RENDEZ-VOUS WITH METRIC SEMANTICS

J.W. de Bakker[1]
CWI, Postbus 4079, NL–1009 AB Amsterdam
& Vrije Universiteit

E.P. de Vink
Department of Mathematics and Computer Science, Vrije Universiteit
De Boelelaan 1081a, NL–1081 HV Amsterdam

Abstract

A comparative semantic study is made of an element of the family of concurrent object-oriented programming languages. Particular attention is paid to two notions: (i) dynamically evolving process structures, including a mechanism to name and refer to processes and a means to create new processes, and (ii) rendez-vous between processes involving the sending and answering of messages and the induced execution of method calls. The methodology of metric semantics is applied in the design of operational and denotational semantics, as well as in the proof of their equivalence. Both semantics employ domains which are determined as fixed points of a contracting functor in the category of complete metric spaces. Moreover, fruitful use is made of the technique of defining semantic meaning functions as fixed points of contracting higher-order mappings. Finally, syntactic and semantics continuations play a pervasive role.

1 Introduction

We shall present a comparative semantic study of a language of the COOP (concurrent object-oriented programming) variety. Particular attention will be paid to the following two phenomena

- dynamically evolving process structures, including a mechanism to name and refer to processes and a means to create new processes;
- a version of *rendez-vous* between processes involving the sending and answering of messages and the ensuing execution of method calls.

The language we consider is a slightly simplified version of the language POOL - the parallel object-oriented language designed by America [Ame89]. Several semantic investigations of this language have appeared already: operational semantics ([ABKR86]), denotational semantics ([ABKR89]), and a comparison of these two ([Rut90b]). Cf. also [AR89a] for a somewhat streamlined version of parts of [ABKR86, ABKR89, Rut90b] – excluding the more difficult sections of the comparison –, and [AR90], where an improvement of POOL's denotational semantics which is organized in three *layers* (for statements, objects and programs) is described. The latter paper is intended as well as a contribution to the issue of the *full abstractness* of the POOL semantics.

[1]Partially supported by ESPRIT Basic Research Action 3020: Integration

The treatments in [ABKR89, Rut90b] are rather complex and demand much from the uninitiated reader. The first aim of the present paper is to provide a more comprehensible version of these investigations, with special emphasis on the comparative issues. Partly, this is achieved by a presentation in two stages, both dealing with dynamically evolving processes, but only in the second one with a facility to name and refer to processes. Also, a careful tuning of the design of the operational and denotational definitions – in particular by the systematic use of so-called syntactic and semantic *continuations* – results in a transparent view of the relationship between the two models. Maybe more importantly, we propose a substantial simplification in the way the rendez-vous concept is handled. Firstly, the operational semantics rule for the rendez-vous is now appealingly simple and, secondly, some of the complexities in the denotational models of [ABKR89, Rut90b], in particular in the definition of the merge operator, are to a large extent avoided. Related to this we find that the equation determining the domain used in POOL's denotational semantics is essentially simplified in our approach. (In the domain equation $P = F(P)$, $F(P)$ has no more subterms of the form $(P \to \cdots)$. See Section 2 for background on this.) In addition, the somewhat extraneous use of the denotational meaning function $\mathcal{D}$ as part of the intermediate operational semantics in [Rut90b] is no more necessary.

The second aim of our paper is to provide a case study in the use of *metric* semantics. Let us first devote a few words to its basic principles. Consider two computations p_1, p_2. A natural *distance* $d(p_1, p_2)$ may be defined in terms of the notion of *initial segment* $p(k)$ of p – roughly, that part of p consisting of the first k steps (if present, otherwise p itself). Now we put $d(p_1, p_2) = 2^{-n}$, where n is the length of the longest common initial segment of p_1 and p_2 (i.e., $n = sup\{k \mid p_1(k) = p_2(k)\}$). Details vary with the form of the p_1, p_2. If computations are given as words (finite or infinite sequences of atomic actions), we take the standard notion of prefix; if p_1, p_2 are trees, we use truncation at depth k for $p(k)$. Other kinds of computations, e.g. involving function application, may be accommodated as well.

Complete metric spaces (*cms*'s) have the characteristic property that Cauchy sequences always have limits; this motivates their use for a smooth handling of infinite behaviour. In addition, each *contracting* function $f : (M, d) \to (M, d)$, for (M, d) a cms, has a *unique* fixed point (by Banach's theorem; see Section 2 for the definition of contracting). Uniqueness of fixed points may conveniently be exploited in a variety of situations:

Firstly, it has been shown that cms's may be used to solve *domain equations* of the form

$$P = F(P) \tag{1.1}$$

or, rather, $(P, d) \cong F(P, d)$, with (P, d) the cms to be determined, $\cong$ isometry, and F a mapping (functor) built from given cms's (A, d_A), the unknown (P, d), and composition rules such as $\dot{\cup}$ (disjoint union), $\times$ (cartesian product), $\to$ (function space), and $\mathcal{P}_{cl}(\cdot)$, $\mathcal{P}_{co}(\cdot)$ (the power sets of all *closed* or *compact* subsets of $\cdot$). See [BZ82], [AR89b] for mathematical details. As an advantage over the more usual cpo framework when used to solve (1.1) we mention that the notions of closed and, especially, compact subset arise very naturally for (the meanings of) many programming constructs. In a cpo setting, one has to choose between the Plotkin-, Smyth-, and Hoare powerdomains (cf. [GS90] for definitions), and it may not be so readily seen how to motivate a choice among these on the basis of a programming (rather than a mathematical) intuition.

Secondly, both denotational ($\mathcal{D}$) and operational ($\mathcal{O}$) semantics may be obtained as fixed points of (contracting) higher-order mappings, say Ψ and Φ. For $\mathcal{D}$ this is fairly traditional: in fact, it subsumes the classical fixed point treatment of recursion. For $\mathcal{O}$ it is less standard: Starting from a transition system $\mathcal{T}$ in the familiar Plotkin SOS style, one may assemble all transitions for a given program π into a meaning $\mathcal{O}(\pi)$. Here the choice as to what kind of domain is used as range for $\mathcal{O}$ (e.g. *linear time*, *branching time* (cf. [BBKM84, BMOZ88]) or bisimulation, interleaving or *noninterleaving* ([BW90]), *failure set* semantics (cf. [Rut89])) is a separate decision, in most cases independent of the design of $\mathcal{T}$. Maybe the most important advantage of this way of defining $\mathcal{O}$ as $fix(\Phi)$ is that is suggests a quite natural method to establish (*) $\mathcal{O} = \mathcal{D}$, viz. by proving that $\mathcal{D} = \Phi(\mathcal{D})$, whence the desired result follows by Banach's theorem (this important proof method is due to [KR90], cf. also [BM88]). Elsewhere ([Rut89, HBR90]) it is discussed how (*) may be strengthened to certain full abstractness results. Recently, investigations have begun concerning the possibility of obtaining $\mathcal{D}$ 'automatically' from a given transition system $\mathcal{T}$. In restricted cases this is indeed possible ([Rut90a]), and it is an interesting problem how this idea may be generalized. We make one further remark on metric vs. cpo semantics: In the latter, one either uses least fixed points, and then has to impose additional conditions to cope with infinite behaviour (e.g. closedness and *boundedness* of [MV88]), or one resorts to greatest fixed points and then continuity may be problematic (see e.g. [Par81]). In the metric framework, once contractivity is satisfied - which is mostly the case - infinite behaviour fits in quite naturally.

Thirdly, unique fixed points may be used to define various semantic operators. In elementary settings, it is no problem to define e.g., sequential or parallel composition. However, if additional features such as infinite behaviour, possibly infinite alphabets, or rendez-vous as part of parallel composition are involved, it is non-trivial how to give rigorous definitions of such operations, and higher-order techniques again turn out to be quite useful.

The present investigation is, partly, a companion to [BV91]. Whereas in that paper we concentrate on so-called *uniform* language notions (the atomic actions are uninterpreted or schematic, and there are no individual variables), we here study a nonuniform (interpreted) language with full-fledged presence of individual variables and non-trivial expressions. The latter necessitate the use, besides of (syntactic and semantic) statement continuations, as well of (syntactic and semantic) expression continuations (in the form proposed in [AB88]). Since in the uniform case certain well-definedness arguments are more perspicuous, we shall occasionally refer below to [BV91] when in need of a justification of some well-definedness property. We refer as well to [BV91] for references to papers where we have used metric semantics for (parallel) *logic* programming (LP). In [Eli91], metric semantics have been described for a language which exhibits, besides the COOP notions studied here, as well LP-like notions such as clausal resolving of goals and backtracking.

We conclude this introduction with a brief overview of the contents of the paper. Section 2 is primarily devoted to a concise presentation of the main ideas concerning the solution of domain equations over (i.e., in the category of) complete metric spaces. In Section 3 we develop comparative semantics for a language ($\mathcal{L}_{pp}$) with 'parallel processes', here to be taken as a dynamically growing system of statements executing in parallel and communicating through (a skeleton version of) the rendez-vous concept. In Section 4 we add to these notions the facility to name and refer to processes, together

with certain refinements of the rendez-vous. The resulting language we call $\mathcal{L}_{po}$, a language with 'parallel objects'. Both for $\mathcal{L}_{pp}$ and $\mathcal{L}_{po}$ we exhibit operational and denotational semantics. We prove that $\mathcal{O} = \mathcal{D}$, for $\mathcal{L}_{pp}$ in some detail and for $\mathcal{L}_{po}$ in outline, in Section 5. Here we find the pay-off from our earlier efforts to obtain a transparent correspondence between the two models, in that the proof of $\mathcal{O} = \mathcal{D}$ is largely syntax-directed, and does not require particular ingenuity.

Acknowledgements We owe much to Pierre America, the designer of the POOL language, and to Jan Rutten who, jointly with Pierre, was responsible for laying its semantics foundations. Jan Rutten also pointed out the need for articulating the notion of resumption in the present paper. We are indebted to Joost Kok for his contributions to the semantic studies of POOL, and, in general, to the members of the Amsterdam Concurrency Group for providing an expert and stimulating forum for discussion on our ongoing research. We thank Franck van Breugel for detailed reading of an earlier version of our paper leading to various improvements.

2 Mathematical preliminaries

2.1 Notations

We use the phrase "let $(x \in)M$ be such that ..." to introduce a set M with variable x ranging over M such that ... . We use $\mathcal{P}_\pi(\cdot)$ for the collection of all subsets of $\cdot$ which have property π. We use $f : X \to Y$ to define a function f with domain X and range (or codomain) Y. If $X = Y$ and $x \in X$ is such that $f(x) = x$, we call x a fixed point of f. If f has a unique fixed point we denote it by $\mathit{fix}(f)$. For $(x \in)M$ any set, we use $\vec{x}$ as a notation for a list (or vector) over M, with $k \geq 1$ elements.

2.2 Domain equations

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

$$P \cong F(P)$$

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

$$P \cong A \cup (B \times P) \tag{2.1}$$

$$P \cong A \cup \mathcal{P}_{co}(B \times P) \tag{2.2}$$

$$P \cong A \cup (B \to P) \tag{2.3}$$

where A and B are given fixed complete metric spaces. In [BZ82] it is first described how to solve these equations in a metric setting Roughly, the approach amounts to the following: In order to solve $P \cong F(P)$ they define a sequence of complete metric spaces $(P_n)_n$ by: $P_0 = A$ and $P_{n+1} = F(P_n)$, for $n>0$, such that $P_0 \subseteq P_1 \subseteq \cdots$. Then they take the *metric completion* of the union of these spaces P_n, say $\bar{P}$, and show: $\bar{P} \cong F(\bar{P})$. In this way they are able to solve equations (2.1), (2.2) and (2.3) above.

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

$$P \cong A \cup (P \xrightarrow{1} G(P)) \tag{2.4}$$

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

In [AR89b] the above approach is generalized in order to overcome this problem. The family of complete metric spaces is made into a *category* $\mathcal{C}$ by providing some additional structure. (For an extensive introduction to category theory we refer the reader to [Mac71].) Then the expression F is interpreted as a *functor* $F : \mathcal{C} \to \mathcal{C}$ which is (in a sense) *contracting*. It is proved that a generalized version of Banach's theorem (see below) holds, i.e., that contracting functors have a fixed point (up to isometry). Such a fixed point, satisfying $P \cong F(P)$, is a solution of the domain equation.

We shall now give a quick overview of these results, omitting many details and all proofs. For a full treatment we refer the reader to [AR89b]. We start by listing the basic definitions and facts of metric topology that we shall need. We assume the following notions to be known (the reader might consult [Dug66] or [Eng89]): metric space, ultra-metric space, complete (ultra-)metric space, continuous function, closed set, compact set. In our definition the distance between two elements of a metric space is always between 0 and 1, inclusive.

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

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

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

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

A function $f : M_1 \to M_2$ is called *contracting* (or a *contraction*) if there exists an ϵ<1 such that for all $x, y \in M_1$

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

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

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

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

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

Definition 2.1 Let (M, d), $(M_1, d_1), \ldots, (M_n, d_n)$ be metric spaces.

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

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

This supremum always exists since the values taken by our metrics are always between 0 and 1.

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

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

If no confusion is possible we often write $\cup$ rather than $\bar{\cup}$.

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

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

4. Let $\mathcal{P}_{cl}(M) = \{X : X \subseteq M \wedge X \text{ is closed}\}$. 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 $d(x, Z) = \inf_{z \in Z}\{d(x,z)\}$ for every $Z \subseteq M$, $x \in M$. (We use the convention that $\sup \emptyset = 0$ and $\inf \emptyset = 1$.) The spaces $\mathcal{P}_{co}(M) = \{X \subseteq M \wedge X \text{ is compact}\}$ and $\mathcal{P}_{nc}(M) = \{X \subseteq M \wedge X \text{ is non-empty and compact}\}$ are supplied with a metric by taking the restriction of d_H.

5. For any real number ϵ with $0 < \epsilon \leq 1$ we define

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

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

Proposition 2.2 Let (M,d), $(M_1,d_1), \ldots, (M_n,d_n)$, d_F, d_U, d_P and d_H be as in Definition 2.1 and suppose that (M,d), $(M_1,d_1), \ldots, (M_n,d_n)$ are complete. Then

$$(M_1 \to M_2, d_F) \quad \text{(a)}$$

$$(M_1 \,\bar{\cup}\, \cdots \,\bar{\cup}\, M_n, d_U) \quad \text{(b)}$$

$$(M_1 \times \cdots \times M_n, d_P) \quad \text{(c)}$$

$$(\mathcal{P}_{cl}(M), d_H), \quad (\mathcal{P}_{co}(M), d_H), \quad (\mathcal{P}_{nc}(M), d_H) \quad \text{(d)}$$

$$\mathrm{id}_\epsilon((M,d)) \quad \text{(e)}$$

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

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

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

Proposition 2.3 Let $(\mathcal{P}_{cl}(M),d_H)$ be as in Definition 1. Let $(X_i)_i$ be a Cauchy sequence in $\mathcal{P}_{cl}(M)$. We have

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

Proofs of Propositions 2.2(d) and 2.3 can be found in, for instance, [Dug66] and [Eng89]. The proofs are also repeated in [BZ82]. The completeness of $\mathcal{P}_{co}(M)$ is proved in [Kur56].

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

Definition 2.4 Let $\mathcal{C}$ denote the category that has complete metric spaces for its objects. The arrows ι in $\mathcal{C}$ are defined as follows: Let M_1, M_2 be complete metric spaces. Then $M_1 \to\!\!\!\!\cdot\ M_2$ denotes a pair of maps $M_1 \rightleftarrows_j^i M_2$, satisfying the following properties:

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

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

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

Definition 2.5 For every arrow $M_1 \to\!\!\!\!\cdot\ M_2$ in $\mathcal{C}$ with $\iota = [i,j]$ we define

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

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

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

Definition 2.6 We call a functor $F : \mathcal{C} \to \mathcal{C}$ contracting whenever the following holds: There exists an ϵ, with $0 \leq \epsilon < 1$, such that, for all $D \to\!\!\!\!\cdot\ E \in \mathcal{C}$,

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

We can now state the analogue of Banach's theorem. (Cf. [Mac71] for the notions of convergence and direct limit:

Theorem 2.7 Let F be a contracting functor $F : \mathcal{C} \to \mathcal{C}$ and let $D_0 \to^{\iota_0} F(D_0) \in \mathcal{C}$. Let the sequence $(D_n, \iota_n)_n$ be defined by $D_{n+1} = F(D_n)$ and $\iota_{n+1} = F(\iota_n)$ for all $n \geq 0$. This sequence is converging, so it has a direct limit $(D, (\gamma_n)_n)$. We have $D \cong F(D)$.

Let us now indicate how this theorem can be used to solve Equations (2.1) to (2.4) above. We define

$$F_1(P) = A \cup \mathrm{id}_{1/2}(B \times P) \tag{2.5}$$

$$F_2(P) = A \cup \mathcal{P}_{co}(B \times \mathrm{id}_{1/2}(P)) \tag{2.6}$$

$$F_3(P) = A \cup (B \to \mathrm{id}_{1/2}(P)) \tag{2.7}$$

If the expression $G(P)$ in Equation (2.4) is, for example, equal to P, then we define F_4 by

$$F_4(P) = A \cup \mathrm{id}_{1/2}(P \xrightarrow{1} P) \tag{2.8}$$

Note that the definitions of these functors specify, for each metric space (P, d_P), the metric on $F(P)$ *implicitly* (see Definition 2.1).

Now it is easily verified that F_1, F_2, F_3, and F_4 are contracting functors on $\mathcal{C}$. Intuitively, this is a consequence of the fact that in the definitions above each occurrence of P is preceded by a factor $\mathrm{id}_{1/2}$. Thus these functors have a fixed point, according to Theorem 2.7, which is a solution for the corresponding equation. (We often omit the factor $\mathrm{id}_{1/2}$ in the reflexive domain equations, assuming that the reader will be able to fill in the details.)

In [AR89b] it is shown that functors like F_1 to F_4 have *unique* fixed points (up to isometry). The results above hold for complete *ultra-metric* spaces too, which can be easily verified.

3 Parallel Processes

3.1 Introduction

We study the language $\mathcal{L}_{pp}$ of 'parallel processes', with particular attention for the programming notions of process creation and rendez-vous. In Section 4, we shall extend $\mathcal{L}_{pp}$ to the language $\mathcal{L}_{po}$ of 'parallel objects', the essence of the extension being the ability to name and refer to processes.

In $\mathcal{L}_{pp}$ we firstly find several conventional and simple programming constructs: assignments, sequential composition, conditionals, and the while statement. Also, a simple block construct introducing initialized (for convenience) local variables is included. Moreover, simple expressions (terms over some signature) appear. Three more advanced notions are furthermore considered:

- Process creation: Assuming that already $n(\geq 0)$ processes are active (i.e. executing in parallel), the effect of the statement **new**(s) will be to create an $n+1$-st process, with body s, to be executed in parallel to the n already active processes. (Note

that no other form of parallel execution, in particular no form of syntactic '||', is present in $\mathcal{L}_{pp}$.)

- Rendez-vous: This appears in the following 'skeleton' version: We introduce so called *methods* m, $\bar{m}$ (with $\bar{\bar{m}} = m$), together with an accompanying declaration d which assigns to each m a statement $d(m) = s$. Synchronized execution of m and $\bar{m}$ in two parallel processes results firstly in the execution of s, and, thereafter, in the resumption (in parallel) of the two remaining statements ('continuations') following m and $\bar{m}$, respectively. (The effect of $m|\bar{m} = s(= d(m))$ should be compared with similar rules $c|\bar{c} = \tau$ (in CCS) or $a|b = c$ (in ACP), the essential difference being that, contrary to s, τ or c are atomic.) In Section 4, we shall dress up this skeleton with some further notions: transmitting parameters, returning a resulting value, and identifying, by the sender, of the receiving component.

- Expressions with side-effects: We introduce here a simple version of side-effects, in order to motivate the mechanism of (syntactic and semantic) expression continuations. Again, a more interesting setting will be provided in Section 4.

3.2 Syntax

Throughout our paper, we use a self-explanatory BNF-like notation for syntactic definitions. We start with the introduction of four basic sets

- $(x \in) IVar$, a countable set of *individual variables*
- $(\alpha, \beta \in) Cons$, a countable set of *constants*
- $(\phi \in) Func$, a countable set of *function symbols* (each with some arity ≥ 1)
- $(m \in) M$, a *finite* set of *method names*. On M, a mapping $\bar{\cdot} : M \to M$, satisfying $\bar{\bar{m}} = m$, is given. (Since it is customary to consider only finite systems of declarations, d's domain M is assumed to be finite. Mathematically, there are no obstacles to dealing with infinite M.)

A program $\pi = (d, s)$ in the language $\mathcal{L}_{pp}$ consists of a declaration d in $Decl_{pp}$ and a statement s in $Stat_{pp}$. A declaration is a mapping from M to $Stat_{pp}$. Statements are conventional (see above), or have the form of the process creation **new**(s) or of a method call m. Expressions (in Exp_{pp}) are conventional, or exhibit a side-effect, in the form of $(s; e)$: an expression which first executes the statement s, and then executes e.

Definition 3.1 (syntax for $\mathcal{L}_{pp}$).

a. $s(\in Stat_{pp}) \quad ::= \quad x := e \mid m \mid (s_1; s_2 \mid \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} \mid$
$\textbf{while } e \textbf{ do } s \textbf{ od} \mid \textbf{new}(s) \mid \textbf{begin var } x := e; s \textbf{ end}$

b. $e(\in Exp_{pp}) \quad ::= \quad \alpha \mid x \mid \phi(\vec{e}) \mid (s; e)$

c. $(d \in) Decl_{pp} = M \to Stat_{pp}$

d. $\pi(\in \mathcal{L}_{pp}) \quad ::= \quad (d, s)$

3.3 Operational semantics

The operational semantics for $\mathcal{L}_{pp}$ is derived from a *transition system* $\mathcal{T}_{pp}$. Transitions are built using so-called *syntactic continuations*, which we use in two varieties:

- $(r \in) SySCo$, the syntactic statement continuations
- $(g \in) SyECo$, the syntactic expression continuations.

The design of these two classes has been motivated partly by our wish to obtain a smooth operational semantics for $\mathcal{L}_{pp}$, partly by the desire to obtain a tractable link with the *semantic continuations* which play a key role in the denotational semantics.

Definition 3.2 (syntactic continuations) Let E be a new symbol, standing for 'termination'.

a. $r(\in SySCo) ::= \mathrm{E} \mid (s : r) \mid (e : g) \mid (r_1, r_2) \mid r\langle\alpha/x\rangle \mid$
$\quad$ **if** β **then** r_1 **else** r_2 **fi** $\mid g(\alpha)$

b. $g(\in SyECo) ::= \lambda\alpha.r$

The continuations $(s : r)$ and $(e : g)$ are of a sequential nature. They should be read as 'execute s and continue with r', or 'evaluate e, pass its value to g, and continue with the result', respectively. Next, (r_1, r_2) denotes (interleaved) parallel execution of r_1 and r_2. The **if** $-$ **then** $-$ **else** $-$ **fi** construct and $g(\alpha)$ should be clear. The construct $r\langle\alpha/x\rangle$ will play a role in elaborating an assignment. Syntactic expression continuations were first used in this way in [AB88].

For the definition of $\mathcal{T}_{pp}$, we need the following basic definitions:

Definition 3.3

a. Let $(\alpha, \beta \in) V = Z \cup \{tt, ff\} \cup \cdots$ be the set of *basic values*. V is assumed to include at least the integers and the truth values tt, ff. Other basic values may be added, if desired. We find it convenient to use the same variables to range over V and over the set of constants C.

b. Let, for ϕ a function symbol with arity k, $\bar{\phi}$ be some element of $V^k \to V$.

c. Let $(\sigma \in) \Sigma = IVar \to V$ denote the set of *states*.

d. Let the auxiliary set $(\tau \in) \mathrm{T}$ be defined as $\mathrm{T} = \Sigma \cup M$.

e. Let $r[\alpha/\beta]$ denote the result of syntactically substituting the constant α for the constant β in r.

f. Let $\sigma[\alpha/x]$ denote the state which satisfies

$$\sigma[\alpha/x](y) = \begin{cases} \alpha & \text{if } x \equiv y \\ \sigma(y) & \text{if } x \not\equiv y \end{cases} .$$

We are now ready for

Definition 3.4 (transitions and transition systems)

a. A *transition* is a five-tuple

$$<r_1, \sigma, d, r_2, \tau> \tag{3.1}$$

in $SySCo \times \Sigma \times Decl_{pp} \times SySCo \times \mathrm{T}$. For (3.1) we usually write

$$<r_1, \sigma> \rightarrow_d <r_2, \tau>$$

b. A *transition system* $\mathcal{T}$ is a finite set of rules of the form

$$\frac{<r_1, \sigma_1> \rightarrow_d <r_1', \tau_1>, \cdots, <r_n, \sigma_n> \rightarrow_d <r_n', \tau_n>}{<r, \sigma> \rightarrow_d <r', \tau>}$$

for some $n \geq 0$. Such a rule should be read as: if we can establish (using $\mathcal{T}$) that the n premises are satisfied, we may infer that the conclusion holds. If $n = 0$, we have an *axiom*, written simply as $<r, \sigma> \rightarrow_d <r', \tau>$.

c. Rules which share the same (list of) premise(s) may be combined into one rule (with more than one consequence).

d. In a transition $<r, \sigma> \rightarrow_d <r', \tau>$ we shall usually suppress mentioning the d. No confusion will arise, since transitions are always to be taken with respect to one fixed d.

e. A rule of the form

$$\frac{<r_1, \sigma> \rightarrow_d <r, \tau>}{<r_2, \sigma> \rightarrow_d <r, \tau>}$$

will be abbreviated to $<r_2, \sigma> \rightarrow_0 <r_1, \sigma>$ or even to $r_2 \rightarrow_0 r_1$. (Read: in order to execute r_2, find out how to execute r_1. The '0' expresses that this requires zero 'steps'.)

f. Each transition system $\mathcal{T}$ determines a relation $\mathcal{R}$ which is defined as the least relation (here subset of $SySCo \times \Sigma \times Decl_{pp} \times SySCo \times \mathrm{T}$) satisfying the given axioms and rules.

Next, we give the definition of the transition system $\mathcal{T}_{pp}$ which will be used to obtain the operational semantics $\mathcal{O}$ for $\mathcal{L}_{pp}$.

Definition 3.5 (transition system $\mathcal{T}_{pp}$ for $\mathcal{L}_{pp}$) The rules in $\mathcal{T}_{pp}$ are organized in groups, for easier structuring. This grouping is not part of the formal system itself.

s-rules

- zero-step

$$\begin{aligned}
(x := e) : r &\rightarrow_0 e : \lambda\alpha.(r<\alpha/x>) \\
(s_1; s_2) : r &\rightarrow_0 s_1 : (s_2 : r) \\
\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} : r &\rightarrow_0 e : \lambda\beta.\textbf{if } \beta \textbf{ then } s_1 : r \textbf{ else } s_2 : r \textbf{ fi} \\
\textbf{new}(s) : r &\rightarrow_0 (s : \mathrm{E}, r) \\
<\textbf{begin var } x := e; s \textbf{ end} : r, \sigma> &\rightarrow_0 <(x := e; s) : r<\sigma(x)/x>, \sigma>
\end{aligned}$$

- axioms

$$\begin{aligned} <m : r, \sigma> &\rightarrow <r, m> \\ <\textbf{while } e \textbf{ do } s \textbf{ od} : r, \sigma> &\rightarrow \\ &<e : \lambda\beta.\textbf{if } \beta \textbf{ then } (s; \textbf{while } e \textbf{ do } s \textbf{ od}) : r \textbf{ else } r \textbf{ fi}, \sigma> \end{aligned}$$

e-rules

- zero-step

$$\begin{aligned} \alpha : \lambda\beta.r &\rightarrow_0 r[\alpha/\beta] \\ \phi(e_1, \cdot\cdot, e_k) : g &\rightarrow_0 e_1 : \lambda\beta_1.(e_2 : \cdot\cdot(e_k : \lambda\beta_k.\bar{\phi}(\beta_1, \cdot\cdot, \beta_k) : g)\cdot\cdot) \\ (s; e) : g &\rightarrow_0 s : (e : g) \end{aligned}$$

- axioms

$$<x : g, \sigma> \rightarrow <\sigma(x) : g, \sigma>$$

r-rules

- zero-step

$$\begin{aligned} \textbf{if } tt \textbf{ then } r_1 \textbf{ else } r_2 \textbf{ fi} &\rightarrow_0 r_1 \\ \textbf{if } ff \textbf{ then } r_1 \textbf{ else } r_2 \textbf{ fi} &\rightarrow_0 r_2 \end{aligned}$$

- axioms

$$<r<\alpha/x>, \sigma> \rightarrow <r, \sigma[\alpha/x]>$$

- rules for parallel execution

(interleaving)

$$\frac{<r, \sigma> \rightarrow <r', \tau>}{\begin{array}{l} <(r, \bar{r}), \sigma> \rightarrow <(r', \bar{r}), \tau> \\ <(\bar{r}, r), \sigma> \rightarrow <(\bar{r}, r'), \tau> \end{array}}$$

(rendez-vous)

$$\frac{<r_1, \sigma> \rightarrow <r', m>, <r_2, \sigma> \rightarrow <r'', \bar{m}>}{<(r_1, r_2), \sigma> \rightarrow <s : (r', r''), \sigma>}, \; d(m) = s$$

Explanation

(assignment): evaluating $x := e$ amounts to first evaluating e, and transmitting the result α to the continuation which will eventually arrange that x is set to α.

(new): the body s is supplied with the termination continuation E, and set in parallel to r (which itself may consist of several continuations in parallel)

(begin .. end): evaluate $x := e; s$, and next reset x to the value $(\sigma(x))$ it had upon block entrance

(m): the method m is stored, available for subsequent use in in the rendez-vous rule

($\phi(\vec{e})$): the arguments $e_1, .., e_k$ are evaluated from left to right, yielding $\beta_1, .., \beta_k$; the interpretation $\bar{\phi}$ of ϕ is then applied to these $\beta_1, .., \beta_k$

($r<\alpha/x>$): this handles the assignment of α to x, resulting in $\sigma[\alpha/x]$

(interleaving): the usual interleaving rule for parallel composition

(rendez-vous): in case r_1 and r_2 can make an m and $\bar{m}$-step, respectively, the rendez-vous succeeds, s is executed, and the execution continues with that of (r', r''). (See also the remark at the end of Section 3 for a possible refinement of the rule.)

We next discuss how to assemble all successive steps prescribed by $\mathcal{T}_{pp}$ for some program (d, s) into one result $\mathcal{O}(d, s)$. Crucial here is the definition of the range P of the mapping $\mathcal{O}: \mathcal{L}_{pp} \to P$. We shall determine P as solution of a domain equation (in the category of complete metric spaces, cf. Section 2), viz.

$$P = \{p_0\} \cup (\Sigma \to \mathcal{P}_{co}((\Sigma \cup M) \times P)) \tag{3.2}$$

Equation (3.2) may be understood as follows: Each element p in P (to be called a *process* as well, but now a mathematical, and not a programming, entity) is either the nil-process p_0, or it is a function in $\Sigma \to \mathcal{P}_{co}(\cdot)$ which, when supplied with a state σ as argument, yields an element X of $\mathcal{P}_{co}(\cdot)$, i.e. a compact subset of $(\Sigma \cup M) \times P$. Thus, the elements of $p(\sigma) = X$ are of the form $<\sigma', p'>$ or $<m, p'>$. The first possibility delivers a next state σ', together with a so-called *resumption* p'. This resumption tells us what to do next: In the operational or denotational setting this will be determined by the syntactic or semantic continuation, respectively. A second possibility for an element X is a pair $<m, p'>$; here m results from a method call, and p' is as before. The rendez-vous rule *resolves* synchronized method calls. However, one-sided method calls which have not synchronized with their partner will leave such a pair $<m, p'>$ as a trace in the result.

The domain P is used in the next definition which introduces (the intermediate) $\mathcal{O}_d$ as fixed point of a contracting higher-order mapping (of meaning functions to meaning functions) Φ_d. To understand the structure of the definition, the reader should look at Lemma 3.7.c. This is the result in the form which is most intuitive, and to justify it we employ the Φ_d-mapping.

Definition 3.6 Let $F \in SySCo \to P$.

a. We define $\Phi_d : (SySCo \to P) \to (SySCo \to P)$ by putting

$$\begin{array}{lcl} \Phi_d(F)(\mathrm{E}) & = & p_0 \\ \Phi_d(F)(r) & = & \lambda\sigma.\{<\tau, F(r')> \mid <r, \sigma> \to <r', \tau>\}, \quad \text{for } r \neq \mathrm{E} \end{array}$$

where $\to$ is determined by $\mathcal{T}_{pp}$.

b. $\mathcal{O}_d = fix(\Phi_d)$, $\mathcal{O}(d, s) = \mathcal{O}_d(s : \mathrm{E})$.

We have

Lemma 3.7

a. $\Phi_d(F)(r) \in P$ for each F, r.

b. Φ_d is contracting in F.

c. $\mathcal{O}_d(\mathrm{E}) = p_0$
$\mathcal{O}_d(r) = \lambda\sigma.\{<\tau, \mathcal{O}_d(r')> \mid <r,\sigma> \rightarrow <r',\tau>\}, \quad$ for $r \neq \mathrm{E}$

Proof

a. Follows from the fact that $\mathcal{T}_{pp}$ is *finitely branching*, i.e. for each r , σ, we have $|\{(r',\tau) \mid <r,\sigma> \rightarrow <r',\tau>\}| < \infty$.

b. Clear by the definition of $\Phi_d(F)$, in particular by the $<\tau,..>$-step in its definition.

c. Immediate by the definitions of Φ_d and $\mathcal{O}_d$. □

Remark The domain P has rather more structure than is usual for an operational semantics. We use the same P for our denotational definitions in the next subsection; the proof that $\mathcal{O} = \mathcal{D}$ (in Section 5) will considerably profit from it. On the other hand, it is not difficult to use the same $\mathcal{T}_{pp}$ to obtain a much simpler (i.e., less structured) operational meaning, say $\mathcal{O}^\star : \mathcal{L}_{pp} \rightarrow P^\star$. Let δ be a new symbol (standing for deadlock), and let $\Sigma_\delta^\infty = \Sigma^* \cup \Sigma^\omega \cup \Sigma^* \cdot \{\delta\}$, i.e., the set of all finite sequences over Σ, possibly postfixed by δ, and all infinite sequences over Σ. We put

$$P^\star = \Sigma \rightarrow (\{\{\epsilon\}\} \cup \mathcal{P}_{nc}(\Sigma_\delta^\infty))$$

and define $\mathcal{O}_d^\star$ to satisfy

$$\mathcal{O}_d^\star(\mathrm{E}) = \lambda\sigma.\{\epsilon\}$$

$$\mathcal{O}_d^\star(r) = \begin{cases} \lambda\sigma.\bigcup\{\sigma'.\mathcal{O}_d^\star(r')(\sigma') \mid <r,\sigma> \rightarrow <r',\sigma'>\} \\ \qquad \text{if the above set } \{\cdot\} \neq \emptyset \qquad \text{for } r \neq \mathrm{E}. \\ \{\delta\} \quad \text{otherwise} \end{cases}$$

$\mathcal{O}_d^\star(r)$ exhibits three essential differences with $\mathcal{O}_d(r)$. Firstly, it has lost the branching structure of the latter. Next, steps $<r,\sigma> \rightarrow <r',m>$ do not contribute to the result (whence the possibillity that the set $\{\cdot\}$ might be empty). Thirdly, the resumptions have disappeared (instead of $<\sigma',p'>$ we now simply employ $p'(\sigma')$). As a consequence, $\mathcal{O}_d^\star$ is not compositional. In particular, no relationship of the form $\mathcal{O}_d^\star(r_1, r_2) = \mathcal{O}_d^\star(r_1) \| \mathcal{O}_d^\star(r_2)$ holds.

3.4 Denotational semantics

We shall define the denotational semantics $\mathcal{D}$ for $\mathcal{L}_{pp}$ in terms of the auxiliary semantic mappings $\mathcal{I}_d$ and $\mathcal{E}_d$:

$$\mathcal{I}_d : Stat_{pp} \rightarrow SeSCo \rightarrow P$$
$$\mathcal{E}_d : Exp_{pp} \rightarrow SeECo \rightarrow P$$

Here $(p \in)P$ is as in Section 3.3, $SeSCo =_{df} P$ is the set of semantic statement continuations, and $SeECo =_{df} (f \in)V \to P$ is the set of semantic expression continuations. The definition of the semantic parallel composition operator '||' will be supplied in Definition 3.9.

Definition 3.8 (denotational semantics for $\mathcal{L}_{pp}$)

a.
$$\begin{aligned}
\mathcal{I}_d(x := e)(p) &= \mathcal{E}_d(e)(\lambda\alpha.\lambda\sigma.\{<\sigma[\alpha/x], p>\}) \\
\mathcal{I}_d(m)(p) &= \lambda\sigma.\{<m, p>\} \\
\mathcal{I}_d(s_1; s_2)(p) &= \mathcal{I}_d(s_1)(\mathcal{I}_d(s_2)(p)) \\
\mathcal{I}_d(\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi})(p) &= \mathcal{E}_d(e)(\lambda\beta.\textbf{if } \beta \textbf{ then } \mathcal{I}_d(s_1)(p) \textbf{ else } \mathcal{I}_d(s_2)(p) \textbf{ fi}) \\
\mathcal{I}_d(\textbf{while } e \textbf{ do } s_1 \textbf{ od})(p) &= \\
\lambda\sigma.\{<\sigma, \mathcal{E}_d(e)(\lambda\beta.\textbf{if } \beta \textbf{ then } \mathcal{I}_d(s_1)(\mathcal{I}_d(\textbf{while } e \textbf{ do } s_1 \textbf{ od})(p)) \textbf{ else } p \textbf{ fi})\} & \\
\mathcal{I}_d(\textbf{new}(s_1))(p) &= \mathcal{I}_d(s_1)(p_0) \parallel p \\
\mathcal{I}_d(\textbf{begin var } x := e; s_1 \textbf{ end})(p) & \\
&= \lambda\sigma.\mathcal{I}_d(x := e; s_1)(\lambda\bar{\sigma}.\{<\bar{\sigma}[\sigma(x)/x], p>\})(\sigma)
\end{aligned}$$

b.
$$\begin{aligned}
\mathcal{E}_d(\alpha)(f) &= f(\alpha) \\
\mathcal{E}_d(x)(f) &= \lambda\sigma.\{<\sigma, f(\sigma(x))>\} \\
\mathcal{E}_d(\phi(e_1, .., e_k))(f) & \\
&= \mathcal{E}_d(e_1)(\lambda\beta_1.\cdots\mathcal{E}_d(e_k)(\lambda\beta_k.f(\bar{\phi}(\beta_1, .., \beta_k)))\cdots) \\
\mathcal{E}_d(s; e)(f) &= \mathcal{I}_d(s)(\mathcal{E}_d(e)(f))
\end{aligned}$$

Some explanations may help.

- $s \equiv x := e$: e is evaluated, the result is passed on to the expression continuation $f = \lambda\alpha.\cdots$, and eventually a change of state – setting x to α – is performed, and f then continues (resumes) with p
- $s \equiv m$: the pair $<m,p>$ will play a role in the definition of $\|$.
- $s \equiv$ **while** e **do** s_1 **od**: A (silent) step is performed, leaving σ unchanged, and then e is evaluated and the usual conditional for the while statement is given. Note that $\mathcal{I}_d$(**while**...**od**) returns on the right-hand side. To turn this into a well-defined formula, we should in fact define $\mathcal{I}_d$ as a (unique) fixed point of some higher-order contraction Ψ_d. (Details of a related case can be found in [BV91].)
- $s \equiv$ **new**(s_1): $\mathcal{I}_d(s_1)$ is supplied with the nil-continuation p_0, and executed in parallel with the already present continuation p. Note that $\mathcal{I}_d$ uses $\|$; below, we shall see that $\|$ uses $\mathcal{I}_d$. A comment on this follows later.
- $s \equiv$ **begin**...**end**: this amounts to executing the assignment and then the statement s_1, and after that resetting x to the value $\sigma(x)$ it had upon entrance of the block. Thus, it mimicks the operational rule.
- $e \equiv x$: As in $\mathcal{T}_{pp}$, a silent step is performed, and then the value $\sigma(x)$ is passed on to the expression continuation f.

We proceed with the definition of the parallel composition operator. Let X, Y range over $\mathcal{P}_{co}(\mathrm{T} \times P)$.

Definition 3.9 Let $p_1, p_2 \in P$.

a. $p_1 \parallel p_2 = \lambda\sigma.((p_1(\sigma) \between p_2) \cup (p_2(\sigma) \between p_1) \cup (p_1(\sigma) \mid_\sigma p_2(\sigma))$

b. $X \between p = \{<\tau, p' \parallel p> \mid <\tau, p> \in X\}$

c. $X \mid_\sigma Y = \{<\sigma, \mathcal{I}_d(s)(p' \parallel p'')> \mid <m, p'> \in X, <\bar{m}, p''> \in Y \text{ and } d(m) = s\}$

In executing $p_1 \parallel p_2$ for argument σ, one either makes a simple step from the left- or right operand (this yields interleaved execution), or the two outcomes $X = p_1(\sigma)$ and $Y = p_2(\sigma)$ communicate (in $X \mid_\sigma Y$) by a rendez-vous of the two steps $<m, p'>$ in X and $<\bar{m}, p''>$ in Y. This leads to the evaluation of $\mathcal{I}_d(s)$, for $s = d(m)$, with continuation $p' \parallel p''$. The circularity in this definition, viz. $\parallel$ defined in terms of ($\between$ and $\mid_\sigma$ defined in terms of) $\parallel$ and $\mathcal{I}_d$, and $\mathcal{I}_d$ defined in terms of $\parallel$, may be circumvented by using a simultaneous higher-order mapping (in two arguments), and defining $<\mathcal{I}_d, \parallel>$ as unique fixed point of this mapping. Considerable detail about this approach is supplied in [BV91]; therefore, we omit this here.

Finally, we put

Definition 3.10 $\mathcal{D}(d, s) = \mathcal{I}_d(s)(p_0)$.

In Section 5 we shall prove

First Main Theorem For each π in $\mathcal{L}_{pp}$, $\mathcal{O}(\pi) = \mathcal{D}(\pi)$.

Remark Though the rendez-vous rule (and the corresponding denotational definitions) yield precisely all successful computations, one might argue that it induces too many deadlock possibilities: Consider, e.g., the situation that $d(m) = m'$, and that $r_1 = (m : \mathrm{E}, \bar{m}' : \mathrm{E})$, $r_2 = \bar{m} : \mathrm{E}$. Since $<r_1, \sigma> \rightarrow <(\mathrm{E}, \bar{m}' : \mathrm{E}), m>$ and $<r_2, \sigma> \rightarrow <\mathrm{E}, \bar{m}>$, we may infer that $<(r_1, r_2), \sigma> \rightarrow <m' : (\mathrm{E}, \bar{m}' : \mathrm{E}), \sigma>$. As a consequence, in the result $<m' : (\mathrm{E}, \bar{m}' : \mathrm{E}), \sigma>$, a rendez-vous between m' and $\bar{m}'$ is no longer possible (since m''s partner $\bar{m}'$ is not accessible in a parallel component, but has been 'hidden' to occur *after* m'). Thus, an extra deadlock possibility has arisen which should have been avoided. A way out of this problem is the introduction (taken from [ABKR89]) of a separation between so-called *dependent* and *independent* resumptions. This works as follows: Right-hand sides of transitions are now of the form $<r', \sigma'>$ or $<r', <r'', m>>$. Here r' is the independent resumption which may continue independently of the success of the rendez-vous involving m, and r'' is the dependent resumption which may resume only *after* the rendez-vous for m has taken place. The induced modifications in $\mathcal{T}_{pp}$ are

- $<m : r, \sigma> \rightarrow <\mathrm{E}, <r, m>>$
- (revised rendez-vous rule)

$$\frac{<r_1, \sigma> \rightarrow <r_1', <r_1'', m>> \quad <r_2, \sigma> \rightarrow <r_2', <r_2'', \bar{m}>>}{<(r_1, r_2), \sigma> \rightarrow <(s : (r_1'', r_2''), (r_1', r_2')), \sigma>} \quad , \quad d(m) = s$$

Also, in the interleaving rule we now take $\tau \in \Sigma \cup (SySCo \times M)$. As a consequence, only the independent resumption (r') in $<r', <r'', m>>$ is involved in interleaving steps. Next, in the definition of P we replace the $M \times P$ term by $M \times P \times P$. Finally, we change the definition of $\Phi_d(F)(r)$, for $r \neq \mathrm{E}$, to read

$$\Phi_d(F)(r) = \lambda\sigma.\ \{<\sigma', F(r')> \mid <r, \sigma> \rightarrow <r', \sigma'>\} \ \cup \\ \{<m, F(r''), F(r')> \mid <r, \sigma> \rightarrow <r', <r'', m>>\},$$

with '$\rightarrow$' with respect to the amended $\mathcal{T}_{pp}$.
As to the denotational definitions, we impose the following changes:

- change in P as just given
- change in definition of $\mathcal{I}_d$: $\mathcal{I}_d(m)(p) = \lambda\sigma\{<m,p,p_0>\}$
- change in definition of $\|$:

$$X \mathbin{\|\!\!\|} p = \{<\sigma,p' \| p> \mid <\sigma,p'> \in X\} \cup \{<m,p'',p' \| p> \mid <m,p'',p'> \in X\}$$
$$X \mid_\sigma Y = \{<\sigma,\mathcal{I}_d(s)(p_1'' \| p_2'') \| p_1' \| p_2'> \mid <m,p_1'',p_1'> \in X, <\bar{m},p_2'',p_2'> \in Y, d(m) = s\}$$

We leave to the reader to work out the required modifications in the equivalence proof of Section 5.

4 Parallel objects

4.1 Introduction

The language $\mathcal{L}_{po}$ extends $\mathcal{L}_{pp}$ with a mechanism to *name* and *refer to* processes. Such a named process will from now on be called an *object.* It includes an 'active' part - comparable to the s in the **new**(s) construct of Section 3 - and a declarative part. In the declarative part we find the information on how a method name m is to be supplied with a method body μ, here taken in the form of a parametrized expression $\lambda\vec{x}.e$. Individual variables may now refer not only to values such as integers or truth values (together called V in Section 3), but as well to (the names of) objects. To be precise we replace V by

$$(\alpha,\beta,\gamma \in) Obj = SObj \cup ObjN$$

where $SObj$, the set of *standard objects,* takes over the role of V, and $ObjN$ is the set of *object names.* Objects are created as instances of a *class*: the relevant information about a class c is contained in the declaration $d(c)$. This is a pair $<d(c)_1, d(c)_2>$, where $d(c)_1 \in M \rightarrow Meth$ tells us how each method name m is provided with a method $\mu \in Meth$ as its body (i.e., $d(c)_1(m) = \mu$), and $d(c)_2 \in Stat_{po}$ is the statement (the 'process' of Section 3) execution of which is initiated (in parallel to the already existing objects) at the moment **new**(c) is executed. In other words, each execution of **new**(c) results in the creation of one more object as instance of class c, and all these objects are executing the (same) body s (determined by c's declaration) in parallel. The execution of **new**(c) furthermore involves the creation of a new name, say α', which is used to identify the newly created object (instance of c). Normally, this name will be stored in some individual variable (occurring in the creat*ing* object), for later reference.

The snapshot in Figure 1 of a creat*ing* α and creat*ed* α' may help (see next page). This picture assumes that $d(c)_1(m') = \mu'$, ..., and that $d(c)_2 = s'$. Details on how the new name α' is to be determined follow in Section 4.3. The picture also reflects that individual variables (from now on rather called *instance* variables) are 'private' to the objects. Private variables are not accessible from other objects. In fact, the only way in which objects may interact is by the sending and receiving of *messages.* This takes place by an extended version of the rendez-vous mechanism. Instead of the earlier synchronized execution of m and $\bar{m}$ occurring in two parallel processes (leading to the execution of

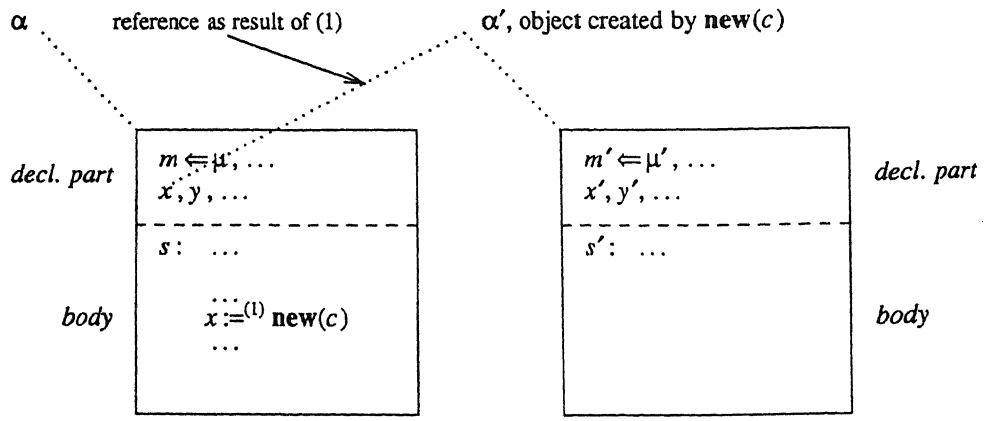

Figure 1: Two Objects

parallel processes (leading to the execution of the body $s = d(m)$ associated with m), we now have the following concept, execution of which is described in a number of steps:

1. a statement **answer**(m), when occurring in the body of an object (named, say, by object name α) indicates willingness to execute the method μ (associated with the method name m in the declaration of the class of which α is an instance) upon request;

2. a so-called send-expression $e!m(\vec{e})$, when occurring in the body of an object (named, say, by object name β) is executed as follows:

   - the value of the expression e is determined, resulting in the object γ; next
   - the values of the expressions $e_1, .., e_k$ are determined from left to right, resulting in $\gamma_1, .., \gamma_k$;
   - a request for execution of the method associated with method name m by the object name γ is issued

   [Step 2 takes place in parallel to Step 1];

3. in case the issue of this request synchronizes with the execution of the answer statement **answer**(m) as meant under 1 (implying that $\alpha = \gamma$), and assuming that $\mu = \lambda\vec{x}.e'$, next

4. the values $\gamma_1, .., \gamma_k$ are assigned to the (formal parameters, i.e., the) instance variables $x_1, .., x_k$, the expression e' is evaluated, the x_i are reset to their earlier values (which they had just before the assignment), and the result $\bar{\alpha}$ is returned to that position in object β where the value of $e!m(\vec{e})$ is required;

5. execution is resumed with the parallel execution (in α) of the statement following **answer**(m) and (in β) with the construct following $e!m(\vec{e})$.

All through the execution of 1. to 5., further parallel objects (different from α or β) will continue independently with their own activities. The only 'waiting' involved is (in α) for completing the evaluation of the method μ , and (in β) for the returning of the value $\bar{\alpha}$.

This brief sketch of the informal semantic of $\mathcal{L}_{po}$ should suffice here. More extensive explanations are contained in various studies on POOL semantics ([ABKR86, ABKR89, Rut90b, AR89a, AR90]. We have aimed at including, in $\mathcal{L}_{po}$, of all essential features of POOL. Concepts not treated are

- temporary variables (in addition to instance variables) and the object **nil**;
- the conditional answer statement, and an answer statement of the form **answer**$(m_1, .., m_k)$, $k \geq 1$;
- the method call (not as part of a rendez-vous);
- a few special cases of expressions;
- (a full treatment of) the standard objects.

Apart from the last item, the missing features can be dealt with without undue effort, by small extensions of the present definitions. Standard objects are more difficult since they are not, by nature, compact (cf. [Rut90b] for more information on this).

4.2 Syntax

The syntax for $\mathcal{L}_{po}$ may be inferred from that of $\mathcal{L}_{pp}$, as amended in the light of the extensions outlined above. Note that the new- and block constructs have been moved from the class of statements to that of expressions.
The following basic sets are used

- $(x \in) IVar$, a countable set of (individual or) instance variables
- $(m \in) M$, a finite set of method names
- $(\alpha, \beta, \gamma \in) SObj$, the syntactic set of *standard objects* (to be identified later with the semantic set of standard objects including the integers, truth values, and maybe more)
- $(c \in) Class$, a finite alphabet of *class names.*

We have no more use for the set *Func* . Finiteness of M and *Class* is, as before, postulated in order to avoid declarations with infinite domain.

Definition 4.1 (syntax for $\mathcal{L}_{po}$)

a. $s(\in Stat_{po}) \quad ::= \quad x := e \mid \textbf{answer}(m \mid (s_1; s_2) \mid \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} \mid \textbf{while } e \textbf{ do } s \textbf{ od}$

b. $e(\in Exp_{po}) \quad ::= \quad \alpha \mid x \mid e!m(\vec{e}) \mid \textbf{new}(c) \mid (s; e) \mid \textbf{begin var } \vec{x} := \vec{e}; e \textbf{ end}$

c. $(d \in) Decl_{po} = Class \rightarrow ((M \rightarrow Meth) \times Stat_{po})$

d. $\mu(\in Meth) \quad ::= \quad \lambda \vec{x}.e$

e. $\pi(\in \mathcal{L}_{po}) \quad ::= \quad (d, \mathbf{new}(c))$

In clause e., we see that the execution of a program starts with the creating of a first object as instance of some (initial) class c.

4.3 Operational semantics

As before, we base the operational semantics on a transition system, now named $\mathcal{T}_{po}$. This will involve a somewhat extended notion of state, as well as an adapted notion of a, possibly labeled, syntactic continuation.

We begin with the introduction of the sets of objects and states.

Definition 4.2

a. $(\alpha, \beta, \gamma \in) Obj = SObj \cup ObjN$
Here $SObj$ is the set of standard objects, and $ObjN$ is a (not further specified) set of object names.

b. $(\sigma \in)\Sigma = (IVar \rightarrow Obj \rightarrow Obj) \times \mathcal{P}_{fin}(Obj)$.

c. The functions $new : \mathcal{P}_{fin}(Obj) \times Class \rightarrow ObjN$ and $class : ObjN \rightarrow Class$ will be introduced below.

d. The notation $\sigma[\beta/x, \alpha]$ abbreviates $<\sigma_1[\sigma_1(x)[\beta/\alpha]/x], \sigma_2>$: σ is changed such that $\sigma[\beta/x, \alpha](x)(\alpha)$ now equals β; elsewhere σ is not changed.

A state is a pair $\sigma = <\sigma_1, \sigma_2>$. For a given instance variable x and object name α, $\sigma_1(x)(\alpha)$ tells us the current value of x (in object α). Note that the 'same' x will have, in general, a different value $\sigma_1(x)(\bar{\alpha})$ in some other object $\bar{\alpha}$. Furthermore, $\sigma_2 \in \mathcal{P}_{fin}(Obj)$ consists of a *finite* subset of Obj which may be read as the collection of all objects currently active. (If one so desires, one may consider some or all of the standard objects (for integers, truth values and the like) as already active and supplied with suitable standard methods. These issues are dealt with at length in [ABKR89], [Rut90b], and are not further treated here.) The function *new* delivers, for a current set of active objects $\xi(\in \mathcal{P}_{fin}(Obj))$ and class c, a new name $new(\xi, c)$ not in ξ, which may be used to name a new instance of class c. The function *class* determines, for each object name α, the class $c = class(\alpha)$ of which α is an instance.
We proceed with the definition of the various continuations.

Definition 4.3

a. $(r \in) SySCo$ is the set of syntactic statement continuations given by

$$r \quad ::= \quad \mathrm{E} \mid (s : r) \mid (e : g) \mid r{<}\alpha/x{>} \mid g(\alpha) \mid \mathbf{if}\ \beta\ \mathbf{then}\ r_1\ \mathbf{else}\ r_2\ \mathbf{fi} \mid {<}\beta, m, \tilde{\beta}{>}: g.$$

b. $(g \in) SyECo$ is the set of syntactic expression continuations given by

$$g \quad ::= \quad \lambda\alpha.r \mid g{<}\alpha/x{>} \mid \chi$$

c. $(\rho \in) LSySCo$ is the set of *labeled* syntactic statement continuations given by

$$\rho \quad ::= \quad <\alpha, r> \mid (\rho_1, \rho_2) \mid \alpha : \chi \mid <\beta, \rho>$$

d. $(\chi \in) LSyECo$ is the set of *labeled* syntactic expression continuations given by

$$\chi \quad ::= \quad <\alpha, g> \mid (\chi, \rho) \mid (\rho, \chi)$$

Anticipating the denotational semantics, we already mention that each ρ will correspond to some (mathematical) process in P, and each χ to some function in $Obj \rightarrow P$. Whereas $<\alpha, r>$ should be read as: have r executed by object α, the construct $\alpha : \chi$ has as intended meaning that the object α is passed as argument to (the function which is the meaning of) χ. The construct $e : \chi$ (special case of $e : g$) is normally evaluated by some object, say β. The value of the expression e is determined (with respect to β); eventually, its value, say γ, is passed on to χ (which itself may be a labeled construct, e.g., $<\alpha, g>$). The construct $<\beta, \rho>$ is auxiliary; the role of β is (eventually) no more than to be thrown away.

Below, we shall make extensive use of pairs $<\rho, \sigma>$ - to be read as: execute the labeled continuation ρ with state σ as argument. We adopt the convention that, in such a pair, ρ is always *consistent* with respect to σ. This requires, by definition, that all α appearing as labels in ρ are element of σ_2 (the set of currently active object names). Here we say that

- α appears as label in $<\alpha, r>$ or $<\alpha, g>$
- if α appears as label in ρ, ρ_1, ρ_2 or g, then α appears as label in (ρ_1, ρ_2), (ρ, χ) , (χ, ρ), $e : g$ or $<\beta, \rho>$.

A *transition* is a five-tuple (written in the arrow notation of Section 3) of one of three forms

- $<\rho, \sigma> \rightarrow_d <\rho', \sigma'>$,
- $<\rho, \sigma> \rightarrow_d <\rho', <\alpha, m>>$,
- $<\rho, \sigma> \rightarrow_d <\chi, <\beta, m, \vec{\beta}>>$.

The first possibility reflects a 'normal' step, the second results from an answer statement: $<\alpha, m>$ indicates that object α is willing to execute the method named by m, and the third results from a send expression, asking object β to execute m with parameters $\vec{\beta}$, with a result to be returned, upon completion of the method execution, to χ. A transition rule has the general form as described in Section 3. Rules of the form

$$\frac{<\rho_1, \sigma> \rightarrow_d \cdots}{<\rho_2, \sigma> \rightarrow_d \cdots}$$

with $\cdots$ standing for the *same* pair, will again be abbreviated to $<\rho_2, \sigma> \rightarrow_0 <\rho_1, \sigma>$, or even to $\rho_2 \rightarrow_0 \rho_1$. If ρ_2, ρ_1 are of the form $<\alpha, r_2>$, $<\alpha, r_1>$, respectively, we further simplify the notation to read $r_2 \rightarrow_0 r_1$.

We next present

Definition 4.4 (transition system $\mathcal{T}_{po}$ for $\mathcal{L}_{po}$)

s-rules

- zero-step

$$\begin{aligned}(x := e) : r &\rightarrow_0 e : \lambda\alpha.(r{<}\alpha/x{>})\\ (s_1; s_2) : r &\rightarrow_0 s_1 : (s_2 : r)\\ \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} : r &\rightarrow_0 e : \lambda\beta.\textbf{if } \beta \textbf{ then } s_1 : r \textbf{ else } s_2 : r \textbf{ fi}\end{aligned}$$

- axioms

$$\begin{aligned}{<}{<}\alpha, \textbf{answer}(m) : r{>}, \sigma{>} &\rightarrow {<}{<}\alpha, r{>}, {<}\alpha, m{>}{>}\\ {<}{<}\alpha, \textbf{while } e \textbf{ do } s \textbf{ od}, \sigma{>} &\rightarrow\\ {<}{<}\alpha, e : \lambda\beta.\textbf{if } \beta \textbf{ then } s : \textbf{while } e \textbf{ do } s \textbf{ od} : r \textbf{ else } r \textbf{ fi}{>}, \sigma{>}&\end{aligned}$$

e-rules

- zero-step

$$\begin{aligned}\alpha : g &\rightarrow_0 g(\alpha)\\ (s; e) : g &\rightarrow_0 s : (e : g)\\ e!m(e_1, .., e_k) : g &\rightarrow_0 e : \lambda\beta.(e_1 : \lambda\beta_1.(\cdots\\ &\qquad e_k : \lambda\beta_k.({<}\beta, m, \beta_1, .., \beta_k{>}: g)\cdots))\\ {<}{<}\alpha, \textbf{begin var } \vec{x} := \vec{e}; e \textbf{ end} : g &\rightarrow_0\\ {<}{<}\alpha, ((x_1 := e_1; ..; x_k := e_k); e) : g{<}\sigma_1(x_1)(\alpha)/x_1{>}..{<}\sigma_1(x_k)(\alpha)/x_k{>}{>}, \sigma{>}&\end{aligned}$$

- axioms

$$\begin{aligned}{<}{<}\alpha, x : g{>}, \sigma{>} &\rightarrow {<}{<}\alpha, \sigma_1(x)(\alpha) : g{>}, \sigma{>}\\ {<}{<}\alpha, \textbf{new}(c) : g{>}, \sigma{>} &\rightarrow {<}({<}\alpha, \beta : g{>}, {<}\beta, s : \mathrm{E}{>}), \sigma'{>}\end{aligned}$$

where $\beta = new(\sigma_2, c)$, $\sigma' = {<}\sigma_1, \sigma_2 \cup \{\beta\}{>}$, and $d(c)_2 = s$

r, ρ, χ-rules

- zero-step

$$\begin{aligned}\textbf{if } tt \textbf{ then } r_1 \textbf{ else } r_2 \textbf{ fi} &\rightarrow_0 r_1\\ \textbf{if } ff \textbf{ then } r_1 \textbf{ else } r_2 \textbf{ fi} &\rightarrow_0 r_2\end{aligned}$$

$$\begin{aligned}(\lambda\alpha.r)(\beta) &\rightarrow_0 r[\beta/\alpha]\\ g{<}\alpha/x{>}(\beta) &\rightarrow_0 g(\beta){<}\alpha/x{>}\\ {<}\alpha, g{>}(\beta) &\rightarrow_0 {<}\alpha, g(\beta){>}\\ (\chi, \rho)(\beta) &\rightarrow_0 (\chi(\beta)), \rho)\\ (\rho, \chi)(\beta) &\rightarrow_0 (\rho, \chi(\beta))\\ {<}\beta, \rho{>} &\rightarrow_0 \rho\end{aligned}$$

- axioms

$$\begin{aligned}{<}{<}\alpha, {<}\beta, m, \vec{\beta}{>}: g{>}, \sigma{>} &\rightarrow {<}{<}\alpha, g{>}, {<}\beta, m, \vec{\beta}{>}{>}\\ {<}{<}\alpha, r{<}\beta/x{>}{>}, \sigma{>} &\rightarrow {<}{<}\alpha, r{>}, \sigma[\beta/x, \alpha]{>}\end{aligned}$$

- rules for parallel execution

(interleaving)

$$\frac{<\rho,\sigma>\rightarrow <\rho',\sigma'>}{\begin{array}{l}<(\rho,\bar{\rho}),\sigma>\rightarrow <(\rho',\bar{\rho}),\sigma'> \\ <(\bar{\rho},\rho),\sigma>\rightarrow <(\bar{\rho},\rho'),\sigma'>\end{array}}$$

and similar rules with $<\alpha,m>$ replacing σ', or with χ' replacing ρ' and $<\beta,m,\vec{\beta}>$ replacing σ'

(rendez-vous)

$$\frac{<\rho_1,\sigma>\rightarrow <\chi',<\beta,m,\vec{\beta}>>,<\rho_2,\sigma>\rightarrow <\rho'',<\beta,m>>}{<(\rho_1,\rho_2),\sigma>\rightarrow <\beta,\textbf{begin var } \vec{x}:=\vec{\beta};e \textbf{ end} : (\chi',\rho'')>,\sigma>}$$

where $d(class(\beta))_1(m) = \lambda\vec{x}.e$

Explanation. Most of the rules should be clear as refinement of those of $\mathcal{T}_{pp}$. We emphasize that (even when no object label α is explicitly written) all calculations take place as part of named objects: eventually, all access to variables is through the function application $\sigma_1(x)(\alpha)$ in the axiom for $<<\alpha,x:g>,\sigma>$. The answer statement executed in α determines a step $<\alpha,m>$; the send expression $e!m(\vec{e})$ evaluates e and $e_1,..,e_k$ from left to right, and makes a step involving the outcome $<\beta,m,<\beta_1,..,\beta_k>>$. The **new**($c$) expression determines a new object name β (on the basis of the current set of names σ_2 and the class name c), and initiates execution of $<\beta,s:\mathrm{E}>$, where s, the body of class c, is retrieved from $d(c)_2$. In the rendez-vous of ρ_1, ρ_2, where ρ_1 may make a send-step $<\chi',<\beta,m,\vec{\beta}>>$ and ρ_2 a (corresponding) answer step $<\rho'',<\beta,m>>$, the body of the method $\mu = \lambda\vec{x}.e$ associated with m in the declaration is, after appropriate initialization with the parameters $\vec{\beta}$, executed, and the result is eventually passed back to χ'. (If desired, one may refine the rendez-vous rule by the introduction of dependent and independent resumptions, cf. the remark at the end of Section 3.)
We next discuss how to use $\mathcal{T}_{po}$ to determine $\mathcal{O}$ for $\mathcal{L}_{po}$. First, we introduce the domain P which serves as range for $\mathcal{O}$. Corresponding to the three kinds of right-hand sides of a transition (viz. $<\rho',\sigma'>$, $<\rho',<\alpha,m>>$, $<\chi,<\beta,m,\vec{\beta}>>$), it is natural to define P as solution of the equation

$$\begin{aligned}P = \{p_0\} \cup (\Sigma \rightarrow \mathcal{P}_{co}(\Sigma \times P \quad \cup \\ Obj \times M \times P \cup Obj \times M \times Obj^* \times (Obj \rightarrow P)))\end{aligned}$$

Using this P, we define $\mathcal{O}$ as fixed point of a contracting higher order operator Φ_d based on $\mathcal{T}_{po}$. Since we now deal with transitions yielding both $<\rho',..>$ and $<\chi',..>$ results, we introduce Φ_d as an operator on *pairs* of meaning functions $F = <F_1,F_2>$:

Definition 4.5

a. Let $F_1 \in LSySCo \rightarrow P$, $F_2 \in LSyECo \rightarrow Obj \rightarrow P$, and let Φ_d have the type
$\Phi_d : (LSySCo \rightarrow P) \times (LSyECo \rightarrow Obj \rightarrow P) \rightarrow$
$(LSySCo \rightarrow P) \times (LSyECo \rightarrow Obj \rightarrow P)$,
where $\Phi_d(F_1,F_2) =_{df} <\hat{F}_1,\hat{F}_2>$ is given as

$\hat{F}_1(\rho) = p_0$ if all r occurring in ρ are equal to E, and otherwise

$$\hat{F}_1(\rho) = \lambda\sigma.\ \{<\sigma', F_1(\rho')> \mid <\rho,\sigma> \rightarrow <\rho',\sigma'>\} \ \cup$$
$$\{<<\beta,m>, F_1(\rho')> \mid <\rho,\sigma> \rightarrow <\rho', <\beta,m>>\} \ \cup$$
$$\{<<\beta,m,\vec{\beta}>, F_2(\chi')> \mid <\rho,\sigma> \rightarrow <\chi', <\beta,m,\vec{\beta}>>\}$$

and

$\hat{F}_2(\chi) = \lambda\alpha.p_0$ if all r occurring in χ are equal to E, and otherwise

$$\hat{F}_2(\chi) = \lambda\alpha.\lambda\sigma.\ \{<\sigma', F_1(\rho')> \mid <\alpha : \chi,\sigma> \rightarrow <\rho',\sigma'>\} \ \cup$$
$$\{<<\beta,m>, F_1(\rho')> \mid <\alpha : \chi,\sigma> \rightarrow <\rho', <\beta,m>>\} \ \cup$$
$$\{<<\beta,m,\vec{\beta}>, F_2(\chi')> \mid <\alpha : \chi,\sigma> \rightarrow <\chi', <\beta,m,\vec{\beta}>>\}$$

b. $\mathcal{O}_d = fix(\Phi_d)_1$, $\mathcal{O}(d, \mathbf{new}(c)) = \mathcal{O}_d(<\alpha, s : \mathrm{E}>)$, where $\alpha = \mathbf{new}(\emptyset, c)$, and $d(c)_2 = s$.

Thus, in order to execute $(d, \mathbf{new}(c))$, the first instance of c is named by α – obtained when the set of active objects is still empty – and execution of the body of this object (given in the declaration of c) is initiated.

4.4 Denotational semantics

Similar to what we did in Section 3, we define the intermediate denotational mappings

$$\mathcal{I}_d : Stat_{po} \rightarrow Obj \rightarrow P \rightarrow P,$$
$$\mathcal{E}_d : Exp_{po} \rightarrow Obj \rightarrow (Obj \rightarrow P) \rightarrow P.$$

Let f range over $Obj \rightarrow P$.

Definition 4.6 (denotational semantics for $\mathcal{L}_{pp}$)

- statements

$$\begin{aligned}
\mathcal{I}_d(x := e)(\alpha)(p) &= \mathcal{E}_d(e)(\alpha)(\lambda\beta.\lambda\sigma.\{<\sigma[\beta/x,\alpha], p>\}) \\
\mathcal{I}_d(m)(\alpha)(p) &= \lambda\sigma.\{<<\alpha,m>, p>\} \\
\mathcal{I}_d(s_1; s_2)(\alpha)(p) &= \mathcal{I}_d(s_1)(\alpha)(\mathcal{I}_d(s_2)(\alpha)(p)) \\
\mathcal{I}_d(\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi})(\alpha)(p) & \\
&= \mathcal{E}_d(e)(\lambda\beta.\mathbf{if}\ \beta\ \mathbf{then}\ \mathcal{I}_d(s_1)(\alpha)(p)\ \mathbf{else}\ \mathcal{I}_d(s_2)(\alpha)(p)\ \mathbf{fi}) \\
\mathcal{I}_d(\mathbf{while}\ e\ \mathbf{do}\ s\ \mathbf{od})(\alpha)(p) &= \lambda\sigma.\{<\sigma, \mathcal{E}_d(e)(\alpha)(\lambda\beta. \\
&\quad \mathbf{if}\ \beta\ \mathbf{then}\ \mathcal{I}_d(s)(\alpha)(\mathcal{I}_d(\mathbf{while}\ e\ \mathbf{do}\ s\ \mathbf{od})(\alpha)(p))\ \mathbf{else}\ p\ \mathbf{fi})>\}
\end{aligned}$$

- expressions

$$\begin{aligned}
\mathcal{E}_d(\beta)(\alpha)(f) &= f(\beta) \\
\mathcal{E}_d(x)(\alpha)(f) &= \lambda\sigma.\{<\sigma, f(\sigma_1(x)(\alpha))>\} \\
\mathcal{E}_d(s; e)(\alpha)(f) &= \mathcal{I}_d(s)(\alpha)(\mathcal{E}_d(e)(\alpha)(f)) \\
\mathcal{E}_d(e!m(\vec{e}))(\alpha)(f) &= \mathcal{E}_d(e)(\alpha)(\lambda\beta.(\mathcal{E}_d(e_1)(\alpha)(\lambda\beta_1.(\cdots \\
&\qquad \mathcal{E}_d(e_k)(\alpha)(\lambda\beta_k.\lambda\sigma.\{<<\beta,m,\vec{\beta}>, f>\})\cdots)))) \\
\mathcal{E}_d(\mathbf{new}(c))(\alpha)(f) &= \lambda\sigma.\{<\sigma', f(\beta)\ ||\ \mathcal{I}_d(s)(\beta)(p_0)>\} \\
&\quad \text{where } \beta = new(\sigma_2, c),\ \sigma' = <\sigma_1, \sigma_2 \cup \{\beta\}> \text{ and } d(c)_2 = s \\
\mathcal{E}_d(\mathbf{begin\ var}\ \vec{x} := \vec{e}; e\ \mathbf{end})(\alpha)(f) & = \\
\lambda\sigma.\mathcal{E}_d(\vec{x} := \vec{e}; e)(\alpha)(\lambda\beta.\lambda\bar{\sigma}.\{<\bar{\sigma}[\sigma_1(x)(\alpha)/x_1]\cdots[\sigma_1(x_k)(\alpha)/x_k], f(\beta)>\})(\sigma)
\end{aligned}$$

The '||'-operator used in the clause for **new**(c) is defined in

Definition 4.7 Let $p_1, p_2 \in P$, $X, Y \in \mathcal{P}_{co}(\cdot)$. We put

$$p_1 \parallel p_2 = \lambda\sigma.((p_1(\sigma) \mathbin{\lfloor\!\lfloor} p_2) \cup (p_2(\sigma) \mathbin{\lfloor\!\lfloor} p_1) \cup (p_1(\sigma) \mid_\sigma p_2(\sigma)))$$

where

$$\begin{aligned} X \parallel p = \; & \{<\sigma, p' \parallel p> \mid <\sigma, p'> \in X\} \quad \cup \\ & \{<<\alpha, m>, p' \parallel p> \mid <<\alpha, m>, p'> \in X\} \quad \cup \\ & \{<<\beta, m, \vec{\beta}>, f \parallel p> \mid <<\beta, m, \vec{\beta}>, f> \in X\} \end{aligned}$$

$$f \parallel p = \lambda\alpha.(f(\alpha) \parallel p)$$

$$\begin{aligned} X \mid_\sigma Y = \{ \; & <\sigma, \mathcal{E}_d(\textbf{begin var } \vec{x} := \vec{\beta}; e \textbf{ end})(\beta)(f \parallel p') \mid \\ & <<\beta, m, \vec{\beta}>, f> \in X, <\beta, m, p'> \in Y \\ & \text{or vice versa, and } d(class(\beta))_1(m) = \lambda\vec{x}.e\} \end{aligned}$$

As in Section 3, the above definitions are circular in that $\mathcal{E}_d$ depends on the definition of $\parallel$, and $\parallel$ depends on the definition of $\mathcal{E}_d$. We again refer to [BV91] for a rigorous definition of a comparable problem. (In the present setting, contractivity of the relevant higher-order operator follows easily from the $<\sigma', \ldots>$ step in the clause for $\mathcal{E}_d(\textbf{new}(c))$ and the $<\sigma, \ldots>$ step in the clause for $X \mid_\sigma Y$.) Also, the definition of $\mathcal{I}_d$ is not well-formed since it is circular in the case of the while statement. This problem as well may be dealt with by the familiar argument.

We are, at last, ready for the final

Definition 4.8 The denotational meaning $\mathcal{D} : \mathcal{L}_{po} \to P$ is given by

$$\mathcal{D}(d, \textbf{new}(c)) = \mathcal{I}_d(s)(\alpha)(p_0)\,,$$

where $\alpha = ncw(\emptyset, c)$ and $s = d(c)_2$.

In Section 5, we shall sketch the proof of the

Second Main Theorem For each $\pi \in \mathcal{L}_{po}$, $\mathcal{O}(\pi) = \mathcal{D}(\pi)$.

5 Equivalence of $\mathcal{O}$ and $\mathcal{D}$

We shall provide a detailed presentation of the proof that $\mathcal{O}$ and $\mathcal{D}$ coincide on $\mathcal{L}_{pp}$. For $\mathcal{L}_{po}$, we shall only outline the main ideas.

We start with the equivalence proof for $\mathcal{L}_{pp}$. We assume the various definitions from Section 3; in addition we give several further definitions which will link the syntactic continuations r to their denotations involving semantics continuations.

Definition 5.1 The mappings

$$\begin{aligned} \mathcal{R}_d &: SySCo \to P \\ \mathcal{G}_d &: SyECo \to V \to P \end{aligned}$$

are given as follows

a.

$$\begin{aligned}
\mathcal{R}_d(\mathrm{E}) &= p_0 \\
\mathcal{R}_d(s : r) &= \mathcal{I}_d(s)(\mathcal{R}_d(r)) \\
\mathcal{R}_d(e : g) &= \mathcal{E}_d(e)(\mathcal{G}_d(g)) \\
\mathcal{R}_d(r_1, r_2) &= \mathcal{R}_d(r_1) \parallel \mathcal{R}_d(r_2) \\
\mathcal{R}_d(\textbf{if } \beta \textbf{ then } r_1 \textbf{ else } r_2 \textbf{ fi}) &= \textbf{if } \beta \textbf{ then } \mathcal{R}_d(r_1) \textbf{ else } \mathcal{R}_d(r_2) \textbf{ fi} \\
\mathcal{R}_d(r{<}\alpha/x{>}) &= \lambda\sigma.\{{<}\sigma[\alpha/x], \mathcal{R}_d(r){>}\} \\
\mathcal{R}_d(g(\alpha)) &= \mathcal{G}_d(g)(\alpha)
\end{aligned}$$

b.

$$\mathcal{G}_d(\lambda\alpha.r) = \lambda\beta.\mathcal{R}_d(r[\beta/\alpha]),\ \beta \text{ fresh}$$

We now state a central lemma relating the transition system $\mathcal{T}_{pp}$ and the $\mathcal{R}_d$-function:

Lemma 5.2 If $r_1 \rightarrow_0 r_2$ then $\mathcal{R}_d(r_1) = \mathcal{R}_d(r_2)$.

Proof In all the cases this is immediate by the definitions of $\mathcal{T}_{pp}$ and of $\mathcal{R}_d$. □

Next, we introduce complexity measures on *SySCo* and *SyECo* (and on $Stat_{pp}$, Exp_{pp}), which will play a role in an inductive argument in the proof of the key theorem below.

Definition 5.3 The mappings $\| \cdot \|_r : SySCo \rightarrow N$ (and analogously $\| \cdot \|_g$, $\| \cdot \|_s$, $\| \cdot \|_e$) are defined by

a. $\| \mathrm{E} \|_r = 0$, $\| s : r \|_r = \| s \|_s + \| r \|_r$, $\| e : g \|_r = \| e \|_e + \| g \|_g$, $\| (r_1, r_2) \|_r = \| r_1 \|_r + \| r_2 \|_r$, $\| r{<}\alpha/x{>} \|_r = \| r \|_r$, $\| g(\alpha) \|_r = \| g \|_g + \| \alpha \|_e$, $\| \textbf{if } \beta \textbf{ then } r_1 \textbf{ else } r_2 \textbf{ fi} \|_r = \max(\| r_1 \|_r, \| r_2 \|_r) + 1$.

b. $\| \lambda\alpha.r \|_g = \| r \|_r$.

c. $\| x := e \|_s = \| x \|_e + \| e \|_e + 1$, $\| m \|_s = 1$, $\| s_1; s_2 \|_s = \| s_1 \|_s + \| s_2 \|_s + 1$, $\| \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} \|_s = \| e \|_e + (\max(\| s_1 \|_s, \| s_2 \|_s) + 1) + 1$, $\| \textbf{while } e \textbf{ do } s \textbf{ od} \|_s = \| e \|_e + \| s \|_s + 1$, $\| \textbf{new}(s) \|_s = \| s \|_s + 1$, $\| \textbf{begin var } x := e; s \textbf{ end} \|_s = \| x := e; s \|_s + 1$.

d. $\| \alpha \|_e = \| x \|_e = 1$, $\| \phi(e_1, .., e_k) \|_e = 1 + \| e_1 \|_e + \cdots + \| e_k \|_e + 1$, $\| s; e \|_e = \| s \|_s + \| e \|_e + 1$.

It is not difficult to verify that

Lemma 5.4 If $r_1 \rightarrow_0 r_2$ then $\| r_1 \|_r > \| r_2 \|_r$.

Proof By the various definitions. Note, e.g., that $\| \phi(e_1, .., e_k) \|_e = 1 + \sum_{i=1}^{k} \| e_i \|_e + 1$, but $\| \bar{\phi}(\alpha_1, .., \alpha_k) \|_e = 1$, since $\bar{\phi}(\alpha_1, .., \alpha_k)$ is an element of V. □

The main step leading to the proof that $\mathcal{O} = \mathcal{D}$ on $\mathcal{L}_{pp}$ now follows. The key idea is to show (following a method from [KR90]) that the denotational mapping $\mathcal{R}_d$ is a fixed point of the contracting higher-order operator Φ_d which we used earlier to *define* $\mathcal{O}_d$. This then implies that $\mathcal{R}_d = \mathcal{O}_d$, from which $\mathcal{O} = \mathcal{D}$ is immediate.

Theorem 5.5 $\Phi_d(\mathcal{R}_d)(r) = \mathcal{R}_d(r)$, for all $r \in SySCo$.

Proof Induction on $\| r \|_r$. If $r = \mathrm{E}$, the result is clear. We now discuss a selection of subcases for r, leaving the most difficult case that $r = (r_1, r_2)$ to the last.

- $r \equiv (x := e) : r_1$

$$\begin{aligned}
&\Phi_d(\mathcal{R}_d)((x := e) : r_1) \\
&\quad = \text{ def. } \Phi_d, \text{ def. } \mathcal{T}_{pp} \\
&\Phi_d(\mathcal{R}_d)(e : \lambda\alpha.r_1{<}\alpha/x{>}) \\
&\quad = \text{ ind. hyp.} \\
&\mathcal{R}_d(e : \lambda\alpha.r_1{<}\alpha/x{>}) \\
&\quad = \text{ Lemma 5.2} \\
&\mathcal{R}_d((x := e) : r_1).
\end{aligned}$$

- $r \equiv (s_1; s_2) : r_1$

$$\begin{aligned}
&\Phi_d(\mathcal{R}_d)((s_1; s_2) : r_1) \\
&\quad = \text{ def. } \Phi_d, \text{ def. } \mathcal{T}_{pp} \\
&\Phi_d(\mathcal{R}_d)(s_1 : (s_2 : r_1)) \\
&\quad = \text{ ind. hyp.} \\
&\mathcal{R}_d(s_1 : (s_2 : r_1)) \\
&\quad = \text{ Lemma 5.2} \\
&\mathcal{R}_d((s_1; s_2) : r_1)
\end{aligned}$$

- $r \equiv \textbf{while } e \textbf{ do } s \textbf{ od} : r_1$

$$\begin{aligned}
&\Phi_d(\mathcal{R}_d)(\textbf{while } e \textbf{ do } s \textbf{ od} : r_1) \\
&\quad = \text{ def. } \Phi_d, \text{ def. } \mathcal{T}_{pp} \\
&\lambda\sigma.\{{<}\sigma, \mathcal{R}_d(e : \lambda\beta.\textbf{if } \beta \textbf{ then } (s; \textbf{while } e \textbf{ do } s \textbf{ od}) : r_1 \textbf{ else } r_1 \textbf{ fi}){>}\} \\
&\quad = \text{ def. } \mathcal{R}_d \\
&\lambda\sigma\{{<}\sigma, \mathcal{E}_d(e)(\lambda\beta.\textbf{if } \beta \textbf{ then } \mathcal{I}_d(s)(\mathcal{I}_d(\textbf{while } e \textbf{ do } s \textbf{ od})(\mathcal{R}_d(r_1))) \textbf{ else } \mathcal{R}_d(r_1) \textbf{ fi}){>}\} \\
&\quad = \text{ def. } \mathcal{I}_d \\
&\mathcal{I}_d(\textbf{while } e \textbf{ do } s \textbf{ od})(\mathcal{R}_d(r_1)) \\
&\quad = \text{ def. } \mathcal{R}_d \\
&\mathcal{R}_d(\textbf{while } e \textbf{ do } s \textbf{ od} : r_1)
\end{aligned}$$

$r \equiv (x : g)$

$$\begin{aligned}
&\Phi_d(\mathcal{R}_d)(x : g) \\
&\quad = \text{ def. } \Phi_d, \ \mathcal{T}_{pp} \\
&\lambda\sigma.\{{<}\sigma, \mathcal{R}_d(\sigma(x) : g){>}\} \\
&\quad = \text{ def. } \mathcal{R}_d \\
&\lambda\sigma.\{{<}\sigma, \mathcal{E}_d(\sigma(x))(\mathcal{G}_d(g)){>}\} \\
&\quad = \text{ def. } \mathcal{E}_d \\
&\lambda\sigma.\{{<}\sigma, \mathcal{G}_d(g)(\sigma(x)){>}\} \\
&\quad = \text{ def. } \mathcal{E}_d \\
&\mathcal{E}_d(x)(\mathcal{G}_d(g)) \\
&\quad = \text{ def. } \mathcal{R}_d \\
&\mathcal{R}_d(x : g)
\end{aligned}$$

- $r \equiv (r_1, r_2)$ Take any σ.

$$
\begin{array}{l}
\Phi_d(\mathcal{R}_d)(r_1,r_2)(\sigma) \\
\quad = \text{ def. } \Phi_d \\
\{<\tau, \mathcal{R}_d(\bar{r})> \mid <(r_1,r_2),\sigma> \rightarrow <\bar{r},\tau>\} \\
\quad = \text{ def. } \mathcal{T}_{pp} \\
\{<\tau', \mathcal{R}_d(r',r_2)> \mid <r_1,\sigma> \rightarrow <r',\tau'>\} \ \cup \\
\{<\tau'', \mathcal{R}_d(r_1,r'')> \mid <r_2,\sigma> \rightarrow <r'',\tau''>\} \ \cup \\
\{<\sigma, \mathcal{R}_d(s:(r',r''))> \mid <r_1,\sigma> \rightarrow <r',m>, <r_2,\sigma> \rightarrow <r'',\bar{m}>, d(m) = s\} \\
\quad = \\
\{<\tau', \mathcal{R}_d(r')> \mid <r_1,\sigma> \rightarrow <r',\tau'>\} \between \mathcal{R}_d(r_2) \ \cup \\
\{<\tau'', \mathcal{R}_d(r'')> \mid <r_2,\sigma> \rightarrow <r'',\tau''>\} \between \mathcal{R}_d(r_1) \ \cup \\
\{<\sigma, \mathcal{I}_d(s)(\mathcal{R}_d(r') \parallel \mathcal{R}_d(r''))> \mid <r_1,\sigma> \rightarrow <r',m>, <r_2,\sigma> \rightarrow <r'',\bar{m}>, d(m) = s\} \\
\quad = \text{ see below for (*)} \\
\Phi_d(\mathcal{R}_d)(r_1)(\sigma) \between \mathcal{R}_d(r_2) \ \cup \ \Phi_d(\mathcal{R}_d)(r_2)(\sigma) \between \mathcal{R}_d(r_1) \cup \\
\quad (*)\Phi_d(\mathcal{R}_d)(r_1)(\sigma) \mid_\sigma \Phi_d(\mathcal{R}_d)(r_2)(\sigma) \\
\quad = \text{ ind. hyp.} \\
\mathcal{R}_d(r_1)(\sigma) \between \mathcal{R}_d(r_2) \ \cup \ \mathcal{R}_d(r_2)(\sigma) \between \mathcal{R}_d(r_1) \ \cup \ \mathcal{R}_d(r_1)(\sigma) \mid_\sigma \mathcal{R}_d(r_2)(\sigma) \\
\quad = \\
(\mathcal{R}_d(r_1) \parallel \mathcal{R}_d(r_2))(\sigma) \\
\quad = \\
\mathcal{R}_d(r_1,r_2)(\sigma)
\end{array}
$$

where the step leading to (*) is justified as follows:

$$
\begin{array}{l}
\Phi_d(\mathcal{R}_d)(r_1)(\sigma) \mid_\sigma \Phi_d(\mathcal{R}_d)(r_2)(\sigma) \\
\quad = \\
\{<\sigma, \mathcal{I}_d(s)(p' \parallel p'')> \mid <m,p'> \in \Phi_d(\mathcal{R}_d)(r_1)(\sigma), \\
\qquad\qquad <\bar{m},p''> \in \Phi_d(\mathcal{R}_d)(r_2)(\sigma), d(m) = s\} \\
\quad = \\
\{<\sigma, \mathcal{I}_d(s)(p' \parallel p'')> \mid \ <m,p'> \in \{<\tau', \mathcal{R}_d(r')> \mid <r_1,\sigma> \rightarrow <r',\tau'>\}, \\
\qquad\qquad\qquad <\bar{m},p''> \in \{<\tau'', \mathcal{R}_d(r'')> \mid <r_2,\sigma> \rightarrow <r'',\tau''>\}, d(m) = s\} \\
\quad = \\
\{<\sigma, \mathcal{I}_d(s)(\mathcal{R}_d(r') \parallel \mathcal{R}_d(r''))> \mid <r_1,\sigma> \rightarrow <r',m>, <r_2,\sigma> \rightarrow <r'',\bar{m}>, d(m) = s\}
\end{array}
$$

□

Finally, we can prove

First Main Theorem For $\pi \in \mathcal{L}_{pp}$, $\mathcal{O}(\pi) = \mathcal{D}(\pi)$.

Proof $\mathcal{O}(d,s) = \mathcal{O}_d(s:\mathrm{E}) = \mathcal{R}_d(s:\mathrm{E}) = \mathcal{I}_d(s)(p_0) = \mathcal{D}(d,s)$ □

Remark The above proof suggests that, once $\mathcal{T}_{pp}$ is in the 'right' form, and $\mathcal{D}$ and the semantic operators follow the structure of $\mathcal{T}_{pp}$, then the proof that $\mathcal{O} = \mathcal{D}$ follows more or less 'automatically', i.e., it may be completely syntax driven without an appeal to additional arguments. In [Rut90a], it has been established that this is indeed the case for transition systems (and associated $\mathcal{D}$) of a restricted format. We conjecture that the approach of [Rut90a] may be generalized to cover the present case as well. This

would require, more specifically, a better understanding of how continuations might be incorporated in the method of [Rut90a].

We next outline how the proof that $\mathcal{O} = \mathcal{D}$ on $\mathcal{L}_{po}$ may be structured extending the above approach. We first provide the counterpart of Definition 5.1.

Definition 5.6

a. The mapping $\mathcal{R}_d : SySCo \rightarrow Obj \rightarrow P$ is given by

$$\begin{aligned}
\mathcal{R}_d(\mathrm{E}) &= p_0 \\
\mathcal{R}_d(s : r)(\alpha) &= \mathcal{I}_d(s)(\alpha)(\mathcal{R}_d(r)(\alpha)) \\
\mathcal{R}_d(e : g)(\alpha) &= \mathcal{E}_d(e)(\alpha)(\mathcal{G}_d(g)(\alpha)) \\
\mathcal{R}_d(\textbf{if } \beta \textbf{ then } r_1 \textbf{ else } r_2 \textbf{ fi})(\alpha) &= \textbf{if } \beta \textbf{ then } \mathcal{R}_d(r_1)(\alpha) \textbf{ else } \mathcal{R}_d(r_2)(\alpha) \textbf{ fi} \\
\mathcal{R}_d(r{<}\beta/x{>})(\alpha) &= \lambda\sigma.\{{<}\sigma[\beta/x, \alpha], \mathcal{R}_d(r)(\alpha){>}\} \\
\mathcal{R}_d(g(\beta))(\alpha) &= \mathcal{G}_d(g)(\alpha)(\beta) \\
\mathcal{R}_d({<}\beta, m, \vec{\beta}{>} : g)(\alpha) &= \lambda\sigma.\{{<}{<}\beta, m, \vec{\beta}{>}, \mathcal{G}_d(g)(\alpha){>}\}
\end{aligned}$$

b. The mapping $\mathcal{G}_d : SyECo \rightarrow Obj \rightarrow Obj \rightarrow P$ is given by

$$\begin{aligned}
\mathcal{G}_d(\lambda\beta.r)(\alpha) &= \lambda\gamma.\mathcal{R}_d(r[\gamma/\beta])(\alpha), \gamma \text{ fresh} \\
\mathcal{G}_d(g{<}\beta/x{>})(\alpha) &= \lambda\gamma.\lambda\sigma.\{{<}\sigma[\beta/x, \alpha], \mathcal{G}_d(g)(\alpha)(\gamma){>}\}, \gamma \text{ fresh} \\
\mathcal{G}_d(\chi)(\alpha) &= \mathrm{R}_d(\chi)
\end{aligned}$$

c. The mapping $\mathrm{R}_d : LSySCo \rightarrow P$ is given by

$$\begin{aligned}
\mathrm{R}_d({<}\alpha, r{>}) &= \mathcal{R}_d(r)(\alpha) \\
\mathrm{R}_d(\rho_1, \rho_2) &= \mathrm{R}_d(\rho_1) \,||\, \mathrm{R}_d(\rho_2) \\
\mathrm{R}_d({<}\beta, \rho{>}) &= \mathrm{R}_d(\rho) \\
\mathrm{R}_d(\alpha : \chi) &= \mathrm{X}_d(\chi)(\alpha)
\end{aligned}$$

d. The mapping $\mathrm{X}_d : LSyECo \rightarrow Obj \rightarrow P$ is given by

$$\begin{aligned}
\mathrm{X}_d({<}\alpha, g{>}) &= \mathcal{G}_d(g)(\alpha) \\
\mathrm{X}_d(\chi, \rho) &= \mathrm{X}_d(\chi) \,||\, \mathrm{R}_d(\rho) \\
\mathrm{X}_d(\rho, \chi) &= \mathrm{R}_d(\rho) \,||\, \mathrm{X}_d(\chi)
\end{aligned}$$

Again we have

Lemma 5.7 If $r_1 \rightarrow_0 r_2$ (with respect to $\mathcal{T}_{po}$), then $\mathcal{R}_d(r_1) = \mathcal{R}_d(r_2)$. □

Similar to the proof of Theorem 5.5 (assuming an appropriate generalization of the complexity measures $\| \cdot \|$), we can now prove

Theorem 5.8 $\Phi_d(R_d, \mathrm{X}_d)(\rho, \chi) = {<}R_d(\rho), \mathrm{X}_d(\chi){>}$. □

From this, the second main theorem follows directly:

Second Main Theorem For $\pi \in \mathcal{L}_{po}$, $\mathcal{O}(\pi) = \mathcal{D}(\pi)$. □

References

[AB88] P. America and J.W. de Bakker. Designing equivalent semantic models for process creation. *Theoretical Computer Science*, 60:109–176, 1988.

[ABKR86] P. America, J.W. de Bakker, J.N. Kok, and J.J.M.M. Rutten. Operational semantics of a parallel object-oriented language. In *Proc. POPL'86*, pages 194–208, St. Petersburg, Florida, 1986.

[ABKR89] P. America, J.W. de Bakker, J.N. Kok, and J.J.M.M. Rutten. Denotational semantics of a parallel object-oriented language. *Information and Computation*, 83:152–205, 1989.

[Ame89] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1:366–411, 1989.

[AR89a] P. America and J.J.M.M. Rutten. A parallel object-oriented language: design and semantic foundations. In J.W. de Bakker, editor, *Languages for Parallel Architectures: Design, Semantics, Implementation Models*, Wiley Series in Parallel Computing, pages 1–49. Wiley, 1989.

[AR89b] P. America and J.J.M.M. Rutten. Solving reflexive domain equations in a category of complete metric spaces. *Journal of Computer and System Sciences*, 39:343–375, 1989.

[AR90] P. America and J.J.M.M. Rutten. A layered semantics for a parallel object-oriented language. Technical Report CS-R9052, CWI, Amsterdam, 1990. To appear in "Foundations of Object-Oriented Languages", LNCS 489, Springer, 1991.

[BBKM84] J.W. de Bakker, J.A. Bergstra, J.W. Klop, and J.-J.Ch. Meyer. Linear time and branching time semantics for recursion with merge. *Theoretical Computer Science*, 34:135–156, 1984.

[BV91] J.W. de Bakker and E.P. de Vink. CCS for OO and LP. In *Proc. TAPSOFT'91*. LNCS, Springer, 1991. To appear.

[BM88] J.W. de Bakker and J.-J.Ch. Meyer. Metric semantics for concurrency. *BIT*, 28:504–529, 1988.

[BMOZ88] J.W. de Bakker, J.-J.Ch. Meyer, E.-R. Olderog, and J.I. Zucker. Transition systems, metric spaces and ready sets in the semantics of uniform concurrency. *Journal of Computer and System Sciences*, 36:158–224, 1988.

[BW90] J.W. de Bakker and J.H.A. Warmerdam. Metric pomset semantics for a concurrent language with recursion. In I. Guessarian, editor, *Proc. 18ème Ecole de Printemps d'Informatique, Semantique du Parallelisme*, pages 21–49. LNCS 469, Springer, 1990.

[BZ82] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54:70–120, 1982.

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

[Eli91] A. Eliëns. *DLP - a Language for Distributed Logic Programming.* PhD thesis, Universiteit van Amsterdam, 1991.

[Eng89] R. Engelking. *General Topology*, volume 6 of *Sigma Series in Pure Mathematics.* Heldermann, revised and completed edition, 1989.

[GS90] C.A. Gunter and D.S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 633–674. North-Holland, 1990.

[HBR90] E. Horita, J.W. de Bakker, and J.J.M.M. Rutten. Fully abstract denotational semantics for nonuniform concurrent languages. Technical Report CS-R9027, CWI, Amsterdam, 1990.

[KR90] J.N. Kok and J.J.M.M. Rutten. Contractions in comparing concurrency semantics. *Theoretical Computer Science*, 76:180–222, 1990.

[Kur56] K. Kuratowski. Sur une méthode de métrisation complète des certains espaces d'ensembles compacts. *Fundamenta Mathematicae*, 42:114–138, 1956.

[Mac71] S. MacLane. *Categories for the working mathematician*, volume 5 of *Graduate texts in mathematics.* Springer, 1971.

[MV88] J.-J.Ch. Meyer and E.P. de Vink. Applications of compactness in the Smyth powerdomain of streams. *Theoretical Computer Science*, 57:251–382, 1988.

[Par81] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proc 5th GI Conference*, pages 167–183. LNCS 104, Springer, 1981.

[Rut89] J.J.M.M. Rutten. Correctness and full abstraction of metric semantics for concurrency. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 628–659. LNCS 354, Springer, 1989.

[Rut90a] J.J.M.M. Rutten. Deriving metric models for bisimulation from transition system specifications. In M. Broy and C.B. Jones, editors, *Proc. IFIP TC2 Working conference on programming concepts and methods*, pages 155–177. North-Holland, 1990.

[Rut90b] J.J.M.M. Rutten. Semantic correctness for a parallel object-oriented language. *SIAM Journal on Computing*, 19:341–383, 1990.