g_m the second component of environment γ is supplied with arguments m and α. This function g_m expects parameters $\bar{\beta}$ and a continuation f, both to be received from an object sending a message specifying the method m. After the execution of the method both the continuation of the sending object and the given continuation p are to be executed in parallel. So the final argument γ_2 is supplied with is

$$\lambda\beta \cdot (f(\beta) \parallel p).$$

Remark. Now that we have defined the semantics of send expressions and answer statements let us briefly return to the definition of $\pi|_\sigma \rho$ (Definition 4.4). Let $\pi = \langle \alpha, m, \bar{\beta}, f, q\rangle$ (the result from the elaboration of a send expression) and $\rho = \langle \alpha, m, g\rangle$ (resulting from an answer statement). Then $\pi|_\sigma \rho$ is defined as

$$\pi|_\sigma \rho = \{\langle \sigma, g(\bar{\beta})(f) \parallel q\rangle\}.$$

We see that the execution of the method m proceeds in parallel with the independent resumption q of the sender. Now that we know how g is defined we have

$$g(\bar{\beta})(f) = \gamma_2(m)(\alpha)(\bar{\beta})(\lambda\beta \cdot (f(\beta) \parallel p)).$$

The continuation of the execution of m is given by $\lambda\beta \cdot (f(\beta) \parallel p)$, the parallel composition of the continuations f and p. This represents the fact

that after the rendezvous, during which the method is executed, the sender and the receiver of the message can proceed in parallel again. (Of course, the independent resumption q may still be executing at this point.) Moreover, the result β of the method execution is passed on to the continuation f of the send expression.

(S4, expressions as statements)

$$[\![e]\!]_S(\gamma)(\alpha)(p) = [\![e]\!]_E(\gamma)(\alpha)(\lambda\beta \cdot p).$$

If an expression occurs as a statement, only its side effects are important. The resulting value is neglected.

(S5, sequential composition)

$$[\![s_1; s_2]\!]_S(\gamma)(\alpha)(p) = [\![s_1]\!]_S(\gamma)(\alpha)([\![s_2]\!](\gamma)(\alpha)(p)).$$

The continuation of s_1 is the execution of s_2 followed by p. We observe that a semantic operator for sequential composition is absent. The use of continuations has made it superfluous.

(S6, conditional)

$$\begin{aligned}
&[\![\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}]\!]_S(\gamma)(\alpha)(p) = \\
&[\![e]\!]_E(\gamma)(\alpha)(\lambda\beta \cdot (\text{if } \beta = tt \\
&\qquad\qquad \text{then } [\![s_1]\!]_S(\gamma)(\alpha)(p) \\
&\qquad\qquad \text{elseif } \beta = ff \\
&\qquad\qquad \text{then } [\![s_2]\!]_S(\gamma)(\alpha)(p) \\
&\qquad\qquad \text{else } \lambda\sigma \cdot \varnothing \\
&\qquad\qquad \text{fi})).
\end{aligned}$$

If $\beta \notin \{tt, ff\}$, then the result is $\lambda\sigma \cdot \varnothing$, indicating abnormal termination due to the occurence of an error.

(S7, loop statement)

$$[\![\textbf{do } e \textbf{ then } s \textbf{ od}]\!]_S(\gamma)(\alpha)(p) = \text{Fixed Point}(\Phi),$$

where $\Phi: P \to P$ is defined by

$$\begin{aligned}
\Phi(q) = [\![e]\!]_E(\gamma)(\alpha)(\lambda\beta \cdot \lambda\sigma \cdot \{\langle \sigma, &\text{ if } \beta = tt \\
&\text{then } [\![s]\!]_S(\gamma)(\alpha)(q) \\
&\text{elseif } \beta = ff \\
&\text{then } p \\
&\text{else } \lambda\sigma \cdot \varnothing \\
&\text{fi}\rangle\}).
\end{aligned}$$

We shall show below (Lemma 4.8(b)) that Φ is contracting.

(S8, select statement)

$$
\begin{aligned}
&[\![\textbf{sel}(e_1 \textbf{ answer } V_1 \textbf{ then } s_1) \textbf{ or } \cdots \textbf{or}(e_n \textbf{ answer } V_n \textbf{ then } s_n) \textbf{ les}]\!]_S(\gamma)(\alpha)(p) = \\
&[\![e_1]\!]_E(\gamma)(\alpha)(\\
&\ \lambda\beta_1 \cdot \text{if } \beta_1 \notin \{tt, ff\} \text{ then } \lambda\sigma \cdot \varnothing \\
&\qquad \text{else } [\![e_2]\!]_E(\gamma)(\alpha)(\\
&\ \cdots \\
&\ \lambda\beta_n \cdot \text{if } \beta_n \notin \{tt, ff\} \text{ then } \lambda\sigma \cdot \varnothing \\
&\qquad \text{else } \lambda\sigma \cdot \\
&\qquad\ (\{\langle \sigma, [\![s_k]\!]_S(\gamma)(\alpha)(p)\rangle : \beta_k = tt \wedge V_k = \varnothing \\
&\qquad\qquad \wedge \forall i < k[\beta_i = tt \Rightarrow V_i \neq \varnothing]\} \\
&\qquad\qquad \cup \{\langle \alpha, m, g_{m,k}\rangle : \beta_k = tt \wedge m \in V_k \\
&\qquad\qquad \wedge \forall i < k[\beta_i = tt \Rightarrow (m \notin V_i \wedge V_i \neq \varnothing)]\}) \\
&\qquad \text{fi} \cdots) \\
&\ \text{fi}),
\end{aligned}
$$

where

$$g_{m,k} = \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to P \cdot \gamma_2(m)(\alpha)(\bar{\beta})(\lambda\beta \cdot (f(\beta) \parallel [\![s_k]\!]_S(\gamma)(\alpha)(p))).$$

The reader is entitled to some explanation. First the guards are evaluated from left to right. If any of them evaluates to something different from *tt* or *ff*, then an error occurs immediately, indicated by $\lambda\sigma \cdot \varnothing$. After the evaluation of the guards we have two sets of possible steps:

The first set is empty or contains a step corresponding with a guarded command that has a true guard and an empty answer set, and for which there does not occur any empty answer set to its left.

The second set contains those steps that result from the selection of a method in one of those guarded commands that have a non-empty answer set V_k. A message specifying the method $m \in V_k$ can be answered if to the left of the kth guarded command there occur no guarded commands with an empty answer set nor with an answer set containing m. This expresses exactly the priority order of the methods as explained in Section 3.2.1. The function $g_{m,k}$ expresses the execution of the method m in the kth guarded command. The only difference with the function g_m used in the definition of the answer statement (S3 above) is that the continuation of the receiving object α (which executes the select statement s) in this case is: $[\![s_k]\!]_S(\gamma)(\alpha)(p)$. It represents the execution of the statement s_k of the kth guarded command, followed by p, the continuation of the entire select statement.

Note that a guarded command for which the guard evaluates to *ff* can never be selected. If *all* guards in the select statement evaluate to *ff*, the result is $\lambda\sigma \cdot \varnothing$, denoting abnormal termination.

LEMMA 4.8. *The semantic functions* $[\![\cdots]\!]_E$ *and* $[\![\cdots]\!]_S$ *of Definitions* 4.6 *and* 4.7 *are well defined*:

(a) *For all* $e \in Exp$, $s \in Stat$, $\gamma \in Env$, $\alpha \in AObj$:

$$[\![e]\!]_E(\gamma)(\alpha) \in (Obj \to P) \to^1 P \qquad \text{and} \qquad [\![s]\!]_S(\gamma)(\alpha) \in P \to^1 P.$$

(b) *The function* Φ: $P \to P$ *used in Definition* 4.7 (S7) *is contracting.*

For the proof see the Appendix (A.2).

4.3. *Standard Objects*

DEFINITION 4.9 (Integers). Let the process p_{INT}, which represents the activity of all integer objects, be such that is satisfies the equation

$$p_{\text{INT}} = \lambda\sigma \cdot (\{\langle n, add, g_n^+ \rangle : n \in \mathbb{Z}\} \cup \{\langle n, sub, g_n^- \rangle : n \in \mathbb{Z}\} \cup \cdots),$$

where

$$g_n^+ = \lambda\beta \in Obj^* \cdot \lambda f \in Obj \to P \cdot$$
$$\text{(if } \beta \in \mathbb{Z} \text{ then } f(n + \beta) \parallel p_{\text{INT}} \text{ else } \lambda\sigma \cdot \varnothing \text{ fi)},$$

$$g_n^- = \lambda\beta \in Obj^* \cdot \lambda f \in Obj \to P \cdot$$
$$\text{(if } \beta \in \mathbb{Z} \text{ then } f(n - \beta) \parallel p_{\text{INT}} \text{ else } \lambda\sigma \cdot \varnothing \text{ fi)},$$

and where the dots stand for similar terms representing the other operations on integers.

Remarks. (1) This definition is self-referential since p_{INT} occurs at the right-hand side of the definition. Formally, p_{INT} can be defined as the fixed point of a suitably defined contraction on P, similar to the definition of the merge operator $\parallel$ as the fixed point of the contraction Φ_{PC} (see A.1 in the Appendix).

(2) We observe that p_{INT} is an infinitely branching process. Such a process fits naturally into our domain. This is the reason why we have chosen $\mathcal{P}_{\text{cl}}(\cdots)$ (closed subsets) in our domain equation rather than $\mathcal{P}_{\text{comp}}(\cdots)$ (compact subsets).

(3) The operational intuition behind the definition of p_{INT} is the following: For every $n \in \mathbb{Z}$ the set $p_{\text{INT}}(\sigma)$ contains, among others, two elements, namely $\langle n, add, g_n^+ \rangle$ and $\langle n, sub, g_n^- \rangle$. These steps indicate that the integer object n is willing to execute its methods *add* and *sub*. If, for

example, by evaluating $n!add(n')$, a certain active object sends a request to integer object n to execute the method *add* with parameter n', then g_n^+, supplied with n' and the continuation f of the active object, is executed. We have that $g_n^+(n')(f)$ is, by definition, the parallel composition of f supplied with the immediate result of the execution of the method *add*, namely $n+n'$, and the process p_{INT}, which remains unaltered: $g_n^+(n')(f) = f(n+n') \parallel p_{\text{INT}}$. If, by mistake, a request for the execution of the method *add* arrives that specifies the wrong type or number of parameters, then $\lambda\sigma \cdot \varnothing$ is the result: the system deadlocks.

DEFINITION 4.10 (Booleans). Let the process p_{BOOL}, which represents the behaviour of the booleans *tt* and *ff*, be such that it satisfies the equation

$$p_{\text{BOOL}} = \lambda\sigma \cdot (\{\langle b, and, g_b^{\wedge}\rangle : b \in \{tt, ff\}\} \cup \{\langle b, \text{or}, g_b^{\vee}\rangle : b \in \{tt, ff\}\} \cup \{\langle b, not, g_b^{\neg}\rangle : b \in \{tt, ff\}\}),$$

where

$$g_b^{\wedge} = \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to P \cdot (\text{if } \bar{\beta} \in \{tt, ff\} \text{ then } f(b \wedge \bar{\beta}) \parallel p_{\text{BOOL}} \text{ else } \lambda\sigma \cdot \varnothing \text{ fi})$$

$$g_b^{\vee} = \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to P \cdot (\text{if } \bar{\beta} \in \{tt, ff\} \text{ then } f(b \vee \bar{\beta}) \parallel p_{\text{BOOL}} \text{ else } \lambda\sigma \cdot \varnothing \text{ fi})$$

$$g_b^{\neg} = \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to P \cdot (\text{if } \bar{\beta} = \langle\,\rangle \text{ then } f(\neg b) \parallel p_{\text{BOOL}} \text{ else } \lambda\sigma \cdot \varnothing \text{ fi})$$

Remark. As with p_{INT}, the definition of p_{BOOL} is self-referential. It can be formally justified along the lines of Remark (1) above. The intuition for this definition is very similar to that of the definition of p_{INT} (see Remark (3) above).

DEFINITION 4.11 (Standard object *nil*). The process p_{NIL}, representing the behaviour of the standard object *nil*, is given by

$$p_{\text{NIL}} = \lambda\sigma \cdot \{\langle nil, m, \lambda\bar{\beta} \cdot \lambda f \cdot \lambda\sigma \cdot \varnothing\rangle : m \in MName\}.$$

Remark. The process p_{NIL}, representing the behaviour of the object *nil*, is willing to execute any method $m \in MName$. The execution of a method consist of immediate (abnormal) termination, indicated by $\lambda\sigma \cdot \varnothing$. In this way, we model that sending messages to *nil* leads to abnormal termination of the entire system.

DEFINITION 4.12 (Standard objects). We define one process for all our standard objects:

$$p_{ST} = p_{INT} \parallel p_{BOOL} \parallel p_{NIL}$$

EXAMPLE. The standard objects are assumed to be present at the execution of every POOL statement s. Therefore the process representing the semantic value of s will be put into parallel with p_{ST}. An example may illustrate how communication with a standard object proceeds. We determine

$$[\![x \leftarrow (2!add(3))]\!]_S(\gamma)(\alpha)(p_0) \parallel p_{ST}$$

for a given $x \in Ivar$, $\gamma \in Env$, and $\alpha \in AObj$. First we compute the semantic value of the assignment:

$$\begin{aligned}
&[\![x \leftarrow (2!add(3))]\!]_S(\gamma)(\alpha)(p_0)\\
&\quad = [\![2!add(3)]\!]_E(\gamma)(\alpha)(f)\\
&\qquad [\text{where } f = \lambda\beta \cdot \lambda\sigma' \cdot \{\langle \sigma'', p_0\rangle\} \text{ with}\\
&\qquad\quad \sigma'' = \langle \sigma_1'\{(\sigma_1'(\alpha)\{\beta/x\})/\alpha\}, \sigma_2', \sigma_3'\rangle]\\
&\quad = [\![2]\!]_E(\gamma)(\alpha)(\lambda\beta_1 \cdot ([\![3]\!]_E(\gamma)(\alpha)(\lambda\beta_2 \cdot \lambda\sigma \cdot \{\langle \beta_1, add, \beta_2, f, p_0\rangle\})))\\
&\quad = [\![3]\!]_E(\gamma)(\alpha)(\lambda\beta_2 \cdot \lambda\sigma \cdot \{\langle 2, add, \beta_2, f, p_0\rangle\})\\
&\quad = \lambda\sigma \cdot \{\langle 2, add, 3, f, p_0\rangle\}.
\end{aligned}$$

Now the parallel composition:

$$\begin{aligned}
&\lambda\sigma \cdot \{\langle 2, add, 3, f, p_0\rangle\} \parallel p_{ST}\\
&\quad = \sigma \cdot \{\langle 2, add, 3, f, p_0\rangle\} \parallel \lambda\sigma' \cdot \{\cdots, \langle 2, add, g_2\rangle, \cdots\} \parallel p_{BOOL} \parallel p_{NIL}\\
&\qquad [\text{where } g_2 = \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to P \cdot\\
&\qquad\quad (\text{if } \bar{\beta} \in \mathbb{Z} \text{ then } f(2 + \bar{\beta}) \parallel p_{INT} \text{ else } \lambda\sigma \cdot \varnothing \text{ fi})]\\
&\quad = \lambda\sigma \cdot \{\langle 2, add, 3, f, p_0\rangle|_\sigma \langle 2, add, g_2\rangle, \cdots\} \parallel p_{BOOL} \parallel p_{NIL}\\
&\qquad [\text{where all steps have been omitted}\\
&\qquad\quad \text{but for the successful communication step}]\\
&\quad = \lambda\sigma \cdot \{\langle \sigma, g_2(3)(f)\rangle, \ldots\} \parallel p_{BOOL} \parallel p_{NIL}\\
&\quad = \lambda\sigma \cdot \{\langle \sigma, f(5) \parallel p_{INT}\rangle, \ldots\} \parallel p_{BOOL} \parallel p_{NIL}\\
&\quad = \lambda\sigma \cdot \{\langle \sigma, (\lambda\sigma' \cdot \{\langle \sigma'', p_0\rangle\}) \parallel p_{INT}\rangle, \ldots\} \parallel p_{BOOL} \parallel p_{NIL}
\end{aligned}$$

where σ'' is as above but with $\beta = 5$.

4.4. *Semantics of a Unit*

4.5.1. *Environments*

If we want to define the semantics of a unit U we obviously need an environment γ_U that contains information about the class definitions and the method definitions of U. It will be defined as the fixed point of a contracting function.

DEFINITION 4.13. Let *Env* be the set of environments as defined in Definition 4.5. Thus

$$Env = (AObj \to P) \times (MName \to AObj \to Obj^* \to (Obj \to P) \to^! P).$$

For every $U \in Unit$, we define a function $\Phi_U : Env \to Env$. Let $\gamma \in Env$, $\gamma = \langle \gamma_1, \gamma_2 \rangle$. Now $\Phi_U(\gamma)$, denoted by $\tilde{\gamma}$, is given as follows: First we determine $\tilde{\gamma}_1$: Let $\alpha \in AObj$ and $C = \tau(\alpha)$. If U specifies a definition for the class C, then we put

$$\tilde{\gamma}_1(\alpha) = [\![s]\!]_S(\gamma)(\alpha)(p_0),$$

where

$$U = \langle ..., C \Leftarrow d, ... \rangle, \; d = \langle ..., s \rangle;$$

otherwise,

$$\tilde{\gamma}_1(\alpha) = \lambda\sigma \cdot \varnothing.$$

Now we define $\tilde{\gamma}_2$. Let $m \in MName$, $\alpha \in AObj$, $\bar{\beta} \in Obj^*$, $f \in Obj \to P$, and put $C = \tau(\alpha)$. If U specifies a definition for C in which m occurs and $length(\bar{\beta})$ is equal to the number of formal parameters of m, then we put

$$\tilde{\gamma}_2(m)(\alpha)(\bar{\beta})(f) = \lambda\sigma \cdot \{\langle \sigma', [\![e]\!]_E(\gamma)(\alpha)(\lambda\beta \cdot \lambda\bar{\sigma} \cdot \{\langle \bar{\sigma}', f(\beta) \rangle\}) \rangle\},$$

where

$$U = \langle ..., C \Leftarrow d, ... \rangle,$$
$$d = \langle ..., (..., m \Leftarrow \mu, ...), ... \rangle,$$
$$\mu = \langle (u_1, ..., u_n), e \rangle,$$
$$\sigma' = \langle \sigma_1, \sigma_2\{h/\alpha\}, \sigma_3 \rangle.$$
$$\bar{\beta} = \langle \beta_1, ..., \beta_n \rangle,$$
$$h(u_i) = \beta_i \text{ for } i = 1, ..., n,$$
$$h(u) = nil \text{ for } u \notin \{u_1, ..., u_n\},$$
$$\bar{\sigma}' = \langle \bar{\sigma}_1, \bar{\sigma}_2\{\sigma_2(\alpha)/\alpha\}, \bar{\sigma}_3 \rangle.$$

Otherwise, we put

$$\tilde{\gamma}_2(m)(\alpha)(\beta)(f) = \lambda\sigma \cdot \varnothing.$$

Remark. If $\tilde{\gamma}_1$ is applied to an object name of which the class is not defined in the unit U, then the empty process, $\lambda\sigma \cdot \varnothing$, is the result, indicating that an error has occurred. The same happens when $\tilde{\gamma}_2$ is supplied with incorrect arguments. The definition of $\tilde{\gamma}_1$ is straightforward. It provides a process representing the body of the appropriate object. If $\tilde{\gamma}_2$ is applied to a method m and object α, we get as a result the semantic value of the expression e that is used in the definition μ of m, preceded by a state transformation in which the temporary variables of α are initialized. After the execution of e these temporary variables are set back to their old values again, and the continuation f is supplied with the resulting value of e. (Here we use the fact that, although evaluation of a method by an object might lead to a nested invocation, this always proceeds in a "last in, first out" fashion.)

LEMMA 4.14. *Let $U \in Unit$ and let Φ_U be defined as in* 4.13. *Then Φ_U is a contraction.*

For the proof see the Appendix (A.3).

DEFINITION 4.15. Let $U \in Unit$, let Φ_U be as in 4.13. We define

$$\gamma_U = \text{Fixed Point}(\Phi_U).$$

4.5.2. *Semantics of a Unit*

The execution of a unit U with $U = \langle C_1 \Leftarrow d_1, ..., C_n \Leftarrow d_n \rangle$ consists of the creation of an object of class C_n and the execution of its body.

DEFINITION 4.16 (Semantics of a unit). We define a function

$$\mathscr{D}: Unit \to P$$

as follows: Let $U \in Unit$. Then

$$\mathscr{D}[\![U]\!] = p_U \parallel p_{ST},$$

where

$$p_U = [\![s]\!]_S(\gamma_U)(\nu(\varnothing, C_n))(p_0),$$

with

$$U = \langle ..., C_n \Leftarrow \langle ..., s \rangle \rangle,$$

and γ_U as given in Definition 4.15.

Remark. The function $[\![s]\!]_S$ is supplied with the environment γ_U, which contains information about the class and method definitions in U, the name $\nu(\varnothing, C_n)$ of the first object, and with p_0, denoting the empty continuation. The standard objects are represented by p_{ST}. They are assumed to be present at the execution of every unit U. Therefore they are composed in parallel together with p_U.

4.5.3. *Paths and Yield*

The semantics of the statement $x \leftarrow 1; x \leftarrow x+1$ executed by object α, and with the continuation p_0 is

$$\lambda\sigma \cdot \{\langle \sigma', \lambda\bar{\sigma} \cdot \{\langle \bar{\sigma}', p_0 \rangle\}\rangle\},$$

where in σ' the value of $\sigma(\alpha)(x)$ is set to 1, and in $\bar{\sigma}'$ the value of $\bar{\sigma}(\alpha)(x)$ is set to $\bar{\sigma}(\alpha)(x)+1$. This process consists of two successive state transformations that are not yet composed. The reason for this is that in our semantics parallelism is modeled by interleaving. If, however, we know that the statement above is the entire POOL program we want to consider, then no further parallel composition, and thus no further interleaving, will take place. Then we are able to compose the two state transformations into one that accumulates their respective effects. For that purpose we introduce the notion of *paths*. Given a process p_1 and a state σ_1, we want to consider computation sequences starting from $\langle \sigma_1, p_1 \rangle$.

DEFINITION 4.17 (Paths). A finite or infinite sequence $(\langle \sigma_i, p_i \rangle)_i$ with $\sigma_i \in \Sigma$, $p_i \in P$ is called a *path* (starting from $\langle \sigma_1, p_1 \rangle$) whenever

(a) $\forall j \geqslant 1[j < \mathit{length}((\langle \sigma_i, p_i \rangle)_i) \Rightarrow \langle \sigma_{j+1}, p_{j+1} \rangle \in p_j(\sigma_j)]$

(b) The sequence satisfies one of the following conditions:

(1) It is infinite. (This represents an infinite computation.)

(2) The sequence terminates with the pair $\langle \sigma_n, p_n \rangle$, where $p_n = p_0$. (This represents normal termination of all the objects in the system.)

(3) The sequence terminates with the pair $\langle \sigma_n, p_n \rangle$, where $p_n(\sigma_n) = \varnothing$. (This represents abnormal termination.)

(4) The sequence terminates with the pair $\langle \sigma_n, p_n \rangle$, where $p_n(\sigma_n) \subset \mathit{Send}_P \cup \mathit{Answer}_P$. (This represents termination by deadlock.)

The set of all paths we shall call *Path*.

Remarks. (1) A path $(\langle p_i, \sigma_i \rangle)_i$ represents a particular execution of the process p_1 starting from the state σ_1. In every component $\langle \sigma_i, p_i \rangle$ of a path starting in $\langle \sigma_1, p_1 \rangle$, the state σ_n is passed on to the resumption process p_n.

(2) In general a set $p_i(\sigma_i)$ may contain elements of $Send_P$ or $Answer_P$, besides elements of $\Sigma \times P$. Since we consider paths of only those processes that represent total (POOL) systems that are not expected to communicate with any environment, we view such elements as unsuccesful attempts at communication. Therefore we do not want to incorporate them in our definition of paths. Note that if $p_i(\sigma_i)$ contains only elements of $Send_P$ and $Answer_P$, then the path ends, and we have the termination by deadlock of case (4) above.

(3) Note that for paths representing the execution of an entire unit case (2) above never arises due to the fact that at least the standard objects are always ready to answer messages. This means that "normal termination" of a POOL program is an instance of case (4) above.

Next we define the function *yield*. It presents us, given a process p and a state σ, with the set of *all* possible paths that start from $\langle \sigma, p \rangle$.

DEFINITION 4.18 (Yield). The function $yield: P \to \Sigma \to \mathcal{P}(Path)$ is defined as follows. Let $p \in P$, $\sigma \in \Sigma$. Then

$$yield(p)(\sigma) = \{(\langle \sigma_i, p_i \rangle)_i : (\langle \sigma_i, p_i \rangle)_i \text{ a path such that } \langle \sigma_1, p_1 \rangle = \langle \sigma, p \rangle\}.$$

If we want to have all computation sequences of the denotational meaning of a given unit U, we can apply this function *yield* to the semantics of U as given in Definition 4.16:

$$yield(\mathcal{D}[\![U]\!])(\sigma_U).$$

The state σ_U we start with must be such that

$$\sigma_1 = \lambda\alpha \cdot \lambda x \cdot nil,$$

$$\sigma_2 = \lambda\alpha \cdot \lambda u \cdot nil,$$

$$\sigma_3 = \{v(\varnothing, C_n)\}$$

(where $U = \langle ..., C_n \Leftarrow d_n \rangle$) in which all variables are initialized to *nil*, and the set of objects names that are currently in use consists of the name of the first active object.

5. FAIRNESS

We shall now introduce the notion of *fairness*. A path will be called fair if it does *not* represent a situation in which an object is infinitely often enabled to take a step but never does so. To determine whether a path is fair or not, for each step that occurs in the path we have to identify the

object that takes it. It appears that the semantics of statements as we have defined it offers too little information to make the desired identification. Therefore a small adaptation of our semantic domain P, the merge operator $\|$ and the semantic functions $[\![\cdots]\!]_E$ and $[\![\cdots]\!]_S$ is required. In our new domain, which we shall still call P, we label every step with the name of the object that takes it. We give the adapted equation that must be satisfied and forget about the details of how to solve it.

DEFINITION 5.1. (Adapted domain P). Let P be such that it satisfies the equation

$$P \cong \{p_0\} \cup id_{1/2}(\Sigma \to \mathcal{P}_{cl}(Step_P))$$

where

$$Step_P = Comp_P \cup Send_P \cup Answer_P,$$

$$Comp_P = \Lambda \times \Sigma \times P \text{ (the set of } \textit{computation} \text{ steps)},$$

$$Send_P = Obj \times Obj \times MName \times Obj^* \times (Obj \to P) \times P,$$

$$Answer_P = Obj \times MName \times (Obj^* \to (Obj \to P) \to^1 P).$$

The set of *labels* Λ, with typical elements κ, is defined by

$$\Lambda = Obj \cup (Obj \times Obj).$$

The set $Answer_P$ is as before, because answer steps were already labeled: their first component indicates the object that is willing to answer the method specified by the second component. The first component of a send step denotes the object that is sending a message; the second indicates the object to which this message is sent. The first component of a computation step (i.e., an element of $Comp_P$) is an element of Λ. It is either an object, indicating the object that is taking an (internal) computation step, or it is a pair of objects, indicating the two participants in a successful communication step (see the definition of the merge operator below).

The definition of the merge operator has to be adapted to this new definition of the domain P.

DEFINITION 5.2. Let $\|: P \times P \to P$ be such that it satisfies, for $p, q \in P$,

$$p \parallel q = \begin{cases} p & \text{if } q = p_0 \\ q & \text{if } p = p_0 \\ \lambda\sigma \cdot (\{\pi \mid\!\!\widehat{\|}\, q : \pi \in p(\sigma) \wedge q(\sigma) \neq \varnothing\} \cup \\ \quad \{\pi \mid\!\!\widehat{\|}\, p : \pi \in q(\sigma) \wedge p(\sigma) \neq \varnothing\} \cup \\ \quad \bigcup \{\pi|_\sigma \rho : \pi \in p(\sigma), \rho \in q(\sigma)\}) & \text{otherwise.} \end{cases}$$

For $\pi \in Step_P$ we distinguish three cases:

(i) $\langle \kappa, \sigma', p' \rangle \hat{\|} q = \langle \kappa, \sigma', p' \| q \rangle$

(ii) $\langle \alpha, \beta, m, \bar{\beta}, f, p \rangle \hat{\|} q = \langle \alpha, \beta, m, \bar{\beta}, f, p \| q \rangle$

(iii) $\langle \alpha, m, g \rangle \hat{\|} q = \langle \alpha, m, \lambda \bar{\beta} \cdot \lambda h \cdot (g(\bar{\beta})(h) \| q) \rangle$.

Finally the set of successful communications between two processes is defined as follows. Let $\pi, \rho \in Step_P$. We have

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

The definition of a *path* (as given in Definition 4.17) has to be altered straightforwardly: A path now contains triples $\langle \kappa_i, \sigma_i, p_i \rangle$. Finally, the definition of $[\![\cdots]\!]_E$ and $[\![\cdots]\!]_S$ ought to be changed. We give one example of a clause of the definition of $[\![\cdots]\!]_E$.

DEFINITION 5.3. Let $[\![\cdots]\!]_E$ and $[\![\cdots]\!]_S$ be as given in Definitions 4.6 and 4.7, but adapted straightforwardly as is illustrated by the following clause. Let $\alpha \in AObj$, $\gamma \in Env$, $f \in Obj \to P$. We define

$$[\![x]\!]_E(\gamma)(\alpha)(f) = \lambda \sigma \cdot \{\langle \alpha, \sigma, f(\sigma_1(\alpha)(x)) \rangle\}.$$

As fairness is a negative constraint let us define which paths are to be excluded.

DEFINITION 5.4 (Unfairness). A path $(\langle \kappa_i, \sigma_i, p_i \rangle)_i$ is called *unfair* whenever one of the following conditions holds:

(i) $\exists \kappa \, \exists i_0 \geqslant 0 \, \forall n \geqslant i_0 \, [\exists p \, \exists \sigma [\langle \kappa, \sigma, p \rangle \in p_n(\sigma_n)] \wedge \kappa \neq \kappa_{n+1}]$.

(ii) $\exists \alpha \, \exists \langle i_0, i_1, \ldots \rangle \, \exists \beta \, \exists m \, \exists \bar{\beta}$,

$$\begin{aligned} & [\forall k \geqslant 0 [1 \leqslant i_k < i_{k+1}] \\ & \wedge \forall n \geqslant i_0 \, \exists f \, \exists p [\langle \alpha, \beta, m, \bar{\beta}, f, p \rangle \in p_n(\sigma_n)] \\ & \wedge \forall k \geqslant 1 \, \exists g [\langle \beta, m, g \rangle \in p_{i_k}(\sigma_{i_k})] \\ & \wedge \forall n > i_0 \, [\kappa_n \neq \langle \alpha, \beta \rangle]]. \end{aligned}$$

(iii) $\exists\alpha\,\exists\langle i_0, i_1, \ldots\rangle\,\exists m,$

$$[\forall k \geqslant 0[1 \leqslant i_k < i_{k+1}]$$
$$\wedge\ \forall n \geqslant i_0\,\exists g[\langle \alpha, m, g\rangle \in p_n(\sigma_n)]$$
$$\wedge\ \forall k \geqslant 1\,\exists\beta\,\exists\bar{\beta}\,\exists f\,\exists p[\langle \beta, \alpha, m, \bar{\beta}, f, p\rangle \in p_{i_k}(\sigma_{i_k})]$$
$$\wedge\ \forall n > i_0\ \neg\exists\beta[\kappa_n = \langle \beta, \alpha\rangle]].$$

Remark. The unfairness of a path satisfying condition (i) is interesting only when $\kappa \in Obj$. Let $\kappa = \alpha$, for an object $\alpha \in Obj$. When condition (i) is informally rephrased, it states that from a certain moment i_0 on, object α is continuously willing to take a step (namely, $\langle \alpha, \sigma, p\rangle$, where σ and p depend on the moment n) but in this path it never does so.

If a path satisfies condition (ii) it is unfair with respect to an object α because this object is neglected in too rude a manner. It tries, from a certain moment i_0 on, to communicate with object β in order to have method m executed. But although there are infinitely many moments i_k at which object β is willing to execute this method m our object α is never chosen as a matching communication partner.

Condition (iii) concerns the academic case that an object α wants to execute method m from moment i_0 on but never does so, although infinitely many matching partners present themselves one after another. (They might all be the same object.) Whenever the first component of a path results from the evaluation of a POOL program, condition (iii) implies condition (ii). For, once an object is willing to send a request to object α for the execution of method m, it is unable to do anything else until α agrees to the request.

DEFINITION 5.5 (Fairness). A path $(\langle \kappa_i, \sigma_i, p_i\rangle)_i$ is called *fair* if it is *not* unfair.

We define a function *fairyield*, which presents us, given a process p, a state σ, and a label κ, with the set of all possible fair paths that start from $\langle \kappa, \sigma, p\rangle$.

DEFINITION 5.6 (Fairyield). The function $fairyield: P \to \Sigma \to \Lambda \to \mathcal{P}(Path)$ is defined as follows. Let $p \in P$, $\sigma \in \Sigma$, $\kappa \in \Lambda$, then

$$fairyield(p)(\sigma)(\kappa) = \{(\langle \kappa_i, \sigma_i, p_i\rangle)_i : \langle \kappa_1, \sigma_1, p_1\rangle = \langle \kappa, \sigma, p\rangle \text{ and } (\langle \kappa_i, \sigma_i, p_i\rangle)_i \text{ is a fair path}\}.$$

(Formally, the choice of a label κ is necessary, but of no importance for the result of $fairyield(p)(\sigma)(\kappa)$.)

The *fair* computation sequences for a unit U are now given by

$$fairyield(\mathcal{D}[\![U]\!])(\sigma_U)(\alpha),$$

where $\mathcal{D}[\![U]\!]$ is as in Definition 4.16, σ_U is as defined at the end of Subsection 4.5.3, and α is an arbitrary label.

6. Conclusions

Now that we have given a semantics for the language POOL, it is time to evaluate our efforts. The first thing to note is that we have succeeded in giving a semantics that is really denotational: It constitutes a rigorously defined mapping from the syntactically correct constructs of the language to a mathematical domain suitable for expressing the behaviour of these constructs. Furthermore, this mapping is defined in a compositional way, in the sense that the semantics of a composite construct is defined in terms of the semantics of its constituents. We think we have given a satisfactory semantics to a parallel language with very powerful constructs: dynamic process (object) creation (the new expression) and flexible communication primitives (send, answer, and select).

The techniques we have used are quite general. We are confident that they can also be used to give a denotational semantics to other parallel languages, such as Ada or Occam.

Giving a denotational semantics to a language is an excellent way of reviewing the language design itself. In doing this for POOL, a simplified version of POOL-T, we have encountered no major semantic anomalies. A minor point is the semantics of the select statement, which appears to be overly complex and difficult to understand. In the design of POOL2, a new member of the POOL family, we have decided not to change the basic semantic primitives of the language, and to introduce only some syntactic "sugar" to enhance its ease of use. The select statement, however, is omitted and its functionality is obtained by the use of a *conditional answer*, which accepts an appropriate message if there is any and otherwise continues without waiting.

Let us now review some of the details of the present work: Why did we use the metric framework instead of the more common order-theoretic framework? We did this because it was possible. One should realize that the main reason to use structured domains instead of plain sets is that we want to be able to solve equations describing the required semantic objects in a recursive way. An equivalent formulation is that we want to construct fixed points of certain operations. Now the order-theoretic approach has

turned out to be very valuable in the situation that the operations under consideration may have many fixed points. Taking the *least* fixed point of a continuous operation on a complete partial order amounts to taking the solution that makes the fewest arbitrary assumptions. In other words, it takes the solution that is only defined insofar as it is defined explicitly by the equation. In contrast, the metric approach is very useful if the equation has only one solution. If the equation is characterized by a contracting operation on a complete metric space, then this implies that the equation has exactly one solution, and that this solution can be approximated by repeatedly applying the corresponding operation, starting from an arbitrary point. In a situation with unique fixed points, we think that the metric approach is more appropriate because it makes this situation manifest.

One could argue that our paper is not very concise, because we have to justify our constructions with proofs that are sometimes very lengthy. But if we compare this with the order-theoretic approach, we see that such proofs are also required there. They are, however, frequently omitted. This is justified on the one hand by the fact that order theory has become rather standard, so that the reader can be assumed to be able to provide the proofs himself, and, on the other hand, by the existence of very general theorems stating that functions (or functors) constructed in certain ways from certain basic building blocks are guaranteed to have fixed points. The metric approach is not yet so well known, so we thought it advisable to include the relevant proofs, but on the other hand, corresponding general theorems about the existence of fixed points for large classes of functors have been developed (see, for example, America and Rutten, 1988). A remarkable point is that the mathematical techniques used to solve reflexive domain equations, which in De Bakker and Zucker (1982) differed greatly from the ones used in the order-theoretic approach, have again converged to the latter in our work.

An important issue is the choice of the concrete mathematical domain in which the meanings of our program fragments reside, the space P of processes. It is certainly complex enough to accommodate all the different constructs in the language. However, in certain respects it appears to be too complex. For example, in the definition of fairness we had to deal extensively with unrealistic situations, processes that could never turn up as the meaning of a program. Intuitively it is clear that if we want to use a single domain of processes to describe the semantics of different constructs like expressions, statements, and units, then this domain cannot be made simpler. So if we want simpler (smaller) domains, we shall have to use different ones for different syntactic categories. Actually there are good reasons for trying to develop another semantics with smaller domains:

First, the semantics given here does not provide a clear view of the basic

concept of the language, the concept of an object. It would be nice to have a semantics in which the objects appear as building blocks of the system and in which their fundamental properties, e.g., with respect to protection, are already clear from the domain used for their semantics.

Second, there is the notion of full abstractness. A semantics is called fully abstract if any two program fragments that behave the same in all possible contexts are assigned equal semantic values. Intuitively speaking, a semantics is fully abstract if it does not provide unnecessary details. This is certainly a pleasant property of a semantics. Now full abstractness assumes a notion of observable behaviour of a program and in the language as we have presented it, programs do not interact at all with the outside world. Therefore such a notion of observability still has to be developed for POOL. Nevertheless it seems extremely unlikely that for any reasonable choice of observable behaviour a semantics along the lines of the current paper will turn out to be fully abstract.

Another unsatisfactory point is the treatment of fairness. The way this is defined here, by first generating all execution paths and then excluding the unfair ones, has a definite non-compositional flavor. It would be much more elegant if processes exhibiting unfair behaviour did not even arise in the whole construction. The most important ingredient would be a fair merge operator, merging two fair processes into one fair process. However, in our framework such a fair merge is impossible because in some situations the resulting process would give rise to non-closed subsets of steps (containing a whole Cauchy sequence, but not its limit). To solve this problem we shall probably need a more general theory of fairness, if possible in the metric framework.

A final point of further work to be done is the comparison of this denotational semantics with the operational one given in (America *et al.*, 1986). An equivalence proof would, of course, be very desirable. For a language that is only slightly simpler than POOL (instead of the rendezvous mechanism it uses simple value transmission) this has been achieved in America and De Bakker (1988). The equivalence of the operational and denotational semantics for the full language POOL is proved in Rutten (1988).

Appendix

In Definition 4.4 we gave an equation for the merge operator $\|$. Here we show that there is exactly one operator in $P \times P \to^1 P$ satisfying that equation. Let $\Phi_{PC}: (P \times P \to^1 P) \to (P \times P \to^1 P)$ be defined as follows: For $\odot \in P \times P \to^1 P$ we define $\Phi_{PC}(\odot)$, which we denote by $\tilde{\odot}$, by

$$\tilde{\odot} q = \lambda\sigma \cdot (\{\pi \tilde{\odot} q : \pi \in p(\sigma) \wedge q(\sigma) \neq \varnothing\} \cup \{\pi \tilde{\odot} p : \pi \in q(\sigma) \wedge p(\sigma) \neq \varnothing\}$$
$$\cup \bigcup \{\pi|_\sigma \rho : \pi \in p(\sigma), \rho \in q(\sigma)\})$$

or all $p, q \in P \setminus \{p_0\}$, and by $p_0 \tilde{\odot} q = q \tilde{\odot} p_0 = p_0$. Here, $\pi \hat{\odot} q$ is defined by

$$\langle \sigma', p' \rangle \hat{\odot} q = \langle \sigma', p' \odot q \rangle,$$
$$\langle \alpha, m, \beta, f, p \rangle \hat{\odot} q = \langle \alpha, m, \beta, f, p \odot q \rangle,$$
$$\langle \alpha, m, g \rangle \hat{\odot} q = \langle \alpha, m, \lambda\beta \cdot \lambda h \cdot (g(\beta)(h) \odot q) \rangle,$$

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

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

LEMMA A.1. (a) *Φ_{PC} is well defined, that is,*

$$\forall \odot \in P \times P \to^1 P [\Phi_{PC}(\odot) \in P \times P \to^1 P],$$

(b) *Φ_{PC} is a contraction.*

Proof. (a) Φ_{PC} is well defined: Let $\odot \in P \times P \to^1 P$; we show

$$\forall p_1, p_2, q_1, q_2 \in P [d_P(p_1 \tilde{\odot} q_1, p_2 \tilde{\odot} q_2) \leqslant \max\{d_P(p_1, p_2), d_P(q_1, q_2)\}],$$

where $\tilde{\odot} = \Phi_{PC}(\odot)$.

Let $p_1, p_2, q_1, q_2 \in P$. We have (recall that P is an ultra-metric space)

$$d_P(p_1 \tilde{\odot} q_1, p_2 \tilde{\odot} q_2) \leqslant \max\{d_P(p_1 \tilde{\odot} q_1, p_1 \tilde{\odot} q_2), d_P(p_1 \tilde{\odot} q_2, p_2 \tilde{\odot} q_2)\}.$$

It suffices to show that

(1) $d_P(p_1 \tilde{\odot} q_1, p_1 \tilde{\odot} q_2) \leqslant d_P(q_1, q_2)$,
(2) $d_P(p_1 \tilde{\odot} q_2, p_2 \tilde{\odot} q_2) \leqslant d_P(p_1, p_2)$.

We treat only the first case, the second being symmetric to it.

If one of p_1, q_1, q_2 is equal to p_0, the result is trivial, so suppose $p_1, q_1, q_2 \neq p_0$. Let $\sigma \in \Sigma$ and let for $i = 1, 2$,

$$X_i = \{\pi \hat{\odot} q_i \mid \pi \in p_1(\sigma) \wedge q_i(\sigma) \neq \varnothing)\}.$$
$$Y_i = \{\pi \hat{\odot} p_1 \mid \pi \in q_i(\sigma) \wedge p_1(\sigma) \neq \varnothing\},$$
$$Z_i = \bigcup \{\pi|_\sigma \rho : \pi \in p_1(\sigma), \rho \in q_i(\sigma)\},$$

so $p_1 \tilde{\odot} q_i(\sigma) = X_i \cup Y_i \cup Z_i$. Because σ is arbitrary, it suffices to show that

$$\tfrac{1}{2}\, d_{\mathscr{P}_{cl}(Step_P)}(X_1 \cup Y_1 \cup Z_1, X_2 \cup Y_2 \cup Z_2) \leqslant d_P(q_1, q_2).$$

The factor $\frac{1}{2}$ is due to the occurrence of $id_{1/2}$ in the domain equation for P (see Definition 4.3). We have

$$\begin{aligned} &d_{\mathscr{P}_{cl}(Step_P)}(X_1 \cup Y_1 \cup Z_1, X_2 \cup Y_2 \cup Z_2) \\ &\quad \leqslant \max\{d_{\mathscr{P}_{cl}(Step_P)}(X_1, X_2), d_{\mathscr{P}_{cl}(Step_P)}(Y_1, Y_2), d_{\mathscr{P}_{cl}(Step_P)}(Z_1, Z_2)\}. \end{aligned}$$

This is a consequence of the fact that the union operator is NDI, which is quite easy to prove. We show: $d_{\mathscr{P}_{cl}(Step_P)}(Z_1, Z_2) \leqslant 2 \cdot d_P(q_1, q_2)$. (The proofs for X_i and Y_i are straightforward.) By the definition of the Hausdorff distance we have

$$d_{\mathscr{P}_{cl}(Step_P)}(Z_1, Z_2) = \max\{\sup_{z_1 \in Z_1}\{d(z_1, Z_2)\}, \sup_{z_2 \in Z_2}\{d(z_2, Z_1)\}\}.$$

We consider only the first supremum:

$$\sup_{z_1 \in Z_1}\{d(z_1, Z_2)\} = \sup_{z_1 \in Z_1} \inf_{z_2 \in Z_2}\{d_{Step_P}(z_1, z_2)\}.$$

Let $z_1 \in Z_1$. There are several possibilities:

1. Suppose $\{z_1\} = \langle \alpha, m, \beta, f, p\rangle \mid_\sigma \langle \alpha, m, g_1\rangle$ with $\langle \alpha, m, \beta, f, p\rangle \in p_1(\sigma)$, $\langle \alpha, m, g_1\rangle \in q_1(\sigma)$.

1(a) If there is a $\langle \alpha, m, g_2\rangle \in q_2(\sigma)$, then we can take $z_2 \in Z_2$ such that

$$\{z_2\} = \langle \alpha, m, \beta, f, p\rangle \mid_\sigma \langle \alpha, m, g_2\rangle$$

Then we have

$$\begin{aligned} d_{Step_P}(z_1, z_2) &= d_{Step_P}(\langle \sigma, g_1(\beta)(f) \odot p\rangle, \langle \sigma, g_2(\beta)(f) \odot p\rangle) \\ &= d_P(g_1(\beta)(f) \odot p, g_2(\beta)(f) \odot p) \\ &\leqslant [\text{since } \odot \in P \times P \to^1 P] \qquad d(g_1, g_2) \\ &= d_{Step_P}(\langle \alpha, m, g_1\rangle, \langle \alpha, m, g_2\rangle). \end{aligned}$$

Now for any $\varepsilon > 0$ we can choose $\langle \alpha, m, g_2\rangle \in q_2(\sigma)$ such that

$$\begin{aligned} d_{Step_P}(\langle \alpha, m, g_1\rangle, \langle \alpha, m, g_2\rangle) &\leqslant d_{\mathscr{P}_{cl}(Step_P)}(q_1(\sigma), q_2(\sigma)) + \varepsilon \\ &\leqslant d_{\Sigma \to \mathscr{P}_{cl}(Step_P)}(q_1, q_2) + \varepsilon \\ &\leqslant 2 \cdot d(q_1, q_2) + \varepsilon. \end{aligned}$$

Therefore

$$d(z_1, Z_2) \leqslant 2 \cdot d(q_1, q_2) + \varepsilon$$

for arbitrary ε, so

$$d(z_1, Z_2) \leqslant 2 \cdot d(q_1, q_2).$$

1(b) If there is no g_2 such that $\langle \alpha, m, g_2 \rangle \in q_2(\sigma)$, then

$$d_{\mathscr{P}_{cl}(Step_P)}(q_1(\sigma), q_2(\sigma)) \geqslant d(\langle \alpha, m, g_1 \rangle, q_2(\sigma)) = 1.$$

Therefore,

$$d_P(q_1, q_2) \geqslant \tfrac{1}{2}\, d_{\mathscr{P}_{cl}(Step_P)}(q_1(\sigma), q_2(\sigma)) \geqslant \tfrac{1}{2}.$$

Now

$$d(z_1, Z_2) \leqslant 1 = 2 \cdot d_P(q_1, q_2).$$

2. The second possibility is that $\{z_1\} = \langle \alpha, m, g \rangle \mid_\sigma \langle \alpha, m, \beta, f_1, p \rangle$, with $\langle \alpha, m, g \rangle \in p_1(\sigma)$, $\langle \alpha, m, \beta, f_1, p \rangle \in q_1(\sigma)$. This case can be treated similarly to the first case.

From 1 and 2 we know that for arbitrary $z_1 \in Z_1$,

$$d(z_1, Z_2) \leqslant 2 \cdot d_P(q_1, q_2).$$

Symmetrically, we have

$$\forall z_2 \in Z_2 [d(z_2, Z_1) \leqslant 2 \cdot d_P(q_1, q_2)].$$

Therefore we can conclude

$$d_{\mathscr{P}_{cl}(Step_P)}(Z_1, Z_2) \leqslant 2 \cdot d_P(q_1, q_2).$$

(b) Φ_{PC} is a contraction. Let $\odot_1, \odot_2 \in P \times P \to^1 P$, let $\tilde{\odot}_i =^{\text{def}} \Phi_{PC}(\odot_i)$. We show that

$$d_{PXP \to^1 P}(\tilde{\odot}_1, \tilde{\odot}_2) \leqslant \tfrac{1}{2}\, d(\odot_1, \odot_2).$$

We have

$$d_{P \times P \to^1 P}(\tilde{\odot}_1, \tilde{\odot}_2) = \sup_{p,q \in P} \{d_P(p \,\tilde{\odot}_1\, q, p \,\tilde{\odot}_2\, q)\}.$$

Let $p, q \in \Sigma \to \mathscr{P}_{\mathrm{cl}}(Step_P)$, $\sigma \in \Sigma$. Let for $i = 1, 2$,

$$X_i =^{\mathrm{def}} \{\pi \mathbin{\hat{\odot}_i} q \mid \pi \in p(\sigma)\},$$
$$Y_i =^{\mathrm{def}} \{\pi \mathbin{\hat{\odot}_i} p \mid \pi \in q(\sigma)\},$$
$$Z_i =^{\mathrm{def}} \bigcup \{\pi|_\sigma \rho : \pi \in p(\sigma), \rho \in q(\sigma)\},$$

so $p \mathbin{\tilde{\odot}_i} q(\sigma) = X_i \cup Y_i \cup Z_i$. We have

$$d_{\mathscr{P}_{\mathrm{cl}}(Step_P)}(X_1 \cup Y_1 \cup Z_1, X_2 \cup Y_2 \cup Z_2)$$
$$\leqslant \max\{d_{\mathscr{P}_{\mathrm{cl}}(Step_P)}(X_1, X_2), d_{\mathscr{P}_{\mathrm{cl}}(Step_P)}(Y_1, Y_2), d_{\mathscr{P}_{\mathrm{cl}}(Step_P)}(Z_1, Z_2)\}.$$

We consider $d_{\mathscr{P}_{\mathrm{cl}}(Step_P)}(X_1, X_2)$. By definition of the Hausdorff distance we have

$$d_{\mathscr{P}_{\mathrm{cl}}(Step_P)}(X_1, X_2) = \max\{\sup_{\pi_1 \in X_1} \{d(\pi_1, X_2)\}, \sup_{\pi_2 \in X_2} \{d(\pi_2, X_1)\}\}.$$

Let $\pi_1 \in X_1$. We show

$$d(\pi_1, X_2) = \inf_{\pi_2 \in X_2} \{d_{Step_P}(\pi_1, \pi_2)\} \leqslant d_{P \times P \to^1 P}(\odot_1, \odot_2).$$

We treat one of the three possible cases for $\pi_1 \in X_1$, say $\pi_1 = \langle \sigma', p' \odot_1 q \rangle$, where $p' \in p(\sigma)$:

$$\inf_{\pi_2 \in X_2} \{d_{Step_P}(\langle \sigma', p' \odot_1 q \rangle, \pi_2)\}$$
$$\leqslant d_{Step_P}(\langle \sigma', p' \odot_1 q \rangle, \langle \sigma', p' \odot_2 q \rangle)$$
$$= d_{\Sigma \times P}(\langle \sigma', p' \odot_1 q \rangle, \langle \sigma', p' \odot_2 q \rangle)$$
$$= d_P(p' \odot_1 q, p' \odot_2 q)$$
$$\leqslant d_{P \times P \to^1 P}(\odot_1, \odot_2).$$

Thus we have

$$\sup_{\pi_1 \in X_1} \{d(\pi_1, X_2)\} \leqslant d_{P \times P \to^1 P}(\odot_1, \odot_2).$$

Similarly

$$\sup_{\pi_2 \in X_2} \{d(\pi_2, X_1)\} \leqslant d_{P \times P \to^1 P}(\odot_1, \odot_2).$$

So

$$d_{\mathscr{P}_{\mathrm{cl}}(Step_P)}(X_1\ X_2) \leqslant d_{P \times P \to^1 P}(\odot_1, \odot_2).$$

And analogously

$$d_{\mathscr{P}_{cl}(Step_P)}(Y_1, Y_2) \leqslant d_{P\times P\to^1 P}(\odot_1, \odot_2).$$

We have, according to the definition of Z_i, that $Z_1 = Z_2$. So

$$\begin{aligned} d_{\mathscr{P}_{cl}(Step_P)}(p\,\tilde{\odot}_1\, q(\sigma),\ p\,\tilde{\odot}_2\, q(\sigma)) &= d_{\mathscr{P}_{cl}(Step_P)}(X_1 \cup Y_1 \cup Z_1,\ X_2 \cup Y_2 \cup Z_2) \\ &\leqslant d_{P\times P\to^1 P}(\odot_1, \odot_2). \end{aligned}$$

This holds for every $\sigma \in \Sigma$. Therefore

$$\begin{aligned} d_P(p\,\tilde{\odot}_1\, q,\ p\,\tilde{\odot}_2\, q) &= \tfrac{1}{2}\, d_{\Sigma\to\mathscr{P}_{cl}(Step_P)}(p\,\tilde{\odot}_1\, q,\ p\,\tilde{\odot}_2\, q) \\ &\leqslant \tfrac{1}{2}\, d_{P\times P\to^1 P}(\odot_1, \odot_2) \end{aligned}$$

and thus

$$d_{P\times P\to^1 P}(\tilde{\odot}_1, \tilde{\odot}_2) \leqslant \tfrac{1}{2}\, d_{P\times P\to^1 P}(\odot_1, \odot_2).$$

LEMMA A.2 (Lemma 4.8). *For every expression e, statement s, environment γ, and active object α we have*

(i) $[\![e]\!]_E(\gamma)(\alpha) \in (Obj \to P) \to^1 P$

(ii) $[\![s]\!]_S(\gamma)(\alpha) \in P \to^1 P$

(iii) $\forall p \in P[\Phi_{e,s,p} \in P \to^{1/2} P]$,

where $\Phi_{e,s,p}: P \to P$ *is defined, for* $q \in P$, *by*

$$\begin{aligned} \Phi_{e,s,p}(q) = [\![e]\!]_E(\gamma)(\alpha) \\ (\lambda\beta \cdot \lambda\sigma \cdot \{\langle \sigma, &\text{if } \beta = tt \text{ then } [\![s]\!]_S(\gamma)(\alpha)(q) \\ &\text{elseif } \beta = ff \text{ then } p \\ &\text{else } \lambda\sigma \cdot \varnothing \\ &\text{fi}\rangle\}). \end{aligned}$$

Proof. We prove this lemma using induction on the complexity of the structure of statements and expressions. The proof consists of two parts. Let $\gamma \in Env$, $\alpha \in AObj$. We show

(a) For all simple (see below) expressions e and statements s we have

$$[\![e]\!]_E(\gamma)(\alpha) \in (Obj \to P) \to^1 P \text{ and } [\![s]\!]_S(\gamma)(\alpha) \in P \to^1 P.$$

(b) Suppose we have proved parts (i) and (ii) of the lemma for statements s_i and expressions e_j. If $s \in Stat$ and $e \in Exp$ are composed of the statements s_i and expressions e_j the lemma holds for e and s.

Part (a). Simple expressions are of the form x, u, **new**(e), **self**, or ϕ, the only type of simple statement is of the form **answer** V. Let e be a simple expression. We have to show that

$$\forall f_1, f_2 \in (Obj \to P)[d_P([\![e]\!]_E(\gamma)(\alpha)(f_1), [\![e]\!]_E(\gamma)(\alpha)(f_2)) \leqslant d_{Obj \to P}(f_1, f_2)].$$

Let $f_1, f_2 \in (Obj \to P)$. For every simple expression e that is not a standard object nor the expression **self**, we even have

$$d_P([\![e]\!]_E(\gamma)(\alpha)(f_1), [\![e]\!]_E(\gamma)(\alpha)(f_2)) \leqslant \tfrac{1}{2} d_{Obj \to P}(f_1, f_2).$$

Intuitively the decrease of distance follows from the fact that the evaluation of these expressions always takes at least one step. In this step the state may be changed and the value of the expression is passed on to the continuation f_i. This may be illustrated by the general form of the semantics of such expressions e,

$$[\![e]\!]_E(\gamma)(\alpha)(f_i) = \lambda\sigma \cdot \{\langle \sigma', \cdots f_i(\beta) \cdots \rangle\}$$

for some $\sigma' \in \Sigma$, $\beta \in Obj$. As an example let us treat one such type of expression. We show that $[\![\textbf{new}(C)]\!]_E(\gamma)(\alpha) \in (Obj \to P) \to^1 P$:

$$\begin{aligned}
&d_P([\![\textbf{new}(C)]\!]_E(\gamma)(\alpha)(f_1), [\![\textbf{new}(C)]\!]_E(\gamma)(\alpha)(f_2)) \\
&\quad = d_P(\lambda\sigma \cdot \{\langle \sigma', \gamma_1(\beta) \parallel f_1(\beta)\rangle\}, \lambda\sigma \cdot \{\langle \sigma', \gamma_1(\beta) \parallel f_2(\beta)\rangle\}) \\
&\quad = \tfrac{1}{2} \sup_{\sigma \in \Sigma} \{d_{Stepp}(\langle \sigma', \gamma_1(\beta) \parallel f_1(\beta)\rangle, \langle \sigma', \gamma_1(\beta) \parallel f_2(\beta)\rangle)\} \\
&\quad = \tfrac{1}{2} \sup_{\sigma \in \Sigma} \{d_P(\gamma_1(\beta) \parallel f_1(\beta), \gamma_1(\beta) \parallel f_2(\beta))\} \qquad \text{[because } \parallel \text{ is NDI]} \\
&\quad \leqslant \tfrac{1}{2} \sup_{\sigma \in \Sigma} \{d_P(f_1(\beta), f_2(\beta))\} \\
&\quad \leqslant \tfrac{1}{2} d_{Obj \to P}(f_1, f_2).
\end{aligned}$$

Here σ' and β are as in Definition 4.6, part E5. For the standard objects we have the following: Let $\phi \in SObj$, then

$$\begin{aligned}
&d_P([\![\phi]\!]_E(\gamma)(\alpha)(f_1), [\![\phi]\!]_E(\gamma)(\alpha)(f_2)) \\
&\quad = d_P(f_1(\phi), f_2(\phi)) \\
&\quad \leqslant d_{Obj \to P}(f_1, f_2),
\end{aligned}$$

and analogously for **self**.

For the only simple statement **answer** V, we have, for given processes $p_1, p_2 \in P$,

$$d_P([\![\textbf{answer } V]\!]_S(\gamma)(\alpha)(p_1), [\![\textbf{answer } V]\!]_S(\gamma)(\alpha)(p_2)) = d_P(\lambda\sigma \cdot \{\langle \alpha, m, g_m^{(1)}\rangle : m \in V\}, \lambda\sigma \cdot \{\langle \alpha, m, g_m^{(2)}\rangle : m \in V\}),$$

where for $j = 1, 2$ and $m \in V$,

$$g_m^{(j)} = \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in (Obj \to P) \cdot \gamma_2(m)(\alpha)(\bar{\beta})(\lambda\beta \cdot (f(\beta) \parallel p_j)).$$

The desired result is straightforward from

$$\begin{aligned} & d_{Obj^* \to (Obj \to P) \to P}(g_m^{(1)}, g_m^{(2)}) \qquad [\text{because } \gamma_2(m)(\alpha)(\bar{\beta}) \in (Obj \to P) \to^1 P] \\ & \leqslant \sup_{f \in (Obj \to P)} \{d_{Obj \to P}(\lambda\beta \cdot (f(\beta) \parallel p_1), \lambda\beta \cdot (f(\beta) \parallel p_2))\} \\ & = \sup_{p \in P} \{d_P(p \parallel p_1, p \parallel p_2)\} \qquad [\text{because } \parallel \text{ is DNI}] \\ & \leqslant d_P(p_1, p_2). \end{aligned}$$

Part (b). Composite expressions are of the form $e!m(e_1, ..., e_n)$, $m(e_1, ..., e_n)$, $e_1 \equiv e_2$, or $s; e$. Composite statements are of the form $x \leftarrow e$, $u \leftarrow e, e, s_1; s_2$, **if** e **then** s_1 **else** s_2 **fi**, **do** e **then** s **od** or **sel** g_1 **or** $\cdots$ **or** g_n **les**. Suppose that we have proved parts (i) and (ii) of the lemma for expressions $e, e_1, ..., e_n \in Exp$ and for $s \in Stat$. We shall treat one composite expression and one composite statement. We show that $[\![e!m(e_1, ..., e_n)]\!]_E(\gamma)(\alpha) \in (Obj \to P) \to^1 P$. Let $f_1, f_2 \in (Obj \to P)$. We have

$$\begin{aligned} & d_P([\![e!m(e_1, ..., e_n)]\!]_E(\gamma)(\alpha)(f_1), [\![e!m(e_1, ..., e_n)]\!]_E(\gamma)(\alpha)(f_2)) \\ & = d_P([\![e]\!]_E(\gamma)(\alpha)(\cdots \lambda\sigma \cdot \{\langle \beta, m, \bar{\beta}, f_1, p_0\rangle\} \cdots), \\ & \qquad [\![e]\!]_E(\gamma)(\alpha)(\cdots \lambda\sigma \cdot \{\langle \beta, m, \bar{\beta}, f_2, p_0\rangle\} \cdots)) \\ & \qquad [\text{by the induction hypothesis for } e] \\ & \leqslant d(\cdots \lambda\sigma \cdot \{\langle \beta, m, \bar{\beta}, f_1, p_0\rangle\} \cdots, \cdots \lambda\sigma \cdot \{\langle \beta, m, \bar{\beta}, f_2, p_0\rangle\} \cdots) \\ & \qquad [\text{by the induction hypotheses for } e_1, ..., e_n] \\ & \leqslant d_P(\lambda\sigma \cdot \{\langle \beta, m, \bar{\beta}, f_1, p_0\rangle\}, \lambda\sigma \cdot \{\langle \beta, m, \bar{\beta}, f_2, p_0\rangle\}) \\ & \leqslant \tfrac{1}{2} d_{Obj \to P}(f_1, f_2). \end{aligned}$$

The most interesting example of a composite statement is the **do** statement. We have that

$$[\![\textbf{do } e \textbf{ then } s \textbf{ od}]\!](\gamma)(\alpha) \in P \to^1 P$$

by the following argument, which at the same time proves part (iii) of the lemma.

First, we show that

$$\forall p \in P[\Phi_{e,s,p} \in P \to^{1/2} P].$$

Let $q_1, q_2 \in P$. We have

$$\begin{aligned}
&d_P(\Phi_{e,s,p}(q_1), \Phi_{e,s,p}(q_2)) \\
&\quad = d_P([\![e]\!]_E(\gamma)(\alpha)(\lambda\beta \cdots q_1 \cdots), [\![e]\!]_E(\gamma)(\alpha)(\lambda\beta \cdots q_2 \cdots)) \\
&\qquad [\text{by the induction hypothesis for } e] \\
&\quad \leqslant d_{Obj \to P}(\lambda\beta \cdot \lambda\sigma \cdot \{\cdots q_1 \cdots\}, \lambda\beta \cdot \lambda\sigma \cdot \{\cdots q_2 \cdots\}) \\
&\quad \leqslant \tfrac{1}{2} d_P([\![s]\!]_S(\gamma)(\alpha)(q_1), [\![s]\!]_S(\gamma)(\alpha)(q_2)) \\
&\qquad [\text{by the induction hypothesis for } s] \\
&\quad \leqslant \tfrac{1}{2} d_P(q_1, q_2).
\end{aligned}$$

Second, let $p_1, p_2 \in P$. We define

$$\begin{aligned}
q_1 &= ^{\text{def}} \text{Fixed Point}(\Phi_{e,s,p_1}), \\
q_2 &= ^{\text{def}} \text{Fixed Point}(\Phi_{e,s,p_2}).
\end{aligned}$$

We have

$$\begin{aligned}
&d_P([\![\textbf{do } e \textbf{ then } s \textbf{ od}]\!]_S(\gamma)(\alpha)(p_1), [\![\textbf{do } e \textbf{ then } s \textbf{ od}]\!]_S(\gamma)(\alpha)(p_2)) \\
&\quad = [\text{by definition}]\, d_P(q_1, q_2) \\
&\quad = d_P(\Phi_{e,s,p_1}(q_1), \Phi_{e,s,p_2}(q_2)) \\
&\qquad [\text{by the same kind of calculation as above,} \\
&\qquad \text{using the induction hypothesis for } e] \\
&\quad \leqslant \tfrac{1}{2} \max\{d_P([\![s]\!]_S(\gamma)(\alpha)(q_1), [\![s]\!]_S(\gamma)(\alpha)(q_2)), d_P(p_1, p_2)\} \\
&\qquad [\text{using the induction hypothesis for } s] \\
&\quad \leqslant \tfrac{1}{2} \max\{d_P(q_1, q_2), d_P(p_1, p_2)\}.
\end{aligned}$$

We see

$$d_P(q_1, q_2) \leqslant \tfrac{1}{2} d_P(p_1, p_2).$$

LEMMA A.3 (Lemma 4.14). *Let for a unit* $U \in Unit$ Φ_U *be defined as in Definition* 4.13. *Then* Φ_U *is a contraction.*

Proof. We shall show

$$\forall \gamma, \delta \in Env[d_{Env}(\tilde{\gamma}, \tilde{\delta}) \leqslant \tfrac{1}{2} d_{Env}(\gamma, \delta)],$$

where $\tilde{\gamma} = \Phi_U(\gamma)$, $\tilde{\delta} = \Phi_U(\delta)$, by proving for $\gamma, \delta \in Env$ the two inequalities:

(a) $d_{Env_1}((\tilde{\gamma})_1, (\tilde{\delta})_1) \leqslant \tfrac{1}{2} d_{Env}(\gamma, \delta)$

(b) $d_{Env_2}((\tilde{\gamma})_2, (\tilde{\delta})_2) \leqslant \tfrac{1}{2} d_{Env}(\gamma, \delta)$.

We have

$$\begin{aligned} & d_{Env_1}((\tilde{\gamma})_1, (\tilde{\delta})_1) \\ & \quad = \sup_{\alpha \in AObj} \{d_P((\tilde{\gamma})_1(\alpha), (\tilde{\delta})_1(\alpha))\} \\ & \quad \leqslant \sup_{s \in Stat, \alpha \in AObj} \{d_P([\![s]\!]_S(\gamma)(\alpha)(p_0), [\![s]\!]_S(\delta)(\alpha)(p_0))\}. \end{aligned}$$

Now it is easy to prove (in the same way as in Lemma 4.8) that, for every $s \in Stat$ and $e \in Exp$,

$$[\![s]\!]_S \in Env \to^{1/2} (AObj \to P \to^1 P),$$
$$[\![e]\!]_E \in Env \to^{1/2} (AObj \to (Obj \to P) \to^1 P).$$

Intuitively this can be explained by the fact that whenever the environment occurs in the semantic equations (the cases E4, E5, S3, and S8), it is "guarded" by $\lambda\sigma \cdot \langle \cdots \rangle$. From this observation it follows that

$$\sup_{s \in Stat, \alpha \in AObj} \{d_P([\![s]\!]_S(\gamma)(\alpha)(p_0), [\![s]\!]_S(\delta)(\alpha)(p_0))\} \leqslant \tfrac{1}{2} d_{Env}(\gamma, \delta),$$

which concludes the proof of part (a).

The proof of part (b) is similar to that of part (a) and therefore we omit it.

ACKNOWLEDGMENTS

We are indebted to the members of the Working Group on Semantics of ESPRIT project 415, especially to Werner Damm who stressed the importance of using continuations at a moment when we had given up on them (at that time the approach in (America and Rutten, 1988) had not yet been conceived, and continuations did not fit into the process domain). We

also wish to thank the following persons for their contribution to the discussions of many of the preliminary ideas on which this report is based: Frank de Boer, Anton Eliëns, Hans Jonkers, Frank van der Linden, John-Jules Meyer, Marly Roncken, and Erik de Vink. Finally we are grateful to the anonymous referees, whose comments on an earlier version of this paper have led to considerable improvements.

RECEIVED September 30, 1986; ACCEPTED September 9, 1987

REFERENCES

AMERICA, P. (1985), "Definition of the programming language POOL-T," ESPRIT Project 415, No. 0091, Philips Research Laboratories, Eindhoven.

AMERICA, P. (1986), "Rationale for the design of POOL," ESPRIT Project 415, No. 0053, Philips Research Laboratories, Eindhoven.

AMERICA, P. (1987), POOL-T—A parallel object-oriented language, *in* "Object-Oriented Concurrent Systems" (A. Yonezawa and M. Tokoro, Eds.), MIT Press, Cambridge, MA.

AMERICA, P., AND DE BAKKER, J. W. (1988), Designing equivalent semantic models for process creation, *Theoret. Comput. Sci.* **60**, 109–176.

AMERICA, P., DE BAKKER, J. W., KOK, J. N., AND RUTTEN J. J. M. M. (1986), Operational semantics of a parallel object-oriented language, *in* "Conference Record of the 13th Symposium on Principles of Programming Languages, St. Petersburg, Florida," pp. 194–208.

AMERICA, P., AND RUTTEN, J. J. M. M. (1988), Solving reflexive domain equations in a category of complete metric spaces, *in* "Proceedings, Third Workshop on Mathematical Foundations of Programming Language Semantics, New Orleans, 1987" (M. Main, A. Melton, M. Mislove, D. Schmidt, Eds.), pp. 254–288, Lecture Notes in Comput. Sci., Vol. 298, Springer-Verlag, New York/Berlin. To appear in *J. Comput. System Sci.*

ANSI (1983), "Reference Manual for the Ada Programming Language," ANSI/MIL-STD 1815 A, U. S. Department of Defense, Washington D. C.

BERGSTRA, J., AND KLOP, J. W. (1984), Process algebra for synchronous communication, *Inform. and Control* **60**, 109–137.

DE BAKKER, J. W., KOK, J. N., MEYER, J.-J. CH., OLDEROG, E.-R., AND ZUCKER, J. I. (1986), Contrasting themes in the semantics of imperative concurrency, *in* "Current Trends in Concurrency, Overviews and Tutorials" (J. W. de Bakker, W. P. de Roever, G. Rozenberg, Eds.), pp. 51–121, Lecture Notes in Comput. Sci., Vol. 224, Springer-Verlag, New York/Berlin.

DE BAKKER, J. W., AND ZUCKER, J. I. (1982), Processes and the denotational semantics of concurrency, *Inform. and Control* **54**, 70–120.

DE BRUIN, A. (1986), "Experiments with Continuation Semantics: Jumps, Backtracking, Dynamic networks," Ph.D. thesis, Free University of Amsterdam.

CLINGER, W. D. (1981), Foundations of actor semantics, Ph.D. thesis, AI-TR-633, Massachusetts Institute of Technology.

DUGUNDJI, J. (1966), "Topology," Allyn & Bacon, Inc., Boston.

ENGELKING, R. (1977), "General Topology," Polish Scientific, Warsaw.

GIERZ, G., HOFMANN, K. H., KEIMEL, K., LAWSON, J. D., MISLOVE, M., AND SCOTT, D. S. (1980), "A Compendium of Continuous Lattices," Springer-Verlag, New York/Berlin.

GOLDBERG, A., AND ROBSON, D. (1983), "Smalltalk-80, The Language and Its Implementation," Addison-Wesley, Reading, MA.

GORDON, M. J. C. (1979), "The Denotational Description of Programming Languages," Springer-Verlag, New York/Berlin.

HAHN, H. (1948), "Reelle Funktionen," Chelsea, New York.

HENNESSY, M., AND PLOTKIN, G. D. (1979), Full abstraction for a simple parallel programming language, *in* "Proceedings, 8th Symposium on the Mathematical Foundations of Computer Science," pp. 108–120, Lecture Notes in Comput. Sci., Vol. 74, Springer-Verlag, New York/Berlin.

HEWITT, C. (1977), Viewing control structures as patterns of passing messages, *Artif. Intell.* **8**, 323–364.

MAC LANE, S. (1971), "Categories for the Working Mathematician," Springer-Verlag, New York/Berlin.

ODIJK, E. A. M. (1987), The DOOM system and its applications: A survey of ESPRIT 415 subproject A, *in* "Parallel Architectures and Languages Europe, Vol. I" (J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, Eds.), pp. 461–479, Lecture Notes in Comput. Sci., Vol. 258, Springer-Verlag, New York/Berlin.

RUTTEN, J. J. M. M. (1988), Semantic correctness for a parallel object-oriented language, report CS-R8843, Centre for Mathematics and Computer Science, Amsterdam. To appear in *SIAM J. Comput.*

VAANDRAGER, F. W. (1986), "Process algebra Semantics of POOL," Technical Report CS-R8629, Centre for Mathematics and Computer Science, Amsterdam.