Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

P. America, J.W. de Bakker, J.N. Kok, J.J.M.M. Rutten

A denotational semantics of a parallel object-oriented language

Computer Science/Department of Software Technology Report CS-R8626 August

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

A 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, 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 designed by America. For this purpose we construct a mathematical domain of processes in the sense of De Bakker and Zucker. Their approach is extended to cover a wider class of reflexive domain equations, including function spaces. A category of metric spaces is considered, and the desired domain is obtained as fixed point of a contracting functor. The domain is sufficiently rich to allow a fully compositional definition of the language constructs in POOL, including advanced concepts such as process 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. A preliminary discussion is provided on how to deal with fairness. Full mathematical details are supplied (in an appendix), with the exception of the general domain construction which is to be described elsewhere.

1980 Mathematics Subject Classification: 68B10, 68C01.
1986 Computing Reviews Categories: D.1.3, D.2.1, D.3.1, F.3.2.
Key Words & Phrases: object-oriented programming, denotational semantics, parallelism, metric spaces, fixed points, fairness, domain equations, continuations.
Note: This work was carried out in the context of ESPRIT project 415: Parallel Architectures and Languages for AIP - a VLSI-directed approach.

Report CS-R8626
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

CONTENTS

1. INTRODUCTION

In this report we give a formal semantics of a language called POOL (Parallel Object-Oriented Language). It is a slightly simplified version of the language POOL-T, which is defined in [Am1] and for which [Am2,3] give an account of the design considerations. After having defined an operational semantics for POOL in [ABKR], in this paper we set out to develop a *denotational* semantics, which will be more abstract. (For a characterization of denotational semantics, as opposed to operational semantics, see the introduction of [BZ].)

An important choice we have made is to use a mathematical framework of complete metric spaces for our semantic description. In this we follow and generalize the approach of [BZ]. (For other applications of this type of semantic framework see [BKMOZ].) 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 the category of complete metric spaces and then constructing a fixed point for this functor. The mathematical means to do this are sketched in section 2 and will be presented in detail in [AR].

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

$$[\![\cdots]\!]_E : Exp \rightarrow Env \rightarrow Obj \rightarrow ExpCont \rightarrow P$$

$$[\![\cdots]\!]_S : Stat \rightarrow Env \rightarrow Obj \rightarrow Cont \rightarrow P$$

where

$$ExpCont = Obj \rightarrow P,$$

$$Cont = P.$$

We give the formal description of the type of these semantic functions, because we want to stress three of their characteristics, namely the use of *environments*, *objects* and *continuations*.
The environments 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 class *Obj* stands for the set of objects. Its appearance in the defining equations reflects that in POOL each expression or statement is evaluated *by an object.* An informal introduction to the language POOL which, in particular, explains this view on how objects have an internal activity in which they execute expressions or statements, is to be found in section 3 (which is almost identical to a section in [ABKR]).
Finally, a continuation will be given as an argument to the semantic functions, describing 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), its type is $Obj \rightarrow P$, whereas the type of continuations of statements is simply P. (For an explanation and many examples of this type of semantics, which is sometimes called: continuation semantics, see [Br].)
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 (as we shall see, POOL itself does not have a syntactic operator for parallel execution) used, e.g., in the equation for the creation of a new object. Then the core of the semantic definitions, in terms of the various semantic equations for the respective classes of expressions and statements, is displayed. Once the reader has understood 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_{STANDARD}, which describes the standard objects (integers, booleans) of the language. Finally, 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_{STANDARD}.
In section 5 the semantic model is adapted a little 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 final form and will be probably developed further in subsequent work. Section 6 presents some conclusions and gives some suggestions for further research.

As related work concerning the semantics of POOL, apart from [ABKR], we only know the paper [Va]. Concerning the semantics of object-oriented programming in general, we refer to [Am4] for an overview. Semantic treatments of *parallel* object-oriented languages are scarce; only [Cl] gives a detailed mathematical model for an actor language.
As to the material in section 2, there is a vast amount of literature on order-theoretic domain theory (see, for instance, [Gi]). Our approach in which the category of metric spaces and (generalizations of) Banach's theorem are central, may be an attractive alternative for a situation where the contractivity of the various functions encountered is a natural phenomenon.

Acknowledgements
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 we had given up on them (at that time the approach in [AR] 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 Eliens, Frank van der Linden, John-Jules Meyer and Erik de Vink.
We are grateful to Mini Middelberg and Carolien Swagerman for their contribution to the typing of this document.

2. Mathematical preliminaries

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

2.1. Metric spaces

Definition 2.1 (Metric space)
A *metric space* is a pair (M,d) with M a set and d a mapping $d:M\times M\to[0,1]$, which satisfies the following properties:
(a) $\forall x,y\in M\,[d(x,y)=0 \Leftrightarrow x=y]$
(b) $\forall x,y\in M\,[d(x,y)=d(y,x)]$
(c) $\forall x,y,z\in M\,[d(x,y)\leqslant d(x,z)+d(z,y)]$.
We call (M,d) an *ultra-metric space* if the following stronger version of property (c) is satisfied:
(c′) $\forall x,y,z\in M\,[d(x,y)\leqslant\max\{d(x,z),d(z,y)\}]$.

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

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

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:
$\forall\epsilon>0\ \exists N\in\mathbb{N}\ \forall n,m>N\ [d(x_n,x_m)<\epsilon]$.
(b) Let $x\in M$. We say that $(x_i)_i$ *converges to* x and call x the *limit* of $(x_i)_i$ whenever we have:
$\forall\epsilon>0\ \exists N\in\mathbb{N}\ \forall n>N\ [d(x,x_n)<\epsilon]$.
Such a sequence we call *convergent.*
(c) The metric space (M,d) is called *complete* whenever each Cauchy sequence converges to an element of M.

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

Proposition 2.4
(a) *Let $(M_1,d_1),(M_2,d_2)$ be metric spaces. For every $A\geqslant 0, f\in M_1\to^A M_2$ we have: f is continuous.*

(b) *(Banach's fixed point theorem)*
Let (M,d) be a complete metric space, $f:M\to M$ a contracting function. Then there exists $x\in M$ 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 $f^{(0)}(x_0)=x_0$.

DEFINITION 2.5 (Closed sets)
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 2.6
Let $(M,d),(M_1,d_1),\ldots,(M_n,d_n)$ be metric spaces.
(a) We define a metric d_F on $M_1\to M_2$ as follows. For every $f_1,f_2\in M_1\to M_2$

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

(b) Whenever we have that $M_1,\ldots,M_n$ are mutually disjoint and $0<\epsilon\leqslant 1$, we define a metric $d_{U,\epsilon}$ on $M_1\cup\cdots\cup M_n$ as follows. For every $x,y\in M_1\cup\cdots\cup M_n$

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

(c) We define a metric d_P on $M_1\times\cdots\times M_n$ by the following clause.
For every $(x_1,\ldots,x_n),(y_1,\ldots,y_n)\in M_1\times\cdots\times M_n$

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

(d) Let $\mathcal{P}_{cl}(M)=^{def}\{X|X\subseteq M|X \text{ is closed and non-empty}\}$. We define a metric d_H on $\mathcal{P}_{cl}(M)$, called the *Hausdorff distance,* as follows. For every $X,Y\in\mathcal{P}_{cl}(M)$

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

where $d(x,Z)=^{def}\inf_{z\in Z}\{d(x,z)\}$ for every $Z\subseteq M,\ x\in M$.

PROPOSITION 2.7
Let $(M,d),(M_1,d_1),\ldots,(M_n,d_n),d_F,d_{U,\epsilon},d_P,d_H$ be as in definition 2.6. We have that
(a) $(M_1\to M_2,d_F)$,
(b) $(M_1\cup\cdots\cup M_n,d_{U,\epsilon})$,
(c) $(M_1\times\cdots\times M_n,d_P)$,
(d) $(\mathcal{P}_{cl}(M),d_H)$
are complete metric spaces. In case (M,d) and (M_i,d_i) are all ultra-metric spaces these composed spaces are again ultra-metric.

If in the sequel we write $M_1\to M_2$, $M_1\times\cdots\times M_n$ or $\mathcal{P}_{cl}(M)$, we mean the metric space with the metric defined above. We wite $M_1\cup^{\epsilon}\cdots\cup^{\epsilon}M_n$ to indicate the ϵ that is used in the metric.
The proofs of proposition 2.7 (a), (b) and (c) are straightforward. Part (d) is more involved. It can be proved with the help of the following characterization of the completeness of $(\mathcal{P}_{cl}(M),d_H)$.

PROPOSITION 2.8
Let $(\mathcal{P}_{cl}(M),d_H)$ be as in definition 2.6. Let $(X_i)_i$ be a Cauchy sequence in $\mathcal{P}_{cl}(M)$. We have:

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

Proofs of proposition 2.7(d) and 2.8 can be found in (for instance) [Du] and [En]. Proposition 2.8 is

due to Hahn [Ha]. The proofs are also repeated in [BZ].

2.2. Inverse limit construction and fixed point theorem

We shall use for our denotational semantics a solution of a recursive domain equation of the following type:

$$P \cong FP,$$

where F is some functor $F:\mathcal{C}\to\mathcal{C}$ on a category of complete metric spaces (to be defined in a moment). (For an extensive introduction into category theory we refer the reader to [ML].) A few examples of such equations are

(1) $P \cong \{p_0\} \cup^{\epsilon} (A \times P)$,
(2) $P \cong \{p_0\} \cup^{\epsilon} \mathcal{P}_{cl}(A \times P)$,
(3) $P \cong \{p_0\} \cup^{\epsilon} (A \to P)$,
(4) $P \cong \{p_0\} \cup^{\epsilon} (P \to^{1} B)$,

where A is some (possibly infinite) set, B some expression (possibly containing P), p_0 some kind of null element, and $0<\epsilon<1$.

The functor we shall need for our semantics will be built up from the different constructs used in the examples (1) through (4). Techniques have been developed to solve equations of type (1), (2) and (3) in a metrical framework by De Bakker and Zucker. For an explanation and many applications of their theory see [BZ,BKMOZ]. However, it is not clear how their theory could be used to solve equations of type (4).

It appears to be possible to extend the application of the inverse limit construction to the category of complete metric spaces and prove the familiar kind of fixed point theorem with respect to that category. This theorem will in some respects be a categorical generalization of the classical fixed point theorem of Banach (proposition 2.4(b), above). It provides us, under certain conditions, with a unique fixed point for each contracting functor. In the remaining part of this section some definitions and lemmas, needed for the formulation of the theorem, as well as the theorem itself are presented. All proofs are omitted. They will be fully described in [AR].

DEFINITION 2.9 (Category of complete metric spaces)

Let $\mathcal{C}$ denote the category that has complete metric spaces for its objects. The arrows ι in $\mathcal{C}$ are defined as follows Let M_1, M_2 be complete metric space. Then $M_1 \to^{\iota} M_2$ denotes a pair of maps $M_1 \underset{j}{\overset{i}{\rightleftarrows}} M_2$, 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 $<i,j>$ for ι.) Composition of the arrows is defined in the obvious way.

DEFINITION 2.10 (Converging tower)

(a) We call a sequence $(D_n, \iota_n)_n$ of complete metric spaces and arrows a *tower* whenever we have that $\forall n \geq 0\ [D_n \to^{\iota_n} D_{n+1} \in \mathcal{C}]$.
(b) The sequence $(D_n, \iota_n)_n$ is called a *converging tower*, when furthermore the following condition is satisfied:
$\forall \epsilon > 0\ \exists N \in \mathbb{N}\ \forall m > n \geq N\ [d(i_{nm} \circ j_{nm}, id_{D_m}) < \epsilon]$,
where $<i_{nm}, j_{nm}> = \iota_{nm} = \iota_{m-1} \circ \cdots \circ \iota_n$.

EXAMPLE

A special case of a converging tower is a sequence $(D_n, \iota_n)_n$ that satisfies the following conditions:
(a) $\forall n \geq 0\ [D_n \to^{\iota_n} D_{n+1} \in \mathcal{C}]$,

(b) $\exists\epsilon\,[0\leqslant\epsilon<1\wedge\forall n\geqslant 0\,[d(i_{n+1}\circ j_{n+1},id_{D_{n+2}})\leqslant\epsilon\cdot d(i_n\circ j_n,id_{D_{n+1}})]]$.

DEFINITION 2.11 (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:

$$\forall n\geqslant 0\,[D_n\rightarrow^{\gamma_n} D\in\mathcal{C}\wedge\gamma_n=\gamma_{n+1}\circ\iota_n].$$

DEFINITION 2.12 (Initial cone)
A cone $(D,(\gamma_n)_n)$ of a tower $(D_n,\iota_n)_n$ is called *initial* whenever for every other cone $(D',(\gamma'_n)_n)$ of $(D_n,\iota_n)_n$ there exists a unique arrow ι in $\mathcal{C}$ such that the following holds:
(a) $D\rightarrow^{\iota}D'\in\mathcal{C}$,
(b) $\forall n\geqslant 0\,[\iota\circ\gamma_n=\gamma'_n]$.

LEMMA 2.13 (Initiality lemma)
Let $(D_n,\iota_n)_n$ be a converging tower with a cone $(D,(\gamma_n)_n)$. Let $\gamma_n=<\alpha_n,\beta_n>$. We have:

$$D \text{ is an initial cone} \Leftrightarrow \lim_{n\rightarrow\infty}\alpha_n\circ\beta_n=id_D.$$

DEFINITION 2.14 (Inverse limit)
Let $(D_n,\iota_n)_n$, with $\iota_n=<i_n,j_n>$, be a converging tower. The *inverse limit* of $(D_n,\iota_n)_n$ is a cone $(D,(\gamma_n)_n)$, with $\gamma_n=<\alpha_n,\beta_n>$, that is defined as follows:

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

a metric $d:D\times D\rightarrow[0,1]$ such that for all $(x_n)_n,(y_n)_n\in D$: $d((x_n)_n,(y_n)_n)=\lim_{n\rightarrow\infty}\{d_n(x_n,y_n)\}$ where d_n is the metric on D_n;
$\alpha_n:D_n\rightarrow D$ is defined by $\alpha_n(x)=(x_k)_k$, where

$$x_k=\begin{cases} j_k\circ j_{k+1}\circ\cdots\circ j_{n-1}(x) & \text{if } k<n\\ x & \text{if } k=n\\ i_{k-1}\circ i_{k-2}\circ\cdots\circ i_n(x) & \text{if } k>n;\end{cases}$$

$\beta_n:D\rightarrow D_n$ is defined by $\beta_n((x_k)_k)=x_n$.

LEMMA 2.15
The inverse limit of a converging tower (as defined in definition 2.14) is an initial cone for that tower.

DEFINITION 2.16 (Contracting functor)
We call a functor $F:\mathcal{C}\rightarrow\mathcal{C}$ contracting whenever the following holds: there exists $\epsilon,0\leqslant\epsilon<1$, such that for all $D\rightarrow^{\iota}E\in\mathcal{C}$, with $\iota=<i,j>$, we have:

$$d(Fi\circ Fj,id_{FE})\leqslant\epsilon\cdot d(i\circ j,id_E).$$

LEMMA 2.17
Let $F:\mathcal{C}\rightarrow\mathcal{C}$ be a contracting functor, let $(D_n,\iota_n)_n$ be a converging tower with an initial cone $(D,(\gamma_n)_n)$. Then $(FD_n,F\iota_n)_n$ is again a converging tower with $(FD,(F\gamma_n)_n)$ as an initial cone.

THEOREM 2.18 (Fixed point theorem)
Let F be a contracting functor $F:\mathcal{C}\rightarrow\mathcal{C}$ and let $D_0\rightarrow^{\iota_0}FD_0\in\mathcal{C}$. Let the tower $(D_n,\iota_n)_n$ be defined by $D_{n+1}=FD_n$ and $\iota_{n+1}=F\iota_n$ for all $n\geqslant 0$. This tower is converging so it has an inverse limit $(D,(\gamma_n)_n)$. We have: $D\cong FD$.

REMARKS

(1) It is possible to impose certain restrictions upon the category $\mathcal{C}$ such that a contracting functor F on $\mathcal{C}$ has a unique fixed point (up to isometry).

(2) If we wish to restrict our attention to the subcategory of $\mathcal{C}$ of complete *ultra-metric* spaces, the definitions, lemma's and the theorem can be adapted straightforwardly.

3. THE LANGUAGE POOL

(This section is almost literally a copy of the sections 2 and 3 of [ABKR].)

3.1 An informal introduction to the language

Programming concurrent systems is a difficult task in which it occurs very often that subtle errors disturb the correct functioning of a program in a most disastrous way. As the number of parallel processes increases, these difficulties tend also to increase dramatically. Here a clear way of structuring such systems is badly needed. Object-oriented programming (of which the language Smalltalk-80 [GR] 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 [He], but our approach is different from his.)

POOL is an acronym for "Parallel Object-Oriented Language". It stands for a family of languages designed at Philips Research Laboratories in Eindhoven in the second half of 1984 and the first half of 1985. Of these languages, POOL-T [Am1] is the latest, and the one that will be implemented. These languages all make use of the structuring mechanisms of object-oriented programming, integrated with the most important mechanisms for expressing concurrency from the language Ada [ANSI]: processes and rendez-vous. This paper describes the semantics of a language that we will simply call POOL. It is undoubtedly a member of the above family, but it is slightly simplified (especially with respect to syntax) in order to facilitate the semantic description.

A POOL program describes the behaviour of a whole system in terms of its constituents, *objects*. Objects possess some internal data, and they have the ability to act on these data. They are entities of a very dynamic nature: they can be created dynamically, they can be modified, and they have even an internal activity of their own. At the same time they are units of protection: the internal data of one object are not directly accessible to other objects.

An object can have *variables* (also called instance variables) to store its internal data in. A variable can contain (a reference to) an object (another object, or, possibly, the object under consideration itself). Changing (assigning to) a variable means making 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 assigned to by the object itself.

The objects may only interact by sending *messages* to each other. Each object states explicitly to which object it sends a certain message, and also when it is prepared to accept one. When an object sends a message, its activity is suspended until the result of that message arrives. When the receiver answers the message, it will execute a so-called *method* (a kind of procedure) and the result of this method execution will be sent back to the sender. The sender of the message indicates which method should be invoked. It can also pass some *parameters* to the method.

Such a method can access the variables of the receiving object. Furthermore it can have some local variables of its own. 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 local variables, of course.

When an object is created, a local process is started up: the object's *body*. When several objects have been created, their bodies may execute in parallel. This is the way parallelism is introduced into the language. Within a body an object states explicitly when it is prepared to answer certain messages. Having returned the answer, the execution of its body is resumed.

Objects are grouped into *classes*. All objects in one class (the *instances* of that class) have the same number and kind of variables, the same methods for answering messages, and the same body. In this way a class describes the behaviour of its instances.

There is a special object, *nil*, which can be considered to be an element of every class. If a message is

sent to this object, an error occurs. Upon the creation of a new object, its instance variables are initialized to *nil*, and when a method is invoked, its local variables are also initialized to *nil*.
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 sending the message with method name add and parameter 4 to the object 2.

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 elements: $x, y, \ldots,$
LVar	(local variables)	with typical elements: $u, v, \ldots,$
CName	(class names)	with typical elements: $C, D, \ldots,$
MName	(method names)	with typical elements: $m, \ldots$.

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

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

($\mathbb{Z}$ is the set of all integers.)
We now define the set *Exp* of expressions, with typical elements $e, \ldots$:

$e ::= x \mid u \mid e\ !\ m\ (e_1, \ldots, e_n) \mid m\ (e_1, \ldots, e_n) \mid$ **new** $(C) \mid s\ ;\ e \mid$ **self** $\mid \phi$

The set *Stat* of statements, with typical elements $s, \ldots$:

$s ::= x \leftarrow e \mid u \leftarrow e \mid$ **answer** $(m_1, \ldots, m_n) \mid e \mid s_1\ ;\ s_2 \mid$ **if** e **then** s_1 **else** s_2 **fi** $\mid$ **do** e **then** s **od** $\mid$ **sel** g_1 **or** $\cdots$ **or** g_n **les**

The set *GCom* of guarded commands, with typical elements $g, \ldots$:

$g ::= [\ e\ ]$ **answer** $(m_1, \ldots, m_n)$ **then** s.

(Here the expression is optional. Note that $n = 0$ is allowed.)

The set *Unit* of units, with typical elements $U, \ldots$:

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

The set *ClassDef* of class definitions, with typical elements $d, \ldots$:

$d ::= < (x_1, \ldots, x_n) , (m_1 \Leftarrow \mu_1, \ldots, m_k \Leftarrow \mu_k) , s >$

And finally the set *MethDef* of method definitions, with typical elements $\mu, \ldots$:

$\mu ::= < (u_1, \ldots, u_n) , (v_1, \ldots, v_k) , e >.$

3.2.1 Informal explanation

Expressions An instance variable or a local variable used as an expression will yield as its value the object that is stored currently 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 parameters are initialized to the objects in the message, the local 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 and this will be the value of the send-expression.
A method call means simply 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. Of this object the instance variables 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.
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 object that is executing this expression.
The evaluation of a standard object ϕ results in the value of that object. For instance the value of the syntactic object 23 will be the natural number 23.

Statements The first two kinds of statements are assignments, to an instance variable and to a local variable. An assignment is executed by first evaluating the expression on the right, and then making the variable refer to the resulting object. Assignments to parameters of a method are not allowed.
The next kind of statement is an answer-statement. This indicates that a message is answered. The object executing the answer-statement waits until a message arrives with a method name that is present in the list. Then it executes the method (after initializing parameters and local 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 every expression may also occur as a statement. Upon execution, the expression is evaluated and the result is discarded. So only the side effects are important.
Sequential composition, if-statements and do-statements 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 (if absent, they default to *tt*). They must all result in an object of class Boolean. The guarded commands, of which the expression has 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. This must be either the first one in which the answer statement contains no method names, or there has arrived a message with its method name in the list, and there is no guarded command before this one with an empty method name list, or with the same method name in the list. (Otherwise stated, the object may choose the first guarded command for which it does not have to answer a message, or it may choose to answer a message, in which case it must select the first guarded command in which that can be done.) If the selected guarded command has a non-empty answer list the above message is answered. After that in either case the statement after **then** is

executed, and the select-statement is terminated.

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

Units These are the programs of POOL. If a unit is to be executed, a single new instance of the *last* class defined in the unit is created and its body is started. This object has the task to start the whole system up, 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 instance variables, the methods and the body each instance of the class will have.

Method definitions A method definition describes, of course, a method. Here u_1 through u_n are the parameters, v_1 through v_k the local variables, and e is the expression to be evaluated when the method is invoked.

3.2.2 Context conditions

For a POOL program to be valid there are a few more conditions to be satisfied. In general, they amount roughly to the requirement that such a program be a translated version of a valid POOL-T program. 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 instance variables in a class definition are different.
- All method names in a class definition are different.
- All parameters and local variables in a method definition are different.
- All instance variables are declared in the current class definition.
- All local variables are declared in the current method definition (they may not occur in a body).
- The class in a new-expression is defined in the current unit. (Here the standard classes are not allowed.)
- For each method name in an answer-statement or method call there is a method definition in the current class definition.
- There should be a consistent typing possible for the whole program.

The last condition means that each (instance or local) variable, each parameter and each method should be assigned a class name, its *type*, in such a way that, if starting from these assigned types, the types of each expression is determined, the following conditions hold:
- The left hand side of an assignment has the same type as its right hand side.
- In a send-expression, the class of the destination has a method with the indicated name, and the number and types of the actual parameters agree with those of the formal parameters. The same holds for the parameters of a method call.
- The types of the expressions in if-statements, do-statements and guarded commands are Boolean.
- The type of the expression in a method definition agrees with the type of the method.

4. Denotational semantics

This section constitutes the heart of our paper. It introduces the domain of processes used in the semantic equations and, for each syntactic construct in POOL an equation yielding a result in this domain is provided. In addition, it presents the definition of the sets of objects and states, and it defines the auxiliary operation of parallel composition, which is used, e.g., in the equation of the new-construct. Next, the semantics for the standard objects (integers and booleans) of POOL are 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, and of the notions of *paths* and *yield* of a process.

4.1 Definitions

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

Definition 4.1 (Objects)
We define the set $AObj$ of *active* (i.e., non-standard) objects by:

$$AObj = \{\hat{n} | n \in \mathbb{N}\}.$$

This set and the set of standard objects $SObj$ together form the set Obj of *objects,* with typical elements $\alpha, \ldots$:

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

(Here $\mathbb{N}$ is the set of natural numbers and $\mathbb{Z}$ is the set of all integers.)

Definition 4.2
The set of states, with typical elements $\sigma, \ldots,$ is defined by

$$\begin{aligned} \Sigma = {} & AObj \rightarrow IVar \rightarrow Obj \\ & \times AObj \rightarrow (LVar \rightarrow Obj)^* \\ & \times \mathbb{N} \\ & \times AObj \rightarrow CName. \end{aligned}$$

Remarks
(1) We denote the four components of $\sigma \in \Sigma$ by $\sigma = <\sigma_1, \sigma_2, \sigma_3, \sigma_4>$.
(2) The first component of a state is used for the values of the instance variables of each (active) object. The second component of a state is used for the values of the local variables of each active method invocation of each object in order to handle recursion. For each state σ and each object α, $\sigma_2(\alpha)$ denotes an element of the set $(LVar \rightarrow Obj)^*$, which can be seen as a stack of frames. We shall need the following operations upon these stacks.
For $f_1, \ldots, f_n, f \in (LVar \rightarrow Obj)$, we define:

$$-pop(<f_1, \ldots, f_n>) = \begin{cases} <f_1, \ldots, f_{n-1}> & \text{if } n > 1 \\ \epsilon & \text{if } n = 1 \end{cases}$$

$$-push(f, <f_1, \ldots, f_n>) = <f_1, \ldots, f_n, f>$$

$-top(<f_1,\ldots,f_n>) = f_n$.

The third component serves as an object counter. We need it in order to give unique names to newly created objects. The fourth component will be used to store for each object its class name, which indicates the class definition that has been used for its creation.

(3) When we compare this definition to its counterpart in [ABKR], we observe that the two components of a state used there are the same as the first and second component of a state according to our new definition. Our third state component was not needed then. Our current fourth component was already present in the former paper. However, there it was called a type function.

4.2 The domain and the merge operator

In order to give a meaning to expressions, statements and units we shall define a mathematical domain P of *processes.* It will be a complete ultra-metric space that satisfies the following equation:

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

where

$$Step_P = (\Sigma \times P) \cup^1 Send_P \cup^1 Answer_P,$$

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

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

Before we present the formal definition of P let us first try to explain intuitively the intended interpretation of this domain.

A process p is either p_0 or of the form $\lambda\sigma{\cdot}X$. The process p_0 is the terminated process. Suppose $p = \lambda\sigma{\cdot}X$. Depending on the state σ, process p has the choice among the *steps* in the set X. Each step $x \in X$ 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 $x \in Step_P$.

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

$$x = <\sigma', p'>.$$

The only action is a change of state: σ' is the new state. The resumption again is a process, called p'. (When $p' = p_0$ no steps can be taken after this step x.)

Secondly, x might be a *send-step,* $x \in Send_P$. In this case we have, say

$$x = <\alpha, m, \bar{\beta}, f>,$$

with $\alpha \in Obj, m \in MName, \bar{\beta} \in Obj^*$ and $f \in (Obj \to P)$. The action involved here consists of the registration of an attempt at communication, in which a message is sent to the object α, specifying method m, together with parameters $\bar{\beta}$. This is the interpretation of the first three components α, m and $\bar{\beta}$. The last component f gives us the resumption of this send-step, when applied to the result of the message.

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

$$x = <\alpha, m, g>$$

with $\alpha \in Obj, m \in MName, g \in (Obj^* \to (Obj \to P) \to^1 P)$. It is then called an *answer-step.* The first two components of x express that the object α is willing to accept a message that specifies the method m. The last component g is the resumption of this answer-step. The function g is applied to the parameters in the message and to the (parameterized) 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.*

(A short remark on the "attenuation factors" above the union signs (cf. definition 2.6 and proposition 2.7). The factor $\frac{1}{2}$ in the first union is necessary to get a *contracting* functor F of which P is a fixed point (see definition 4.3 and lemma 4.4). Here any ϵ with $0 < \epsilon < 1$ would do. On the other hand, the

factor in the second union should not be less than 1, because otherwise the operator $|_\sigma$ in definition 4.6 is not NDI anymore.)

Let us now proceed with the formal definition of P. We define a contracting functor $F:\mathcal{C}\to\mathcal{C}$, where from here on, $\mathcal{C}$ denotes the category of complete ultra-metric spaces. Then we apply the fixed point theorem of section 2 (theorem 2.18).

DEFINITION 4.3
Let F, a functor $F:\mathcal{C}\to\mathcal{C}$, be defined by the following clauses.
(a) Let $<P,d>$ be a complete ultra-metric space.
$FP =^{def} \{p_0\} \cup \frac{1}{2}(\Sigma\to\mathcal{P}_{cl}(Step_P))$
with $Step_P$ as defined above.
$Fd:FP\times FP\to[0,1]$
is the ultra-metric on FP induced by the ultra-metric d on P, the discrete metrics on $\Sigma, Obj, MName$ and Obj^* and the canonic definitions of a metric on a function space, a disjoint union of spaces, a product space and a set of closed subspaces (see definition 2.6).
(b) Suppose $P\to'Q\in\mathcal{C}$ with $\iota=<i,j>$. We have to define $F\iota=<Fi,Fj>$ such that $FP\to^{F\iota}FQ\in\mathcal{C}$.
The function $Fi:FP\to FQ$ is defined by: $Fi(p_0)=p_0$ and for all $f\in\Sigma\to\mathcal{P}_{cl}(Step_P)$:

$$Fi(f) = \lambda\sigma\cdot(\{<\sigma',i(p)>|<\sigma',p>\in f(\sigma)\}\cup$$
$$\{<\alpha,m,\bar{\beta},i\circ g>|<\alpha,m,\bar{\beta},g>\in f(\sigma)\}\cup$$
$$\{<\alpha,m,\lambda\bar{\alpha}\in Obj^*\cdot\lambda h\in(Obj\to Q)\cdot i(g(\bar{\alpha})(j\circ h))>|<\alpha,m,g>\in f(\sigma)\}).$$

Note that $Fi(f)(\sigma)$ is a closed set and that $\lambda h\cdot i\circ g(\bar{\alpha})(j\circ h)\in(Obj\to Q)\to^1 Q$, both necessary conditions to ensure that $Fi(f)\in FQ$.
The second component of $F\iota$, the function $Fj:FQ\to FP$ is defined as follows: $Fj(p_0)=p_0$ and for all $f\in\Sigma\to\mathcal{P}_{cl}(Step_Q)$:

$$Fj(f) = \lambda\sigma\cdot \text{closure}(\{<\sigma',j(q)>|<\sigma',q>\in f(\sigma)\}\cup$$
$$\{<\alpha,m,\bar{\beta},j\circ g>|<\alpha,m,\bar{\beta},g>\in f(\sigma)\}\cup$$
$$\{<\alpha,m,\lambda\bar{\alpha}\in Obj^*\cdot\lambda h\in(Obj\to P)\cdot j(g(\alpha)(i\circ h))>|<\alpha,m,g>\in f(\sigma)\}).$$

Note that we have in this case ensured that $Fj(f)(\sigma)$ is a closed set by taking the closure of the set above. In general this is necessary because j need not to be a closed mapping.

LEMMA 4.4
Let F be as in definition 4.3. We have:
(a) $F:\mathcal{C}\to\mathcal{C}$
(b) *F is contracting.*

For a proof, see the appendix. Now we are able to apply theorem 2.18, which requires a contracting functor $F:\mathcal{C}\to\mathcal{C}$.

DEFINITION 4.5 (Domain P)
Let F be as in definition 4.3.
Let $\iota_0=<i_0,j_0>$ be given by :

$$i_0:\{p_0\}\to F\{p_0\}, j_0:F\{p_0\}\to\{p_0\},$$
$$i_0(p_0)=p_0,$$
$$j_0(p)=p_0, \text{ for all } p\in F\{p_0\}.$$

We have: $\{p_0\} \to^{\iota_0} F\{p_0\} \in \mathcal{C}$. Let P be defined as the inverse limit of the tower $(F^n\{p_0\}, F^n\iota_0)_n$.
By theorem 2.18 we have

$$P \cong \{p_0\} \cup^{\frac{1}{2}} \mathcal{P}_{cl}(Step_P)$$

where

$$Step_P = (\Sigma \times P) \cup^1 Send_P \cup^1 Answer_P,$$
$$Send_P = Obj \times MName \times Obj^* \times (Obj \to P),$$
$$Answer_P = Obj \times MName \times (Obj^* \to (Obj \to P) \to^1 P).$$

We now define an operator for the *parallel composition* (or the *merge*) of two processes, for which we shall use the symbol $\|$.
As we intend to model parallel composition by interleaving, the merge of two processes p and q will consist of three parts. The first part contains all possible first steps of p followed by the parallel composition of its resumption with q. The second part contains similarly the first steps of q. The last part contains the communication steps, that result from two matching communication steps taken simultaneously by process p and q. So our merge operator

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

should satisfy an equation of the following type:

$$p \| q = \lambda\sigma.(\{x \| q : x \in p(\sigma)\} \cup \{x \| p : x \in q(\sigma)\} \cup \bigcup \{x|_a y : x \in p(\sigma), y \in q(\sigma)\})$$

for all $p,q \in P \setminus \{p_0\}$, where the three sets of which $p \| q$ is composed should be defined such that they correspond with the parts described above. When we specify this equation further by defining

$$<\sigma', p'> \| q = <\sigma', p' \| q>$$

(thus treating one case of $x \| q$), the recursive character of the equation becomes clear.
One might solve this equation by defining $\|$ with induction on the complexity of the processes p and q. This would involve the details of the construction of P. (For several examples see [BZ].) We choose another approach by defining $\|$ as the fixed point of a contracting higher-order function on the set of binary operators on P.

DEFINITION 4.6
We define a function

$$\Phi_{PC} : (P \times P \to^1 P) \to (P \times P \to^1 P)$$

as follows. Let $\odot \in P \times P \to^1 P$, let $\tilde{\odot} =^{def} \Phi_{PC}(\odot)$. For $p,q \in P$ we define

$$p \tilde{\odot} q = \begin{cases} p & \text{if } q = p_0 \\ q & \text{if } p = p_0 \\ \lambda\sigma.\ (\{x \tilde{\odot} q : x \in p(\sigma)\} \cup \{x \tilde{\odot} p : x \in q(\sigma)\} \cup \bigcup \{x|_a y : x \in p(\sigma), y \in q(\sigma)\}) & \text{otherwise.} \end{cases}$$

For $x \in Step_P$ we distinguish three cases.

(i) $<\sigma',p'>\tilde{\odot}q = <\sigma',p'\odot q>$
(ii) $<\alpha,m,\bar{\beta},f>\tilde{\odot}q = <\alpha,m,\bar{\beta},\lambda\beta\cdot(f(\beta)\odot q)>$
(iii) $<\alpha,m,g>\tilde{\odot}q = <\alpha,m,\lambda\bar{\beta}\cdot\lambda h\cdot(g(\bar{\beta})(h)\odot q)>$.

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

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

LEMMA 4.7
Let Φ_{PC} be as in definition 4.6. We have
(a) *Φ_{PC} is well defined, that is: $\Phi_{PC}(\odot) \in P\times P \to^1 P$ for all $\odot \in P\times P\to^1 P$.*
(b) *Φ_{PC} is a contraction.*

For the proof see the appendix.

DEFINITION 4.8 (Parallel composition)
$\| =^{def}$ Fixed Point (Φ_{PC}).

4.3 Semantics of statements and expressions

In this section we define the semantics of statements by specifying a function $[\![\cdots]\!]_S$ of the following 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 set of environments is given in definition 4.9 below.
The second parameter of $[\![s]\!]_S$, the object α, represents the object that is currently executing 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 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

$$Cont_E = Obj \to P$$

is the set of continuations of expressions. 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 of the expression, will yield the process that is to be executed after the evaluation of the expression.
Another advantage of the use of continuations lies in the treatment of the new-expression, the POOL construct for process creation. This will be elucidated below.

DEFINITION 4.9 (Environments)
The set of environments is defined as follows.

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

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 (of classes and methods) that are present in U. It will be needed in the computation of the semantics of U. In general, the first component γ_1 of an environment γ is a function, which gives the denotational value of the body of a class definition C by: $\gamma_1(C)(\alpha)$, supposed that this body is executed by an object α. We shall need this first component when we want to define the semantics of a new-expression.
When we supply γ_2 with arguments $m, C, \alpha, \overline{\beta}$ and f, the value of the body of method m (that has been defined in the class definition of C) will be presented as: $\gamma_2(m)(C)(\alpha)(\overline{\beta})(f)$. Here α is again the object that is currently executing this body and $\overline{\beta}$ is a (possibly empty) sequence of objects that are the parameters of m. The function f is the continuation of m.

DEFINITION 4.10 (Semantics of expressions)
We define a function

$$[\![\cdots]\!]_E : Exp \to Env \to AObj \to (Obj \to P) \to {}^1P$$

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

(E1, instance variable)

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

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 supplied with the resulting value.

(E2, local variable)

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

The value of u is looked up in the top frame of the stack $\sigma_2(\alpha)$.

(E3, send-expression)

$$[\![e!m(e_1, \ldots, e_n)]\!]_E(\gamma)(\alpha)(f) =$$

$$[\![e]\!]_E(\gamma)(\alpha)($$
$$\lambda\beta\cdot([\![e_1]\!]_E(\gamma)(\alpha)($$
$$\lambda\beta_1\cdot([\![e_2]\!]_E(\gamma)(\alpha)($$
$$\ldots$$
$$\lambda\beta_{n-1}\cdot([\![e_n]\!]_E(\gamma)(\alpha)($$
$$\lambda\beta_n\cdot\lambda\sigma\cdot\{<\beta,m,\overline{\beta},f>\}))\cdots)))))$$

where

$$\overline{\beta} = <\beta_1, \ldots, \beta_n>.$$

The expressions $e, e_1, \ldots, e_n$ are evaluated from left to right. Their results correspond to the formal parameters $\beta, \beta_1, \ldots, \beta_n$ of their respective continuations. Object β represents the name of the object to which the message is sent. The sequence $<\beta_1, \ldots, \beta_n>$ represents the parameters for the execution of method m.
Besides these values and the method name m the final step $<\beta, m, \overline{\beta}, f>$ also contains the continuation f of the send-expression. If the attempt at communication succeeds, this continuation will be supplied with the result of the method execution. (See section 4.2.)

(E4, method call)

$$[\![m(e_1, \ldots, e_n)]\!]_E(\gamma)(\alpha)(f) =$$
$$[\![e_1]\!]_E(\gamma)(\alpha)($$
$$\lambda\beta_1\cdot([\![e_2]\!]_E(\gamma)(\alpha)($$
$$\ldots$$
$$\lambda\beta_{n-1}\cdot([\![e_n]\!]_E(\gamma)(\alpha)($$
$$\lambda\beta_n\cdot\lambda\sigma\cdot\{<\sigma,\gamma_2(m)(\sigma_4(\alpha))(\alpha)(\overline{\beta})(f)>\}))\cdots)))$$

where

$$\overline{\beta} = <\beta_1, \ldots, \beta_n>.$$

Here the final step is not a communication step. It represents the execution of the method m when supplied with parameters $\overline{\beta}$.

(E5, new-expression)

$$[\![\textbf{new}(C)]\!]_E(\gamma)(\alpha)(f) = \lambda\sigma\cdot\{<\sigma', \gamma_1(C)(\hat{\sigma}_3) \| f(\hat{\sigma}_3)>\}$$

where

$$\sigma' = <\sigma_1\{\lambda x\cdot nil/\hat{\sigma}_3\}, \sigma_2, \sigma_3 + 1, \sigma_4\{C/\hat{\sigma}_3\}>.$$

A new object is created, its name is the current value of the object counter σ_3 supplied with a festive hat on the occasion of this happy event. The hat (furthermore) serves the purpose of distinguishing this new *active* object $\hat{\sigma}_3$ from the *standard* object σ_3 (which is a natural number).
The new activity, as given by $\gamma(C)(\hat{\sigma}_3)$, is composed in parallel with the continuation f supplied with $\hat{\sigma}_3$, the value of the new-expression.
We observe that we are able to perform this parallel composition because we know from f what should happen after the evaluation of this new-expression.

(E6)

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

The definition of $[\![\cdots]\!]_S$ is given below in definition 4.11. Lemma 4.12 states that $[\![\cdots]\!]_E$ and $[\![\cdots]\!]_S$ are well defined, although their definitions refer to each other.

(E7, **self**)

$$[\![\textbf{self}]\!]_E(\gamma)(\alpha)(f) = \lambda\sigma\cdot\{<\sigma, f(\alpha)>\}.$$

The continuation f is supplied with the value of the expression **self**, that is the name of the object executing this expression.

(E8, standard objects)

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

We use $f(\phi)$ instead of $\lambda\sigma\cdot\{<\sigma, f(\phi)>\}$ in this definition, wishing to express that the value of a standard object is immediately present: it does not take a step to evaluate ϕ.

DEFINITION 4.11 (Semantics of statements)
The function

$$[\![\cdots]\!]_S : Stat \rightarrow Env \rightarrow AObj \rightarrow P \rightarrow {}^1P$$

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

(S1, assignment to an instance variable)

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

where

$$\sigma' = <\sigma_1\{(\sigma_1(\alpha)\{\beta/x\})/\alpha\}, \sigma_2, \sigma_3, \sigma_4>.$$

The expression e is evaluated, the result assigned to x.

(S2, assignment to a local variable)

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

where

$$\sigma' = <\sigma_1, \sigma_2\{S/\alpha\}, \sigma_3, \sigma_4>,$$
$$\sigma_2(\alpha) = <f_1, \ldots, f_n>,$$
$$S = <f_1, \ldots, f_{n-1}, f_n\{\beta/u\}>.$$

(S3, answer-statement)

$$[\![\textbf{answer}(m_1, \ldots, m_n)]\!]_S(\gamma)(\alpha)(p) = \lambda\sigma\cdot\{<\alpha, m_i, g_i> | 1 \leqslant i \leqslant n\}$$

where for $1 \leqslant i \leqslant n$

$$g_i = \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \rightarrow P \cdot \gamma_2(m_i)\ (\sigma_4(\alpha))\ (\alpha)\ (\bar{\beta})\ (\lambda\beta\cdot(f(\beta)\|p)).$$

For each method m_i the function g_i represents its execution followed by its continuation. In the definition of g_i the second component of environment γ is supplied with arguments m_i, the classname $\sigma_4(\alpha)$ and α.

This function g_i expects some parameters $\bar{\beta}$ and some continuation f, both to be received from an object sending a message specifying the method m_i while offering a sequence of parameters $\bar{\beta}$ and its own continuation f. After the execution of the method both the continuations of the requesting object and object α are to be executed in parallel. So the final argument γ_2 is supplied with is :

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

REMARK
Now that we have defined the semantics of send-expressions and answer-statements let us briefly return to the definition of $x|_\sigma y$ (definition 4.6).
Let $x=<\alpha,m,\bar{\beta},f>$ (the result from the elaboration of a send-expression) and $y=<\alpha,m,g>$ (resulting from an answer-statement). Then $x|_\sigma y$ is defined as

$$x|_\sigma y = \{<\sigma,g(\bar{\beta})(f)>\}.$$

Now that we know how g is defined we have

$$g(\bar{\beta})(f) = \gamma_2(m)(\sigma_4(\alpha))(\alpha)(\bar{\beta})(\lambda\beta\cdot(f(\beta)\|p)).$$

We observe that the continuation of the execution of m is given by $\lambda\beta\cdot(f(\beta)\|p)$, the parallel composition of the (local) continuations f and p. Moreover we note that the result of the method execution, as parameterized by β, is passed on to the continuation of the send-expression f.

(S4, expressions as statements)

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

Expressions may be used for their side effects only. 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, if-statement)

$$[\![\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}]\!]_S(\gamma)(\alpha)(p) =$$

$$[\![e]\!]_E(\gamma)(\alpha)(\lambda\beta\cdot(\text{if } \beta = tt \text{ then } [\![s_1]\!]_S(\gamma)(\alpha)(p) \text{ else } [\![s_2]\!]_S(\gamma)(\alpha)(p) \text{ fi})).$$

(S7, do-statement)

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

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

$$\Phi(q) = [\![e]\!]_E(\gamma)(\alpha)(\lambda\beta\cdot\lambda\sigma\cdot\{<\sigma, \text{ if } \beta = tt \text{ then } [\![s]\!]_S(\gamma)(\alpha)(q) \text{ else } p \text{ fi}>\}).$$

We shall show below (lemma 4.12(b)) that Φ is contracting.

(S8, select-statement)
(a) Evaluation of the guards

$$[\![\textbf{sel (answer } V_1 \textbf{ then } s_1) \textbf{ or } \cdots \textbf{ or (answer } V_{k-1} \textbf{ then } s_{k-1}) \textbf{ or}$$
$$(e_k \textbf{ answer } V_K \textbf{ then } s_k) \textbf{ or } g_{k+1} \textbf{ or } \cdots \textbf{ or } g_n \textbf{ les}]\!]_S(\gamma)(\alpha)(p) =$$
$$[\![e_k]\!]_E(\gamma)(\alpha)(\lambda\beta\cdot(\textbf{if } \beta = tt$$
$$\textbf{then } [\![\textbf{ sel } \cdots \textbf{ or (answer } V_k \textbf{ then } s_k) \textbf{ or } g_{k+1} \textit{ or } \cdots \textbf{ les }]\!](\gamma)(\alpha)(p)$$
$$\textbf{else } [\![\textbf{ sel } \cdots \textbf{ or (answer } V_{k-1} \textbf{ then } s_{k-1}) \textbf{ or } g_{k+1} \cdots \textbf{ les }]\!]_S(\gamma)(\alpha)(p)$$
$$\textbf{fi }))$$

(b) Select-statement without guards
Let the statement s be defined by

$$s \equiv^{def} \textbf{sel (answer } V_1 \textbf{ then } s_1) \textbf{ or } \cdots \textbf{ or (answer } V_n \textbf{ then } s_n) \textbf{ les,}$$

let $\alpha \in AObj, \gamma \in Env, p \in P$. We want to define $[\![s]\!]_S(\gamma)(\alpha)(p)$. Before we can give the semantics of s we remove from the method sets V_i those methods that can never be selected for communication. Let

$$<W_1, \ldots, W_n> =^{def} <V_1, V_2 - V_1, \ldots, V_n - (V_1 \cup \cdots \cup V_{n-1})>.$$

The methods in s have a priority order from left to right. Therefore, when a method m occurs in a certain guarded command, all occurrences of m in guarded commands to the right of this command can be discarded.
For every non-empty W_k and $m \in W_k$ we define

$$g_{m,k} = \lambda\overline{\beta} \in Obj \cdot \lambda f \in Obj \to P \cdot$$
$$\gamma_2(m)(\sigma_4(\alpha))(\alpha)(\overline{\beta})(\lambda\beta\cdot(f(\beta)\|([\![s_k]\!]_S(\gamma)(\alpha)(p)))).$$

This function $g_{m,k}$ expresses the meaning of method m. The only difference with the function g_m used in the definition of the answer-statement (S3 above) is that the local continuation of 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 guarded command g_k followed by p, the continuation of the entire select-statement s.
Because a guarded command with an empty message set has priority over all the guarded commands to its right, we distinguish the following two cases.
(1) $V_j = \emptyset$ for some j, $1 \leqslant j \leqslant n$ and $\forall 1 \leqslant i < j [V_i \neq \emptyset]$:

$$[\![s]\!]_S(\gamma)(\alpha)(p) =^{def} \lambda\sigma\cdot(\{<\sigma, [\![s_j]\!]_S(\gamma)(\alpha)(p)>\} \cup \{<\alpha, m, g_{m,k}> | m \in W_k, 1 \leqslant k < j\}).$$

(2) $\forall 1 \leqslant j \leqslant n [V_j \neq \emptyset]$:

$$[\![s]\!]_S(\gamma)(\alpha)(p) =^{def} \lambda\sigma\cdot\{<\alpha, m, g_{m,k}> | m \in W_k, 1 \leqslant k \leqslant n\}.$$

We note that according to these definitions a method that occurs to the right of an empty message set will never be selected.

LEMMA 4.12
The semantic functions $[\![\cdots]\!]_E$ *and* $[\![\cdots]\!]_S$ *of definitions 4.10 and 4.11 are well defined. That is*
(a) *For all* $e \in Exp, s \in Stat, \gamma \in Env, \alpha \in AObj$:

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

(b) *The function $\Phi: P \to P$ used in definition 4.11 (S7) is contracting.*

For the proof see the appendix.

4.4 Standard objects

DEFINITION 4.13 (Integers)
We define a process P_{INT}, that represents the activity of all integer objects, as the limit of the following sequence of processes.

$$\begin{aligned} &(0)\quad Q_0 = p_0 \\ &(k+1)\quad Q_{k+1} = \lambda\sigma\cdot\ (\{<n, \mathsf{add}, g_n> \mid n \in \mathbb{Z}\} \cup \\ &\qquad\qquad\qquad\qquad \{<n, \mathsf{sub}, \tilde{g}_n> \mid n \in \mathbb{Z}\}) \end{aligned}$$

where

$$\begin{aligned} g_n &= \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to P \cdot \\ &\quad (\text{if } \bar{\beta} \in \mathbb{N}^1 \text{ then } f(n + (\bar{\beta})_1) \| Q_k \text{ else } p_0 \text{ fi}), \\ \tilde{g}_n &= \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to P \cdot \\ &\quad (\text{if } \bar{\beta} \in \mathbb{N}^1 \text{ then } f(n - (\bar{\beta})_1) \| Q_k \text{ else } p_0 \text{ fi}). \end{aligned}$$

We define

$$P_{\mathrm{INT}} = \lim_{k \to \infty} Q_k.$$

REMARKS
(1) The limit of $(Q_k)_k$ exists because we have

$$\forall k \geqslant 1 [d(Q_k, Q_{k+1}) \leqslant (\tfrac{1}{2})^k].$$

(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 $\mathcal{P}_{cl}(\cdots)$ (closed subsets) in our domain equation rather then $\mathcal{P}_{comp}(\cdots)$ (compact subsets).

DEFINITION 4.14 (Booleans)
The process that represents the booleans *tt* and *ff* is defined as the limit of the following sequence.

$$\begin{aligned} &(0)\quad S_0 = p_0 \\ &(k+1)\quad S_{k+1} = \lambda\sigma\cdot\ (\{<b, \mathsf{and}, g_b> \mid b \in \{tt, ff\}\} \cup \\ &\qquad\qquad\qquad\qquad \{<b, \mathsf{or}, \tilde{g}_b> \mid b \in \{tt, ff\}\} \cup \\ &\qquad\qquad\qquad\qquad \{<b, \mathsf{not}, \bar{g}_b> \mid b \in \{tt, ff\}\}) \end{aligned}$$

where

$$\begin{aligned} g_b &= \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to P \cdot \\ &\quad (\text{if } \bar{\beta} \in \{tt, ff\}^1 \text{ then } f(b \wedge (\bar{\beta})_1) \| S_k \text{ else } p_0 \text{ fi}) \\ \tilde{g}_b &= \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to P \cdot \\ &\quad (\text{if } \bar{\beta} \in \{tt, ff\}^1 \text{ then } f(b \vee (\bar{\beta})_1) \| S_k \text{ else } p_0 \text{ fi}) \end{aligned}$$

$$\bar{g}_b = \lambda\bar{\beta}\in Obj^*\cdot\lambda f\in Obj\to P\cdot$$
$$\text{(if } \bar{\beta} = \epsilon \text{ then } f(\neg b)\|S_k \text{ else } p_0 \text{ fi)}$$

We define

$$P_{\text{BOOL}} = \lim_{k\to\infty} S_k.$$

REMARK
We have

$$\forall k \geqslant 1[d(S_k, S_{k+1}) \leqslant (\tfrac{1}{2})^k],$$

so P_{BOOL} is well defined.

DEFINITION 4.15 (Standard objects)
We define one process for all our standard objects:

$$P_{\text{STANDARD}} = {}^{def} P_{\text{INT}}\|P_{\text{BOOL}}.$$

EXAMPLE 4.16
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_{STANDARD}. An example may illustrate how communication with a standard object proceeds.
We determine

$$[\![x\leftarrow(2!\text{add}(3))]\!]_S(\gamma)(\alpha)(p_0)\|P_{\text{STANDARD}}$$

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

$[\![x\leftarrow(2!\text{add}(3))]\!]_S(\gamma)(\alpha)(p_0) =$

$[\![2!\text{add}(3)]\!]_E(\gamma)(\alpha)(f)$

[where $f = \lambda\beta\cdot\lambda\sigma'\cdot\{<\sigma'', p_0>\}$ with $\sigma'' = <\sigma'_1\{(\sigma_1(\alpha)\{\beta / x\} / \alpha\}, \sigma'_2, \sigma'_3, \sigma'_4>] =$

$[\![2]\!]_E(\gamma)(\alpha)(\lambda\beta_1\cdot([\![3]\!]_E(\gamma)(\alpha)(\lambda\beta_2\cdot\lambda\sigma\cdot\{<\beta_1, \text{add}, f>\}))) =$

$[\![3]\!]_E(\gamma)(\alpha)(\lambda\beta_2\cdot\lambda\sigma\cdot\{<2, \text{add}, \beta_2, f>\}) =$

$\lambda\sigma\cdot\{<2, \text{add}, 3, f>\}.$

Now the parallel composition:

$\lambda\sigma\cdot\{<2, \text{add}, 3, f>\}\|P_{\text{STANDARD}} =$

$\lambda\sigma\cdot\{<2, \text{add}, 3, f>\}\|\lambda\sigma'\cdot\{\cdots, <2, \text{add}, g_2>, \cdots\}\|P_{\text{BOOL}}$

[where $g_2 = \lambda\bar{\beta}\in Obj^*\cdot\lambda f\in Obj\to P\cdot$(if $\bar{\beta}\in\mathbb{N}^1$ then $f(2 + (\bar{\beta})_1)\|P_{\text{INT}}$ else p_0 fi)] =

$\lambda\sigma\cdot\{<2, \text{add}, 3, f>|_\sigma <2, \text{add}, g_2>\}\|P_{\text{BOOL}}$

[where all steps have been omitted but for the successful communication step] =

$\lambda\sigma\cdot\{<\sigma, g_2(3)(f)>\}\|P_{\text{BOOL}} =$

$\lambda\sigma\cdot\{<\sigma, f(5)\|P_{\text{INT}}>\}\|P_{\text{BOOL}} =$

$\lambda\sigma\cdot\{<\sigma, (\lambda\sigma'\cdot\{<\sigma'', p_0>\})\|P_{\text{INT}}>\}\|P_{\text{BOOL}}$

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

4.5 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 the following contracting function.

DEFINITION 4.17

Let *Env* be the set of environments as defined in definition 4.9. Thus

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

For every $U \in$ Unit, we define a function

$$\Phi_U : Env \to Env$$

as follows. Let $\gamma \in Env, \gamma = <\gamma_1, \gamma_2>$. Let $\tilde{\gamma} =^{def} \Phi_U(\gamma)$ be given by

(a) $\tilde{\gamma}_1 = \lambda C \cdot \lambda \alpha \cdot [\![s]\!]_S(\gamma)(\alpha)(p_0)$ where

$$U = < \cdots, C \Leftarrow d, \cdots >,$$
$$d = < \cdots, s >.$$

(b) $\tilde{\gamma}_2 = \lambda m \cdot \lambda C \cdot \lambda \alpha \cdot \lambda \bar{\beta} \cdot \lambda f \cdot$
if $lenght(\bar{\beta}) = n$
then $\lambda \sigma \cdot \{<\sigma', [\![e]\!]_E(\gamma)(\alpha)(\lambda \beta \cdot \lambda \bar{\sigma} \cdot \{<\bar{\sigma}', f(\beta)\})>\}$
else p_0
fi

where

$$U = < \cdots, C \Leftarrow d, \cdots >,$$
$$d = < \cdots, (\cdots, m \Leftarrow \mu, \cdots), \cdots >,$$
$$\mu = <(u_1, \ldots, u_n), (v_1, \ldots, v_k), e>,$$
$$\sigma' = <\sigma_1, \sigma_2\{push\ (h, \sigma_2(\alpha))/\alpha\}, \sigma_3, \sigma_4>,$$
$$\bar{\beta} = <\beta_1, \ldots, \beta_n>,$$
$$h(u_i) = \beta_i \text{ for } i = 1, \ldots, n,$$
$$h(v_j) = nil \text{ for } j = 1, \ldots, k,$$
$$\bar{\sigma}' = <\bar{\sigma}_1, \bar{\sigma}_2\{pop(\sigma_2(\alpha))/\alpha\}, \bar{\sigma}_3, \bar{\sigma}_4>.$$

REMARK

The function Φ_U takes an environment γ as an input. The output $\tilde{\gamma}$ generally contains more information about the unit U. The definition of $\tilde{\gamma}_1$ is hopefully clear. The second component $\tilde{\gamma}_2$ needs some further explanation.

In short the execution of a method m amounts to the execution of the expression e, which is given by the definition μ of m. This execution of e is preceded by a stack operation *(push)* upon $\sigma_2(\alpha)$, which initializes the parameters and local variables of m. After the execution of e the topframe of the stack $\bar{\sigma}_2(\alpha)$ is popped again. The final resumption is given by $f(\beta)$, where β stands for the resulting value of e.

LEMMA 4.18
Let $U \in Unit$, let Φ_U be defined as in 4.17. Then: Φ_U is a contraction.

For the proof see the appendix.

DEFINITION 4.19
Let $U \in Unit$, let Φ_U be as in 4.17. We define

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

4.5.2 Paths and yield
The following definition introduces the notion of *paths*. Given a process p_1 and a state σ_1, we want to consider computation sequences starting from $<\sigma_1,p_1>$.

DEFINITION 4.20 (Paths)
A finite or infinite sequence $(<\sigma_i,p_i>)_i$ with $\sigma_i \in \Sigma, p_i \in P$, is called a *path* (starting from $<\sigma_1,p_1>$) whenever
(a) $\forall j \geq 1\ [j < |(<\sigma_i,p_i>)_i| \Rightarrow <\sigma_{j+1},p_{j+1}> \in p_j(\sigma_j)]$
(b) The sequence is either infinite or, when finite, terminates with $<\sigma_n,p_0>$ for some $n \geq 1$.
The set of all paths we shall call *Path*.

REMARKS
(1) A path $(<p_i,\sigma_i>)_i$ represents a particular execution of the process p_1 starting from the state σ_1. In every component $<\sigma_n,p_n>$ of a path starting in $<\sigma_1,p_1>$, the state σ_n is passed on to the resumption process p_n. Those paths that terminate with a component $<\sigma_n,p_0>$, present us with terminal states σ_n.
(2) We observe that in general a set $p_n(\sigma_n)$ may contain elements of $Send_P$ or $Answer_P$, besides elements of $\Sigma \times P$. When we assume that the process p_n will not be composed in parallel with some other process, we may consider such elements as unsuccessful attempts at communication. Therefore we do not want to incorporate them in our definition of paths.

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 $<\sigma,p>$.

DEFINITION 4.21 (Yield)
The function $yield : P \to \Sigma \to \mathcal{P}(Path)$ is defined as follows. Let $p \in P, \sigma \in \Sigma$. Then

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

4.5.3 Semantics of a unit
The execution of a unit U with

$$U = <C_1 \Leftarrow d_1, \ldots, C_n \Leftarrow d_n>$$

consists of the creation of an object of class C_n (by convention) and the execution of its body. This object initiates the program by creating new objects and putting them to work. Let d_n be given by

$$d_n = <\cdots, s>.$$

Then we can give the semantics of U in terms of s and a suitable environment γ_U.
We also need an initial state for every U.

DEFINITION 4.22 (Initial state)
Let $U \in Unit$. We define an initial stat σ_U for U by

$$(\sigma_U)_1 = \lambda\alpha{\cdot}\lambda x{\cdot}nil$$

$$(\sigma_U)_2 = \lambda\alpha{\cdot}\epsilon$$

$$(\sigma_U)_3 = 2$$

$$(\sigma_U)_4 = \lambda\alpha{\cdot}C_n.$$

REMARK
The only values of σ_U that are of importance are $(\sigma_U)_1(\hat{1})$, $(\sigma_U)_2(\hat{1})$,$(\sigma_U)_3$ and the value of $(\sigma_U)_4(\hat{1})$. The others are to be considered default values.

Finally we are able to define the semantics of units.

DEFINITION 4.23 (Semantics of units)
We define a function

$$\mathfrak{D} : Unit \rightarrow \mathcal{P}(Path)$$

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

$$\mathfrak{D}[\![U]\!] = yield\ ([\![s]\!]_S(\gamma_U)(\hat{1})(p_0) \| P_{\text{STANDARD}})\ (\sigma_U)$$

where $U = < \cdots, C_n \Leftarrow < \cdots, s >>$, γ_U as defined in 4.19, σ_U as defined in 4.22.

REMARK
The standard objects are represented by P_{STANDARD}. They are assumed to be present at the execution of every unit U. Therefore they are composed in parallel together with $[\![s]\!]_S(\gamma_U)(\hat{1})(p_0)$.

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 adaption 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. Formally we should change the definition of our contracting functor F as defined in definition 4.3. However we only give the adapted equation that must be satisfied and forget about the tedious details of how to solve it.

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

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

where

$$Step_P = Comp_P \cup^1 Send_P \cup^1 Answer_P,$$
$$Comp_P = \Lambda \times \Sigma \times P \text{ (the set of } computation \text{ steps)},$$
$$Send_P = Obj \times Obj \times MName \times Obj^* \times (Obj \to P),$$
$$Answer_P = Obj \times MName \times (Obj^* \to (Obj \to P) \to^1 P).$$

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
We define a function

$$\Phi_{PC} : (P \times P \to^1 P) \to (P \times P \to^1 P)$$

as follows. Let $\odot \in P \times P \to^1 P$, let $\tilde{\odot} =^{def} \Phi_{PC}(\odot)$. For $p,q \in P$ we define

$$p \tilde{\odot} q = \begin{cases} p & \text{if } q = p_0 \\ q & \text{if } p = p_0 \\ \lambda\sigma \cdot \; (\{x \tilde{\odot} q : x \in p(\sigma)\} \cup & \text{otherwise.} \\ \quad \{x \tilde{\odot} p : x \in q(\sigma)\} \cup & \\ \quad \bigcup \{x|_\sigma y : x \in p(\sigma), y \in q(\sigma)\}) & \end{cases}$$

For $x \in Step_P$ we distinguish three cases.
(i) $<\kappa,\sigma',p'> \tilde{\odot} q = <\kappa,\sigma',p' \odot q>$
(ii) $<\alpha,\beta,m,\bar{\beta},f> \tilde{\odot} q = <\alpha,\beta,m,\bar{\beta},\lambda\beta'\cdot(f(\beta') \odot q)>$
(iii) $<\alpha,m,g> \odot q = <\alpha,m,\lambda\beta\cdot\lambda h\cdot(g(\beta)(h) \odot q)>$.
Finally the set of successful communications between two processes is defined as follows. Let $x,y \in Step_P$. We have

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

The new merge operator is defined as the fixed point of Φ_{PC} (c.f. lemma 4.7 and definition 4.8).

The definition of a *path* has to be altered straightforwardly. Finally the definition of $[\![\cdots]\!]_E$ and $[\![\cdots]\!]_S$ ought to be changed. We treat 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 4.10 and 4.11, but adapted straightforwardly as is illustrated by the following clause. Let $\alpha \in AObj, \gamma \in Env, f \in Obj \to P$. We define

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

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

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

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

(ii)

$$\exists\alpha\ \exists <i_0,i_1,\cdots>\ \exists\beta\ \exists m\ \exists\bar{\beta}$$
$$[\forall k \geqslant 0\ [1 \leqslant i_k < i_{k+1}]$$
$$\wedge \forall n \geqslant i_0\ \exists f\ [<\alpha,\beta,m,\bar{\beta},f> \in p_n(\sigma_n)]$$
$$\wedge \forall k \geqslant 1\ \exists g\ [<\beta,m,g> \in p_{i_k}(\sigma_{i_k})]$$
$$\wedge \forall n > i_0\ [\kappa_n \neq <\alpha,\beta>]].$$

(iii)

$$\exists\alpha\ \exists <i_0,i_1,\cdots>\ \exists m$$
$$[\forall k \geqslant 0\ [1 \leqslant i_k < i_{k+1}]$$
$$\wedge \forall n \geqslant i_0\ \exists g\ [<\alpha,m,g> \in p_n(\sigma_n)]$$
$$\wedge \forall k \geqslant 1\ \exists\beta\ \exists\bar{\beta}\ \exists f\ [<\beta,\alpha,m,\bar{\beta},f> \in p_{i_k}(\sigma_{i_k})]$$
$$\wedge \forall n > i_0\ \neg\exists\beta\ [\kappa_n = <\beta,\alpha>]].$$

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 $<\alpha,\sigma,p>$, where σ and p depend on moment n) but in this path 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, 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 $(<\kappa_i,\sigma_i,p_i>)_i$ is called *fair* if it is *not* unfair.

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

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

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

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

We conclude this section with a "fair" variant of definition 4.23.

DEFINITION 5.7 (Fair semantics of units)
Let $\mathcal{D}_{fair}: Unit \to \mathcal{P}(Path)$ be defined as follows. For $U \in Unit$ we have

$$\mathcal{D}_{fair}[\![U]\!] = fairyield\ ([\![s]\!]_S(\gamma_U)(\hat{1})(p_0) \| P_{\text{STANDARD}})\ (\sigma_U)(\hat{1})$$

where U, γ_U and σ_U are as in definition 4.23.

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 succeeded to give 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 statements).

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 object 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 approached 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 to 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 are being developed (see for example [AR]).

A remarkable point is that the mathematical techniques used to solve reflexive domain equations, which in [BZ] differed greatly from the ones used in the order-theoretic approach, have again converged to the latter in our work. They are compared more extensively in [AR].

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 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.

Secondly, 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 [ABKR]. An equivalence proof would, of course, be very desirable.

7. References

[Am1] Pierre America, *Definition of the Programming Language POOL-T,* ESPRIT project 415, Doc. No. 0091, Philips Research Laboratories, Eindhoven, September 1985.

[Am2] Pierre America, *Rationale for the Design of POOL,* ESPRIT project 415, Doc. No. 0053, Philips Research Laboratories, Eindhoven, January 1986.

[Am3] Pierre America, *POOL-T — A Parallel Object-Oriented Language,* in: Object-Oriented Concurrent Systems (Akinori Yonezawa, Mario Tokoro, eds.), MIT Press, 1986.

[Am4] Pierre America, *Object-Oriented Programming: A Theoretician's Introduction,* Bulletin of the EATCS 29, 1986, pp. 69-84.

[ABKR] Pierre America, Jaco de Bakker, Joost N. Kok, Jan Rutten, *Operational Semantics of a Parallel Object-Oriented Language,* 13th ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida, January 13-15, 1986, pp. 194-208.

[AR] Pierre America, Jan Rutten, *Solving Reflexive Domain Equations in a Category of Metric Spaces,* Technical Report, Centre for Mathematics and Computer Science, Amsterdam, to appear (1986).

[ANSI] *Reference Manual for the Ada Programming Language,* ANSI / MIL-STD 1815 A, United States Department of Defense, Washington D.C., January 1983.

[BKMOZ] J.W. de Bakker, J.N. Kok, J.-J. Ch. Meyer, E.-R. Olderog, J.I. Zucker, *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.), Springer, LNCS 224, 1986, pp. 51-121.

[BZ] J.W. de Bakker, J.I. Zucker, *Processes and the Denotational Semantics of Concurrency,* Information and Control 54 (1982), pp. 70-120.

[Br] A. de Bruin, *Experiments with Continuation Semantics: Jumps, Backtracking, Dynamic Networks,* dissertation, Free University of Amsterdam, 1986.

[Cl] William D. Clinger, *Foundations of Actor Semantics,* Ph. D. thesis, Massachusetts Institute of Technology (AI-TR-633), 1981.

[Du] J. Dugundji, *Topology,* Allen and Bacon, Rockleigh, N.J., 1966.

[En] R. Engelking, *General Topology,* Polish Scientific Publishers, 1977.

[Gi] G. Gierz et al., *A Compendium of Continuous Lattices,* Springer, 1980.

[GR] Adele Goldberg, David Robson, *Smalltalk-80, The Language and its Implementation,* Addison-Wesley, 1983.

[Ha] H. Hahn, *Reelle Funktionen,* Chelsea, New York, 1948.

[HP] M. Hennessy, G.D. Plotkin, *Full Abstraction for a Simple Parallel Programming Language,* Proc. 8th Symp. Mathematical Foundations of Computer Science, Springer, LNCS 74, 1979, pp. 108-120.

[He] C. Hewitt, *Viewing Control Structures as Patterns of Passing Messages,* Artificial Intelligence, 8, 1977, p.p. 323-364.

[ML] S. MacLane, *Categories for the Working Mathematician,* Springer, 1971.

[Va] Frits W. Vaandrager, *Process Algebra Semantics of POOL,* Technical Report, Centre for Mathematics and Computer Science, Amsterdam, 1986.

APPENDIX

LEMMA 4.4
Let F be as in definition 4.3. We have:
(a) $F:\mathcal{C}\to\mathcal{C}$
(b) F is contracting

PROOF.
(a) $F:\mathcal{C}\to\mathcal{C}$:
Let $P\to' Q\in\mathcal{C}$, with $\iota=<i,j>$. We show
(a1) for d an ultra-metric on P that
(FP,Fd) is a complete ultra-metric space,
(a2) Fi is isometric,
(a3) Fj is NDI,
(a4) $Fj\circ Fi=id_{FP}$.

(a1) $(FP,FD)\in\mathcal{C}$:
The fact that Fd is an ultra-metric, with respect to which FP is complete is straightforward from proposition 2.7.
(a2) Fi is isometric:
Let $f_1,f_2\in\Sigma\to\mathcal{P}_{cl}(Step_P)$. We want to show:

$$d_{FP}(f_1,f_2)=d_{FQ}(Fi(f_1),Fi(f_2)),$$

which follows from

$$\forall\sigma\in\Sigma[d_{\mathcal{P}_{cl}(Step_P)}(f_1(\sigma),f_2(\sigma))=d_{\mathcal{P}_{cl}(Step_Q)}(Fi(f_1)(\sigma),Fi(f_2)(\sigma))].$$

It suffices to show, given $\sigma\in\Sigma$:

$$\forall x'\in f_1(\sigma)\forall x''\in f_2(\sigma)[d_{Step_P}(x',x'')=d_{Step_Q}(Fi(x'),Fi(x''))].$$

(Note that we still should formally define $Fi(x)$ for $x\in Step_P$. The intended meaning is clear; for instance $Fi(<\sigma',p'>)=<\sigma',i(p')>$.)
We consider only the interesting cases for x' and x''.
(i) $x'=<\sigma',p'>,x''=<\sigma'',p''>$:
$d_{Step_P}(x',x'')=$ [i is isometric]
$d_{Step_Q}(<\sigma',i(p')>,<\sigma'',i(p'')>)=$
$d_{Step_Q}(Fi(x'),Fi(x''))$
(ii) $x'=<\alpha',m',\bar\beta',f'>$, $x''=<\alpha'',m'',\bar\beta'',f''>$:
$d_{Step_P}(<\alpha',m',\bar\beta',f'>,<\alpha'',m'',\bar\beta'',f''>)=$ [i is isometric]
$d_{Step_Q}(<\alpha',m',\bar\beta',i\circ f'>,<\alpha'',m'',\bar\beta'',i\circ f''>)=$
$d_{Step_Q}(Fi(x'),Fi(x''))$
(iii) $x'=<\alpha',m'g'>$, $x''=<\alpha'',m'',g''>$:
It suffices to show (we omit subscripts of d like $Obj^*\to(Obj\to P)\to^1 P$):
$d(g',g'')=d(\lambda\bar\alpha\cdot\lambda h\cdot i\circ g'(\bar\alpha)(j\circ h),\ \lambda\bar\alpha\cdot\lambda h\cdot i\circ g''(\bar\alpha)(j\circ h))$.
(Then $d_{Step_P}(x',x'')=d_{Step_Q}(Fi(x'),Fi(x''))$.) On the one hand we have for $\bar\alpha\in Obj^*,h\in Obj\to Q$:
$d_Q(i\circ g'(\bar\alpha)(j\circ h),i\circ g''(\bar\alpha)(j\circ h))=$ [i is isometric]
$d_P(g'(\bar\alpha)(j\circ h),g''(\bar\alpha)(j\circ h))\leqslant d(g',g'')$.
On the other hand, for every $\delta>0$ we can choose $\bar\alpha\in Obj^*$ and $h\in(Obj\to P)$ such that
$d_P(g'(\bar\alpha)(h),g''(\bar\alpha)(h))\geqslant d(g',g'')-\delta$.

Let $h =^{def} i \circ h'$. Now we have
$d_P(i \circ g'(\bar{\alpha})(j \circ h), i \circ g''(\bar{\alpha})(j \circ h)) =$
$d_P(i \circ g'(\bar{\alpha})(j \circ i \circ h'), i \circ g''(\bar{\alpha})(j \circ i \circ h')) =$
$d_P(i \circ g'(\bar{\alpha})(h'), i \circ g''(\bar{\alpha})(h')) =$ [i is isometric]
$d_Q(g'(\bar{\alpha})(h'), g''(\bar{\alpha})(h')) \geqslant d(g', g'') - \delta$.
Because δ was arbitrary we have
$d(g', g'') \leqslant d(\lambda\bar{\alpha} \cdot \lambda h \cdot i \circ g'(\bar{\alpha})(j \circ h),\ \lambda\bar{\alpha} \cdot \lambda h \cdot i \circ g''(\bar{\alpha})(j \circ h))$.

(a3) Fj in NDI:
As above it suffices to consider elements x' and x'' of $Step_Q$. The Hausdorff distance is not changed by taking closures. We show:

$$\forall x', x'' \in Step_Q [d_{Step_P}(Fj(x'), Fj(x'')) \leqslant d_{Step_Q}(x', x'')].$$

Again there are only three interesting cases.

(i) $x' = \langle \sigma', q' \rangle, x'' = \langle \sigma'', q'' \rangle$:
$d_{Step_P}(Fj(x'), (Fj(x'')) =$
$d_{Step_P}(\langle \sigma', j(q') \rangle, \langle \sigma'', j(q'') \rangle) \leqslant$ [j is NDI]
$d_{Step_Q}(\langle \sigma', q' \rangle, \langle \sigma'', q'' \rangle) =$
$d_{Step_Q}(x', x'')$.

(ii) Similarly for $x', x'' \in Send_Q$.

(iii) $x' = \langle \alpha', m', g' \rangle, x'' = \langle \alpha'', m'', g'' \rangle$:
It suffices to show: $d(\lambda\bar{\alpha} \cdot \lambda h \cdot j \circ g'(\bar{\alpha})(i \circ h), \lambda\bar{\alpha} \cdot \lambda h \cdot j \circ g''(\bar{\alpha})(i \circ h)) \leqslant d(g', g'')$.
Let $\bar{\alpha} \in Obj^*$ and $h \in (Obj \to P)$. We have
$d(j \circ g'(\bar{\alpha})(i \circ h), j \circ g''(\bar{\alpha})(i \circ h)) \leqslant$ [j is NDI]
$d(g'(\bar{\alpha})(i \circ h), g''(\bar{\alpha})(i \circ h)) \leqslant d(g', g'')$.

(a4) $Fj \circ Fi = id_{FP}$:
We only show: $Fj \circ Fi(x) = x$ for $x \in Answer_P$.
Let $x \in Answer_P, x = \langle \alpha, m, f \rangle$. We have
$Fj \circ Fi(x) =$
$\langle \alpha, m, Fj(\lambda\bar{\alpha} \cdot \lambda h \in (Obj \to Q) \cdot i \circ f(\bar{\alpha})(j \circ h)) \rangle =$
$\langle \alpha, m, \lambda\bar{\alpha} \cdot \lambda h' \in (Obj \to P) \cdot j \circ i \circ f(\bar{\alpha})(j \circ i \circ h') \rangle =$
$\langle \alpha, m, \lambda\bar{\alpha} \cdot \lambda h' \in (Obj \to P) \cdot f(\bar{\alpha})(h') \rangle =$
$\langle \alpha, m, f \rangle = x$.

(b) F is contracting:
We show:

$$\forall P \to^{\langle i,j \rangle} Q \in \mathcal{C} [d_{FQ \to FQ}(Fi \circ Fj, id_{FQ}) \leqslant \tfrac{1}{2} \cdot d_{Q \to Q}(i \circ j, id_Q)].$$

Let $P \to^{\langle i,j \rangle} Q \in \mathcal{C}$, we define $\rho = d_{Q \to Q}(i \circ j, id_Q)$. Because of

$$d_{FQ \to FQ}(Fi \circ Fj, id_{FQ}) =$$
$$\sup_{y \in FQ}\{d_{FQ}(Fi \circ Fj(y), y)\} =$$
$$[Fi \circ Fj(p_0) = p_0]$$
$$\sup_{y \in \Sigma \to \mathcal{P}_\omega(Step_Q)}\{d_{FQ}(Fi \circ Fj(y), y)\} =$$
$$\tfrac{1}{2} \cdot \sup_{y \in \Sigma \to \mathcal{P}_\omega(Step_Q)}\{d_{\Sigma \to \mathcal{P}_\omega(Step_Q)}(Fi \circ Fj(y), y)\} =$$
$$\tfrac{1}{2} \cdot \sup_{y \in \Sigma \to \mathcal{P}_\omega(Step_Q)} \sup_{\sigma \in \Sigma}\{d_{\mathcal{P}_\omega(Step_Q)}(Fi \circ Fj(y(\sigma)), y(\sigma))\} \leqslant$$
$$\tfrac{1}{2} \cdot \sup_{x \in Step_Q}\{d_{Step_Q}(Fi \circ Fj(x), x)\}$$

it suffices to show:

$$\forall x \in Step_Q[d_{Step_Q}(Fi \circ Fj(x),x) \leqslant \rho].$$

Let $x \in Step_Q$. We distinguish three cases.

(i) $x \in \Sigma \times Q, x = <\sigma,q>$:
$d_{Step_Q}(Fi \circ Fj(x),x) =$
$d_{Step_Q}(<\sigma,i \circ j(q)>,<\sigma,q>) =$
$d_{\Sigma \times Q}(<\sigma,i \circ j(q)>,<\sigma,q>) =$
$d_Q(i \circ j(q),q) \leqslant$
$\sup_{q \in Q}\{d_Q(i \circ j(q),q)\} = \rho$.

(ii) $x \in Send_Q$: similarly.

(iii) $x \in Answer_Q, x = <\alpha,m,g>$:
$d_{Step_Q}(Fi \circ Fj(x),x) =$
$d_{Step_Q}(<\alpha,m,\lambda\bar{\alpha} \cdot \lambda h \in (Obj \to Q) \cdot i \circ j \circ g(\bar{\alpha})(i \circ j \circ h)>,<\alpha,m,g>) =$
$d_{Answer_Q}(<\alpha,m,\lambda\bar{\alpha} \cdot \lambda h \in (Obj \to Q) \cdot i \circ j \circ g(\bar{\alpha})(i \circ j \circ h)>,<\alpha,m,g>) =$
$\sup_{\bar{\alpha} \in Obj^*, h \in (Obj \to Q)}\{d_Q(i \circ j \circ g(\bar{\alpha})(i \circ j \circ h),g(\bar{\alpha})(h))\}$.
Let $\bar{\alpha} \in Obj^*, h \in (Obj \to Q)$. We have
$d_Q(i \circ j \circ g(\bar{\alpha})(i \circ j \circ h),g(\bar{\alpha})(h)) \leqslant \max\{d_Q(i \circ j \circ g(\bar{\alpha})(i \circ j \circ h),g(\bar{\alpha})(i \circ j \circ h)),d_Q(g(\bar{\alpha})(i \circ j \circ h),g(\bar{\alpha})(h))\}$
because d_Q is an ultra-metric. This maximum is at most ρ because:
$d_Q(i \circ j(g(\bar{\alpha})(i \circ j \circ h)),g(\bar{\alpha})(i \circ j \circ h)) \leqslant \sup_{q \in Q}\{d_Q(i \circ j(q),q)\} = \rho$
and
$d_Q(g(\bar{\alpha})(i \circ j \circ h),g(\bar{\alpha})(h)) \leqslant$ [because $g(\bar{\alpha}) \in (Obj \to Q) \to^1 Q$] $\leqslant d_{Obj \to Q}(i \circ j \circ h,h) \leqslant \rho$.
Note that this is the only place where we need the restriction that we only use functions in $(Obj \to Q) \to Q$ that are NDI. Now we can conclude
$d_{Step_Q}(Fi \circ Fj(x),x) \leqslant \rho$.

LEMMA 4.7
Let Φ_{PC} be as in definition 4.6. We prove
(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

$$\forall p_1,p_2,q_1q_2 \in P[d_P(p_1 \tilde{\odot} q_1,p_2 \tilde{\odot} q_2) \leqslant \max\{d_p(p_1,p_2),d_p(q_1,q_2)\}]$$

where $\tilde{\odot} = \Phi_{PC}(\odot)$.
Let $p_1,p_2,q_1,q_2 \in P$. We have

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

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.
Suppose $p_1,q_1,q_2 \neq p_0$. Let $\sigma \in \Sigma$. Let for $i = 1,2$

$$X_i =^{def} \{x \tilde{\odot} q_i | x \in p_1(\sigma)\},$$

$$Y_i =^{def} \{x \tilde{\odot} p_1 | x \in q_i(\sigma)\},$$

$$Z_i =^{def} \bigcup \{x|_\sigma y : x \in p_1(\sigma), y \in q_i(\sigma)\},$$

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

$$\frac{1}{2} \cdot d_{\mathcal{P}_d(Step_P)}(X_1 \cup Y_1 \cup Z_1, X_2 \cup Y_2 \cup Z_2) \leqslant d_P(q_1, q_2).$$

We have

$$d_{\mathcal{P}_d(Step_P)}(X_1 \cup Y_1 \cup Z_1, X_2 \cup Y_2 \cup Z_2) \leqslant$$

$$\max\{d_{\mathcal{P}_d(Step_P)}(X_1, X_2), d_{\mathcal{P}_d(Step_P)}(Y_1, Y_2), d_{\mathcal{P}_d(Step_P)}(Z_1, Z_2)\}.$$

We show: $d_{\mathcal{P}_d(Step_P)}(Z_1, Z_2) \leqslant 2 \cdot d_P(q_1, q_2)$. (The proofs for X_i and Y_i are straightforward.) By the definition of the Hausdorff distance we have

$$d_{\mathcal{P}_d(Step_P)}(Z_1, Z_2) = \max\{\sup_{z_1 \in Z_1}\{d(z_1, Z_2)\}, \sup_{z_2 \in Z_2}\{d(z_2, Z_1)\}\}$$

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

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

We consider only the first supremum.

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

$$\sup_{z_1 \in Z_1}\{d_{Step_P}(z_1, z_2) \mid \exists x \in p_1(\sigma)\ \exists y_1 \in q_1(\sigma)\ \exists y_2 \in q_2(\sigma)$$

$$[\{z_1\} = x|_\sigma y_1 \wedge \{z_2\} = x|_\sigma y_2]\}.$$

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

1. Suppose $\{z_1\} = <\alpha, m, \bar{\beta}, f> |_\sigma <\alpha, m, g_1>$ with $<\alpha, m, \bar{\beta}, f> \in p_1(\sigma)$, $<\alpha, m, g_1> \in q_1(\sigma)$.

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

$$\{z_2\} = <\alpha, m, \bar{\beta}, f> |_\sigma <\alpha, m, g_2>$$

Then we have

$$d_{Step_P}(z_1, z_2) = d_{Step_P}(<\sigma, g_1(\bar{\beta})(f)>, <\sigma, g_2(\bar{\beta})(f)>)$$

$$= d_P(g_1(\bar{\beta})(f), g_2(\bar{\beta})(f))$$

$$\leqslant d(g_1, g_2)$$

$$= d_{Step_P}(<\alpha, m, g_1>, <\alpha, m, g_2>).$$

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

$$d_{Step_P}(<\alpha, m, g_1>, <\alpha, m, g_2>) \leqslant d_{\mathcal{P}_d(Step_P)}(q_1(\sigma), q_2(\sigma)) + \epsilon$$

$$\leqslant d_{\Sigma \to \mathcal{P}_d(Step_P)}(q_1, q_2) + \epsilon$$

$$\leqslant 2 \cdot d(q_1, q_2) + \epsilon.$$

Therefore

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

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 $<\alpha, m, g_2> \in q_2(\sigma)$, then

$$d_{\mathcal{P}_{cl}(Step_P)}(q_1(\sigma),q_2(\sigma)) \leqslant d(<\alpha,m,g_1>,q_2(\sigma)) = 1.$$

Therefore

$$d_P(q_1,q_2) \geqslant \frac{1}{2}.$$

Now

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

2. The second possibility is that $\{z_1\} = <\alpha,m,g> |_\sigma <\alpha,m,\bar{\beta},f_1>$,
with $<\alpha,m,g> \in p_1(\sigma)$, $<\alpha,m,\bar{\beta},f_1> \in q_1(\sigma)$. 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

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

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

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

We have

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

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

$$X_i =^{def} \{x\,\tilde{\odot}_i q | x \in p(\sigma)\},$$

$$Y_i =^{def} \{x\,\tilde{\odot}_i p | x \in q(\sigma)\},$$

$$Z_i =^{def} \bigcup \{x|_\sigma y : x \in p(\sigma), y \in q(\sigma)\},$$

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

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

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

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

$$d_{\mathcal{P}_{cl}(Step_P)}(X_1,X_2) = \max\{\sup_{x_1\in X_1}\{d(x_1,X_2)\}, \sup_{x_2\in X_2}\{d(x_2,X_1)\}\}$$

Let $x_1 \in X_1$. We show

$$\inf_{x_2\in X_2}\{d_{Step_P}(x_1,x_2)\} \leqslant d_{P\times P\to {}^1P}(\odot_1, \odot_2).$$

We treat one of the three possible cases for $x_1 \in X_1$, say $x_1 = <\sigma', p' \odot_1 q>$, where $p' \in p(\sigma)$:

$$\inf_{x_2\in X_2}\{d_{Step_P}(<\sigma', p' \odot_1 q>, x_2)\} \leqslant$$

$$d_{Step_P}(<\sigma', p' \odot_1 q>, <\sigma', p' \odot_2 q>) =$$

$$d_{\Sigma\times P}(<\sigma', p' \odot_1 q>, <\sigma', p' \odot_2 q>) =$$

$$d_P(p' \odot_1 q, p' \odot_2 q) \leqslant$$
$$d_{P \times P \to {}^1P}(\odot_1, \odot_2).$$

Thus we have

$$\sup_{x_1 \in X_1}\{d(x_1, X_2)\} \leqslant d_{P \times P \to {}^1P}(\odot_1, \odot_2).$$

Similarly

$$\sup_{x_2 \in X_2}\{d(x_2, X_1)\} \leqslant d_{P \times P \to {}^1P}(\odot_1, \odot_2).$$

So

$$d_{\mathcal{P}_{cl}(\text{step}_P)}(X_1, X_2) \leqslant d_{P \times P \to {}^1P}(\odot_1, \odot_2).$$

And analogously

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

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

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

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

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

and thus

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

LEMMA 4.12

For every expression e, statement s, environment γ and active object α we have:

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

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

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

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

$$\Phi_{e,s,p}(q) = [\![e]\!]_E(\gamma)(\alpha)($$
$$\lambda\beta \cdot \lambda\sigma \cdot \{<\sigma, \text{ if } \beta = tt \text{ then } [\![s]\!]_S(\gamma)(\alpha)(q) \text{ else } p \text{ fi}>\}).$$

PROOF.

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

(a) For all simple expressions e and statements s we have $[\![e]\!]_E(\gamma)(\alpha) \in (Obj \to P) \to {}^1P$ and $[\![s]\!]_S(\gamma)(\alpha) \in P \to {}^1P$.

(b) Suppose we have proved part (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 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 statements is of the form **answer**$\{m_1, \ldots, m_n\}$.
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 we even have:

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

Intuitively the decrease of distance follows from the fact that the evaluation of these expression 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:

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

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

$$d_P([\![\textbf{new}(C)]\!]_E(\gamma)(\alpha)(f_1), [\![\textbf{new}(C)]\!]_E(\gamma)(\alpha)(f_2)) =$$

$$d_p(\lambda\sigma\cdot\{<\sigma', \gamma_1(C)(\hat{\sigma}_3) \| f_1(\hat{\sigma}_3)>\}, \lambda\sigma\cdot\{<\sigma', \gamma_1(C)(\hat{\sigma}_3) \| f_2(\hat{\sigma}_3)>\} =$$

$$\tfrac{1}{2} \cdot \sup_{\sigma\in\Sigma}\{d_{Step_P}(<\sigma', \gamma_1(C)(\hat{\sigma}_3) \| f_1(\hat{\sigma}_3)>, <\sigma', \gamma_1(C)(\hat{\sigma}_3) \| f_2(\hat{\sigma}_3)>)\} =$$

$$\tfrac{1}{2} \cdot \sup_{\sigma\in\Sigma}\{d_P(\gamma_1(C)(\hat{\sigma}_3) \| f_1(\hat{\sigma}_3), \gamma_1(C)(\hat{\sigma}_3) \| f_2(\hat{\sigma}_3))\} \leqslant$$

[because $\|$ is NDI] $\leqslant$

$$\tfrac{1}{2} \cdot \sup_{\sigma\in\Sigma}\{d_P(f_1(\hat{\sigma}_3), f_2(\hat{\sigma}_3))\} \leqslant$$

$$\tfrac{1}{2} \cdot d_{Obj\to P}(f_1, f_2).$$

For the standard objects we have the following. Let $\phi \in SObj$:

$$d_P([\![\phi]\!]_E(\gamma)(\alpha)(f_1), [\![\phi]\!]_E(\gamma)(\alpha)(f_2)) =$$

$$d_P(f_1(\phi), f_2(\phi)) \leqslant$$

$$d_{Obj\to P}(f_1, f_2).$$

For the only simple statement **answer**$\{m_1, \ldots, m_n\}$ we have, for given processes $p_1, p_2 \in P$:

$$d_P([\![\textbf{answer}\{m_1, \ldots, m_n\}]\!]_S(\gamma)(\alpha)(p_1), [\![\textbf{answer}\{m_1, \ldots, m_n\}]\!]_S(\gamma)(\alpha)(p_2)) =$$

$$d_P(\lambda\sigma\cdot\{<\alpha, m_i, g_i^{(1)}> | 1 \leqslant i \leqslant n\}, \lambda\sigma\cdot\{<\alpha, m_i, g_i^{(2)}> | 1 \leqslant i \leqslant n\})$$

where for $j = 1,2$, $i = 1,2,3,4$

$$g_i^{(j)} = \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in (Obj \to P) \cdot \gamma_2(m_i)(\sigma_4(\alpha))(\alpha)(\bar{\beta})(\lambda\beta\cdot(f(\beta) \| p_j)).$$

The desired result is straightforward from:

$$d_{Obj^*\to(Obj\to P)\to P}(g_i^{(1)}, g_i^{(2)}) \leqslant$$

[because $\gamma_2(m_i)(\sigma_4(\alpha))(\alpha)(\bar{\beta}) \in (Obj \to P) \to^1 P$] $\leqslant$

$$\sup_{f\in(Obj\to P)}\{d_{Obj\to P}(\lambda\beta\cdot(f(\beta) \| p_1), \lambda\beta\cdot(f(\beta) \| p_2))\} =$$

$$\sup_{p\in P}\{d_P(p \| p_1, p \| p_2)\} \leqslant$$

[because || is NDI]$\leqslant$

$d_P(p_1,p_2)$.

Part (b)

Composite expressions are of the form $e!m(e_1,\ldots,e_n)$, $m(e_1,\ldots,e_n)$ 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 part (i) and (ii) of the lemma for expressions $e,e_1,\ldots,e_n\in Exp$ and for $s\in Stat$. We shall treat one composite expression and one composite statement.

We show that $[\![e!m(e_1,\ldots,e_n)]\!]_E(\gamma)(\alpha)\in(Obj\to P)\to^1 P$. Let $f_1,f_2\in(Obj\to P)$. We have:

$d_P([\![e!m(e_1,\ldots,e_n)]\!]_E(\gamma)(\alpha)(f_1),[\![e!m(e_1,\ldots,e_n)]\!]_E(\gamma)(\alpha)(f_2)) =$

$d_P([\![e]\!]_E(\gamma)(\alpha)(\cdots\lambda\sigma\cdot\{<\beta,m,\bar{\beta},f_1>\}\cdots),[\![e]\!]_E(\gamma)(\alpha)(\cdots\lambda\sigma\cdot\{<\beta,m,\bar{\beta},f_2>\}\cdots))\leqslant$

[by the induction hypothesis for e]$\leqslant$

$d(\cdots\lambda\sigma\cdot\{<\beta,m,\bar{\beta},f_1>\}\cdots,\cdots\lambda\sigma\cdot\{<\beta,m,\bar{\beta},f_2>\}\cdots)\leqslant$

[by the induction hypotheses for $e_1,\ldots,e_n$]$\leqslant$

$d(\lambda\sigma\cdot\{<\beta,m,\bar{\beta},f_1>\},\lambda\sigma\cdot\{<\beta,m,\bar{\beta},f_2>\})\leqslant$

$\frac{1}{2}\cdot d_{Obj\to P}(f_1,f_2)$.

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

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

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

Firstly we show that

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

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

$d_P(\Phi_{e,s,p}(q_1),\Phi_{e,s,p}(q_2)) =$

$d_P([\![e]\!]_E(\gamma)(\alpha)(\lambda\beta\cdots q_1\cdots),[\![e]\!]_E(\gamma)(\alpha)(\lambda\beta\cdots q_2\cdots))\leqslant$

[by the induction hypothesis for e]$\leqslant$

$d_{Obj\to P}(\lambda\beta\cdot\lambda\sigma\cdot\{\cdots q_1\cdots\},\lambda\beta\cdot\lambda\sigma\cdot\{\cdots q_2\cdots\}) =$

$\frac{1}{2}\cdot d_P([\![s]\!]_S(\gamma)(\alpha)(q_1),[\![s]\!]_S(\gamma)(\alpha)(q_2))\leqslant$

[by the induction hypothesis for s]$\leqslant$

$\frac{1}{2}\cdot d_P(q_1,q_2)$.

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

$q_1 =^{def}$ Fixed Point (Φ_{e,s,p_1}),

$q_2 =^{def}$ Fixed Point (Φ_{e,s,p_2}).

We have

$d_P([\![$**do** e **then** s **od**$]\!]_S(\gamma)(\alpha)(p_1),[\![$**do** e **then** s **od**$]\!]_S(\gamma)(\alpha)(p_2)) =$

[by definition] =

$d_P(q_1,q_2) =$

$d_P(\Phi_{e,s,p_1}(q_1),\Phi_{e,s,p_2}(q_2)) \leqslant$

[by the same kind of calculation as above, using the induction hypothesis for e]

$\frac{1}{2}\cdot\max\{d_P([\![s]\!]_S(\gamma)(\alpha)(q_1),[\![s]\!]_S(\gamma)(\alpha)(q_2)),d_p(p_1,p_2)\} \leqslant$

[using the induction hypothesis for s] $\leqslant$

$\frac{1}{2}\cdot\max\{d_P(q_1,q_2),d_P(p_1,p_2)\}$.

We see:

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

LEMMA 4.18
Let for a unit $U \in Unit$ Φ_U be defined as in definition 4.17. We have

Φ_U is a contraction.

PROOF.
We shall show:

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

where $\tilde{\gamma}=\Phi_U(\gamma)$, $\tilde{\rho}=\Phi_U(\rho)$, by proving for $\gamma,\rho \in Env$ the following two inequalities:
(a) $d_{Env_1}((\tilde{\gamma})_1,(\tilde{\rho})_1) \leqslant \frac{1}{2}\cdot d_{Env}(\gamma,\rho)$
(b) $d_{Env_2}((\tilde{\gamma})_2,(\tilde{\rho})_2) \leqslant \frac{1}{2}\cdot d_{Env}(\gamma,\rho)$.
We have

$d_{Env_1}((\tilde{\gamma})_1,(\tilde{\rho})_1) =$

$\sup_{C\in CName,\alpha\in AObj}\{d_P((\tilde{\gamma})_1(C)(\alpha),(\tilde{\rho})_1(C)(\alpha))\} \leqslant$

$\sup_{s\in Stat,\alpha\in AObj}\{d_P([\![s]\!]_S(\gamma)(\alpha)(p_0),[\![s]\!]_S(\rho)(\alpha)(p_0)\}$.

Now it is easy to prove (in the same way as in lemma 4.12) that for every $s \in Stat$ and $e \in Exp$:

$[\![s]\!]_S \in Env \to^{\frac{1}{2}} (AObj \to P \to^1 P)$,

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

From this observation it follows that

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

which concludes the proof of part (a).
The proof of part (b) is similar to that of part (a) and therefore omitted.