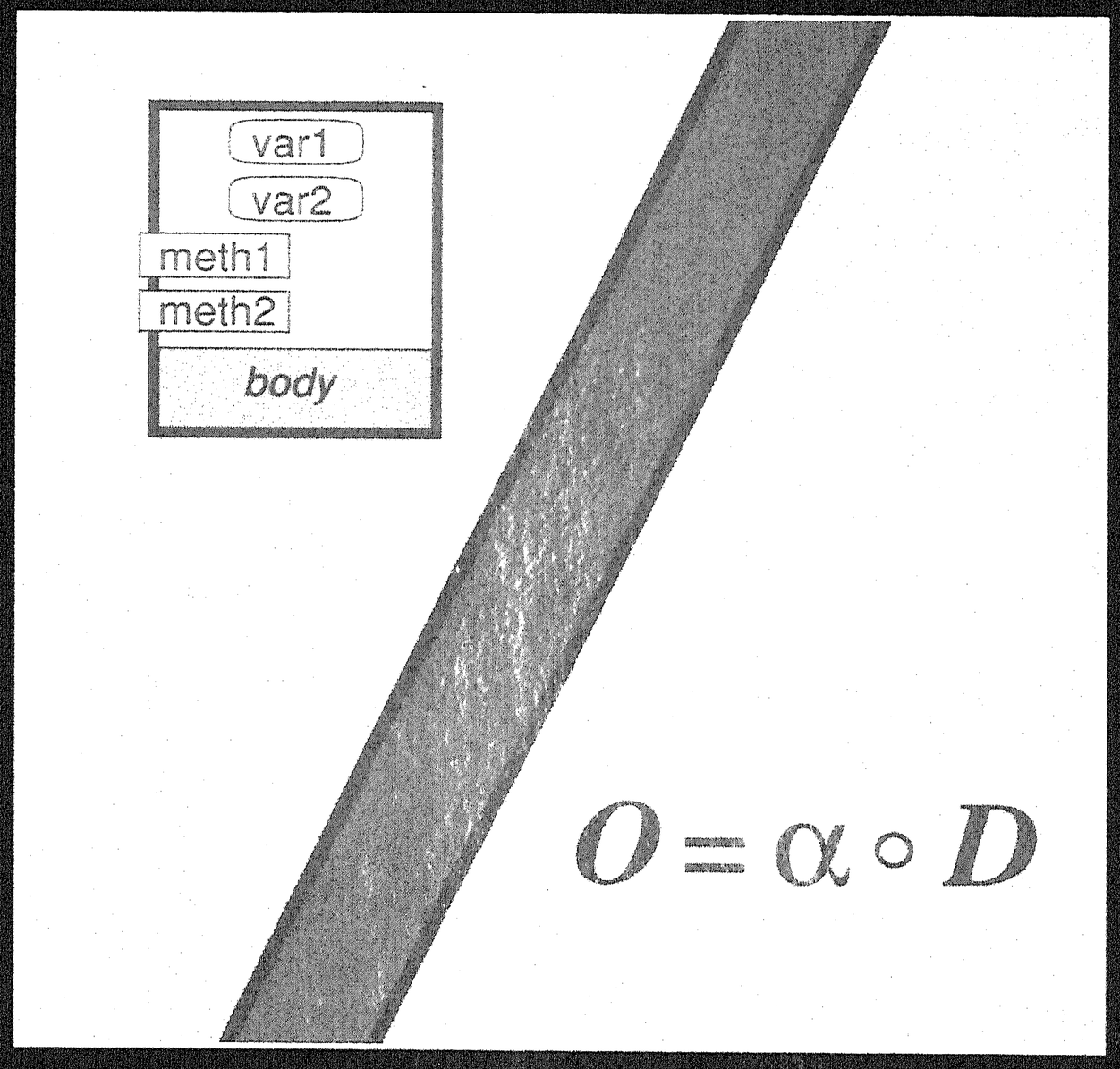
A PARALLEL OBJECT-ORIENTED LANGUAGE:

DESIGN AND SEMANTIC FOUNDATIONS

Pierre America

Jan Rutten



VRIJE UNIVERSITEIT TE AMSTERDAM

A PARALLEL OBJECT-ORIENTED LANGUAGE: DESIGN AND SEMANTIC FOUNDATIONS

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit te Amsterdam,
op gezag van de rector magnificus
dr. C. Datema,
hoogleraar aan de faculteit der letteren,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der wiskunde en informatica
op woensdag 17 mei 1989
in het hoofdgebouw van de universiteit, De Boelelaan 1105

om 13.30 uur door

Petrus Hubertus Maria America
geboren te Kerkrade

en
om 15.00 uur door

Johannes Josephus Martinus Matheus Rutten
geboren te Tilburg

Centrum voor Wiskunde en Informatica
1989

Promotor: prof. dr. J. W. de Bakker
Referent: prof. dr. G. D. Plotkin

The work by P. H. M. America as described in this thesis has been carried out at the Philips Research Laboratories Eindhoven as part of the Philips Research programme.

The work by J. J. M. M. Rutten as described in this thesis has been carried out at the Centre for Mathematics and Computer Science (CWI) in Amsterdam.

The work of both authors has taken place in the context of ESPRIT Project 415:
Parallel Architectures and Languages for Advanced Information Processing:
A VLSI-directed Approach.

Acknowledgements by Pierre America

The first person to thank at this place is of course Jaco de Bakker. He did not only play an important role in the research described in this thesis, but also in introducing me to the national and international scientific community.

The Philips Research Laboratories in Eindhoven, in particular the group *Computer Architecture*, have been an extremely stimulating environment, in which a broad range of disciplines within computer science work together closely and fruitfully. In Henk Bosma, Loek Nijman, and Eddy Odijk, I would like to thank the management for making this possible.

Working in an international research project is a very special experience. I thank all the people involved in ESPRIT project 415 for making this experience such a positive one that I am looking forward to repeat it.

Numerous people have contributed to useful discussions about the design of the languages in the POOL family. Especially the members of the DOOM (ESPRIT 415-A) and PRISMA projects, who have been working very hard to implement POOL and to write programs in it, deserve my deep gratitude.

Frans Kruseman Aretz and Willem-Paul de Roever have helped me by comments, suggestions, and discussions at a high scientific level.

My thanks also extend to Gordon Plotkin, who has refereed this thesis.

Finally I would like to thank my parents, who, with their love and support, have always stimulated me to pursue very ambitious goals, and Twan en Mieke, whose friendship and hospitality over the last years has meant more to me than they might think.

Acknowledgements by Jan Rutten

First of all, I would like to thank Jaco de Bakker, who was my supervisor during the last four years. He taught me many things; in particular, I was inspired by his systematic approach to problems in general, which is always directed towards the isolation of their kernel. Moreover, I have enjoyed his presence as a colleague, here at the CWI and in the context of ESPRIT project 415 activities (ranging from technical discussions to visiting the opera).

I am grateful to Gordon Plotkin for his kind willingness to referee this thesis.

I thank all my collegues at the CWI. Thanks to them it is an inspiring and pleasant place to work. Two of them I would like to thank in particular: Frank de Boer and Joost Kok. Gradually they changed from colleagues to friends and it has given me great pleasure to be in their company, both on scientific and on less serious occasions.

I thank the members of the Amsterdam Concurrency Group: Jaco de Bakker, Frank de Boer, Arie de Bruin, Joost Kok, John-Jules Meyer and Erik de Vink, for a fruitful and enjoyable exchange of ideas.

I thank the members of the Working Group on Semantics of ESPRIT project 415, in which much of the material of this thesis has been discussed. My thanks is in particular due to Frank van der Linden, who has participated in many discussions at Philips on chapters 3 and 4 of this thesis.

I thank Jeff Zucker for an inspiring period of joint work on the problems of fairness, of which the last chapter of this thesis bears witness.

Finally, I am grateful to my parents, Jan Rutten and Corry Rutten-Leys, who have supported and encouraged me, always and in many different ways.

Contents

Introduction

The work described in this thesis has been inspired by the parallel object-oriented language POOL. The thesis describes the design of the language itself and the techniques that have been used to give it a formal semantics. The language POOL, or more precisely, this family of languages, has been developed as a vehicle for writing application programs for a parallel computer. Programming such a parallel machine is considerably more difficult than programming a sequential machine, but if it works, a parallel machine can do the job faster and cheaper than a sequential one.

POOL is directed towards symbolic applications, in contrast to numerical ones. Due to their irregularity, symbolic applications are more difficult to implement correctly and efficiently on a parallel machine. POOL tries to alleviate these problems by supporting an object-oriented programming style, which is currently the best available technique to structure large and complex software systems. In an *object*, pieces of data are closely integrated with the operations that can be applied to them and together they are protected from the outside world by an explicit interface: The internals of an object can only be reached by sending it *messages* of a precisely determined kind. In POOL, such an object also contains a local process, so that it can operate in parallel with the other objects in the system. The same message interface protects the sequential inside of an object against the parallel outside world.

This thesis consists of a collection of papers, several of which have been published separately before:

- Pierre America.
  Issues in the design of a parallel object-oriented language.
  ESPRIT Project 415 Document 452, Philips Research Laboratories, Eindhoven, March 1989.

- Pierre America and Jan Rutten.
  Solving reflexive domain equations in a category of complete metric spaces.
  In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Language Semantics*, pp. 254–288, Lecture Notes in Computer Science 298, Springer-Verlag, 1988. To appear in *Journal of Computer and System Sciences.*

- Pierre America and Jaco de Bakker and Joost Kok and Jan Rutten.
  Denotational semantics of a parallel object-oriented language.
  Report CS-R8626, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, August 1986. The present thesis contains a significantly revised version, which is to appear in *Information and Computation.*

- Pierre America and Jaco de Bakker.
  Designing equivalent semantic models for process creation.
  Theoretical Computer Science, Vol. 60, No. 2, September 1988, pp. 109–176.

- Joost Kok and Jan Rutten.
  Contractions in comparing concurrency semantics.
  Report CS-R8755, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, November 1987. An extended abstract appeared in T. Lepistö and A. Salomaa, editors, *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, pp. 317–332, Lecture Notes in Computer Science 317, Springer-Verlag, 1988.

- Jan Rutten.
  Semantic correctness for a parallel object-oriented language.
  Report CS-R8843, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, October 1988.

- Jan Rutten and Jeffery Zucker.
  A semantic approach to fairness.
  Report CS-R8759, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, November 1987.

The first paper, "Issues in the design of a parallel object-oriented language", gives a more extensive introduction to the language POOL2, the member of the POOL family that is currently being used in subproject A of ESPRIT Project 415. It also discusses the factors that have influenced the most important decisions in the design of this language. Among others, it presents the basic principles of object-oriented programming, it compares several alternative ways of integrating parallelism into an object-oriented language, and it explains the viewpoint taken in POOL towards typing and inheritance. It also gives an overview of the studies on formal aspects of POOL.

The rest of the papers are concerned with formal semantic models for parallel languages, in particular POOL. Object-oriented programming has grown out of an intuitive understanding of what are the important issues in the organization of large software systems. The development of a formal basis for this programming style has been somewhat neglected for a long time. Recently it has become clearer and clearer that such a formal understanding is indispensable in order to make the right choices in the complex process of designing object-oriented systems. This is even more important in *parallel* systems, where we can rely even less on our intuition.

In this thesis we discuss two styles of formal semantics that have been developed for POOL: *operational* semantics and *denotational* semantics. The operational formalism describes the execution of a POOL program in terms of a sequence of transitions between states. The possible transitions are described by a transition relation, which is defined inductively by axioms and rules, corresponding to the syntactic structure of the language. For POOL, an operational semantics along these lines was first described in the paper "Operational Semantics of a Parallel Object-Oriented Language", by Pierre America, Jaco de Bakker, Joost Kok, and Jan Rutten, which appeared in the Conference Record of the 13th Symposium on Principles of Programming Languages, St. Petersburg, Florida, January 13–15, 1986, pp. 194–208. Chapter 7 of this thesis also defines such an operational semantics for POOL.

The denotational semantics works by defining for each syntactic category (e.g., statements, expressions) a *meaning function* that maps a syntactic construct to an element of some mathematical domain. Here the main point is the principle of compositionality: the meaning of a composite construct only depends on the meaning of its constituent parts, not on their actual form. This denotational semantics is described in the third paper included here, "Denotational semantics of a parallel object-oriented language". The mathematical domain used here is a *complete metric space*, which is obtained as a solution of a reflexive domain equation. The second paper in this collection, "Solving reflexive domain equations in a category of complete metric spaces", develops a category-theoretic technique by which a large class of these domain equations can be solved (uniquely up to isomorphism).

The next three papers are devoted to establishing the relationship between operational and denotational semantics. The paper entitled "Designing equivalent semantic models for process creation" investigates this relationship in the context of four languages, ranging from a very simple language with uninterpreted atomic action and a static process structure to a language where the individual processes can store and communicate data and where new processes can be created dynamically. For each of these languages it is proved that the operational semantics and the denotational semantics are *equivalent*, or in other words, that the denotational semantics *is* correct with respect to the operational semantics. This means that there exists an abstraction operator that takes the denotational semantics of a program and, by stripping away the structure necessary for compositionality, produces exactly the operational semantics.

The technique used to prove this is essentially based on the introduction of semantic operators that replace the so-called *continuations* used in the denotational semantics. Unfortunately this leads to long and complicated proofs. Therefore, in the next paper, "Contractions in comparing concurrency semantics", a different technique is developed, which defines the semantic functions themselves as fixed point of some higher-order operators and relates these operators to each other. In the sixth paper, "Semantic correctness for a parallel object-oriented language", this technique is applied to the language POOL, with all its semantically essential constructs, thereby establishing the correctness of the denotational semantics with respect to the operational semantics.

The last paper in this collection, "A semantic approach to fairness", deals with fair processes and fair operations on processes, in the same context of complete metric spaces as the preceding papers. For a simple semantic model, which can be used for the denotational semantics of languages with uninterpreted atomic actions, it is shown how to derive from any process a fair version, which does not postpone certain alternatives forever when repeatly choices must be made. Moreover, it shows how a fair version of the infinite iteration of a single process can be constructed.

Issues in the design of a parallel object-oriented language

Pierre America

Abstract

This document discusses the considerations that have played a role in the design of the language POOL2. This language integrates the structuring techniques of object-oriented programming with mechanisms for expressing parallelism. We introduce the basic principles of object-oriented programming and its significance for program development methodologies. Several approaches for integrating objects and parallelism are compared and arguments for the choices made in POOL2 are presented. We also explain why inheritance is not yet included in POOL2. A brief overview of the research in formal aspects of POOL is given. Finally we indicate some directions for future developments.

1 Introduction

It is generally accepted that the speed of computers that are organized according to the traditional Von Neumann model is approaching its physical limits. In this model, instructions and data are transported back and forth between processor and memory through the famous "Von Neumann bottleneck" and as memories become larger and processors faster, we come closer and closer to the limit that the speed of light imposes on the bandwidth of this bottleneck. A large number of solutions to this problem have been proposed, ranging in radicality from caches [Smi82], which serve as a kind of "impedance adapters" between a fast processor and a slow memory, to completely different computer organizations that are to be combined with revolutionary models of computations (see, e.g., [FFGL88,TBH82]).

An approach in between these extremes proposes the use of a number of traditional processors, each with its own private memory and connected together by a network by which they can exchange information. Provided the network is designed carefully, this organization is scalable to a very large number (several thousands) of processors. Several concrete architectures are based on this general principle [Hil85,Odi87,Sei85].

A problem at least as difficult as designing such parallel machines is how to program them. Traditional programming languages such as Fortran and Pascal are closely related to the von Neumann architecture: they describe a single sequence of actions that the computer should perform. It is not at all an easy task to transform such a program automatically to an equivalent program that makes efficient use of the opportunities for parallelism provided by the hardware. Only for numeric applications, which often have a simple control structure that is largely independent of the actual values of the data, attempts in this direction have been successful, first for vector computers [Ken80] and more recently also for MIMD computers [ACK87]. For symbolic computations, with their more irregular and data-dependent structure, the automatic exploitation of parallelism in traditional programming languages is much more difficult and it has not yet lead to results that are useful in practice.

A drastic approach to this problem is to use pure functional [Bac78,Tur85] or logic [Kow79] programming languages. The idea is that a program in these languages only expresses *what* information the programmer wants, not *how* this should be obtained. By only giving the essential dependencies between input and output, this should leave enough freedom for the implementation to detect and exploit the parallelism automatically. And in fact, the detection of potential parallelism in these languages is quite easy. However, its exploitation has turned out a much more difficult task than it had been initially assumed. Despite the advances made in the last years, the use of implicit parallelism in purely functional and logic languages is not yet understood well enough to be able to base a complete system exclusively on this kind of languages. Therefore it remains necessary to use languages that provide explicit mechanisms for expressing and controlling parallelism.

Dealing with parallelism is not the only problem in programming symbolic applications for parallel machines. The organization of the software itself, data structures, algorithms, etc., for large and complex applications is often a very difficult matter,

where reliability, flexibility, and user-friendliness are important issues. This establishes a real challenge for software technology. A promising approach to meet this challenge is object-oriented programming [Cox86,Mey88]. Object-oriented programming languages offer excellent support for modularity and encapsulation. Object-oriented software development methods provide a relatively high degree of flexibility and reusability.

In this paper we discuss the most important issues that have played a role in the design of the programming language POOL2 [Ame88a]. This language integrates the structuring principles of object-oriented programming with mechanisms for expressing parallelism. It is intended for formulating applications in the area of symbolic computing such that they can be executed on a parallel machine called DOOM (Decentralized Object-Oriented Machine) [Odi87]. The language POOL2 is developed from earlier languages in the POOL family, in particular POOL-T [Ame85b,Ame87a].

Section 2 explains the basic principles of object-oriented programming and briefly discusses its impact on software technology. In section 3 we introduce and compare several different ways in which object-oriented programming can be integrated with mechanisms for expressing parallelism. Section 4 then gives an overview of several new language concepts in POOL2 and section 5 explains why inheritance is not one of them. Then in section 6 an overview is given of the formal studies related to POOL. Finally, section 7 presents some conclusions and indicates some possible directions for future developments.

2 Object-oriented programming

2.1 Basic principles

In the object-oriented programming style a system is described as a collection of *objects* (see figure 1). An object is best defined as an integrated unit of *data* and *procedures* acting on these data. One can think of it as a box that stores some data and has the possibility to act on these data. The data in an object are stored in *variables*. The contents of a variable can be changed by executing an assignment statement.

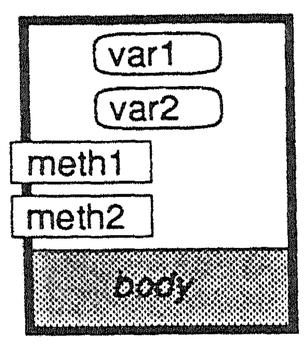


Figure 1: A POOL object

A very important principle is that one object's variables are not accessible to other objects: they are strictly private. In other words, the box has a thick wall around it, which separates the inside from the outside. The only way in which objects can interact is by sending *messages* to each other (see figure 2). Such a message is in fact a request from the sender for the receiver to execute a procedure. This kind of procedures, which are executed in response to messages, are called *methods* in POOL. The receiver decides whether and when it executes such a method, and in some cases it even depends on the receiver which method is executed (see section 5). In general, the sender of the message can include some parameters to be passed to the method and the method can return a result, which is passed back to the receiver. In this way objects can cooperate and communicate. It is important to note that this interaction between objects can only occur according to this precisely determined message interface. Thus every object has the possibility and the responsibility to maintain its own local data in a consistent state.

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

Several object-oriented programming languages use different mechanisms to de-

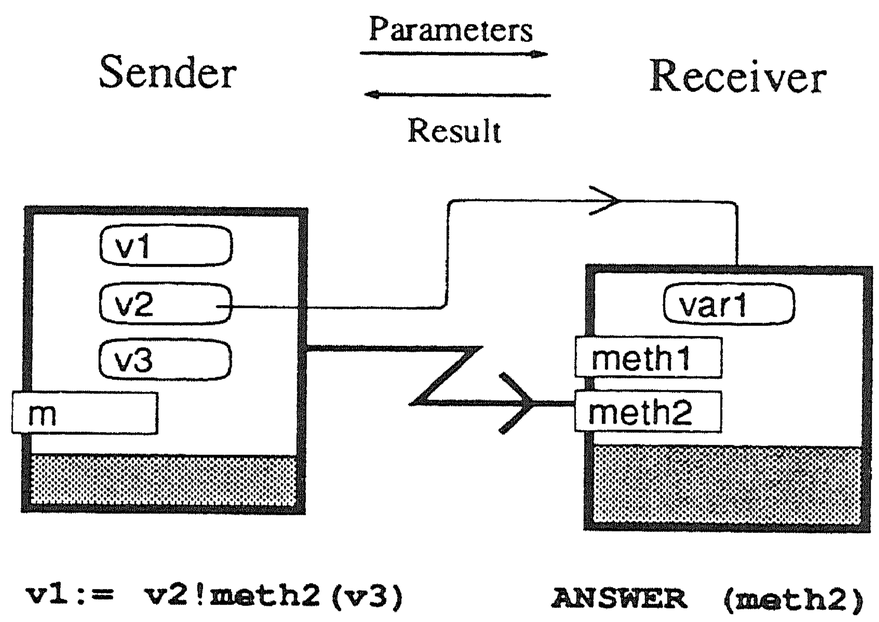


Figure 2: Sending a message

scribe object creation. In general it is agreed upon that creating new objects is not a natural task for the existing instances of the same class (where would the first instance come from?) but rather for the class itself. In Smalltalk-80 [GR83] classes are considered to be objects themselves: they can also be created and changed dynamically. Therefore it is natural to describe object creation in *class methods:* a new object can be created by sending an appropriate message to the class. In POOL it is not natural to consider classes as objects, because we do not want them to change during program execution (see also section 5). Therefore in POOL the creation of new objects is done by *routines*, a kind of procedures different from methods. Routines are not associated with certain objects and they do not have direct access to any object's variables. Instead, in general (but see section 4.1) a routine is associated with a *class*, and it can be executed by any object that knows it. By encapsulating the creation of new objects in routines, it can be ensured that such a new object is properly initialized before it is used.

It is interesting to discuss the nature of the data that is stored and manipulated in the objects. In general, a variable contains a *reference* to some object. Also in parameters and results of methods, references are transferred. Some languages, like Objective-C [Cox86] and Eiffel [Mey87], in addition have some other built-in data types, like integers and characters, that can be manipulated by the objects. These languages are sometimes called *hybrid* object-oriented languages. By contrast, in *pure* object-oriented languages, like Smalltalk-80 [GR83] and POOL, every data item is represented by (a reference to) an object. In these languages, even very simple things like integers are conceptually modelled as objects. For example the addition 3+4 is performed by sending to the object 3 a message mentioning the method add and having (a reference to) the object 4 as a parameter. In response to this message, the object 3 somehow knows how to add itself to the parameter object and it returns the result, a reference to the object 7, to the sender of this message. Of course, this is just

the conceptual view: in an actual implementation some optimizations will take place so that these operations can be performed much more efficiently using the hardware facilities for integer addition.

2.2 A simple example

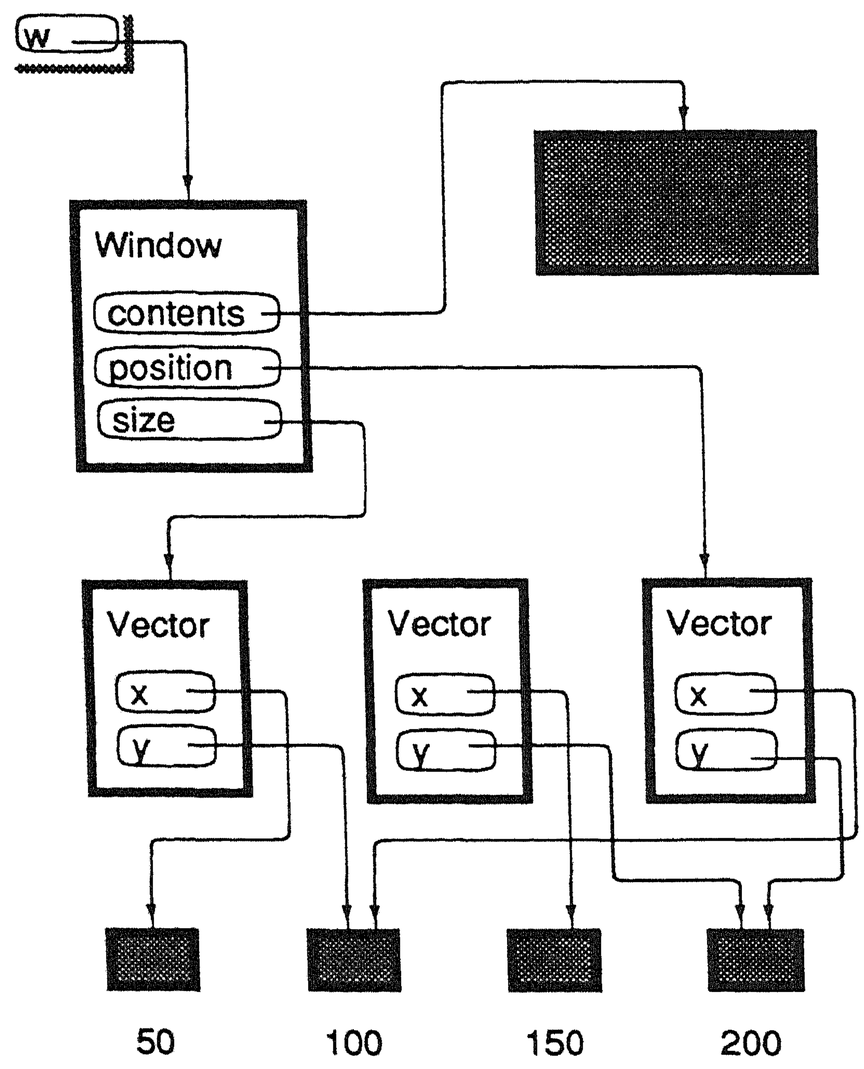


Figure 3: A few objects

Let us illustrate the concepts mentioned above by means of an example. Figure 3 shows a few objects in a certain state during the execution of a program. We see an instance of the class `Window`, which has three variables, `contents`, `position`, and `size`. The variable `contents` refers to an unspecified other object (drawn as a "black box"). The variables `position` and `size` each refer to an object of the class `Vector`. Instances of the class `Vector` have two variables, `x` and `y`. Both `x` and `y` refer to integers, which are also objects, instances of the class `Integer`. Integers are drawn as small "black" boxes. Note that these small black boxes do not represent objects in which integers are *stored*. Rather, they themselves *are* the integers. This is also illustrated by the fact that whenever two different variables, possibly in different objects, have the same integer value (e.g., 50) then they refer to the same integer object, instead of storing each a separate copy of the integer or referring to different objects that store the same

integer.

In this section we shall concentrate on the class Window. The following piece of POOL2 code sketches how it could be described.

```
CLASS Window

VAR contents        : Object
    position, size : Vector

METHOD move (to : Vector) : Window
BEGIN
   position := to;
   display_contents ();     %% a method call
   RESULT SELF
END move

METHOD display_contents () : Window
%% only for internal use!
BEGIN
   ...          %% actual text not relevant here
   RESULT SELF
END display_contents

METHOD where () : Vector
BEGIN
   RESULT position
END where

ROUTINE create (cont : Object, pos, siz : Vector) : Window
TEMP w : Window
BEGIN
   w := Window.new (); %% the standard routine new
   w ! init (cont, pos, siz);
   RESULT w
END create

METHOD init (cont : Object, pos, siz : Vector) : Window
%% for internal use only!
BEGIN
   contents := cont;
   position := pos;
   size     := siz;
   display_contents ();
   RESULT SELF
END init
```

```
END Window
```

A little explanation is appropriate here. First the variables are declared, each with its type. A variable may only refer to instances of the class that is indicated by the type of this variable. Then the methods and routines are defined. In the method `move` another method, `display_contents`, is called directly (without sending a message). In some other object-oriented languages this is done by sending a message to the expression `SELF`, which always indicates the object executing this expression.

We see that for every access to the internal variables of an object, a method is needed: the method `where` simply reads the value of the variable `position`, whereas the method `move` essentially assigns a new value to this variable. In the latter case the method also ensures that the internal consistency is maintained, which here means that the window is actually displayed at the point indicated by the variable `position`. This illustrates how methods can be used to provide the outside world with a controlled access to the internal variables of an object.

Note that methods and routines can have *temporary variables*, which only exist during the invocation of the method/routine. To distinguish them from the variables that exist from the object's creation onward, the latter are also called *instance variables*. Within a method or routine, the parameters can also occur as expressions, just like the temporary and instance variables, but parameters are not allowed as the left-hand side of an assignment.

This example also illustrates the typical use of routines for the creation and initialization of new objects. The routine `create` takes care of this. The creation of a new object is done by calling the routine `new`, which is automatically provided for the class `Window` by the language. Then the routine `create` immediately sends an initializing message to the newly created object. In our example this is indicated by a so-called send statement:

```
w ! init (cont, pos, siz)
```

which sends a message to the object referred to by the variable `w`, specifying the method `init` and as parameters the values of the expressions `cont`, `pos`, and `siz`. In response to this message, the object initializes its variables with the information contained in these parameters and brings itself into a consistent state (by executing the method `display_contents`). Only then the routine `create` returns the new object to its caller. (In fact, the language POOL2 provides a more convenient notation to make sure that newly created objects are properly initialized. We have not used this here in order to illustrate clearly the basic principles.)

Let us finally give a small example of code that uses this class `Window`. This code could appear, for instance, in the definition of another class. We assume that `w` is a variable of type `Window` and that `v`, `v1`, and `v2` are variables of type `Vector` that have already been initialized. Finally `something` should refer to an arbitrary object that can be displayed in our window. In these circumstances, the following piece of code creates a new window, moves it around and asks its new position:

```
w := Window.create (something, v1, v2);
w ! move (v);
v1 := w ! where ()
```

Note that to the users of the class Window we would like to grant only the routine `create` and the methods `move` and `where`, and to hide the routine `new` and the methods `init` and `display_contents`. In POOL, this is made possible by the unit mechanism, explained in section 3.3.

2.3 Relationship with modules and abstract data types

One may argue that object-oriented programming as introduced above is just a formulation of well-known principles in new terminology: One can compare an object with a record and sending a message with a procedure call (in fact, this comes very close to the way in which many object-oriented languages are implemented). For structuring software there are already concepts like modules and abstract data types. Let us therefore first look at the relationship between modules, abstract data types, and objects.

First we consider the notion of *modules*, as it appears in, e.g., Modula-2 [Wir82] and Ada [ANS83] (where the name *package* is used). Such a module is nothing more than a collection of declarations of data types, variables, procedures, etc., provided with an interface that specifies which of these declarations can be used outside the module. The programmer has a large amount of freedom in choosing what to put in one module and where to place the boundaries between modules. It is intended that the grouping of declarations into modules is a meaningful one [Mey82], but the language does not enforce this in any way. It only enforces that the interfaces, once made explicit, are observed.

In programming with *abstract data types*, as exemplified by the notions of 'cluster' in CLU [LAB*81] and 'form' in Alphard [Sha81], there is a clear notion of what is contained in a "module", a data type definition, and what it is about: Such a definition should describe *one* data type, its internal representation and the operations that can be performed on its instances. The interface with the outside world consists of the names of those operations that are to be available outside the data type definition, together with some specification of their behaviour (which is mostly limited to the types of the parameters and results). The internal representation is not accessible from outside the data type definition. With respect to modular programming, abstract data types are much more restrictive in the choice of the boundaries between program units, but on the other hand they offer a much clearer conceptual view of the meaning of these units.

Note that both modules and abstract data types only offer the guarantee that the facilities defined in a program unit are used correctly (that the interfaces are observed) in a *statically typed language* (a language where for every expression it is possible to determine the type of the object it denotes from the program text alone, see section 5.2). In other languages, the use of modules and abstract data types is not

completely useless, because it can give a clearer structure to the program, but it does not offer such a high degree of security as in statically typed languages.

Object-oriented programming is even more restrictive than abstract data types about the allowed constructs in a class definition. In the definition of an abstract data type A, the operations performed on the type can access the internal details of *all* their arguments that are of type A, and there may be more than one of these. In object-oriented programming, however, a method can only access the variables of the object it is associated with (the destination of the corresponding message). So the internal details of only *one object at a time* can be accessed.

Let us illustrate this with an example. Below is the definition of an abstract data type of complex numbers with addition as its only operation (we use an imaginary syntax):

```
TYPE Complex
VAR re, im : Float

OP add (x, y : Complex) : Complex
TEMP z : Complex
BEGIN z := Complex.new ();
      z.re := x.re + y.re;
      z.im := x.im + y.im;
      RESULT z
END add
```

We see that the code of the operation add has access to the re and im variables of both its arguments plus the new object that is to be the result. The difference will be clear with the following corresponding class definition in an object-oriented style:

```
CLASS Complex
VAR re, im : Float

METHOD add (y : Complex) : Complex
%% the first operand is the destination of the message
TEMP z : Complex
BEGIN z : Complex.new ();
      z ! put_re (re + y ! get_re ());
      z ! put_im (im + y ! get_im ());
      RESULT z
 END add

 METHOD put_re (new_re : Float) : Complex
BEGIN re := new_re;
      RESULT SELF
END put_re

METHOD get_re () : Float
```

```
BEGIN RESULT re
END get_re

METHOD put_im (new_im : Float) : Complex
BEGIN im := new_im;
      RESULT SELF
END put_im

METHOD get_im () : Float
BEGIN RESULT im
END get_im
```

Here the code of the method add only has direct access to the variables `re` and `im` of its destination object. Messages must be sent to obtain the real and imaginary part of the second operand y and to fill the variables of the resulting object z.

One can express this difference between modules and abstract data types on the one hand and object-oriented programming on the other hand by saying that with modules and abstract data types, protection takes place at a *syntactic* level (each module is protected against the other modules), whereas with object-oriented programming the protection is at a *semantic* level (each object is protected against the other objects). This results in a finer granularity, because different object are protected against each other even if they are described by the same class definition.

2.4 Impact on software development

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

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

It is true that modules already provide the possibility of encapsulating pieces of

software. However, they do not give guidelines about which definition should be placed together in a module. By choosing a wrong subdivision of a system into modules, it is very well possible to arrive at a collection of modules that not easily adaptable or reusable. The extra value of abstract data types and object-oriented programming is that in addition these techniques give an idea about what belongs in one "module": A class definition describes one class of entities, together with all the operations that can be performed on them. It has turned out in practice that this indeed leads to a better system structure.

Object-oriented programming also leads to a different way of designing software. The common technique of top-down functional design starts from the required end-to-end functionality of a complete program and divides this iteratively into subfunctions until basic language primitives are obtained. The resulting software is not very adaptable to changing requirements, because in practice the changes mostly pertain exactly to this end-to-end functionality. Moreover it is very unlikely that the subfunctions into which the program is divided coincide precisely with subfunctions in another program, which would allow reuse of software, because these subfunctions are obtained in an ad hoc way for each program separately. By contrast, object-oriented design initially focuses on the basic entities (objects) manipulated by the program and it grows towards the required end-to-end functionality in a rather bottom-up way. The resulting software is often easier to adapt to changing circumstances, because these basic entities are not very likely to change. Moreover, this way of designing software leads more often to meaningful software components that can be re-used. (A more extensive discussion of these issues can be found in [Mey88].)

In a strongly typed, sequential environment the extra protection offered by object-oriented programming when compared with abstract data types does not seem very important. Indeed, it may even be a nuisance, as in the above example about complex numbers. One of the reasons nevertheless to choose an object-oriented language in this situation might be that good object-oriented languages are available [Mey87,Str86], whereas languages that directly support abstract data types are not so widely available.

However, as soon as we leave this safe environment, the extra protection becomes really useful. As an example we mention dynamically typed languages, i.e., languages in which every data item has a well-defined type but where it is in general not possible to determine this type from the program text only. A well-known example is Lisp [MAE*80,Ste84]. This kind of languages are often used for *rapid prototyping*, a technique where a "quick and dirty" preliminary version of a program is produced in order to experiment with certain aspects, in particular the user interface. Since the resulting prototype program will not be used for production purposes (it is to be hoped!), reliability is not such an important issue, but flexibility *is* important, because a prototype program must be changed often and quickly. Therefore the use of a dynamically typed language is justified.

Some of the problems with programming in such a language are that it is in general not very well possible to make the structure of the data explicit (everything is coded in lists) and that errors are detected too late and at too low a level (a common error message in Lisp says that you have tried to extract the first element of an empty

list). Object-oriented programming can help here: A class describes the organization of a certain kind of data very clearly: it gives the internal representation and the available operations. Furthermore the object-against-object protection mechanism, unlike the type-against-type protection of abstract data types, also functions in a dynamically typed situation. Therefore errors can be caught earlier: the most common error message in a dynamically typed object-oriented language is that a message has been sent to an object that does not have an appropriate method, which occurs as soon as a data entity is being used in a wrong way.

Object-oriented programming in dynamically typed languages does not stand in the way of flexibility, but it can help in making the structure of a system explicit. This is probably an important reason why object-oriented languages like Smalltalk-80 [GR83] and object-oriented extensions of Lisp [WM80,BDG*87] are so popular for rapid prototyping. Another reason is the support for reusability: If a prototype must be made from scratch, the amount of work this costs can be prohibitive. If however one can make use of a good collection of well-organized software for recurring tasks (for example, handling windows and menus on a bit-mapped display) then sophisticated systems can be constructed very quickly because one only has to take care of the essentials.

Another area in which an object-oriented approach has proved to be valuable is operating systems [Jon78,MT86,WLH81]. Here, again, it is impossible to check statically whether certain operations are permitted and the object-oriented approach gives a good model along which the dynamic checks can be organized. In these systems, the object-oriented principles are often complemented by using *capabilities* instead of just object references. Such a capability not only indicates the identity of an object, but it also explicitly determines the set of operations that may be performed on the object by the holder of the capability. This set may be smaller that the set of all the operations that the object itself admits. It must be admitted that the techniques used in these kinds of operating systems are often quite expensive, so that objects should be fairly large and message should not be sent too often in order to maintain a reasonable performance. Often traditional mechanisms are used to describe the actions of the system on a lower level.

Apart from dynamically typed systems, also parallel programming constitutes an area where the more fine-grained protection of object-oriented programming presents a clear advantage above abstract data types. This is the subject of section 3. Furthermore, even in statically typed systems, there is a structuring mechanism, *inheritance*, which can be used with object-oriented programming but not in general with abstract data types. This mechanism is discussed in detail in section 5.1.

3 Parallelism

3.1 Integrating parallelism in an object-oriented language

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

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

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

Now one can think of several ways to introduce parallelism to object-oriented languages. One possibility is to add processes as an orthogonal concept to the language. In some sense this can be seen as eliminating restriction 1. Several processes can be active at the same time, each one executing an object-oriented program in the way described in section 2.1. These processes act on the same collection of objects; it is even possible that they are executing the same method in the same object at the same time. This way of dealing with parallelism has been adopted by some languages that were initially meant to be purely sequential, such as Smalltalk-80 [GR83] and Trellis/Owl [SCB*86,MK87].

While this approach seems appealing theoretically, it is not so attractive in practice. The point is that it does not at all solve the problems associated with parallelism (we shall come back to this point later). There are still extra facilities needed for synchronization and mutual exclusion. To that end, the above languages provide some built-in classes, for example, semaphores. Even then, the facilities for parallel programming remain rather primitive.

The second approach can be clearly described as relaxing restriction 2 above: Instead of letting an object wait for the result after sending a message, one allows the sender to go on immediately with its own activities. This is called *asynchronous communication*. In this way the sender can execute in parallel with the receiver of the message. It is possible to obtain a large degree of parallelism after a number of messages have been sent. This scheme has been adopted most notably by the family of *actor languages* [Hew77,Lie81,The83,Agh86] but also in [Lan82].

A quite different scheme can be obtained by relaxing the last restriction. Now an object does not always wait quietly until it receives a message, but has an activity of its own, which we shall call its *body*. Execution of the body is started as soon as the object is created, and it takes place in parallel with the other objects in the system. At certain explicitly indicated points the body can be interrupted in order to answer a message. This takes place in the form of a rendez-vous: the sender and the receiver synchronize

(the one that is first willing to communicate waits until the other is ready, too), the parameters are passed to the receiver's method, which is then executed, and finally the result is passed back to the sender (not necessarily at the *end* of the method execution), after which both objects again pursue their own computations independently. This is called *synchronous* communication. In this approach, too, a large degree of parallelism can be obtained, by creating a sufficient number of objects whose bodies can execute in parallel. The languages from the POOL family use bodies as their main mechanism to describe parallelism.

3.2 Comparing different approaches

Let us first compare the above approaches with respect to the criterion of how they help to solve the problems of parallel programming. The key to parallel programming is handling the nondeterminism that results from the unknown relative execution speed of the processes: This nondeterminism should be reduced as much as possible, but a certain amount of it is necessary to make effective use of the parallelism. Now the degree of nondeterminism is increasing very quickly not only with the number of processes, but also with the number of atomic actions in each process, or otherwise stated, with the number of places in each process where it may interact with other processes[1].

Now the disadvantages of the first approach (processes as orthogonal concepts) become very clear: In this approach, a process must expect interaction (perhaps we should call it interference here) from other processes at *every* point of its execution. Therefore the number of different execution sequences of which the programmer has to take care is very large. The extra mechanisms [MK87] added in order to restrict this nondeterminism, for example semaphores, require a disciplined use, which is not enforced by the language. Therefore it is clear that this approach is not suitable for extensive parallel programming.

In fact this issue also plays a role in the traditional dichotomy in parallel programming between shared variables and message passing (see also [AS83]). How cumbersome it is to work with shared variables, when compared with message passing, can also be seen by considering the formalisms for verifying such programs: The classical system to formally verify shared-variable programs [OG76] requires that every assertion in any of the processes is left invariant by every action in every other process. For n processes having each m actions with $m+1$ assertions around them, this requires $n(m+1)m^{n-1}$ checks. Reducing n would reduce the degree of parallelism. Reducing m could be done by increasing the size of atomic actions, and this is precisely what happens with message passing. Moreover, because the communication partner is often indicated explicitly, the checks can be restricted to the set of pairs of corresponding communication statements, which in general leads to a much smaller number of checks. This has been formalized in [AFR80].

---

[1] For example, if we have m processes that do not influence each other's behaviour, and the ith process has n_i atomic actions, then the number of possible interleavings is equal to the multinomial coefficient $(n; n_1, \ldots, n_m) = n!/(n_1! \cdots n_m!)$ where $n = n_1 + \cdots + n_m$.

Of course, the choice between shared variables and message passing is also influence by the underlying machine architecture. In machines with a shared memory between processors (or sequential machines, where the parallelism is virtual) implementing shared variables is trivial, while message passing requires some work. On the other hand, in machines without a shared memory, where the processors exchange information over a communication network, message passing can be mapped directly to the architecture. In these machines it is possible, but very cumbersome and inefficient to implement shared variables. This seems to indicate that even if the machine architecture is not fixed in advanced, it is best to choose message passing instead of shared variables. (DOOM [Odi87], the machine for which POOL2 was developed, does not have shared memory between processors; they communicate via a packet-switching network.)

All these arguments imply that for integrating parallelism in object-oriented languages the two other approaches (asynchronous communication or bodies) are superior: Here the concepts of object and process are effectively unified into one concept, so that the terms 'object' and 'process' have become synonymous. Processes now only interact at clearly defined points: only where messages are sent or answered. Moreover, the possible ways of interaction are limited: only parameters or results may be passed. The variables of each object are protected from access by other objects. If a certain piece of data must be shared among different processes, it can be put in an object of its own. The way in which it can be accessed is then clearly defined by the available methods (and possibly its own body). The language supports to a large degree the discipline necessary in using these mechanisms. Note that inside an object everything happens sequentially. This sequential, deterministic inside is protected from the parallel, nondeterministic outside world by the message interface. Allowing multiple parallel processes to be active inside the same object (as is done, e.g., in Emerald [BHJ*87]) would spoil this comfortable situation.

The choice between asynchronous message passing and the use of bodies for achieving parallelism is much less obvious than the choice against the first approach. Asynchronous communication leads to more flexibility, because the sender does not need to synchronize with the receiver in order to communicate. In this way it is easier in certain cases to keep the available processors in a system busy. On the other hand, asynchronous communication has certain problems associated with it:

For the programmer it is important to realize that the lack of synchronization with asynchronous communication not only increases the system's flexibility in exploiting its resources, but it also increases the degree of nondeterminism: there are more possible executions of such a program than for synchronous communication, and the programmer must ensure that all of these lead to a correct result. Furthermore, the set of messages that have been sent but not yet received constitutes a component of the system's state that does not occur explicitly in the program but is nevertheless very important. (Most formal techniques for asynchronous communication, e.g. [BKT84] explicitly represent these travelling messages; the notable exception is temporal logic [KVR83].)

The most important problem for the implementation is the buffering of messages

that have been sent but not yet received. In principle, it is not admissible just to reserve a fixed buffer space and to block a sender if it tries to send more messages than fit in this buffer, because it would lead to deadlock in programs that are semantically correct. For example, if the sender transmits n messages labelled a and then a message labelled b whereas the receiver first wants to get a b message before answering the a messages, then a deadlock will occur if n is larger than the number of messages that fits in the buffer. On the other hand, in most cases the communication pattern is simpler that this (the receiver does not require such a peculiar order of messages) and in these circumstances one would like to slow down the sender when it gets too far ahead of the receiver. It does not seem possible to solve this problem in general.

Another issue is whether to guarantee that messages travelling from the same sender to the same receiver should arrive in the order in which they were sent. This can be ensured by either using an end-to-end protocol, or by employing a fixed routing between every pair of nodes in a network and making sure that messages are kept in order at each stage of their transmission. In both cases this decreases the performance of the communication system and this penalty would have to be paid even by programs that do not need order preservation.

Let us remark here that it is easy to implement asynchronous communication in a language that has only bodies and synchronous communication: For every message that is to be sent asynchronously, a buffer object is created. The message is sent (synchronously) to the buffer and later the buffer will send it (again synchronously) to the destination. The other way around, implementing synchronous communication with asynchronous communication is also possible, but in certain systems, including actor languages, this is quite cumbersome. The problem is here that an actor cannot selectively wait for messages of a certain kind. Therefore, after a message has been sent and the sender is waiting for the result, it must accept every message that arrives and determine whether it is indeed the expected answer. If not, the message must be stored for later use, or the actor can send it to itself (which would result in some kind of busy waiting).

In POOL we have chosen to use bodies and synchronous communication as the basic mechanism to express parallelism. In most cases, this turns out to be the most natural way to program an application. If the programmer does not explicitly indicate a body, a default body is taken, which continuously answers one message after the other in the order in which they arrive. A method may return its result to the sender of the message before it actually terminates. In this way parallelism can arise with synchronous communication and the default body (this is illustrated by the example in section 3.3). However, in some case an explicit body is needed because it allows to answer messages selectively, indicating the specific kind of messages that are welcome. For example, a buffer might wish to answer only insert messages while it is empty, only extract messages if it is full, and both kinds of messages otherwise. In other cases, the use of a body is not strictly necessary but just more natural, especially in objects that are really active and not just waiting for a request to arrive.

Receiving a message is done in an answer statement, which contains a list of method names. This indicates that exactly one message is to be answered, in principle the first

message that mentions a method occurring in the list. Note that the sender is not indicated (whereas in sending a message the destination is given explicitly). This gives the answering object to react flexibly on the supply of messages, taking the one that comes first without having to commit itself to a specific communication partner. If such a commitment is nevertheless desired, it is often possible to revert the roles of sender and receiver, because synchronous communication transmits information in both directions anyway.

POOL2 also provides a conditional answer statement, that specifies that a message should be answered if a suitable one has already arrived. If no such message is available at the moment, the conditional answer statement will not wait for one but terminate immediately. Again this contributes to an object's flexibility in reacting on the other objects. It would be possible to increase this flexibility even more by allowing an object to indicate a collection of send and receive actions with the intention that one of these actions is performed, preferably the first that can take place. This could be expressed by a generalization of an Ada-like select statement [FY85]. A mechanism to implement this in a POOL context has been developed [Wou88]. Whether this will be incorporated in a future version of the languages will depend on actual performance figures.

In POOL2, asynchronous communication is provided in addition to synchronous communication. Conceptually it is considered as an abbreviation for the mechanism that creates a buffer object for each asynchronous message. This also means that preservation of message ordering is not guaranteed, because the buffer objects may proceed with unknown relative speed. Of course, an implementation is encouraged to use more efficient mechanisms, as long as they have the same semantics. In principle, the programmer is responsible to ensure that a sender of asynchronous messages does not get too far ahead of the receiver.

If we compare POOL with traditional parallel programming languages [AS83] we can easily classify it with respect to criteria like the following:

1. Shared variables or message passing? (POOL: message passing.)
2. Value passing or remote procedure call? (POOL: remote procedure call.)
3. Synchronous or asynchronous message passing? (POOL2: both.)
4. Channels or direct naming of communication partner? (POOL: direct naming of receiver by sender, no naming of sender by receiver.)
5. Static or dynamic process structure? (POOL: dynamic.)

Most of these choices follow directly from the wish to change as little as possible in the mechanisms as we know them from sequential object-oriented programming. The others have been discussed above. The most obvious way in which POOL distinguishes itself from traditional parallel programming languages is by unifying data structures and processes in a single concept of object. This gives rise to a typical style of programming, which is illustrated by the example in the next section.

Finally, let us make some remarks on another issue that is always important in concurrent systems: *fairness*. In POOL there are two requirements on the execution of a program that ensure a certain kind of fairness: The first is the fact that the execution "speed" of any object is arbitrary, but positive. This means that whenever an object can proceed with its execution without having to wait for a message or a message result, it will eventually do so. Clearly this is a very natural and necessary requirement to be imposed on the implementation of a concurrent language. Requiring more precise guarantees about the relative execution speeds of different objects would necessitate a way of measuring those speeds, and even in languages specifically meant for real-time applications (for example Ada [ANS83]) those guarantees are considered too involved to be included in a language definition.

The second requirement on the execution of a POOL program is the condition that all messages sent to a certain object will be stored there in one queue in the order in which they arrive. When that object executes an answer statement, the first appropriate message in the queue will be answered (here 'appropriate' means that the message mentions a method occurring in the answer statement). This condition ensures that it is impossible that an object is sent a message and it executes infinitely many answer statements in which the message could have been answered, without answering this one message. In fact, it is not difficult to see that the latter condition (a message will eventually be answered) is exactly equivalent to the first one (messages are stored in a queue), when one takes the arbitrary but positive speed of the sending object into account.

Note the contrast here with the situation in, e.g., Ada. In Ada, each *entry* (in this situation corresponding with a method name in POOL) has its own queue, and fairness is not guaranteed between different queues. Then it is possible that infinitely many messages with one name are answered without answering a message with another name, even when these messages are answered in a select statement where there is always another open branch for answering the second message. We consider this situation definitely undesirable. In POOL, it may be a little more difficult to implement the de-queuing operation efficiently, but the mechanism is much more convenient for the programmer.

Of course, fairness is only a worst-case guarantee from the language, in a situation where better, quantitative guarantees cannot be given. In practice, it is intended that an object that does not have to wait for another one proceeds as quickly as possible, that messages travel as fast as possible from the sender to the receiver, and that they are answered in an order that approximates as well as possible the first-come-first-serve principle. This is also the reason why message-answering fairness in the language definition is formulated in terms of queues instead of infinite neglection.

3.3 An example: parallel symbol tables

In this section we present a small programming example that shows a typical way of programming in POOL. In this example we implement a parallel version of a symbol table, a data structure that can associate keys with other pieces of information. We

also illustrate the use of a few other elements of the language POOL2, units and generic classes.

Units are the largest pieces of a POOL2 program. They come in two kinds: *implementation* units and *specification* units. An implementation unit contains a collection of class definitions, giving the full details of each class. The corresponding specification unit indicates the interface it other units: it lists the classes that can be used outside of the current unit and for each of these it gives the headers of the available methods and routines. Another unit can import these facilities by mentioning the first unit in its so-called *use list*.

Below is a specification unit that describes the class ST, each instance of which represents a symbol table.

```
SPEC UNIT Symbol_Table

CLASS ST (Info)
%% Each instance is a symbol table containing
%% pairs of a string and an instance of the class Info.

ROUTINE new () : ST (Info)
%% Delivers a new, empty symbol table

METHOD insert (key : String, i : Info) : ST (Info)
%% Inserts a new pair into the destination symbol table.
%% key must not be NIL.
%% If the key is already present, the old Info is overwritten.

ROUTINE search (st : ST (Info), key : String) : Info
%% Retrieves the info stored with this key.
%% If this key is not present in the symbol table, NIL is returned.

END ST
```

The class ST is *generic*, that is, it has a class parameter `Info`, for which an arbitrary class can be filled in when the class ST is used. This allows us to define the class in such a general way that it can be used in many different circumstances without modifying the text. (It would also be possible to make the type of the key a parameter of the class definition. However, in the implementation unit we shall need the fact that keys are ordered. In section 4.1 it is explained how the ordering on keys can be made known to the symbol tables, such that the type of the keys can indeed become a parameter of the class ST.)

The class ST provides its users with two routines and a method. The routine `new` creates and returns a new symbol table object, which is empty initially. The method `insert` adds a new piece of information to the symbol table, consisting of a key, which is an object of type `String`, and an instance of the class `Info`. Finally we have the routine `search`, which tries to look up the `Info` associated with a given key in a symbol table. We shall see below why `search` is a routine instead of a method.

Now here is the first part of the corresponding implementation unit:

```
IMPL UNIT Symbol_Table

CLASS ST (Info)

VAR my_key  : String      %% key stored here
    my_info : Info        %% Info stored here
    left    :   ST (Info) %% all pairs with key < my_key
    right   :   ST (Info) %% all pairs with key > my_key

%% new is a standard routine

METHOD insert (key : String, i : Info) : ST (Info)
BEGIN
   RESULT SELF;          %% rendez-vous ends here
   IF my_key == NIL      %% I am empty
   THEN my_key := key; my_info := i;
        left := new (); right := new ()
   ELSIF key = my_key    %% the key is stored here
   THEN my_info := i
   ELSIF key < my_key
   THEN left  ! insert (key, i)
   ELSE right ! insert (key, i)
   FI
END insert
```

Now we see that a symbol table is internally organized as a tree. Each node in the tree can contain a single key and its associated `Info`. Furthermore it contains references to the left and right subtrees, which contain the other symbol table entries. The routine `new` need not be described explicitly. A routine with this name is supplied automatically by the language. It creates and delivers a new object of the associated class, with all the variables initialized to `NIL` (a reference to *no* object). In our example, this is exactly what an empty symbol table looks like.

The method `insert` returns its result to the sender of the message right at the beginning. In this way, the sender and the receiver are synchronized, but the sender need not wait until the execution of the method is completed. Instead, the rest of the method can execute in parallel with the sender. The actual value returned by the method is not important. Therefore the convention is followed that the object executing the method returns a reference to itself. This is indicated by the expression `SELF`, which denotes the object that is executing the expression.

After having returned the result, the method `insert` determines what to do with the new piece of information. If the symbol table is empty, the new information is stored locally. Otherwise, if the new key happens to be the same as the key already stored here, the local `Info` is simply overwritten. In all other cases, the new key/`Info`

pair is sent to one of the subtrees. In the program text, the operators '=' and '<' are a short-hand notation for message sending operations. For example, `key < my_key` is an abbreviation for the send expression `key ! less (my_key)`, which sends a message to the object referred to by key, requesting the execution of the method `less` with parameter `my_key`. The operator '==', however, is an abbreviation for a call of the routine `id`, which is available for every class. This routine checks whether its two parameters refer to the same object.

One can now ask where the parallelism comes into this example. We have already seen that the sender of an `insert` message does not have to wait until the new information is actually stored in the symbol table. The sender can proceed with its own activities after having handed over the information to the symbol table, and the symbol table will process it in parallel with the sender's activities. The same holds, of course, for a symbol table object that inserts a key/Info pair into one of its subtrees; again it can proceed with a new request immediately after it has given the pair to the subtree. This means that every node in the tree, in particular the top node, needs only a fixed amount of time to process an insertion, independent of the actual size of the symbol table. By contrast, in a sequential system such an insertion would cost an amount of time that in the best case increases logarithmically with the size of the symbol table. To put it otherwise, our parallel symbol table is able to process insertion requests with a constant throughput, whereas in a corresponding sequential symbol table, the throughput rate would decrease as the symbol tables grows.

We would like to maintain this advantage even when look-up requests are sent to the symbol table. However, here it is not possible just to hand over some information to the symbol table, but a reply is desired. Determining this reply will cost an amount of time that increases with the size of the symbol table. So an individual user of the symbol table will inevitably have to wait longer for a reply to his look-up request. What we can do, however, is to maintain the constant throughput rate of the symbol table when there are several, parallel users. This is done as follows: A look-up request is sent to the top node of the tree. This top node returns a result, which just indicates that the request is received. The actual reply will be sent later. If the top node does not store the requested information itself, it delegates the request to one of its subtrees, and so on. When finally the information is found, it is sent directly to the sender of the initial request, without passing via the higher nodes in the tree. In this way we can retain the constant throughput property of the symbol table.

There is one problem here: The reply must be sent to the object that sent the initial look-up request to the top node, and it cannot be the result of this request message. Therefore it must be sent in a separate message from some node in the tree to the requesting object. However, we want to make our symbol table available to an object of any arbitrary class, and we cannot make sure that such an object has an appropriate method to handle the message. To solve this, we introduce a new class, called `Searcher` (this class is hidden from the users of the unit `Symbol_Table`). The instances of the class `Searcher` serve as intermediaries to help other objects in doing look-ups in symbol tables. For each look-up request, a dedicated `Searcher` object is created, it is sent a message (with method go) specifying the symbol table and the key

of the requested information. The method go in the Searcher object sends a request to the symbol table and starts waiting for the reply. When this reply has arrived, the requested information can be passed back to the requesting object as a result of the method go.

Here is the code:

```
ROUTINE search (st : ST (Info), key : String) : Info
TEMP s : Searcher (Info)
BEGIN
   s := Searcher(Info).new ();
   RESULT s ! go (st, key)
END search

METHOD look_up (key : String, client : Searcher (Info)) : ST (Info)
%% Not in SPEC UNIT; used by class Searcher
BEGIN
   RESULT SELF;          %% rendez-vous ends here
   IF my_key == NIL
   THEN client !! reply (NIL)
   ELSIF key = my_key
   THEN client !! reply (my_info)
   ELSIF key < my_key
   THEN left  ! look_up (key, client)
   ELSE right ! look_up (key, client)
   FI
END look_up

%% Class ST needs no explicit body:
%% Incoming messages are anwered in order of arrival
%% by the default body.

END ST

CLASS Searcher (Info)     %% Note: not in SPEC UNIT!

VAR info : Info

%% new is standard routine

METHOD go (st : ST (Info), key : String) : Info
BEGIN
   st ! look_up (key, SELF);
   ANSWER (reply);       %% now the result is in info
   RESULT info
END go
```

```
METHOD reply (new_info : Info)
%% invoked asynchronously
BEGIN info := new_info
END reply

BODY ANSWER (go)          %% each Searcher is used only once!
YDOB
END Searcher
```

What we see in this example is a programming style that is different from tradi
parallel programming: We do not have a collection of processes on the one hand
a collection of data structures on the other hand, such that the processes act o
data structure, and where we must ensure that it will not happen that two pro
are accessing the same data structure at the same time. Instead, the processe
data structures are closely integrated. One could say that each data structure per
the necessary operations on itself. In this way, synchronization and mutual excl
are much easier to handle. In addition, the advantages of sequential object-ori
programming (section 2.4) are maintained.

4 More details about POOL2

The language POOL2 is based on the principles explained in sections 2 and 3. Briefly summarized, these principles amount to describing a system as a collection of objects, each having variables, methods, and a body, where the objects can be created dynamically, are grouped in classes, and interact exclusively by sending messages to each other. However, POOL2 is not the simplest possible language based on these principles (this predicate would be more appropriate for POOL-T [Ame85b] or even better for POOL-S [Ame85a], an early language that was never implemented). While such a simple language has a surprising expressive power, it is nevertheless more convenient for a language used for complex and realistic applications to provide some more facilities.

4.1 Special language elements

The additional language constructs of POOL2 are all based on the idea of "syntactic sugar", a special notation, intended to be more convenient and more natural, for something that is already expressible in the language by other means. As an example, in section 3.3 we have already seen how operators in expressions can be used to abbreviate send expressions. E.g., the expression `3+4` is an abbreviation for `3!add(4)`. POOL2 takes this idea rather far. For some kinds of syntactic sugar the language definition states explicitly into which more primitive form the sugared notation is expanded. In this way the programmer can make the new notation available for one of his own classes by defining a suitable method for this class. For example, the operator `+` can be used for any class that has a synchronous method add with one parameter. Let us call this *explicit* syntactic sugar. In other cases the actual expansion is hidden from the programmer, so that he can only access these features using the special notation (*implicit* syntactic sugar). This applies, for example, to the notions of globals and routine objects, which are discussed below.

The extra facilities provided by POOL2 in addition to the basic primitives of parallel object-oriented programming include the following:

- a lot of explicit syntactic sugar
- implementation and specification units
- generic classes
- asynchronous communication
- new-parameters
- global names for objects
- routines being considered as objects
- enumeration classes
- a collection of standard classes and standard units

We have already encountered several of these constructs in the previous sections. The others are briefly discussed below. A more extensive discussion can be found in [Ame88d], where it is also indicated how the functionality of the constructs can be obtained using only the basic primitives.

For the creation and initialization of new objects, POOL2 provides the built-in routine new, which is automatically included for each programmer-defined class. The parameters of this routine can be specified by the programmer. These so-called new-parameters are passed to the newly created object, where they remain available throughout the lifetime of this object. Before the new object is handed over to the caller of the routine new, it first initializes its variables, according to expressions in the variable declarations and/or by executing explicit initialization statements. In this way, the designer of a class can make sure that all the existing instances of the class are properly initialized. Of course, this can also be done by sending every new object an initializing message, as was illustrated in section 2.2, but since some form of explicit initialization is needed for almost every class, it seems more than justified to introduce some more convenient syntax for it.

In POOL2 it is possible to define a global name to be bound to a specific object. Any other object in the system can then refer to this object by this name. Such globals are defined in global definitions:

```
GLOBAL my_name : String := "Pierre"
       n : Int := (3+4) * (first ! get_number ())
       first := Big_Object.new()
```

Conceptually, what happens here is that for each global implicitly an object is created, which we shall call a global manager. This global manager starts to evaluate the expression in the global definition. As soon as this terminates, it stores the resulting value, and from that moment on this value is available for any other object in the system. If an object tries to determine the value of the global before this is known, the object becomes blocked until the global manager has finished evaluating the global. This mechanism ensures that even dependencies among globals (as in the above example, where n depends on first) are handled correctly, as long as they remain acyclic; otherwise a deadlock occurs.

The manner in which the execution of a system is initiated is also based on globals: A few objects are created by declaring them to be globals. These objects may create other ones and so on until the desired degree of parallelism is reached. In the above example, the class Big_Object might be defined in such a way that it sets the whole system running.

We have already seen that POOL distinguishes between methods, a kind of procedures that are associated with individual objects, and routines, which are in general associated with a class. Because of the fact that a method can directly access the variables of the object it is associated with, it is not possible to consider it as an object itself: If a method could be stored in variables, passed around in messages and executed by any object that had access to it, the protection of the original object that owned the

method could not be guaranteed. (Note that we are talking about the method itself, not the program text that defines it.) However, no such restriction applies to routines, because these are not associated with individual objects and have no direct access to any object's variables. Therefore, POOL2 takes the point of view that routines can be considered as objects. That is, they can be stored in variables, passed as parameters or results of messages, and even new ones can be created dynamically: In addition to the routines associated with a class (we call them *class routines*), it is possible to write *routine expressions*, which indicate the creation of a new routine object. For example, the expression

```
ROUTINE (p : Int) : Int
BEGIN RESULT p ** 2 - 3
END
```

creates a routine that represents the function mapping any integer z to $z^2 - 3$. This routine will be an instance of the class `ROUTINE (Int) : Int`. It is even possible to pass a kind of new-parameters to such a newly created routine object. For example, we can write the expression

```
ROUTINE (p : Int) : Int
TEMP i : Int
BEGIN
    i := p**2 - t;
    IF i < 0 THEN RESULT 0 ELSE RESULT i
END
```

where `t` is an integer variable of the object executing this routine expression. At the moment this expression is evaluated, the value of `t` is determined and this is stored with the routine object. This value will be the one that is used whenever the routine is called, even if the original variable `t` changes its value. In this way the routine does not need and does not have access to the variable `t` after its creation.

Such a routine object can be called by any other object that has a reference to it. For example, if the variable `f` is of type `ROUTINE (Int) : Int`, then the expression

```
f(3) + f(5 - f(2))
```

will lead to three calls of the routine object to which `f` refers.

Note that the possibility to pass routines as parameters considerably enhances the usefulness of generic classes. For example, in section 3.3 we could have made the type of the keys into a parameter of the class ST. The desired ordering on these keys could be passed as a new-parameter to every symbol table object. The class specification would then look like this:

```
CLASS ST (Key, Info)
ROUTINE new (less : ROUTINE (Key, Key) : Bool) : ST (Key, Info)
    ...
END ST
```

Now a symbol table that stores pairs of integers could be created by the expression

```
ST (Int, Int).new (ROUTINE (n, m : Int) : Bool BEGIN RESULT n < m END)
```

In addition to powerful mechanisms for defining classes, POOL2 also provides a number of *standard classes* [Ame88b]. These are available in every program unit without explicit importation via a use list. For standard classes more efficient implementations are provided than would have been possible if they were defined by the programmer. Moreover, for some of these classes, e.g., `Int` and `String`, a special notation is available so that the instances can be represented in a program in a natural way.

The collection of standard classes comprises classes of small, fixed-size entities: booleans, characters, integers, and floating point numbers, but also of potentially large objects: strings and (multidimensional) arrays. In addition, there are generic standard classes for tuples and unions: A tuple contains a fixed number of components, possibly of different types, whereas a union contains one object reference out of a fixed number of possible types (a tuple can be compared with a fixed record in Pascal [BSI82], a union to a variant record).

All these classes, with the exception of arrays, have been defined in such a way that their instances are *immutable* objects, i.e., once they are created, their contents cannot be changed any more. The advantage of this is that an implementation may freely make multiple copies of such an object without changing the behaviour of the system. In a machine like DOOM, without shared memory, it is much more efficient sending a copy of a small object than sending a reference so that the receiver must send several more messages to determine the contents of the object. For objects like strings or tuples, it is even possible to include a copy in a message that travels from one processor to another but to include a reference if the message is local.

On the other hand, the arrays in POOL2 are even more dynamic than in most other programming languages: they can even change their size at run-time. The rationale behind this is that the language implementation can do this much cheaper than a POOL programmer could (for example, by dealing cleverly with pages in a virtual memory system, it can be avoided to copy a complete large array that must grow a little more). If these facilities are not used, they do not cost anything extra.

A number of other facilities, which are not so basic to the language, are included in *standard units* [Ame88c]. The facilities of these units can be imported without the programmer having to define them. Currently POOL2 provides standard units for doing input/output on files, for communicating with the Unix operating system on a host machine, and for controlling the allocation of objects to processors in DOOM.

The latter brings us to another issue: the mapping of a POOL program to a machine like DOOM [Odi87]. This machine consists of a number of processors (called *nodes*), each with its own private memory, which communicate via a message passing network. There is a very natural way of implementing POOL on such a machine: every object is allocated to a certain node, where its data are stored and its body and methods are executed. In general, there are many objects on each node, sharing the processor. These objects must be scheduled one after another, so that they cannot really run in

parallel. On the other hand, having multiple objects on a node makes it possible for the processor to do useful work even if many of these objects are waiting for messages or method results.

At the creation of a new object, the programmer can influence the choice of the node by *allocation pragmas*. These can serve as annotations to calls of the standard routine `new`, and they indicate possible choices for the new object's location. For example, in the expression

```
C.new (par) (* ALLOC HERE, WITH obj, set1 & set2 *)
```

it is indicated that the new object is to be allocated preferably on the same node as its creator. If that is not possible, the object should be placed at the same node as the object in the variable/parameter `obj`. If even that is not possible, it should be placed on a node that is in the intersection of `set1` and `set2`, which are variables/parameters of type `Node_Set`, representing sets of nodes. Whenever there are several possibilities for the allocation of the new object, the intention is that the least occupied node is chosen.

If a message must be sent between two objects on the same node, this can be handled locally, without involvement of the communication network. Only if a message is transmitted between different nodes this network is used. In general, nonlocal communication is more expensive than local communication, because the data must be copied several times and larger messages must be split up in packets and reassembled again. Note that the same POOL send and answer constructs are used for both local and nonlocal communication.

In DOOM, the communication network has been implemented in such a way that the distance between two nodes in the network is not very important for the cost of communication between them. Therefore the most important decision in allocating objects is whether they should be on the same node or on different nodes. If they are on the same node, communication between them is cheap, but they cannot actually run in parallel. If they are on different nodes, they can run in parallel, but communication is more expensive. A thorough understanding of a program is necessary to make the optimal decision. Therefore advice from the programmer is invaluable. Fortunately, in most cases this advice can be limited to allocation pragmas of the form `HERE`, `~ HERE`, `WITH obj`, or `~ WITH obj`.

The general idea is that allocation decisions are made by the programmer and the run-time system *together*, where the programmer supplies the knowledge of the program, and the run-time system the knowledge of the current situation with respect to node occupation.

4.2 Another example program

We shall illustrate some of the abovementioned language constructs in the following example. It implements a parallel version of priority queues. Such a priority queue can store a collection of items; when these are retrieved from the queue the one with the highest priority is output first. Items with the same priority are treated in a first-in-first-out way. Here is the specification unit:

```
SPEC UNIT Prio_Queue

CLASS PQ (Item)
%% Instances of this class are priority queues that store
%% elements of the class Item.

ROUTINE new (higher : ROUTINE (Item, Item) : Bool) : PQ (Item)
%% Creates and returns a new, empty priority queue.
%% The routine higher determines the priority ordering.
%% If it returns TRUE, the first argument is assumed to
%% have a higher priority than the second.

METHOD put (i : Item) : PQ (Item)
%% Stores the item i in the queue; returns SELF.
%% The argument i should not be NIL.

METHOD get () : Item
%% Deletes and returns the item with the highest priority.
%% This method will not be answered when the queue is empty.

END PQ
```

We see that the class PQ is defined in a generic way and that the priority criterion, the routine higher, is passed as a new-parameter to every instance. The corresponding implementation unit is somewhat more interesting. The same technique is used as in section 3.3: Every PQ object only stores one item and delegates the rest to another priority queue. Here is the code:

```
IMPL UNIT Prio_Queue

CLASS PQ (Item)

NEWPAR (higher : ROUTINE (Item, Item) : Bool)
%% The routine new, which creates and returns a new, empty priority
%% queue, is defined automatically with the above parameter list.

VAR max  : Item       %% the highest-priority element in the queue
    rest : PQ (Item)  %% a PQ that stores all the other elements
    %% Both variables are automatically initialized to NIL.

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

METHOD put (i : Item) : PQ (Item)
%% Stores the item i in the queue; returns SELF.
```

```
BEGIN
   RESULT SELF;    %% end of rendez-vous: sender can continue
   IF max == NIL   %% queue is empty
   THEN max := i;
        IF rest == NIL
        THEN rest := PQ (Item).new (higher)
        FI
   ELSIF higher(i, max)       %% only if i has a higher priority
   THEN rest ! put (max);     %% we replace max by i
        max := i
   ELSE rest ! put (i)
   FI
END put

METHOD get () : Item
%% Deletes and returns the item with the highest priority.
%% This method will not be answered if the queue is empty.
%% Therefore we know that max ~== NIL, so rest ~== NIL.
BEGIN
   RESULT max;     %% end of rendez-vous: sender can continue
   max := rest ! get_largest_or_NIL ()
END get

METHOD get_largest_or_NIL () : Item
%% Returns NIL if the queue is empty. Otherwise it deletes
%% the item with the highest priority and returns it.
BEGIN
   RESULT max;     %% end of rendez-vous: sender can continue
   IF max ~== NIL
   THEN max := rest ! get_largest_or_NIL ()
   FI
END get_largest_or_NIL

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

END PQ
```

The class PQ has an explicit body, which makes sure that messages asking for the method **get** are only answered when the queue is not empty. In this way an object that asks for a new element from the queue is automatically delayed until such an element is actually available. For internal purposes an additional method **get_largest_or_NIL** is needed, which is always answered but returns NIL if the queue is empty. (This method might be useful even for users of the class PQ, so it could be mentioned in the specification unit, too.)

The above unit is used by the following program. This will sort pairs of integers and strings, which it reads from the standard input file:

```
IMPL UNIT Sorting

USE File_IO Prio_Queue

GLOBAL root := Sorter.new()

CLASS Sorter
%% An instance of this class will read pairs of integers and strings
%% from the standard input file until a negative integer is found.
%% Then it will print the preceding pairs in ascending order of the
%% integers. Pairs with the same integer will be printed in the order
%% in which they were input.

ALIAS Pair = [Int, String]

VAR compare := ROUTINE (p1, p2 : Pair) : Bool
                 BEGIN RESULT p1 @ 1 < p2 @ 1
                 END
    pq := PQ (Pair).new (compare)
    n : Int := standard_in ! read_Int ()
    s : String

BODY
   WHILE n >= 0
   DO s := standard_in ! read_String ();
      pq ! put ([n, s]);
      n := standard_in ! read_Int ()
   OD;

   DO    %% until deadlock occurs
      [n, s] := pq ! get ();
      standard_out ! write_Int (n, 10)
                   ! write_String (s + "\n")
   OD
YDOB
```

```
END Sorter
```

The use list of this unit mentions the above unit `Prio_Queue` in addition to the standard unit `File_IO`. The system is started by the definition of the global `root`, which is made to refer to a new object of the class `Sorter`. In the definition of the class `Sorter` first an alias is defined, a synonym for the class name `[Int, String]`, which in turn is an abbreviation for `Tuple2 (Int, String)`. Each instance of this standard class stores a pair containing an integer and a string. We shall insert this type of objects into our priority queue.

The routine that determines the priority criterion on pairs is defined in the initialization of the variable `compare`. (It is possible to use this routine expression directly as an argument for the routine new below, but this is not so readable.) In this routine, the expression `p1 @ 1` is a piece of explicit syntactic sugar, which stands for the send expression `p1 ! get_1()`, extracting the first component from the tuple `p1`.

The standard input and output files are denoted by the globals `standard_in` and `standard_out` respectively, which are exported by the standard unit `File_IO`. These globals are referring to elements of the classes `Read_File` and `Write_File`. More instances of these classes can be created by calling the appropriate routines, which either create new files or associate POOL objects to existing files. Actual input and output can then be performed by sending messages to these objects.

The expression `[n, s]` is syntactic sugar for the routine call

```
Tuple2 (Int, String).new (n, s)
```

which creates a new tuple object with `n` as the first component and `s` as the second. However, when `[n, s]` appears at the left-hand side of the assignment, it makes sure that the tuple yielded by the right-hand side is analyzed, storing its first component in `n` and the second component in `s`.

The operator `+` for strings denotes concatenation: it delivers a new string containing the characters in the first operand followed by the characters in the second operand. The string `"\n"` contains a single character, a line feed.

To illustrate the function of the program, consider the following sample input:

```
 1 jumps over
 0 the quick brown fox
 1 the lazy black dog
-100000
```

The corresponding output will be as follows:

```
0 the quick brown fox
1 jumps over
1 the lazy black dog
```

5 Inheritance and typing

In this section we deal with two issues that seem quite independent at first, but at a closer look turn out to be closely related. We explain the concept of inheritance and indicate the problems associated with it, which justify that inheritance is *not* included in POOL2. We also sketch some directions along which solutions to these problems could be found. This requires a careful analysis of the relationship between inheritance and typing.

For the largest part, this section is quite independent of the specific properties of POOL. It is applicable to a large class of object-oriented languages. Therefore we shall, for the moment, forget about the peculiarities of POOL, such as bodies and routines.

5.1 Inheritance

The concept of inheritance was already present in the first object-oriented languages, like Simula [DN66] and Smalltalk-80 [GR83]. The basic idea is that in defining a new class it is often very convenient to start with all the variables and methods of an existing class and only to add some more in order to get the desired new class. The new class is said to *inherit* the variables and methods of the old one. (Note that inheritance is a relationship between classes, not between instances.)

This inheritance mechanism constitutes a very successful way of incorporating facilities for *code sharing* in a programming language. Both the programmer and the implementor can take advantage of it. For the programmer the most important thing is not that he need not write the inherited code several times: for this task a text editor can offer enough help. It is important, however, that the sharing of code has been made explicit. In reading a program, it is not necessary to compare pieces of code in order to see whether they are the same or in what aspects they differ: all this is clearly indicated in the code itself. Moreover, if the program is changed, the changes automatically apply to all the classes inheriting the code. Therefore consistency is guaranteed. The implementation can profit from code sharing by producing more compact code, occupying less computer memory. Especially in systems without shared memory, where code must often be duplicated over many nodes, this is a considerable advantage.

As an example, the piece of code below shows how a class `Bordered_Window` could be described as a subclass of the class `Window`, defined in section 2.2. The new class represents windows that have been adorned with a border of a certain width and colour. Note that it is only necessary to indicate the things that have changed. We can add new variables and methods, we can override the existing methods, and all the time we have access to the features of the superclass (even to the overridden method `move`).

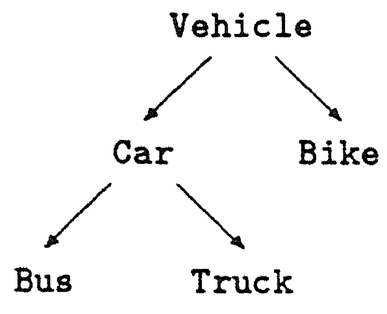
```
CLASS Bordered_Window
INHERIT Window

VAR border_width  : Integer
    border_colour : Colour
```

```
METHOD change_border (new_width  : Integer,
                      new_colour : Colour) : Bordered_Window
BEGIN
   border_width  := new_width;
   border_colour := new_colour;
   display_border ();
   RESULT SELF
END change_border

METHOD move (to : Vector) : Bordered_Window    %% redefined
BEGIN
   Window.move(to);          %% the method of the superclass!
   display_border ();
   RESULT SELF
END move

METHOD display_border () : Bordered_Window
%% only for internal use
BEGIN
   ...         %% actual text not relevant here
   RESULT SELF
END display_border

ROUTINE create (cont          : Object,
                pos, siz      : Vector,
                border_width  : Integer,
                border_colour : Colour) : Bordered_Window
TEMP w : Bordered_Window
BEGIN
   w := Bordered_Window.new ();       %% standard routine new
   w ! init (cont, pos, siz);
   w ! change_border (border_width, border_colour);
   RESULT w
END create
```

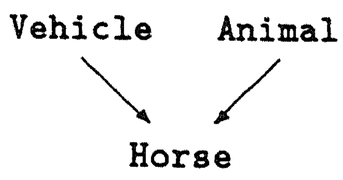
But there is more to inheritance than only code sharing: Suppose that the class B has inherited all the variables and methods from the class A. Then, in a way, we can consider every instance of B equally well as an object of class A: At any point where an object of A is expected (because certain messages are sent to it), any object of class B will satisfy our needs, because it will accept all the messages that an object of class A would accept. Therefore the instances of B can be considered as *specialized versions* of the ones in class A. This can be expressed by calling the class B a *subclass* of A and A a *superclass* of B (note the correspondence with the terminology of subset and superset in set theory).

Conceptually, if we consider objects in a program as representations of entities in the real world (for example in a database or simulation system) and if we use them mainly as collections of variables in which attributes can be stored, then this makes good sense. For example, if we have defined a class `Vehicle` with variables to store the owner and the maximum speed, it is convenient to define the class `Car` as a subclass of `Vehicle` so that we only have to add a variable to store the licence number. An instance of class `Car` is then automatically considered as an element of the type associated with class `Vehicle`.

Of course, this procedure can be repeated several times. For instance, we can define a class `Truck` as a subclass of `Car`, with an extra variable to store the load capacity, we can define `Bus` as another subclass of the class `Car`, with a variable for the number of seats, and we can define `Bike` to be a subclass of `Vehicle`, adding a variable containing the number of speeds. In this way we can get a whole hierarchy of classes, which has the form of a tree:



Moreover, it is possible to allow a new class to inherit from more than one existing class. This mechanism is called *multiple inheritance*, in contrast to *linear* inheritance. For example, a horse can be considered as an animal (having, for example, a father and a mother) and as a vehicle, and therefore the class `Horse` can be defined conveniently as a subclass of both `Animal` and `Vehicle`. In the case of multiple inheritance, the class hierarchy is not a tree any more; it becomes an acyclic directed graph:



A mechanism like the one described above is included in most object-oriented programming languages. In those languages that are statically typed, like Trellis/Owl [SCB*86] and Eiffel [Mey87], this mechanism is linked with typing in a way that is discussed below. In dynamically typed languages, like Smalltalk-80 [GR83], the connotation of specialization, associated with inheritance, is nowhere enforced explicitly by the language. The organization of classes into such a hierarchy based on specialization is only formulated in some informal advice to the programmer [HO87].

Even in these dynamically typed languages, there are some problems with inheritance. The most difficult one is the phenomenon of *name clashes* in multiple inheritance. Such a class occurs when a class tries to inherit from two superclasses that both

have a variable or method with the same name, so that they cannot both be included in the new class. A large variety of solutions to this problem has been proposed, ranging from mandatory explicit renaming by the programmer [Mey88] to imposing a priority ordering on all the superclasses, often together with mechanisms describing how several methods should be combined [BDG*87].

5.2 Relationship with typing

In a language that have a notion of static typing, in the sense that for each expression it is possible to determine from the program text the type of object it denotes, it is possible to make the implications of the specialization in subclasses explicit. For example, if B is a subclass of A, then it should be allowed to use an expression having type B wherever an object of type A is expected. For example, such an expression may be assigned to a variable of type A and a message mentioning a method of class A can be sent to it.

A few extra conditions are necessary to make this absolutely safe: In particular, suppose that the class A has a method m and that B redefines this method. If in class A the method m has parameter types $P_1^A, \ldots, P_n^A$ and result type R^A, while in class B it has parameter types $P_1^B, \ldots, P_l^B$ and result type R^B then we must require that the number of parameters are equal ($n = l$), that $P_1^A \preceq P_1^B, \ldots, P_n^A \preceq P_n^B$, and that $R^B \preceq R^A$, where $X \preceq Y$ expresses that X is either equal to Y or it is a subclass of Y (possible via a number of other subclasses).

We can see that this is necessary if we consider the situation where we send a message listing the method m to an object contained in a variable of type A. Then we expect that the method takes n parameters of types $P_1^A, \ldots, P_n^A$. However, the actual object stored in the variable may be an instance of class B. Therefore, the number of the parameters must be the same: $n = l$. Furthermore this object expects parameters of types $P_1^B, \ldots, P_n^B$ and it may apply to them all the operations allowed by these classes. For, e.g., the first parameter this will not lead to problems, provided that $P_1^A \preceq P_1^B$, because under this condition the actual parameter (of type P_1^A) will indeed admit all the operations defined for P_1^B. This is called the contravariant parameter type rule, because for parameters the inclusion sign $\preceq$ point in the other direction than for the classes A and B themselves. For the result types we have a covariant rule, because the method m of the class B may return any result of type R^B, and because this will be treated as an object of type R^A, we must require $R^B \preceq R^A$.

These rules have been studied in their purest form in [Car88]. They are incorporated in the language Trellis/Owl [SCW85]. Whereas in Eiffel [Mey87] it is not explicitly stated that these rules are enforced (and [Mey88] even gives an example that does not obey the contravariant parameter type rule), circumstantial evidence nevertheless indicates that these rules are intended to be satisfied.

While a statically typed language that enforces the above rules is indeed completely safe with respect to type checking (in the sense that never a message will be sent to an object that does not have a method for it), there are still some problems. For example, the simplest solution to name clashes in multiple inheritance, explicit renaming of inherited

variables and methods, cannot be applied: With renaming it cannot be guaranteed that if class A has a method m, every subclass will also have a method m, because the subclass could have renamed it. (In Eiffel this problem is attacked by nevertheless taking the renamed method in such a case, which is at least confusing.)

A more serious problem that in some cases we want code sharing without subtyping or subtyping without code sharing. For example, in implementing a class `Stack`, it might be convenient to inherit the code from the class `Array`, in order to be able to store stack elements. However, we do not want Stack to be a subtype of Array, because we do not want all the array operations (storing and retrieving elements at arbitrary places) to be applicable to stacks, too. In another case, adding a new method to a number of existing ones may violate an invariant on which the correct functioning of the old methods is based. In this case, an object of the new class will *not* behave as a specialized version of the old one.

The other way around, we can think of several different ways in which a stack can be implemented. While these implementations have completely different code, we do not want to consider them as different types, because they have the same behaviour. As another example, we would like to be able to consider the class `Int` as a subtype of a class `Ordering`, where the latter only specifies that a method `less` should be present that gives rise to a total ordering on the elements. This should be possible even though `Int` is a built-in class and `Ordering` a programmer-defined one, so that code sharing is utterly impossible.

Therefore we propose, for a future object-oriented language, to make a clear distinction between inheritance and subtyping. Inheritance deals with the internal structure of the objects: their variables and the code they execute for their methods. Subtyping, on the other hand, deals with the externally observable behaviour of the objects: the messages that they accept (in particular the method names and parameter types) and the results they return. Note that this distinction is analogous to the separation between the implementation of an abstract data type (or class) and its interface to the outside world. By separating these concepts, many of the problems currently associated with inheritance can be solved easily [Ame87b].

In this context, it is useful to distinguish between the notions of class and type. Let us continue to consider a class as a collection of objects that have the same internal structure: the same variables, methods, and body. Then we can use the term 'type' for a collection of objects that have certain common properties with respect to their behaviour. In other words, whereas a class groups together the objects that have been *built* in the same way, a type comprises a collection of objects that can be *used* in a certain way.

With these definitions, inheritance can take place without the connotations of subtyping. Pieces of code can be imported under the only condition that they perform some useful function. Redefinition and renaming is never a problem, because the new class is in principle unrelated to the existing ones. Subtyping can now be done a posteriori: after a class has been defined it can be determined whether or not its instances belong to a certain type, and for any two types it can be determined whether one is a subtype of the other. This should be done on the basis of a specification of the ex-

ternally observable behaviour of the class's instances, without regarding their internal implementation.

At this point it is not so easy to see how this notion of subtyping can be formalized. In any case, a specification formalism should be devised that considers the behaviour of an object, but not its internal structure. It is not at all trivial to devise such a formalism (see also section 6). However, suppose that we have an appropriate formalism available. Then we can associate each type τ with a specification $\phi(x)$, by which we mean that the type τ consists of all the objects β for which $\phi(\beta)$ holds. Now we can say that a class A belongs to τ (or rather, that all the instances of A belong to τ) precisely if $\forall \beta \in A\, \phi(\beta)$. Moreover, if the type σ is characterized by the specification $\psi(x)$, then we can say that σ is a subtype of τ, or $\sigma \preceq \tau$, precisely if $\forall \beta\, \psi(\beta) \rightarrow \phi(\beta)$.

As an example, we can consider the specifications associated with the types Bag and Stack. Elements of both these types can store integers. They have a method put to insert a new integer and a method get to retrieve an integer. The difference between the two is that the type Stack requires that the integers are retrieved in a last-in-first-out order, while the type Bag does not specify a certain order. Now it can be seen that Stack is a subtype of Bag, because every object that satisfies the specification associated with Stack will automatically satisfy the specification of the type Bag. For a concrete class that implements stacks, e.g., using arrays, it can be established that the instances are members of the type Stack by verifying that they satisfy the specification of Stack. In that case it is clear that they also belong to the type Bag. (In [Ame89] it is shown how these specifications can be given in a formal way using techniques from abstract data types.)

5.3 Integrating it into POOL

Once we have decided that inheritance and subtyping are separate things, we can deal with them separately. In order to introduce inheritance into POOL, we can imagine the concept of inheritance package, a set of variables and methods, which may or may not comprise all the variables and methods of a given class, as long as it is complete: if a method in the package accesses a variable, this is also in the package. It might even be useful to give an explicit *inheritance interface* to such a package, which only lists a number of variables with their types and method headers. The advantage of this is that, in addition to making the interface explicit and easier to read, it is possible to hide certain variables and methods so that they cannot be used in a wrong way.

Inheriting routines does not make sense, because these can anyway called from any point in the system. Bodies, however, require some special attention. Unfortunately there seems to be no natural way of inheriting bodies, especially in the case of multiple inheritance. The only part of the body that is a natural candidate for inheritance is the initialization of the variables. By including this in such an inheritance package, it can be ensured that all variables, even the hidden ones, are correctly initialized.

For objects where the body plays an important role, inheritance might not be a very useful mechanism. But for other objects, with a more server-like role, which wait for messages and then process them, it might be just as useful as in sequential languages.

Also remember that the programming style advocated for POOL consists of letting the objects themselves perform the operations on them. If many classes of objects have a different parallel behaviour, but similar operations, sharing the code of these operations might be very useful.

For the integration of subtyping withing POOL we propose the same approach as sketched broadly above. However, for a parallel language it seems even more difficult to develop an appropriate specification formalism, which is based exclusively on the external behaviour of the objects, without depending on their internal structure (work in this direction is briefly sketched in section 6). Moreover, it is highly improbable that such a formalism would be suitable in practice, i.e., that it would be possible to specify formally every type in every program, and that a compiler could verify the relationships between classes and types. Therefore it would probably be best, for a practical language, to leave a part of these specifications informal. But at the very least it would be possible to formalize the requirements on the availability of methods and the restrictions on their types (e.g., the contravariant parameter type rule).

6 Formal aspects

The POOL family of languages has been the subject of extensive research in the area of formal techniques. Several frameworks have been employed to give a formal description of the semantics of POOL and the relationships between these semantic models have been investigated. Furthermore, significant progress has been made to develop a formalism in which the correctness of a program with respect to a certain specification can be verified. In this section, we give a brief overview of these studies. For more details, the reader is referred to the original documents.

In order not to obscure the semantic essentials of the language with many syntactical details, the syntax has been simplified considerably in these formal studies. All the syntactic sugar present in POOL2 has been removed, and even several important facilities of POOL-T (e.g., units and routines) have disappeared. Furthermore the types of variables, parameters, and method results are no longer explicitly mentioned in the program. In this way we arrive at a language with a very simple syntax. Nevertheless it is straightforward to translate an arbitrary POOL2 program to this simple language: For the special POOL2 elements, it is indicated in [Ame88d] how they can be reformulated in POOL-T terms. Translating POOL-T into this simple language is also easy: Units are merged together (taking care of name clashes between hidden classes), each routine definitions is transferred into method definitions in the classes where the routine is called, and finally all the typing information is omitted. Via these translation steps we can say that we have given a formal description of POOL2.

6.1 Semantics

6.1.1 Operational Semantics

The simplest semantic technique is the use of *transition systems* to define an *operational* semantics. This technique has been introduced by Hennessy and Plotkin [HP79,Plo81,Plo83]. It describes the behaviour of a system in terms of sequences of *transitions* between *configurations*. A configuration describes the system at one particular moment during the execution. Apart from a component describing the values of the variables, it typically contains as a component that part of the program that is still to be executed. The possible transitions are described by a *transition relation*, a binary relation between configurations (by having a relation instead of a function, it is possible to model nondeterminism). This transition relation is defined by a number of *axioms* and *rules*. Because of the presence of (the rest of) the program itself in the configurations, it is possible to describe the transition relation in a way that is closely related to the syntactic structure of the language.

The term "operational" can now be understood as follows: The set of configurations defines a (very abstract) model of a machine, and the transition relation describes how this machine operates: each transition corresponds to an action that the machine can perform. The fact that the semantic description follows the syntactic structure of the language so closely (as we shall see below) is a definite advantage of the transition system approach to operational semantics.

The operational semantics of POOL [ABKR86] uses configurations having four components:

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

The first component is a finite set of *labelled statements:*

$$\{\langle \alpha_1, s_1 \rangle, \ldots, \langle \alpha_n, s_n \rangle\}$$

Here each α_i is an object name and the corresponding s_i is the statement (or sequence of statements) that the object is about to execute. This models the fact that the objects $\alpha_1, \ldots, \alpha_n$ are executing in parallel. The second component is a *state* $\sigma \in \Sigma$, which records the values of the instance variables and temporary variables of all the objects in the system. The third component is a typing function $\tau \in Type$, assigning to each object name the class of which the object is an instance. Finally, the last component is the complete POOL program or *unit*, which is used for looking up the declarations of methods (whenever a message is sent) and bodies (when new objects are created).

The transition relation $\rightarrow$ between configurations is defined by axioms and rules. In general, an axiom describes the essential operation of a certain kind of statement or expression in the language. For example, the axiom describing the assignment statement has the following form:

$$\left\langle X \cup \{\langle \alpha, x := \beta \rangle\}, \sigma, \tau, U \right\rangle \rightarrow \left\langle X \cup \{\langle \alpha, \beta \rangle\}, \sigma\{\beta/\alpha, x\}, \tau, U \right\rangle$$

Here, X is a set of labelled statements, which are not active in this transition, α is the name of the object that executes the assignment, β is another object name, a special case of the expression that can in general appear at the right-hand side of an assignment, and $\sigma\{\beta/\alpha, x\}$ denotes the state that results from changing in the state σ the value of the variable x of the object α into the object name β.

Rules are generally used to describe how to evaluate the components of a composite statement or expression. For example, the following rule describes how the (general) expression at the right-hand side of an assignment is to be evaluated:

$$\frac{\left\langle X \cup \{\langle \alpha, e \rangle\}, \sigma, \tau, U \right\rangle \rightarrow \left\langle X' \cup \{\langle \alpha, e' \rangle\}, \sigma', \tau', U \right\rangle}{\left\langle X \cup \{\langle \alpha, x := e \rangle\}, \sigma, \tau, U \right\rangle \rightarrow \left\langle X' \cup \{\langle \alpha, x := e' \rangle\}, \sigma', \tau', U \right\rangle}$$

According to this rule, *if* the transition above the line is a member of the transition relation, *then* so is the transition below the line. In this way the rule reduces the problem of evaluating the expression in an assignment to evaluating the expression on its own. The latter is described by specific axioms and rules dealing with the several kinds of expressions in the language. Note that as soon as the right-hand side expression has been evaluated completely, so that an concrete object name β results, the assignment axiom above applies and the assignment proper can be performed.

The semantics of a whole program can now be defined as the set of all maximal sequences of configurations $\langle c_1, c_2, c_3, \ldots \rangle$ that satisfy $c_i \rightarrow c_{i+1}$. Each of these sequences represents a possible execution of the program.

6.1.2 Denotational semantics

The second form of semantic description that has been used to describe POOL is *denotational* semantics. Whereas operational semantics uses an abstract machine that can perform certain actions, denotational semantics assigns a mathematical value, a "meaning", to each individual language construct. Here, the most important issue is compositionality: the meaning of a composite construct can be described in terms of only the *meanings* of its syntactic constituents.

For sequential languages, it is very natural that the value associated with a statement is a function from states to states: when applied to the state before the execution of the statement, this function delivers the state after the execution. However, for parallel languages, this is no longer appropriate. The first problem is that parallel languages are in general *nondeterministic:* it is no longer possible to predict a single output state for each given input state. In general, there is a *set* of possible output states.

The second problem is that describing the set of possible output states for each input states does not provide enough information to be able to compose a statement in parallel with other statements: information on the intermediate states is also required. This leads us to the concept of *resumptions* (introduced by Plotkin [Plo76]). Instead of delivering the final state after the execution of the statement has completed, we divide the execution of the statement into its atomic (indivisible) parts, and we deliver a pair $\langle \sigma', r \rangle$, where σ' is the state after the execution of the first atomic action and r is the resumption, which describes the execution from this point on. In this way, it is possible to put another statement in parallel with this one: the execution of the second statement can be interleaved with the original one in such a way that between each pair of subsequent atomic actions of the first statement an arbitrary number of atomic actions of the second one can be executed. Each atomic action can inspect the state at the beginning of its execution and possibly modify it.

For a very simple language (not yet having the power of POOL) we get the following equation for the set (the *domain*) in which the values reside that we want to assign to our statements:

$$P \cong \{p_0\} \cup \Big(\Sigma \to \mathcal{P}(\Sigma \times P)\Big). \tag{1}$$

The intended interpretation of this equation is the following: Let us call the elements of the set P *processes* and denote them with letters p, q, and r. Then a process p can either be the terminated process p_0, which cannot perform any action, or it is a function that, when provided with an input state σ, delivers a set X of possible actions. Each element of this set X is a pair $\langle \sigma', q \rangle$, where σ' is the state after this action and q is a process that describes the rest of the execution.

It is clear that equation (1) cannot be solved in the framework of *sets*, because the cardinality of the right-hand side would always be larger than that of the left-hand side. In contrast to many other workers in the field of denotational semantics of parallelism, who use the framework of complete partial orders (CPOs) to solve this kind of equations (see, e.g., [Plo76]), we have chosen to use the framework of *complete metric spaces*. (Readers unfamiliar with this part of mathematics are referred to standard topology

texts like [Dug66,Eng77] or to [BZ82].) The most important reason for this choice is the possibility to uses Banach's fixed point theorem:

> Let M be a complete metric space with distance function d and let $f : M \to M$ be a function that is *contracting*, i.e., there is a real number ϵ with $0 < \epsilon < 1$ such that for all $x, y \in M$ we have $d(f(x), f(y)) \leq \epsilon . d(x, y)$. Then f has a unique fixed point.

This ensures that whenever we can establish the contractivity of a function we have a *unique* fixed point, whereas in CPO theory mostly we can only guarantee the existence of a *least* fixed point.

Another reason for using complete metric spaces is the naturalness of the power domain construction. Whereas in CPO theory there are several competing definitions (see, e.g., [Plo76,Smy78]) all of which are somewhat hard to understand, in complete metric spaces there is a very natural definition:

> If M is a metric space with distance d, then we define $\mathcal{P}(M)$ to be the set of all *closed* subsets of M, provided with the so-called *Hausdorff distance* d_H, which is defined as follows:
>
> $$d_H(X, Y) = \max\Big\{\sup_{x \in X}\{d(x, Y)\}, \sup_{y \in Y}\{d(y, X)\}\Big\}$$
>
> where $d(x, Z) = \inf_{z \in Z}\{d(x, z)\}$ (with the convention that $\sup \emptyset = 0$ and $\inf \emptyset = 1$).

(A few variations on this definition are sometimes useful, such as taking only the nonempty subsets of M or only the compact ones. The metric is the same in all cases.)

The domain equation that we use for the denotational semantics of POOL (see [ABKR88]) is somewhat more complicated than equation (1), because it also has to accommodate for communication among objects. For POOL, the domain P of processes is defined as follows:

$$P \cong \{p_0\} \cup \Big(\Sigma \to \mathcal{P}(Step_P)\Big)$$

where the set $Step_P$ of *steps* is given by

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

with

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

and

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

The interpretation of these equations (actually, they can be merged into one large equation) is as follows: As in the first example, a process can either terminate directly, or it can take one out of a set of steps, where this set depends on the state. But in addition to internal steps, which are represented by giving the new state plus a resumption process, we now also have communication steps. A *send step* gives the

destination object, the method name, a sequence of parameters, and *two* resumptions. The first one, the *dependent* resumption, is a function from object names to processes. It describes what should happen after the message has been answered and the result has been returned to the sender. To do that, this function should be applied to the name of the result object, so that it delivers a process that describes the processing of that result in the sending object. The other resumption, called the *independent* resumption, describes the actions that can take place in parallel with the sending and processing of the message. These actions do not have to wait until the message has been answered by the destination object. (Note that for a single object the independent resumption will always be p_0, because a sending object cannot do anything before the result has arrived. However, for the correct parallel composition of more objects, the independent resumption is necessary to describe the actions of the objects that are not sending messages.) Finally we have an *answer step:* This consists of the name of the destination object and the method name, plus an even more complicated resumption. This resumption takes as input the sequence of parameters in the message plus the dependent resumption of the sender. Then it returns a process describing the further execution of the receiver and the sender *together.*

Equations like (1) can be solved by a technique explained in [BZ82]: An increasing sequence of metric spaces is constructed, its union is taken and then the metric completion of the union space satisfies the equation. The equation for POOL processes cannot be solved in this way, because the domain variable P occurs at the left-hand side of the arrow in the definition of answer steps. A more general, category-theoretic technique for solving this kind of domain equations has been developed to solve this problem. It is described in [AR88]. Let us only remark here that it is necessary to restrict ourselves to the set of *non-distance-increasing* functions (satisfying $d(f(x), f(y)) \leq d(x, y)$), which is denoted by $\rightarrow^1$ in the above equation.

Let us now give more details about the semantics of statements and expressions. These are described by the following two functions:

$$[\![\ldots]\!]_S : Stat \rightarrow Env \rightarrow AObj \rightarrow Cont_S \rightarrow^1 P$$

$$[\![\ldots]\!]_E : Exp \rightarrow Env \rightarrow AObj \rightarrow Cont_E \rightarrow^1 P.$$

The first argument of each of these function is a statement (from the set *Stat*) or an expression (from *Exp*), respectively. The second argument is an *environment*, which contains the necessary semantic information about the declarations of methods and bodies in the program (for more details, see [ABKR88]). The third argument is the name of the (active) object executing the statement/expression. The last argument is a *continuation.* This certainly deserves some explanation. It seems natural that the semantic function of a statement returns a process describing just the execution of that statement. However, we would get into trouble then, because in defining the semantics of the sequential composition $s_1; s_2$ of two statements we would have to determine the sequential composition of the corresponding processes. This turns out to be impossible, the main source of trouble being the fact that s_1 can create a new object which should execute in parallel not only with the rest of s_1, but also with s_2. In

the same way, one would expect that the semantic function for expressions just returns a value (an object name) as its result. This approach, however, would leave us with the problem of describing the possible side-effects of expression evaluation, which can be quite complicated, e.g., involving message sending.

Continuations form the most convenient and elegant solution to these problems. (For a nice introduction to the use of continuations in a sequential setting, see [Gor79].) The semantic function for statements is provided with a continuation, which is just a process ($Cont_S = P$), describing the execution of all the statements following the current one. The semantic function then delivers a process that describes the execution of the current statement plus the following ones. Analogously, the semantic function for expressions is fed with a continuation, which in this case is a function that maps object names to processes ($Cont_E = Obj \to P$). This function, when applied to the name of the object that is the result of the expression, gives a process describing everything that should happen in the current object after the expression evaluation. Again, the semantic function delivers a process describing the expression evaluation plus the following actions.

Now we are ready to give some examples of clauses that appear in the definition of the semantic functions $[\![\ldots]\!]_S$ and $[\![\ldots]\!]_E$. Let us start with a relatively simple example, the assignment statement:

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

This equation says that if the statement $x := e$ is to be executed in an environment γ (recording the effect of the declarations), by the object α, and with continuation p (describing the actions to be performed after this assignment), then first the expression e is to be evaluated, with the same environment γ and by the same object α, but its resulting object is to be fed into an expression continuation $\lambda\beta.\{\langle\sigma', p\rangle\}$ that delivers a process of which the first action is an internal one leading to the new state σ' and having the original continuation p as its resumption. Here, of course, the new state σ' is equal to $\sigma\{\beta/\alpha, x\}$, only different from σ in that the value of the variable x in the object α is now equal to β.

The semantic definition of sequential composition is easy with continuations:

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

Here the process describing the execution of the second statement s_2 just serves as the continuation for the first statement s_1.

As a simple example of a semantic definition of an expression let us take an instance variable:

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

Evaluating the expression x takes a single step, in which the value $\sigma(\alpha)(x)$ of the variable is looked up in the state σ. The resumption of this first step is obtained by feeding this value into the expression continuation f (which is a function that maps object names into processes).

As a final example of a semantic definition, let us take object creation: The expression new(C) creates a new object of class C and its value is the name of this object. Its semantics is defined as follows:

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

Here β is a fresh object name, determined from σ in a way that does not really interest us here, and σ' differs from σ only in that the variables of the new object β are initialized to NIL. We see that execution of this new-expression takes a single step, of which the resumption consists of the parallel composition of the body $\gamma(C)(\beta)$ of the new object with the execution of the creator, where the latter is obtained by applying the expression continuation f to the name of the new object β (which is, after all, the value of the new-expression). The parallel composition operator $\parallel$ is a function in $P \times P \to P$, which can be defined as the unique fixed point of a suitable contracting higher-order function $\Phi_{PC} : (P \times P \to P) \to (P \times P \to P)$ (an application of Banach's fixed point theorem).

From the above few equations it can already be seen how the use of continuations provides an elegant solution to the problems that we have mentioned.

There are a number of further steps necessary before we arrive at the semantics of a complete program. One interesting detail is that in the denotational semantics, sending messages to standard objects is treated in exactly the same way as sending messages to programmer-defined objects. The standard objects themselves (note that there are infinitely many of them!) are represented by a (huge) process p_{ST}, which is able to answer all the messages sent to standard objects and immediately returning the correct results. This process p_{ST} is composed in parallel with the process p_U, which describes the execution of the user-defined objects in order to give the process describing the execution of the whole system. From this process it is possible to derive a set of possible execution sequences that resemble the ones that we had with the operational semantics.

6.1.3 Equivalence of operational and denotational semantics

Despite the fact that the two forms of semantics described above, the operational and the denotational one, are formulated in widely different frameworks, it turns out that it is possible to establish an important relationship between them:

$$\mathcal{O} = \mathit{abstr} \circ \mathcal{D},$$

which in some sense says that the different forms of semantics of POOL are *equivalent*. Here $\mathcal{D}$ is the function that assigns a process to a POOL program according to the denotational semantics and $\mathcal{O}$ assigns to each program a set of (finite or infinite) sequences of states, which can be extracted from the sequences of configurations obtained from the operational semantics. Finally, *abstr* is an abstraction operator that takes a process and maps it into the set of sequences of states to which the process gives rise. The complete equivalence proof can be found in [Rut88]. In the present section we

shall give a rough sketch of this proof, which proceeds in several steps. We apologize for sometimes being somewhat sloppy in our notation.

The first step leads to an operational semantics that delivers a process, instead of a set of sequences of states. For this, it is most convenient to switch to a *labelled* transition system, a transition system in which the transition relation is ternary, and where the extra component gives some more information on the nature of the transition (e.g., whether it is an internal step or a communication action). Now it is possible to define another operational semantics $\mathcal{O}^*$ as follows:

$$\mathcal{O}^*(X) = \begin{cases} p_0 & \text{if } X \text{ has terminated} \\ \lambda\sigma.\left\{\langle\sigma', \mathcal{O}^*(X')\rangle \;\middle|\; \langle X,\sigma\rangle \xrightarrow{\text{int}} \langle X',\sigma'\rangle\right\} \cup \ldots & \text{otherwise} \end{cases}$$

Here X and X' are finite sets of labelled statements (see section 6.1.1), $\langle X,\sigma\rangle$ stands for a configuration containing X and σ, $\xrightarrow{\text{int}}$ stands for a transition that is labelled as an internal one, and the dots $(\ldots)$ stand for additional, more complicated terms dealing with communication actions. Note that this definition is a recursive one: $\mathcal{O}^*$ also occurs at the right-hand side. However, it can be made into a well-formed definition by taking $\mathcal{O}^*$ as the unique fixed point of a suitable contracting higher-order function. Now it is possible to prove that

$$\mathcal{O} = \mathit{abstr} \circ \mathcal{O}^*,$$

which completes the first step of the equivalence proof.

The second step consists mainly of getting rid of the continuations. Two different techniques have been developed for this. The first technique defines a number of additional semantic operators that allow a denotational (compositional) style of semantic definitions *without* the use of continuations. The resulting semantics can be proved to be equivalent both with $\mathcal{O}^*$ and with the denotational semantics *with* continuations. For a language slightly simpler than POOL (instead of methods it has a CSP-like value communication) this approach has been explored in [AB88a]. For POOL itself, this has not yet been tried, but it seems feasible. The advantage of this technique is that it provides a clear intuitive idea of the proof, but unfortunately it leads to a large number of tedious calculations (the advantages of using continuations become very clear when one tries to avoid them).

The second technique describes the semantic functions themselves as fixed points of suitable higher-order contractions over different domains, the one with and the other without continuations. By defining mappings between these domains and showing that they commute with the contractions, it can be shown that the two semantic functions are equivalent. This technique has been introduced in [KR87] and it is used in [Rut88] to prove the equivalence of POOL semantics. This technique is more difficult to explain, but it definitely leads to a shorter proof.

The final step consists of dealing with a number of details in the semantic definitions that are not yet solved by the above steps. For example, the standard objects are described by special axioms in the operational semantics, but in the denotational semantics there is a large process p_{ST} to describe them. The problem is that the above

two steps only work if in the domain equation for P we take, in the power domain $\mathcal{P}(X)$, only the *compact* subsets instead of all the closed ones. (This is because a continuous function maps each compact set into a compact one, which is not necessarily true for closed sets.) However, the process p_{ST} does not fit in this domain. The problem can be solved by proving that if p resides in the "compact" domain then $abstr(p \parallel p_{ST})$ is compact. (For more details, see [Rut88]).

6.1.4 Other forms of semantics for POOL

In addition to the operational and denotational semantics described above, POOL has been the subject of a number of other semantic studies. Let us first mention [Vaa86]. In this paper, the semantics of POOL is defined by means of *process algebra* [BK84,BK85]. This is done as follows: with the help of an attribute grammar, each POOL program is mapped unto a specification of a process in the ACP formalism with a number of additional operators. This specification in turn can be interpreted in each of the different semantic models that exist for ACP, e.g., bisimulation [BBK87] and failure semantics [BKO86].

In addition to describing the semantics of POOL, [Vaa86] also studies a number of related issues. One of them is the implementation of a fair (in a technical sense) communication mechanism with the help of message queues. The analysis in [Vaa86] detected a small error in the language manual. After this had been corrected, the correctness of this implementation could not yet be proved, unfortunately. In bisimulation semantics it can be shown that the process that results from using explicit message queues is different from the process that does not use these queues. However, this difference is due to the strict notion of equivalence in bisimulation semantics: all we are interested in is that the two processes can not be distinguished by observation from outside. This notion is captured by failure semantics (leading to some additional axioms of equality in the formalism), but unfortunately [Vaa86] does not go as far as giving the equivalence proof in this case, because of its expected complexity.

Another semantic technique, which is currently explored for its suitability to describe POOL, uses *graph grammars*. In [JR87], a special type of graph grammars, called *actor grammars*, are used to describe the semantics of actor languages, a different type of concurrent object-oriented languages [Agh86,Cli81,Hew77] (see also section 3.1). In this model, the execution of a program can be seen as a sequence of rewritings of a graph which represents the system. Production rules in the graph grammar describe how these rewritings should take place. In [Lei88] an initial study is made of the viability of such a technique for describing the semantics of POOL.

Finally, we should mention here some work which describes POOL on a different level. In [DD86,DDH87] a description is given of an abstract POOL machine. In contrast to the "abstract machine" employed in the operational semantics described above, this abstract POOL machine is intended to be the first step in a sequence of refinements which ultimately lead to an efficient implementation on real parallel hardware (DOOM). This abstract POOL machine is described formally in AADL, an Axiomatic Architecture Description Language.

6.2 Proof theory

Developing a formal proof system for verifying the correctness of POOL programs is an even harder task than giving a formal semantics for this language. Therefore this work has been done in several stages.

First the proof theory of SPOOL, a sequential version of POOL, has been studied (see [Ame86]). This language is obtained by omitting the bodies (and the possibility to return a result before a method ends) from POOL, such that now at any moment there is only one active object and we have a sequential object-oriented language. For this language a Hoare-style [Apt81,Hoa69] proof system has been developed. The main contribution from the proof theory of SPOOL was a formalism to deal with dynamically evolving pointer structures. This reasoning should take place at an abstraction level that is at least as high as that of the programming language. More concretely, this means the following:

1. The only operations on "pointers" (references to objects) are
   - testing for equality
   - dereferencing (determining the value of an instance variable of the referenced object)
2. In a given state of the system, it is only possible to reason about the objects that exist in that state, i.e., an object that does not exist (yet) cannot play a role.

Requirement 1 can be met by only admitting the indicated operations to the assertion language (however, this excludes the approach where pointers are explicitly modelled as indices in a large array that represents the "heap"). In order to satisfy requirement 2, variables are forbidden to refer to nonexisting objects and the range of quantifiers is restricted to the existing objects. (The consequence is that the range of quantification depends on the state!)

It is somewhat surprising that even with these restrictions it is possible to describe, e.g., the creation of a new object. The trick is that the reference to the new object can be removed from the precondition of the new-statement if one takes the properties of the new object into account. In fact, the SPOOL proof system has recently been proved to be complete [AB88b], i.e., every correctness formula that is true can be proved in this system. (The only addition with respect to "classical" proof systems is the possibility to quantify over finite sequences of object references, which is not uncommon in dealing with abstract data types [TZ88].)

Another contribution of the SPOOL proof system is a proof rule for message passing and method invocation (in a sequential setting). In this rule the context switching between sending and receiving object and the transmission of parameters and result are representing by appropriate substitution operations.

Along a different track a proof theory was developed to deal with parallelism, in particular with dynamic process creation. In [Boe86] a proof system was given for a language that essentially only differs from POOL in that message passing only consists

of transmitting a single value from the sender to the receiver (like in CSP [Hoa78]). The approach in this proof system is similar to that in [AFR80]: It consists of a local proof system, in which each process is verified separately, using assumptions on the behaviour of the communication statements, and a global proof system, in which these assumptions are proved, using a global invariant.

Whereas the proof system in [Boe86] uses an explicit coding of object references by numbers, an integration with the work on SPOOL has lead to a more abstract proof system for the same programming language [AB88c]. Again this proof system has been proved to be sound and complete.

6.3 Future work

Especially in the area of formal aspects there remain many unsolved problems, so that there is ample opportunity for future work. With respect to semantics, our next goal is to define a semantics in which there is a clear notion of the behaviour of a single object. Ideally, this semantics would be *fully abstract*, which means that it only leads to different meanings for constructs that can actually be *observed* to be different. In order to reach this ideal, the semantics must abstract away from the internal details of the object, because these cannot be observed from outside.

Another interesting issue connected with semantics is formal verification of an implementation of the language. In the case of the POOL2 implementation on DOOM, this is utterly infeasible, if we exclude miracles. However, certain aspects may turn out to be tractable. Some of these (e.g., message queues, see section 6.1.4) have already been tackled. The most promising implementation aspects pertain to optimizations: If a program satisfies certain syntactic conditions, objects of a certain class have a specific behaviour, which allows a simpler and more efficient implementation. Formally proving this kind of properties can make sure that the optimized version of a program indeed has the same semantics as the original one.

In the area of proof theory, the next goal is the development of a sound and complete proof system for the full language POOL, i.e., with dynamic process creation and rendez-vous communication. This is certainly not a trivial extension to the languages that have already been dealt with. Rendez-vous communication is more complex than simple value passing, especially if nested rendez-vous are possible (note that [GR84] makes use of the fact that the nesting depth is statically bounded, which is not the case in POOL).

Another goal, possibly for the more remote future, would be a *compositional* proof system, which would allow the separate verification of a single class and the construction of a complete program on the basis of external specifications of the component classes. Such compositional proof systems exist already (see, e.g., [ZREB85]), but they rely on the fact that the interconnection structure of the processes is determined statically. The dynamic structure of POOL systems will probably require totally new techniques.

An issue where both semantics and proof theory could be of considerable help is inheritance. Formalization of the approach sketched in section 5.3 would ideally consist

of a fully abstract semantics for POOL plus a specification/verification formalism that respects this semantics. It is clear that a large effort will be required to reach these goals.

7 Conclusions

In the design of POOL2 several aspects have played a role. In the first place, POOL2 should be a tool that allows a professional programmer to construct rather large and complex applications that run correctly and efficiently on a parallel machine. Extensive experience with POOL-T has shown that this language offers considerable help in this respect, by providing adequate concepts that can be used in a flexible way. POOL2 is certainly a more complex language, but once it is mastered it is even more convenient for the programmer. Many small and several medium-sized applications have been written in it with good results.

Implementing POOL efficiently on a parallel machine is not an easy task, due to the quite luxurious facilities that the language offers to the programmer. Therefore a considerable effort is needed in the construction of the compiler and the run-time system. At the moment of this writing, the first POOL2 programs are running on our parallel machine, DOOM. It is too early to give a realistic impression of the performance.

A large research effort has been directed at the formal aspects of this language. Especially developing a denotational semantics for a language is a quite severe test of the soundness and validity of its concepts. In the semantic analysis of POOL we have found no significant flaws in the language design. We hope that in the near future we can develop a formalism for the verification of POOL programs. While it is unrealistic to assume that large programs can be verified completely with such a formalism, it might nevertheless give directions towards a better informal basis for software design.

The future developments in the POOL language family will depend to a large extent on the experience with execution of POOL2 programs on a parallel machine. Experimental data will become available in the near future. This can help us to discover where the language or its implementation needs further improvement. Another possibility for the future is the integration of inheritance/subtyping into the language. In order to do this in the right way, more theoretical research is needed. New application areas may also necessitate new language concepts. For example, we can think of the notion of persistency in connection with advanced data and knowledge bases. The overall goal is a clean and consistent combination of concepts in a language of moderate complexity.

References

[AB88a] Pierre America and Jaco de Bakker. Designing equivalent semantic models for process creation. *Theoretical Computer Science*, 60(2):109–176, September 1988.

[AB88b] Pierre America and Frank de Boer. *A proof theory for a sequential version of POOL.* ESPRIT Project 415 Document 188, Philips Research Laboratories, Eindhoven, the Netherlands, July 1988.

[AB88c] Pierre America and Frank de Boer. *A proof system for a parallel language with dynamic process creation.* ESPRIT Project 415 Document 445, Philips Research Laboratories, Eindhoven, the Netherlands, October 1988.

[ABKR86] Pierre America, Jaco de Bakker, Joost N. Kok, and Jan Rutten. Operational semantics of a parallel object-oriented language. In *Conference Record of the 13th Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 13–15, 1986, 194–208.

[ABKR88] Pierre America, Jaco de Bakker, Joost N. Kok, and Jan Rutten. *Denotational semantics of a parallel object-oriented language.* ESPRIT Project 415 Document 190, Philips Research Laboratories, Eindhoven, the Netherlands, January 1988. To appear in *Information and Computation.*

[ACK87] Randy Allen, David Callahan, and Ken Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of 14th POPL*, Munich, West Germany, January 21–23, 1987, 63–76.

[AFR80] Krzysztof R. Apt, Nissim Francez, and Willem Paul de Roever. A proof system for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, July 1980.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Ame85a] Pierre America. *A sketch of POOL-S, a simplified version of POOL1.* ESPRIT Project 415 Document 27, Philips Research Laboratories, Eindhoven, the Netherlands, February 1985.

[Ame85b] Pierre America. *Definition of the programming language POOL-T.* ESPRIT Project 415 Document 91, Philips Research Laboratories, Eindhoven, the Netherlands, September 1985.

[Ame86] Pierre America. *A proof theory for a sequential version of POOL.* ESPRIT Project 415 Document 188, Philips Research Laboratories, Eindhoven, the Netherlands, October 1986.

[Ame87a] Pierre America. POOL-T — a parallel object-oriented language. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, 199–220, MIT Press, 1987.

[Ame87b] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman, editors, *ECOOP'87: European Conference on Object-Oriented Programming*, Paris, France, June 15–17, 1987, 234–242, Lecture Notes in Computer Science 276, Springer-Verlag.

[Ame88a] Pierre America. *Definition of POOL2, a parallel object-oriented language.* ESPRIT Project 415 Document 364, Philips Research Laboratories, Eindhoven, the Netherlands, April 1988.

[Ame88b] Pierre America. *Standard classes for POOL2.* ESPRIT Project 415 Document 365, Philips Research Laboratories, Eindhoven, the Netherlands, April 1988.

[Ame88c] Pierre America. *Standard units for POOL2.* ESPRIT Project 415 Document 366, Philips Research Laboratories, Eindhoven, the Netherlands, April 1988.

[Ame88d] Pierre America. *Rationale for the design of POOL2.* ESPRIT Project 415 Document 393, Philips Research Laboratories, Eindhoven, the Netherlands, May 1988.

[Ame89] Pierre America. A behavioural approach to subtyping in object-oriented programming languages. In *Workshop on Inheritance Hierarchies in Knowledge Representation and Programming Languages*, Viareggio, Italy, February 6–8, 1989. Also to appear in *Philips Journal of Research.*

[ANS83] ANSI. *The Programming Language Ada Reference Manual*, ANSI/MIL-STD-1815A-1983, approved 17 February 1983. Lecture Notes in Computer Science 155, Springer-Verlag, 1983.

[Apt81] Krzysztof R. Apt. Ten years of Hoare logic: a survey — part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[AR88] Pierre America and Jan Rutten. Solving reflexive domain equations in a category of complete metric spaces. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Language Semantics*, 1988, 254–288, Lecture Notes in Computer Science 298, Springer-Verlag.

[AS83] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[Bac78] John Backus. Can programming be liberated from the Von Neumann style? — a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[BBK87] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. On the consistency of Koomen's fair abstraction rule. *Theoretical Computer Science*, 51(1, 2):129–176, 1987.

[BDG*87] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. Keene, G. Kiczales, and D. A. Moon. *Common Lisp Object System specification.* Technical Report, ANSI Common Lisp, 1987.

[BHJ*87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[BK84] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[BK85] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, May 1985.

[BKO86] J. A. Bergstra, J. W. Klop, and E.-R. Olderog. Failures without chaos: a new process semantics for fair abstraction. In Martin Wirsing, editor, *Formal Description of Programming Concepts III — Proceedings of the Third IFIP WG 2.2 Working Conference*, Gl. Avernæs, Ebberup, Denmark, August 25–28, 1986, 77–102, North-Holland.

[BKT84] J. A. Bergstra, J. W. Klop, and J. V. Tucker. *Process algebra with asynchronous communication mechanisms.* Report CS-R8410, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, 1984.

[Boe86] Frank S. de Boer. A proof rule for process creation. In Martin Wirsing, editor, *Formal Description of Programming Concepts III — Proceedings of the Third IFIP WG 2.2 Working Conference*, Gl. Avernæs, Ebberup, Denmark, August 25–28, 1986, 23–50, North-Holland.

[BSI82] BSI. *Specification for the computer programming language Pascal.* Standard BS 6192, British Standards Institution, Herts, United Kingdom, 1982.

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

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Cli81] William Douglas Clinger. *Foundations of actor semantics.* Technical Report 633, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, May 1981.

[Cox86] Brad J. Cox. *Object-Oriented Programming.* Addison-Wesley, 1986.

[DD86] W. Damm and G. Dõhmen. *The POOL-machine: a top level specification for a distributed object-oriented machine.* ESPRIT Project 415 Document 1, Lehrstuhl für Informatik, RWTH Aachen, Aachen, West Germany, October 3, 1986.

[DDH87] W. Damm, G. Dõhmen, and P. den Haan. Using AADL to specify distributed computer architectures — a case study. In J. W. de Bakker, editor, *Deliverable D3 of the Working Group on Semantics and Proof Techniques*, chapter 1.4, ESPRIT Project 415, Philips Research Laboratories, Eindhoven, the Netherlands, October 1987.

[DN66] Ole-Johan Dahl and Kristen Nygaard. Simula: an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, September 1966.

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

[Eng77] R. Engelking. *General Topology.* Polish Scientific Publishers, 1977.

[FFGL88] Jerome A. Feldman, Mark A. Fanty, Nigel H. Goddard, and Kenton J. Lyne. Computing with structured connectionist networks. *Communications of the ACM*, 170–187, February 1988.

[FY85] Nissim Francez and Shaula A. Yemini. Symmetric intertask communication. *ACM Transactions on Programming Languages and Systems*, 7(4):622–636, October 1985.

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

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley, 1983.

[GR84] Rob Gerth and Willem Paul de Roever. A proof system for concurrent Ada programs. *Science of Computer Programming*, 4(2):159–204, August 1984.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[Hil85] W. Daniel Hillis. *The Connection Machine.* M.I.T. Press, 1985.

[HO87] Daniel C. Halbert and Patrick D. O'Brien. Using types and inheritance in object-oriented programming. *IEEE Software*, 71–79, September 1987.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

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

[HP79] Matthew Hennessy and Gordon Plotkin. Full abstraction for a simple parallel programming language. In J. Bečvář, editor, *Proceedings of the 8th Symposium on Mathematical Foundations of Computer Science*, 1979, 108–120, Lecture Notes in Computer Science 74, Springer-Verlag.

[Jon78] Anita K. Jones. The object model: a conceptual tool for structuring software. In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems — An Advanced Course*, 1978, 7–16, Springer-Verlag.

[JR87] D. Janssens and G. Rozenberg. Basic notions of actor grammars: a graph grammar model for actor computation. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph-Grammars and Their Application to Computer Science*, 1987, 280–298, Lecture Notes in Computer Science 291, Springer-Verlag.

[Ken80] Ken Kennedy. *Automatic translation of FORTRAN programs to vector form.* Technical Report 476-029-4, Rice University, October 1980.

[Kow79] Robert Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[KR87] Joost N. Kok and Jan J. M. M. Rutten. *Contractions in comparing concurrency semantics.* Report CS-R8755, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, November 1987.

[KVR83] Ron Koymans, Jan Vytopil, and Willem P. de Roever. Real-time programming and asynchronous message passing. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, Montreal, Canada, August 1983.

[LAB*81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual.* Lecture Notes in Computer Science 114, Springer-Verlag, 1981.

[Lan82] Charles Richard Lang, Jr. *The extension of object-oriented languages to a homogeneous, concurrent architecture.* Ph.D. thesis, California Institute of Technology, Computer Science Department, Pasadena, California, May 1982. Technical Report 5014:TR:82.

[Lei88] George Leih. *Actor graph grammars and POOL2.* PRISMA Project Document 265, University of Leiden, Department of Computer Science, February 1988.

[Lie81] Henry Lieberman. *A preview of Act 1.* A.I. Memo 625, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, June 1981.

[MAE*80] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* The MIT Press, 1980.

[Mey82] Bertrand Meyer. Principles of package design. *Communications of the ACM*, 25(7):419–428, July 1982.

[Mey87] Bertrand Meyer. Eiffel: programming for reusability and extendibility. *ACM SIGPLAN Notices*, 22(2):85–99, February 1987.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[MK87] J. Eliot B. Moss and Walter H. Kohler. Concurrency features for the Trellis/Owl language. In Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman, editors, *ECOOP'87: European Conference on Object-Oriented Programming*, Paris, France, June 15–17, 1987, 171–180, Lecture Notes in Computer Science 276, Springer-Verlag.

[MT86] Sape J. Mullender and Andrew S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29:289–299, August 1986.

[Odi87] Eddy A. M. Odijk. The DOOM system and its applications: a survey of ESPRIT 415 subproject A. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *Proceedings of PARLE: Parallel Architectures and Languages Europe. Volume I: Parallel Architectures*, Eindhoven, the Netherlands, June 15–19, 1987, 461–479, Lecture Notes in Computer Science 258, Springer-Verlag.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[Plo76] Gordon D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, September 1976.

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

[Plo83] Gordon D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Formal Description of Programming Concepts II*, 1983, 199–223, North-Holland.

[Rut88] Jan Rutten. *Semantic correctness for a parallel object-oriented language.* Report CS-R8843, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, October 1988.

[SCB*86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, Portland, Oregon, September 1986, 9–16.

[SCW85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. *Trellis object-based environment — language reference manual.* Technical Report DEC-TR-372, Digital Equipment Corporation, Eastern Research Lab, Hudson, Massachusetts, November 25, 1985.

[Sei85] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[Sha81] Mary Shaw, editor. *ALPHARD: Form and Content.* Springer-Verlag, 1981.

[Smi82] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[Smy78] Michael B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16:23–36, 1978.

[Ste84] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, 1984.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[TBH82] Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins. Data driven and demand driven computer architecture. *ACM Computing Surveys*, 14(1):93–143, March 1982.

[The83] Daniel G. Theriault. *Issues in the design and implementation of Act2.* Technical Report 728, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, June 1983.

[Tur85] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, 1985, 1–16, Lecture Notes in Computer Science 201, Springer-Verlag.

[TZ88] John V. Tucker and Jeffery I. Zucker. *Program Correctnes over Abstract Data Types, with Error-State Semantics.* CWI Monographs 6, North-Holland, 1988.

[Vaa86] Frits W. Vaandrager. *Process algebra semantics for POOL.* Report CS-R8629, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, August 1986.

[Wir82] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, 1982.

[WLH81] William A. Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System*. McGraw-Hill, 1981.

[WM80] Daniel Weinreb and David Moon. *Flavors: message passing in the Lisp machine*. AI Memo 602, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, November 1980.

[Wou88] Rick Wouters. *A generalized select statement for POOL*. ESPRIT Project 415 Document 430, Philips Research Laboratories, Eindhoven, the Netherlands, August 1988.

[ZREB85] Job Zwiers, Willem Paul de Roever, and Peter van Emde Boas. Compositionality and concurrent networks: soundness and completeness of a proof system. In *Proceedings of the 12th International Colloquium on Automata, Languages and Programming (ICALP)*, Nafplion, Greece, July 15–19, 1985, 509–519, Lecture Notes in Computer Science 194, Springer-Verlag.

The following paper will appear in *Journal of Computer and System Sciences* and is included in this thesis with kind permission of Academic Press, Inc..

Solving Reflexive Domain Equations in a Category of Complete Metric Spaces

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

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

This paper presents a technique by which solutions to reflexive domain equations can be found in a certain category of complete metric spaces. The objects in this category are the (non-empty) metric spaces and the arrows consist of two maps: an isometric embedding and a non-distance-increasing left inverse to it. The solution of the equation is constructed as a fixed point of a functor over this category associated with the equation. The fixed point obtained is the direct limit (colimit) of a convergent tower. This construction works if the functor is *contracting*, which roughly amounts to the condition that it maps every embedding to an even denser one. We also present two additional conditions, each of which is sufficient to ensure that the functor has a *unique* fixed point (up to isomorphism). Finally, for a large class of functors, including function space constructions, we show that these conditions are satisfied, so that they are guaranteed to have a unique fixed point. The techniques we use are so reminiscent of Banach's fixed-point theorem that we feel justified to speak of a category-theoretic version of it.

1980 Mathematical Subject classification: 68B10, 68C01.
1986 Computing Reviews Categories: D.1.3, D.3.1, F.3.2.
Key words and phrases: domain equations, complete metric spaces, category theory, converging towers, contracting functors, Banach's fixed-point theorem.

Note: This work was carried out in the context of ESPRIT project 415: Parallel Architectures and Languages for Advanced Information Processing — a VLSI-directed approach.

1. Introduction

The framework of complete metric spaces has proved to be very useful for giving a denotational semantics to programming languages, especially concurrent ones. For example, in the approach of De Bakker and Zucker [BZ] a process is modelled as the element of a suitable metric space, where the distance between two processes is defined in such a way that the smaller this distance is, the longer it takes before the two processes show a different behaviour.

In order to construct a suitable metric space in which processes are to reside, we must solve a

reflexive domain equation. For example, a simple language, where a process is a fixed sequence of uninterpreted atomic actions, gives rise to the equation

$$P \cong \{p_0\} \overline{\cup} (A \times P).$$

(Here $\overline{\cup}$ denotes the disjoint union operation.) In [BZ] an elementary technique was developed to solve such equations. Roughly, this consisted of starting with a small metric space, enriching it iteratively, and taking the metric completion of the union of all the obtained spaces.

In many cases this technique is sufficient to solve the equation at hand, but there are equations for which it does not work: equations where the domain variable P occurs in the left-hand side of a function space construction, e.g.,

$$P \cong \{p_0\} \overline{\cup} (P \rightarrow P).$$

This kind of equation arises when the semantic description is based on *continuations* (see for example [ABKR]). In this paper we present a technique by which these cases can also be solved, at least when we restrict the function space at hand to the *non-distance-increasing* functions.

The structure of this report is as follows: In section 2 we list some mathematical preliminaries. In section 3 we introduce our category $\mathcal{C}$ of complete metric spaces, we define the concepts of converging tower and contracting functor. We show that a converging tower has a direct limit and that a contracting functor preserves such a limit. Then we see how a contracting functor gives rise to a converging tower and that the limit of this tower is a fixed point of the functor.

Section 4 presents two cases in which we can show that the fixed point we construct is the unique fixed point (up to isomorphism) of the contracting functor at hand. One case arises when we work in a base-point category: a category where every space has a specially designated base-point and where every map preserves this base-point. The other case is where the functor is not only contracting, but also hom-contracting: it is a contraction on every function space.

Finally, in section 5, we present a large class of functors (including most of the ones we are interested in), for which we can show that each of them has a unique fixed point.

Acknowledgements
We would like to thank Jaco de Bakker, Frank de Boer, Joost Kok, Frank van der Linden, John-Jules Meyer and Erik de Vink for useful discussions on the contents of this paper. We are also grateful to Marino Delusso and Eline Meijs, who have typed this report.

2. Mathematical preliminaries

In this section we collect some definitions and properties concerning metric spaces, in order to refresh the reader's memory or to introduce him to this subject.

2.1. Metric spaces

DEFINITION 2.1 (Metric space)

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

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

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

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

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

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

Note that we consider only metric spaces with bounded diameter: the distance between two points never exceeds 1.

Example

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

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

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*. Notation: $\lim_{i\to\infty}x_i=x$.

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

DEFINITION 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 $A\geqslant 0$ and $f\in M_1\to^A M_2$ we have: f is continuous.*

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

DEFINITION 2.5 (Closed subsets)

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

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

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

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

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

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

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

(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 $Z\subseteq M$, $x\in M$.
An equivalent definition would be to set $V_r(X)=\{y\in M\mid\exists x\in X\,[d(x,y)<r]\}$ for $r>0$, $X\subset M$, and then to define

$$d_H(X,Y)=\inf\{r>0|\ X\subset V_r(Y)\wedge Y\subset V_r(X)\}.$$

PROPOSITION 2.7
Let (M,d), $(M_1,d_1),\ldots,(M_n,d_n)$, d_F, d_U, d_P and d_H be as in definition 2.6 and suppose that (M,d), $(M_1,d_1),\ldots,(M_n,d_n)$ are complete. We have that
(a) $(M_1\to M_2,d_F)$, $(M_1\to^A M_2,d_F)$,
(b) $(M_1\cup\cdots\cup M_n,d_U)$,
(c) $(M_1\times\cdots\times M_n,d_P)$,
(d) $(\mathcal{P}_{cl}(M),d_H)$
are complete metric spaces. If (M,d) and (M_i,d_i) are all ultra-metric spaces these composed spaces are again ultra-metric. (Strictly spoken, for the completeness of $M_1\to M_2$ and $M_1\to^A M_2$ we do not need the completeness of M_1. The same holds for the ultra-metric property.)

If in the sequel we write $M_1\to M_2$, $M_1\to^A M_2$, $M_1\cup\cdots\cup M_n$, $M_1\times\cdots\times M_n$ or $\mathcal{P}_{cl}(M)$, we mean the metric space with the metric defined above.
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].

THEOREM 2.9 (Metric completion)
Let M be an arbitrary metric space. Then there exists a metric space $\overline{M}$ (called the completion of M) together with an isometric embedding $i: M \to \overline{M}$ such that:
(1) *$\overline{M}$ is complete*
(2) *For every complete metric space M' and isometric embedding $j: M \to M'$ there exists a unique isometric embedding $\bar{j}: \overline{M} \to M'$ such that $\bar{j} \circ i = j$.*

PROOF
The space $\overline{M}$ is constructed by taking the set of all Cauchy sequences in M and dividing it out by the equivalence relation $\equiv$ defined by

$$(x_n)_n \equiv (y_n)_n \ =^{\text{def}} \lim_{n\to\infty} d(x_n, y_n) = 0.$$

The metric d_c on $\overline{M}$ is defined by

$$d_c([(x_n)]_{\equiv}, [(y_n)]_{\equiv}) \ =^{\text{def}} \lim_{n\to\infty} d(x_n, y_n)$$

and the embedding i will map every $x \in M$ to the equivalence class of the sequence of which all elements are equal to x:

$$i(x) = [(x)_n]_{\equiv}.$$

It is easy to show that $\overline{M}$ and i satisfy the above properties.

3. A CATEGORY OF COMPLETE METRIC SPACES

In this section we want to generalize the technique of solving reflexive domain equations of De Bakker and Zucker ([BZ]). We shall first give an example of their approach and then explain how it can be extended.
Consider a domain equation

$$P \cong \{p_0\} \ \overline{\cup} \ (A \times P),$$

with A an arbitrary set. In [BZ] a complete metric space that satisfies this equation is constructed as follows. An increasing sequence $A^{(0)} \subseteq A^{(1)} \subseteq \cdots$ of metric spaces is defined by

$$(0)\ A^{(0)} = \{p_0\},\ d_0 \text{ trivial},$$

$$(n+1)\ A^{(n+1)} = \{p_0\} \cup A \times A^{(n)},$$

$$d_{n+1}(p_0, q) = 1 \text{ if } q \in A^{(n+1)},\ q \neq p_0,$$

$$d_{n+1}(\langle a_1, p_1\rangle, \langle a_2, p_2\rangle) = \begin{cases} 1 & \text{if } a_1 \neq a_2 \\ \frac{1}{2} \cdot d_n(p_1, p_2) & \text{if } a_1 = a_2. \end{cases}$$

Note that for every $i \geqslant 0$, $A^{(i)}$ is a subspace of $A^{(i+1)}$. Their union is defined as

$$A^* = \bigcup_{n \in \mathbb{N}} A^{(n)},$$

and a domain A^∞ is defined as the *metric completion* of this union:

$$A^{\infty} = \overline{A^*}.$$

It is then proved that A^{∞} satisfies the equation. (We observe that A^* is isometric to the set of all finite sequences of elements of A, while A^{∞} is isometric to the set of all finite and infinite sequences, in both cases with a suitable metric.)
In order to extend this approach, we shall formulate a number of category-theoretic generalizations of some of the concepts used in the construction described above.

First we shall define a *converging tower* to be the counterpart of an increasing sequence of metric spaces; then the construction of a *direct limit* of such a tower will be the generalization of the metric completion of the union of such a sequence. Finally we shall give a generalized version of Banach's fixed-point theorem.
For this purpose we define a category $\mathcal{C}$ of complete metric spaces.

DEFINITION 3.1 (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 spaces. Then $M_1 \rightarrow^{\iota} M_2$ 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.

REMARK
For the basic definitions from category theory we refer the reader to [ML].

We can consider M_1 as an approximation of M_2: in a sense the set M_2 contains more information than M_1, because M_1 can be isometrically embedded into M_2. Elements in M_2 are approximated by elements in M_1. For an element $m_2 \in M_2$ its (best) approximation in M_1 is given by $j(m_2)$. (The reason why j should be NDI is, at this point, difficult to motivate.)
When we informally rephrase clause (c), it states that the approximation in M_1 of the embedding of an element $m_1 \in M_1$ into M_2 is again m_1. Or, in other words, that M_2 is a consistent extension of M_1.

DEFINITION 3.2
For every arrow $M_1 \rightarrow^{\iota} M_2$ in $\mathcal{C}$ with $\iota = <i,j>$ we define

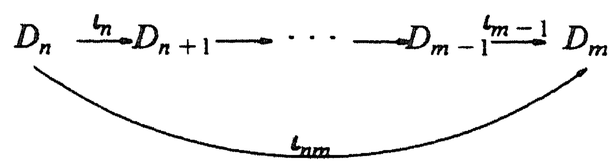
$$\delta(\iota) = d_{M_2 \rightarrow M_2}(i \circ j, id_{M_2}) \; (= \sup_{m_2 \in M_2} \{d_{M_2}(i \circ j(m_2), m_2)\}).$$

This number plays an important role in our theory. It can be regarded as a measure of the quality with which M_2 is approximated by M_1: the smaller $\delta(\iota)$, the denser M_1 is embedded into M_2.
We next try to formalize a generalization of increasing sequences of metric spaces by the following definition.

DEFINITION 3.3 (Converging tower)
(a) We call a sequence $(D_n, \iota_n)_n$ of complete metric spaces and arrows a *tower* whenever we have that $\forall n \in \mathbb{N}\ [D_n \rightarrow^{\iota_n} D_{n+1} \in \mathcal{C}]$.

$$D_0 \to^{\iota_0} D_1 \to^{\iota_1} D_2 \to \cdots \to D_n \to^{\iota_n} D_{n+1} \to \cdots$$

(b) The sequence $(D_n, \iota_n)_n$ is called a *converging tower* when furthermore the following condition is satisfied:
$\forall \epsilon > 0\ \exists N \in \mathbb{N}\ \forall m > n \geqslant N\ [\delta(\iota_{nm}) < \epsilon]$, where $\iota_{nm} = \iota_{m-1} \circ \cdots \circ \iota_n : D_n \to D_m$.



EXAMPLE 3.4
A special case of a converging tower is a sequence $(D_n, \iota_n)_n$ that satisfies the following conditions:
(a) $\forall n \in \mathbb{N}\ [D_n \to^{\iota_n} D_{n+1} \in \mathcal{C}]$,
(b) $\exists \epsilon\ [0 \leqslant \epsilon < 1 \wedge \forall n \in \mathbb{N}\ [\delta(\iota_{n+1}) \leqslant \epsilon \cdot \delta(\iota_n)]]$.
(Note that $\delta(\iota_{nm}) \leqslant \delta(\iota_n) + \cdots + \delta(\iota_{m-1}) \leqslant \epsilon^n \cdot \delta(\iota_0) + \cdots + \epsilon^{m-1} \cdot \delta(\iota_0) \leqslant \frac{\epsilon^n}{1-\epsilon} \cdot \delta(\iota_0)$.)

EXAMPLE 3.5
Let $A^{(0)} \subseteq A^{(1)} \subseteq \cdots$ be the sequence of metric spaces defined at the beginning of this chapter. We show how it can be transformed into a converging tower, by defining a sequence of arrows $(\iota_n)_n$ (with $\iota_n = \langle i_n, j_n \rangle$) with induction on n:

(0) $i_0(p_0) = p_0$, j_0 trivial,

$(n+1)$ $i_{n+1} : A^{(n+1)} \to A^{(n+2)}$, trivial $(i_{n+1}(p) = p)$,

$j_{n+1} : A^{(n+2)} \to A^{(n+1)}$,

$j_{n+1}(p_0) = p_0$,

$j_{n+1}(\langle a, p \rangle) = \langle a, j_n(p) \rangle$ for $\langle a, p \rangle \in A^{(n+2)}$.

It is not difficult to see that we have obtained a tower

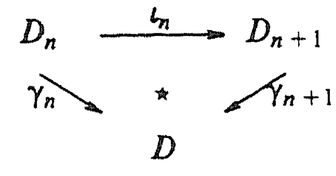
$$A^{(0)} \to^{\iota_0} A^{(1)} \to^{\iota_1} \cdots,$$

which is converging.

3.1 The direct limit construction
In this subsection we show that in our category $\mathcal{C}$ every converging tower has an *initial cone*. The construction of such an initial cone for a given tower (the *direct limit* construction) generalizes the technique of forming the metric *completion* of the union of an increasing sequence of metric spaces.
Before we treat the inverse limit construction, we first give the definition of a cone and an initial cone and then formulate a criterion for the initiality of a cone.

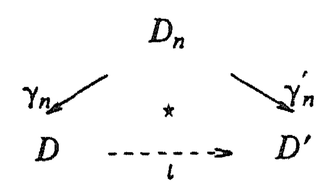
DEFINITION 3.6 (Cone)
Let $(D_n, \iota_n)_n$ be a tower. Let D be a complete metric space and $(\gamma_n)_n$ a sequence of arrows. We call $(D, (\gamma_n)_n)$ a *cone* for $(D_n, \iota_n)_n$ whenever the following condition holds:

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



DEFINITION 3.7 (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 $\iota:D\to D'$ in $\mathcal{C}$ such that:

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



LEMMA 3.8 (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:

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

PROOF
$\Leftarrow$
Suppose $\lim_{n\to\infty}\alpha_n\circ\beta_n=id_D$. Let $(D',(\gamma'_n)_n)$, with $\gamma'_n=<\alpha'_n,\beta'_n>$, be another cone for $(D_n,\iota_n)_n$. We have to prove the existence of a unique arrow $D\to^{\iota}D'\in\mathcal{C}$ such that

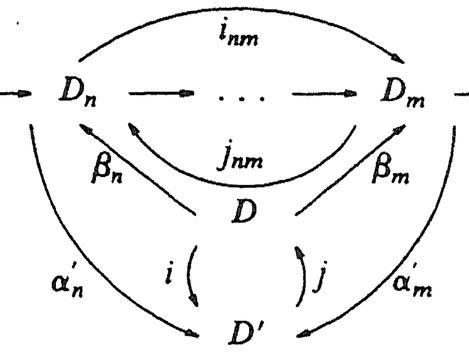
$$\forall n\in\mathbb{N}\ [\iota\circ\gamma_n=\gamma'_n].$$

First we construct an embedding $i:D\to D'$, then a projection $j:D'\to D$. Next, the arrow ι will be defined as $\iota=<i,j>$.
For every $n\in\mathbb{N}$ we have

$$\alpha'_n\circ\beta_n\in D\to D'.$$

We show that $(\alpha'_n\circ\beta_n)_n$ is a Cauchy sequence in $D\to D'$ and then use the completeness of this function space to define i as the limit of that sequence.
Let $m>n\geq 0$. We have

$$d_{D\to D'}(\alpha'_m\circ\beta_m,\ \alpha'_n\circ\beta_n)=$$
$$d_{D\to D'}(\alpha'_m\circ\beta_m,\ \alpha'_m\circ i_{nm}\circ j_{nm}\circ\beta_m)=$$
$$\sup_{x\in D}\{d_{D'}(\alpha'_m\circ\beta_m(x),\ \alpha'_m\circ i_{nm}\circ j_{nm}\circ\beta_m(x))\}=$$
[because α'_m is isometric]
$$\sup_{x\in D}\{d_{D_m}(\beta_m(x),\ i_{nm}\circ j_{nm}\circ\beta_m(x))\}=$$
[because β_m is surjective]
$$\sup_{x\in D_m}\{d_{D_m}(x,\ i_{nm}\circ j_{nm}(x))\}=$$
$$d_{D_m\to D_m}(id_{D_m},\ i_{nm}\circ j_{nm}))=\delta(\iota_{nm}).$$



Let $\epsilon>0$. Because $(D_n, \iota_n)_n$ is a converging tower there is an $N\in\mathbb{N}$ such that

$$\forall m>n\geqslant N\ [\ \delta(\iota_{nm})<\epsilon\ ].$$

Thus $(\alpha'_n\circ\beta_n)_n$ is a Cauchy sequence. We define

$$i = \lim_{n\to\infty}\alpha'_n\circ\beta_n\ .$$

We prove that i is isometric by showing:

$$\forall x, y\in D\ [\ d_{D'}\ (i(x), i(y)) = d_D(x, y)\ ]$$

Let $x, y\in D$, we have

$$d_{D'}(i(x), i(y)) =$$
$$d_{D'}(\lim_{n\to\infty}\alpha'_n\circ\beta_n(x), \lim_{n\to\infty}\alpha'_n\circ\beta_n(y)) =$$
$$\lim_{n\to\infty}d_{D'}(\alpha'_n\circ\beta_n(x), \alpha'_n\circ\beta_n(y)) =$$
[because α'_n is isometric]
$$\lim_{n\to\infty}d_{D_n}(\beta_n(x), (\beta_n(y)) =$$
[because α_n is isometric]
$$\lim_{n\to\infty}d_D(\alpha_n\circ\beta_n(x), \alpha_n\circ\beta_n(y)) =$$
$$d_D(\lim_{n\to\infty}\alpha_n\circ\beta_n(x), \lim_{n\to\infty}\alpha_n\circ\beta_n(y)) =$$
$$d_D(x, y)\ .$$

Thus i is isometric.

Similar to the definition of i we choose

$$j = \lim_{n\to\infty}\alpha_n\circ\beta'_n\ .$$

We have that j is NDI, because, for $x, y\in D'$:

$$d_D(j(x), j(y)) =$$
$$d_D(\lim_{n\to\infty}\alpha_n\circ\beta'_n(x), \lim_{n\to\infty}\alpha_n\circ\beta'_n(y)) =$$
$$\lim_{n\to\infty}d_D(\alpha_n\circ\beta'_n(x), \alpha_n\circ\beta'_n(y)) =$$
[because α_n is isometric]
$$\lim_{n\to\infty}d_{D_n}(\beta'_n(x), (\beta'_n(y)) \leqslant$$
[because β'_n is NDI]

$\lim_{n\to\infty} d_{D'}(x, y) =$
$d_{D'}(x, y)$.

We also show : $j \circ i = id_D$. Let $x \in D$, then

$$j \circ i(x) =$$
$$j(\lim_{n\to\infty} \alpha'_n \circ \beta_n(x)) =$$
$$\lim_{n\to\infty} j \circ \alpha'_n \circ \beta_n(x) =$$
$$\lim_{n\to\infty} \lim_{m\to\infty} \alpha_m \circ \beta'_m \circ \alpha'_n \circ \beta_n(x) =$$
$$\lim_{n\to\infty} \alpha_n \circ \beta'_n \circ \alpha'_n \circ \beta_n(x) =$$
$$[\text{ because } \beta'_n \circ \alpha'_n = id_{D_n}]$$
$$\lim_{n\to\infty} \alpha_n \circ \beta_n(x) = x .$$

Now we can define

$$\iota = \langle i, j \rangle ,$$

of which we have so far proved : $D \to^{\iota} D' \in \mathcal{C}$.

Next we have to verify that ι satisfies the condition

$$\forall m \in \mathbb{N} [\ \iota \circ \gamma_m = \gamma'_m\ ].$$

This amounts to

$$\forall m \in \mathbb{N} [\ i \circ \alpha_m = \alpha'_m \wedge \beta_m \circ j = \beta'_m\ ].$$

Let $m \geqslant 0$. We only prove the first part of the conjunction. We have

$$\begin{aligned} i \circ \alpha_m &= (\lim_{n\to\infty} \alpha'_n \circ \beta_n) \circ \alpha_m \\ &= (\lim_{n\to\infty} \alpha'_{n+m} \circ \beta_{n+m}) \circ \alpha_m \\ &= \lim_{n\to\infty} \alpha'_{n+m} \circ \beta_{n+m} \circ \alpha_m \\ &= \lim_{n\to\infty} \alpha'_{n+m} \circ \beta_{n+m} \circ \alpha_{n+m} \circ i_{m,\, m+n} \\ &= \lim_{n\to\infty} \alpha'_{n+m} \circ id_{D_{n+m}} \circ i_{m,\, m+n} \\ &= \lim_{n\to\infty} \alpha'_m = \alpha'_m . \end{aligned}$$

Finally we show that ι is *unique*. Suppose $D \to^{\iota'} D'$, with $\iota' = \langle i', j' \rangle$, is another arrow in $\mathcal{C}$, that satisfies

$$\forall m \in \mathbb{N} [\ \iota' \circ \gamma_m = \gamma'_m\ ].$$

We only show that $i' = i$, leaving the proof of $j' = j$ to the reader:

$$\begin{aligned} i' &= i' \circ id_D \\ &= i' \circ \lim_{m\to\infty} \alpha_m \circ \beta_m \\ &= \lim_{m\to\infty} i' \circ \alpha_m \circ \beta_m \\ &= \lim_{m\to\infty} \alpha'_m \circ \beta_m \\ &= i . \end{aligned}$$

$\Rightarrow$

Suppose now that $(D, (\gamma_n)_n)$ is an initial cone of the converging tower $(D_n, \iota_n)_n$. We have to prove that

$$\lim_{m\to\infty} \alpha_n \circ \beta_n = id_D \ .$$

By an argument similar to the proof for $(\alpha'_n \circ \beta_n)_n$ above, we have that $(\alpha_n \circ \beta_n)_n$ is a Cauchy sequence. We define

$$f = \lim_{n\to\infty} \alpha_n \circ \beta_n \ ,$$
$$D' = \{\ x \mid x \in D \mid f(x) = x\ \}.$$

We set out to prove that $D' = D$.
The set D' is a closed subset of D, so it again constitutes a complete metric space. For each $n \in \mathbb{N}$ we have

$$\alpha_n : D_n \to D'$$

because of the following argument. Let $d \in D_n$, then:

$$f(\alpha_n(d)) =$$
$$\lim_{m\to\infty} \alpha_m \circ \beta_m(\alpha_n(d)) =$$
$$\lim_{m\to\infty} \alpha_{n+m} \circ \beta_{n+m} \circ (\alpha_n(d)) =$$
$$\lim_{m\to\infty} \alpha_{n+m} \circ \beta_{n+m} \circ \alpha_{n+m} \circ i_{n,\,n+m}(d) =$$
$$\lim_{m\to\infty} \alpha_{n+m} \circ i_{n,\,n+m}(d) =$$
$$\lim_{m\to\infty} \alpha_n(d) =$$
$$\alpha_n(d) \ .$$

So $f(\alpha_n(d)) = \alpha_n(d)$, and thus $\alpha_n(d) \in D'$.
Next we define, for each $n \in \mathbb{N}$:

$$\alpha'_n = \alpha_n \ ,$$
$$\beta'_n = \beta_n 1 D' \ \ (\beta_n \text{ restricted to } D'),$$
$$\gamma'_n = <\alpha'_n, \beta'_n> \ .$$

It is clear that $(D', (\gamma'_n)_n)$ is another cone for $(D_n, \iota_n)_n$. Because $(D, (\gamma_n)_n)$ is initial, there exists a unique arrow $D \to^{\iota_1} D' \in \mathcal{C}$ with $\iota_1 = <i_1, j_1>$ such that

$$\forall n \in \mathbb{N}\ [\ \iota_1 \circ \gamma_n = \gamma'_n\ ] \ .$$

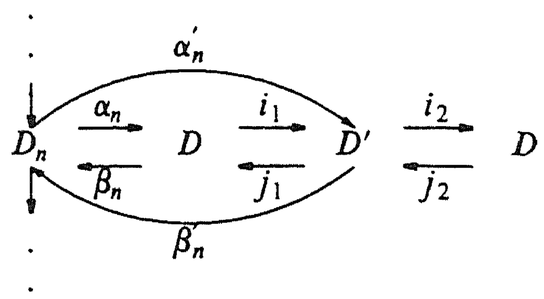
The set D' can also be embedded into D: let $D' \to^{\iota_2} D$, with $\iota_2 = <i_2, j_2>$, be defined by

$$i_2 = id_{D'} \ ,$$
$$j_2 = i_1 \ .$$

Then $D' \to^{\iota_2} D \in \mathcal{C}$. For i_2 is isometric, j_2 is NDI and the following argument shows that $j_2 \circ i_2 = id_{D'}$. Let $d \in D'$. Then

$$j_2 \circ i_2(d) = j_2(d)$$
$$= i_1(d)$$
$$= [\ \text{because } d \in D', \text{ we have } f(d) = d \ ;$$
$$\text{in other words, } (\lim_{n\to\infty} \alpha_n \circ \beta_n)(d) = d\ ]$$

$$
\begin{aligned}
& (i_1 \circ (\lim_{n\to\infty} \alpha_n \circ \beta_n))(d) \\
= \ & \lim_{n\to\infty} (i_1 \circ \alpha_n \circ \beta_n)(d) \\
= \ & \lim_{n\to\infty} (\alpha'_n \circ \beta_n)(d) \\
= \ & \lim_{n\to\infty} (\alpha_n \circ \beta_n)(d) = d \, .
\end{aligned}
$$



Now we are able to define $D \to {}^{\iota} D$ by

$$
\begin{aligned}
\iota &= \iota_2 \circ \iota_1 \\
&= \langle i_2 \circ i_1, j_1 \circ j_2 \rangle \, .
\end{aligned}
$$

It is easy to verify that

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

By the initiality of D we have that

$$\iota = \langle id_D \, , \, id_D \rangle \, .$$

Thus $i_2 \circ i_1 = id_D$. This implies $D = D'$.
Conclusion:

$$\lim \alpha_n \circ \beta_n = id_D \, .$$

The initiality lemma will appear to be very useful in the sequel, where we shall construct a cone for an arbitrary converging tower and prove that it is initial.

DEFINITION 3.9 (Direct limit construction)
Let $(D_n, \iota_n)_n$, with $\iota_n = \langle i_n, j_n \rangle$, be a converging tower. The *direct limit* of $(D_n, \iota_n)_n$ is a cone $(D, (\gamma_n)_n)$, with $\gamma_n = \langle \alpha_n, \beta_n \rangle$, that is defined as follows:

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

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

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

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

LEMMA 3.10
Let (D, d) be as defined above. We have:

(D,d) is a complete metric space.

PROOF
Let $(x_n)_n$, $(y_n)_n \in D$. Let $m>n\geqslant 0$, then

$$\begin{aligned} d_{D_n}(x_n, y_n) &= d_{D_n}(j_{nm}(x_m), j_{nm}(y_m)) \\ &\leqslant [\text{ because } j_{nm} \text{ is NDI }] \\ &\quad d_{D_m}(x_m, y_m)\,. \end{aligned}$$

Thus $(d_{D_n}(x_n, y_n))_n$ is an increasing sequence. It is bounded by 1, thus its supremum exists, and is equal to the limit. It is not difficult to show that d is a metric.
We shall prove the completeness of D with respect to this metric. Let $(\bar{x}^i)_i$, with $\bar{x}^i=(x_0^i, x_1^i, x_2^i, \ldots)$ be a Cauchy sequence in D. Because for all k and for all n and m:

$$\begin{aligned} d_{D_k}(x_k^n, x_k^m) &\leqslant \sup_{k\in\mathbb{N}}\{d_{D_k}(x_k^n, x_k^m)\} \\ &= d\,(\bar{x}^n, \bar{x}^m) \end{aligned}$$

and $(\bar{x}^i)_i$ is a Cauchy sequence, we have, for all $k\in\mathbb{N}$, that $(x_k^i)_i$ is a Cauchy sequence in D_k. For every k we set

$$x_k = \lim_{i\to\infty} x_k^i\,.$$

We have $j_k(x_{k+1})=x_k$, since

$$\begin{aligned} j_k(x_{k+1}) &= j_k(\lim_{i\to\infty} x_{k+1}^i) \\ &= \lim_{i\to\infty} j_k(x_{k+1}^i) \\ &= \lim_{i\to\infty} x_k^i \\ &= x_k\,. \end{aligned}$$

Thus $(x_k)_k$ is an element of D.
Because the convergence of the sequences $(x_k^i)_i$ for $k\in\mathbb{N}$ was uniform, we have

$$\forall\epsilon>0\ \exists N\in\mathbb{N}\ \forall k\in\mathbb{N}\ \forall n>N\ [\ d_{D_k}(x_k^n, x_k)<\epsilon\ ]\,.$$

This fact implies that $(x_k)_k$ is the limit of $(\bar{x}^i)_i$, since, for $\epsilon>0$,

$$\begin{aligned} d((x_k)_k, \bar{x}^n) &= \sup_{k\in\mathbb{N}}\{d_{D_k}(x_k, x_k^n)\} \\ &\leqslant \epsilon \end{aligned}$$

for n bigger than a suitable N.

RELATION BETWEEN THE DIRECT LIMIT CONSTRUCTION AND METRIC COMPLETION
We can look upon the construction of the direct limit for a tower $(D_n,i_n)_n$ as a generalization of taking the metric completion of the union of a sequence of metric spaces. We define

$$\begin{aligned} D_0' &= \{0\}\times D_0 \\ D_{n+1}' &= \{n+1\}\times(D_{n+1}\setminus i_n(D_n)) \cup D_n', \end{aligned}$$

and take $l_n: D_n\to D_n'$ as follows:

$$l_0(d) = <0,d> \quad \text{for } d\in D_0,$$

$$l_{n+1}(d) = \begin{cases} l_n(d') & \text{if } d=i_n(d')\in D_{n+1} \text{ with } d'\in D_n \\ <n+1,d> & \text{if } d\notin i_n(D_n). \end{cases}$$

Because each i_n is an injection, this construction works, and we see that each l_n is a bijection. Therefore, we can use $(l_n)_n$ in the obvious way to define a metric d'_n on each D'_n and suitable $i'_n: D'_n \to D'_{n+1}$ and $j'_n: D'_{n+1} \to D'_n$.

Now we have an isomorphic copy of our original tower, which satisfies the condition that each $i'_n: D'_n \to D'_{n+1}$ is a subset embedding. From now on we leave out the primes, and just suppose that $i_n: D_n \to D_{n+1}$ satisfies this condition.

If we define U as the union of $(D_n)_n$, and $d: U \times U \to [0,1]$ by

$$d(x,y) = d_{D_k}(x,y),$$

whenever $x \in D_n, y \in D_m$ and $k \geqslant m,n$, we have that (U,d) is a metric space. Generally, it will not be complete. The direct limit of $(D_n,i_n)_n$ can be regarded as the completion of (U,d) in the following sense.

In U we consider only such sequences $(x_n)_n$, for which:

$$\forall n \in \mathbb{N}[x_n \in D_n] \tag{1}$$

and

$$\forall n \in \mathbb{N}[x_n = j_n(x_{n+1})]. \tag{2}$$

It follows that $(x_n)_n$ is a Cauchy sequence. For $m > n$ we have

$$\begin{aligned} d(x_m,x_n) &= d_{D_n}(x_m, i_{nm}(x_n)) \\ &= d_{D_n}(x_m, i_{nm} \circ j_{nm}(x_m)) \\ &\leqslant d_{D_m \to D_m}(id_{D_m}, i_{nm} \circ j_{nm}) \\ &= \delta(\iota_{nm}). \end{aligned}$$

This number is small for large n and m, because $(D_n,i_n)_n$ is a converging tower.

For every $(x_n)_n$ and $(y_n)_n$ in U, that both satisfy (1) and (2), we have:

$$\text{if } \lim_{n\to\infty} d_{D_n}(x_n,y_n) = 0, \text{ then } (x_n)_n = (y_n)_n,$$

because of:

$$\begin{aligned} d_{D_n}(x_n,y_n) &= d_{D_n}(j_n(x_{n+1}), j_n(y_{n+1})) \\ &\leqslant d_{D_{n+1}}(x_{n+1}, y_{n+1}) \end{aligned}$$

(expressing that $(d_{D_n}(x_n,y_n))_n$ is a monotonic, non-decreasing sequence with limit 0, so all its elements are 0).

Of course it is not the case that every Cauchy sequence satisfies (1) and (2), but we can find in each class of Cauchy sequences that will have the same limit a representative sequence, which satisfies (1) and (2), and which by the above is unique. Let $(x_n)_n$ be an arbitrary Cauchy sequence in U. As a representative of the class of Cauchy sequences with the same limit as $(x_n)_n$, we take the sequence $(y_n)_n$, defined by

$$y_n = \lim_{m\to\infty} x^n_m,$$

with

$$x^n_m = \begin{cases} x_m & \text{if } x_m \in D_n \\ j_{nk}(x_m) & \text{if } x_m \notin D_n, \text{ and } k > n \text{ is the least number with } x_m \in D_k \end{cases}$$

(Remember that $k > n \Rightarrow D_k \supset D_n$). It is not very difficult to show, that we have indeed:

$$\lim_{n\to\infty} d_{D_n}(x_n,y_n) = 0,$$

and that $(y_n)_n$ satisfies (1) and (2). Finally we remark that the direct limit D of $(D_n, \iota_n)_n$ consists of exactly those sequences in U, that satisfy (1) and (2), and thus can be viewed as the metric completion of (U,d).

Remember from theorem 2.9 that the metric completion $\overline{M}$ of a metric space M is the smallest complete metric space, into which M can be isometrically embedded, in the following sense: $\overline{M}$ can be isometrically embedded into every other complete metric space with that property.
For the direct limit of a converging tower, we have a similar initiality property:

LEMMA 3.11
The direct limit of a converging tower (as defined in definition 3.9) is an initial cone for that tower.

PROOF
Let $(D_n, \iota_n)_n$ and $(D, (\gamma_n)_n)$ be as defined in definition 3.9. According to the initiality lemma (3.8), it suffices to prove

$$\lim_{n\to\infty} \alpha_n \circ \beta_n = id_D ,$$

which is equivalent to

$$\forall \epsilon > 0\ \exists N \in \mathbb{N}\ \forall n > N\ [\ d(\alpha_n \circ \beta_n, id_D) < \epsilon\ ]$$

Let $\epsilon > 0$. Because $(D_n, \iota_n)_n$ is a converging tower, we can choose $N \in \mathbb{N}$ such that

$$\forall m > n \geqslant N\ [\ d(i_{nm} \circ j_{nm}, id_{D_m}) < \epsilon\ ].$$

Let $n > N$. Let $(x_m)_m \in D$, we define

$$(y_m)_m = \alpha_n \circ \beta_n((x_m)_m).$$

For every $m > n$ we have

$$\begin{aligned} d_{D_m}(y_m, x_m) &= d_{D_m}(i_{nm}(x_n), x_m) \\ &= d_{D_m}(i_{nm} \circ j_{nm}(x_m), x_m) \\ &\leqslant d(i_{nm} \circ j_{nm}, id_{D_m}) \\ &< \epsilon \end{aligned}$$

Therefore

$$d_D((y_m)_m, (x_m)_m) = \sup\{d_{D_m}(y_m, x_m)\} \leqslant \epsilon .$$

Because $(x_n)_n \in D$ was arbitrary, we have

$$d(\alpha_n \circ \beta_n, id_D) < \epsilon$$

for all $n > N$.

3.2 A fixed-point theorem
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 3.12 (Contracting functor)
We call a functor $F: \mathcal{C} \to \mathcal{C}$ contracting whenever the following holds: there exists an ϵ, with $0 \leqslant \epsilon < 1$, such that for all $D \to' E \in \mathcal{C}$ we have:

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

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

LEMMA 3.13
Let $F:\mathcal{C}\to\mathcal{C}$ 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.

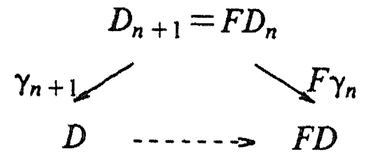
The proof, which may use the initiality lemma, is left to the reader.

THEOREM 3.14 (Fixed-point theorem)
Let Cat be a category and let $F:Cat\to Cat$ be a functor. Let $D_0\to^{\iota_0} FD_0\in Cat$. 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 $n\geqslant 0$. If this tower has an initial cone $(D,(\gamma_n)_n)$ and if this tower and its cone are preserved under F, that is, if $(FD_n,F\iota_n)_n$ has $(FD,(F\gamma_n)_n)$ as an initial cone, then we have: $D\cong FD$.

PROOF
We have that

$$(FD_n,\ F\iota_n)_n\ =\ (D_{n+1},\ \iota_{n+1})_n.$$

This implies that $(D,\ (\gamma_n)_n)$ and $(FD,\ (F\gamma_n)_n)$ are both initial cones of $(D_{n+1},\ \iota_{n+1})_n$. It follows from the definition of an initial cone that D and FD are isomorphic.



COROLLARY 3.15 *Let F be a contracting functor $F:\mathcal{C}\to\mathcal{C}$ and let $D_0\to^{\iota_0} FD\in\mathcal{C}$. Then F has a fixed point, that is, there exist a $D\in\mathcal{C}$ with $D\cong FD$.*

PROOF Consider the tower $(D_n,\iota_n)_n$ defined by $D_{n+1}=FD_n$ and $\iota_{n+1}=F\iota_n$ for all $n\geqslant 0$. This tower can be seen to be converging in the same way as in example 3.4. Thus it has a direct limit $(D,(\gamma_n)_n)$, which is (according to lemma 3.11) an initial cone for this tower. According to lemma 3.13, F preserves towers and their initial cones. Now we can apply theorem 3.14, which yields: $D\cong FD$.

REMARK
It is always possible to find an arrow $D_0\to^{\iota_0} FD_0\in\mathcal{C}$: Take $D_0=\{p_0\}$; because FD_0 is non-empty we can choose an arbitrary $p_1\in FD_0$, and put $\iota_0=<i_0,j_0>$ with $i(p_0)=p_1$ and $j(x)=p_0$, for $x\in FD_0$.

4. UNIQUENESS OF FIXED POINTS

We know that a contracting function $f:M\to M$, on a complete metric space M, has a *unique* fixed point. We would like to prove a similar property for contracting functors on $\mathcal{C}$.
Let us consider a contracting functor F on the category of complete metric spaces $\mathcal{C}$. By corollary

3.15 we know that F has a fixed point, that is there exists $D \in \mathcal{C}$ and an isometry κ such that

$$D \xrightarrow[\cong]{\kappa} FD.$$

Suppose we have another fixed point D' with an isometry λ, such that

$$D' \xrightarrow[\cong]{\lambda} FD'.$$

We know by the construction of D that it is the direct limit of the converging tower $(D_n, \iota_n)_n$, where $D_0 \to^{\iota_0} FD_0 \in \mathcal{C}$ is a given embedding and $D_{n+1} = FD_n$, $\iota_{n+1} = F\iota_n$.
If we have that D' is also (the endpoint of) a cone for that tower, the initiality of D implies that there exists an isometric embedding $D \to^{\iota} D' \in \mathcal{C}$. If we moreover can demonstrate that this ι is an isometry, then we can conclude that the functor F has a unique fixed point, which would be quite satisfactory.
A proof for ι being an isometry might look like:

$$\begin{aligned} \delta(\iota) &= (?)\, \delta(F\iota) \\ &\leqslant \epsilon \cdot \delta(\iota), \end{aligned}$$

implying (once the question-mark has been eliminated) that $\delta(\iota)=0$, thus ι is an isometry.
It turns out that we can guarantee that the second fixed point D' is also a cone for the converging tower $(D_n, \iota_n)_n$ in one of *two* ways. Firstly, we can restrict our functor F to the *base-point* category of complete metric spaces (to be defined in a moment). Secondly, we can require F to be contracting in yet another sense, to be called *hom*-contracting below.
We shall proceed in both directions, first exploring the unicity of fixed points of contracting functors on the base-point category, then focusing on functors on $\mathcal{C}$ that are contracting and hom-contracting.
In both cases it appears to be possible to prove the equality marked by (?) above. Unfortunately (for good mathematicians, who are said to be lazy), this takes some serious effort, to which the proof of the following theorem bears witness.
First we give the definition of the base-point category:

DEFINITION 4.1 (Base-point category of complete metric spaces)
Let $\mathcal{C}^*$ denote the base-point category of complete metric spaces, which has triples

$$< M, d, m >$$

for its objects. Here (M,d) is a complete metric space and m is an arbitrary element of M, called the base-point of M. The arrows in $\mathcal{C}^*$ are as in $\mathcal{C}$ (see definition 3.1), but for the constraint that they map base-points onto base-points, i.e. for $< M,d,m > \to^{<i,j>} < M',d',m' > \in \mathcal{C}^*$ we also require that $i(m)=m'$, and $j(m')=m$.

REMARK
The definitions of cone, functor etcetera can be adapted straightforwardly. Moreover, lemmas 3.8, 3.11, 3.13 and corollary 3.15 still hold.

THEOREM 4.2 (Uniqueness of fixed points)
Let F be a contracting functor $F: \mathcal{C}^* \to \mathcal{C}^*$. Then F has a unique fixed point up to isometry, that is to say: there exists a $D \in \mathcal{C}^*$ such that

(1) $FD \cong D$, and
(2) $\forall D' \in \mathcal{C}^*\ [FD' \cong D' \Rightarrow D \cong D']$.

PROOF

We define a converging tower $(D_n, \iota_n)_n$ by

$$D_0 = <\{p_0\}, d_{\{p_0\}}, p_0> ,$$

$$D_{n+1} = FD_n \text{ for all } n \geqslant 0 ,$$

$$\iota_0 : D_0 \to D_1, \text{ trivial,}$$

$$\iota_{n+1} = F\iota_n \text{ for all } n \geqslant 0 .$$

Let $(D, (\gamma_n)_n)$ be the direct limit of this tower. As in theorem 3.14, we have that both $(D, (\gamma_n)_n)$ and $(FD, (F\gamma_n)_n)$ are initial cones of $(D_n, \iota_n)_n$. The initiality of $(D_n, (\gamma_n)_n)$ implies the existence of a unique arrow $D \to^{\kappa} FD$, such that for $n \geqslant 0$,

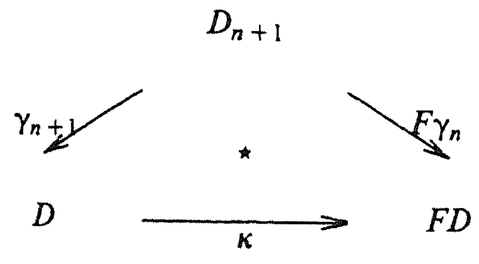


FIGURE 1

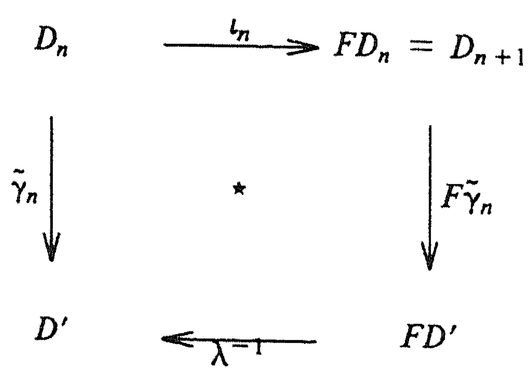
Because also $(FD, (F\gamma_n))_n$ is initial, we know that κ must be isometric.

Now let $D' \in \mathcal{C}$ be another fixed point of F, say $D' \xrightarrow[\cong]{\lambda} FD'$ for an isometry λ. We define $(\tilde{\gamma}_n)_n$ such that $(D', (\tilde{\gamma}_n)_n)$ is a cone for $(D_n, \iota_n)_n$:

$$\tilde{\gamma}_0 : D_0 \to D' \text{ is the unique arrow, which maps base-point to base-point,}$$

$$\tilde{\gamma}_{n+1} = \lambda^{-1} \circ F\tilde{\gamma}_n .$$

We have that $(D', (\tilde{\gamma}_n)_n)$ is indeed a cone for $(D_n, \iota_n)_n$ because of the commutativity of the following diagram, for all $n \in \mathbb{N}$:



We prove it by induction on n :

(0) Because the arrows in $\mathcal{C}$ map base-points onto base-points, we have that $(\lambda^{-1} \circ F\tilde{\gamma}_0 \circ \iota_0)_1(p_0)$ and $(\tilde{\gamma}_0)_1(p_0)$ are both equal to the base-point of D', and for any $x \in D'$, that $(\lambda^{-1} \circ F\tilde{\gamma}_0 \circ \iota_0)_2(x) = (\tilde{\gamma}_0)_2(x) = p_0$.

Note that this is the *only* place, where we make use of the base-point structure of $\mathcal{C}^*$.

$(n+1)$ Suppose that we have $\lambda^{-1} \circ F\tilde{\gamma}_n \circ \iota_n = \tilde{\gamma}_n$. Then

$$\begin{aligned}\lambda^{-1} \circ F\tilde{\gamma}_{n+1} \circ \iota_{n+1} &= \lambda^{-1} \circ F(\tilde{\gamma}_{n+1} \circ \iota_n) \\ &= \lambda^{-1} \circ F(\lambda^{-1} \circ F\tilde{\gamma}_n \circ \iota_n) \\ &= \lambda^{-1} \circ F\tilde{\gamma}_n \\ &= \tilde{\gamma}_{n+1} .\end{aligned}$$

Again by the initiality of $(D, (\gamma_n)_n)$ there is a unique arrow $D \to' D'$ such that, for all $n \in \mathbb{N}$:

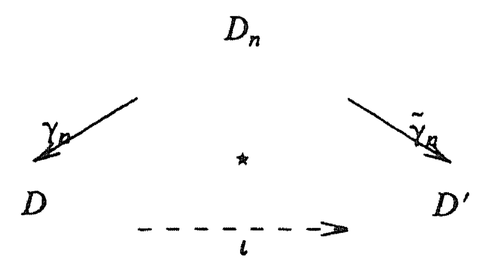
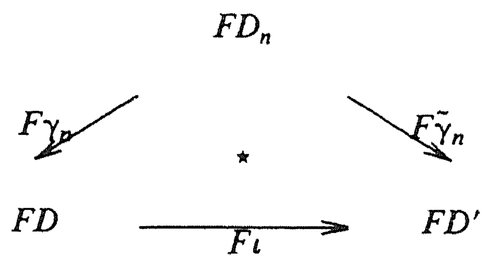
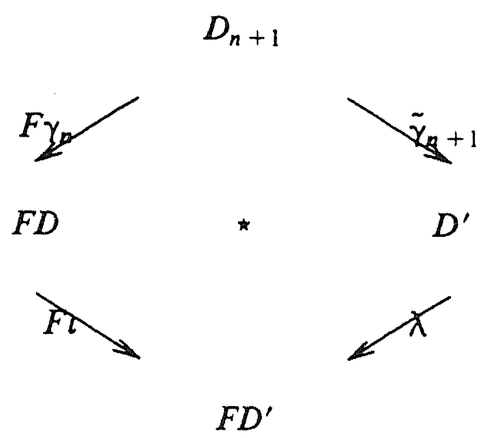


FIGURE 2

As indicated above, we now set out to prove that ι is an isometry. When we apply F to figure 2, we get

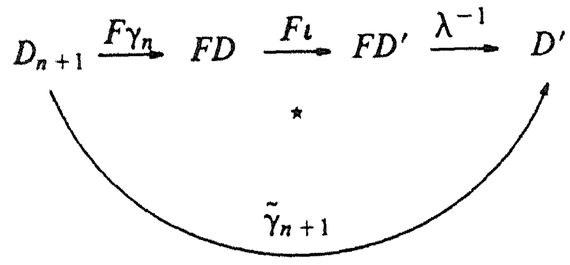


which leads to:

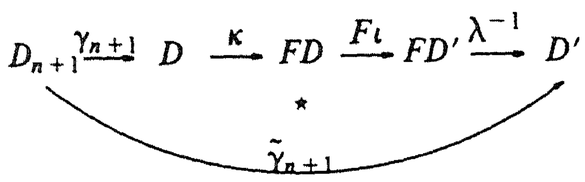


(because $\tilde{\gamma}_{n+1} = \lambda^{-1} \circ F\tilde{\gamma}_n$, so $F\tilde{\gamma}_n = \lambda \circ \gamma_{n+1}$), or, replacing λ by λ^{-1} and reversing the corresponding

arrow:



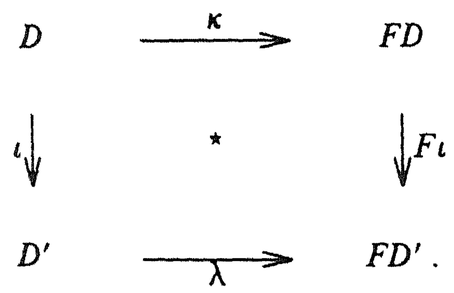
Substituting $\kappa \circ \gamma_{n+1}$ for $F\gamma_n$ (figure 1) yields:



or: $(\lambda^{-1} \circ F\iota \circ \kappa) \circ \gamma_{n+1} = \tilde{\gamma}_{n+1}$ (this equality also holds for γ_0 and $\tilde{\gamma}_0$). But according to figure 2, ι is the only arrow with: $\forall n \in \mathbb{N}\ [\iota \circ \gamma_n = \tilde{\gamma}_n]$. Thus

$$\iota = \lambda^{-1} \circ F\iota \circ \kappa ,$$

or, in other words:



This commutativity, together with the fact that κ and λ are isometries implies:

$$\delta(\iota) = \delta(F\iota) .$$

(For the definition of δ see definition 3.2.)
Now the proof can be concluded, following the train of thought indicated above:

$$\begin{aligned} \delta(\iota) &= \delta(F\iota) \\ &\leqslant \epsilon \cdot \delta(\iota), \end{aligned}$$

for some $0 \leqslant \epsilon < 1$, since F is a contraction. This implies

$$\delta(\iota) = 0 ,$$

so (if $\iota = <i, j>$)

$$i \circ j = id_{D'} .$$

At last we can draw the desired conclusion:

$$D \underset{\cong}{\overset{\iota}{\rightarrow}} D'.$$

Now we return again to our original category $\mathcal{C}$ of complete metric spaces and provide for, as promised above, another criterion for functors on $\mathcal{C}$, that, together with contractivity, will appear to be sufficient to ensure uniqueness of their fixed points.

DEFINITION 4.3 (Hom-contractivity)
We call a functor $F:\mathcal{C}\to\mathcal{C}$ *hom-contracting,* whenever

$$\forall P\in\mathcal{C}\,\forall Q\in\mathcal{C}\,\exists\epsilon<1\;[F_{P,Q}:(P\to^{\mathcal{C}}Q)\to^{\epsilon}(FP\to^{\mathcal{C}}FQ)]$$

where

$$P\to^{\mathcal{C}}Q=\{\iota\,|\,\iota:P\to Q\,|\,\iota \text{ is an arrow in } \mathcal{C}\},\quad F_{P,Q}(\iota)=F\iota$$

REMARKS
Because arrows in $\mathcal{C}$ are pairs, we have on $P\to^{\mathcal{C}}Q$ the standard metric for the Cartesian product. So let $\iota_1,\ \iota_2:P\to Q$, $\iota_1=<i_1,j_1>$ and $\iota_2=<i_2,j_2>$. Then their distance is defined by

$$d(\iota_1,\iota_2)=\max\{d_{P\to Q}(i_1,i_2),d_{Q\to P}(j_1,j_2)\}.$$

It is not the case that every hom-contracting functor is also contracting, which follows from the following example.
Let $A=\{0\}$ and $B=\{1,2\}$ be discrete metric spaces. We define a functor $F:\mathcal{C}\to\mathcal{C}$ as follows. For every complete metric space $P\in\mathcal{C}$ let

$$FP=\begin{cases}A & \text{if } P \text{ contains exactly 1 element}\\ B & \text{otherwise.}\end{cases}$$

For $\iota:P\to Q$ we define $F\iota$:

$$F\iota=\begin{cases}1_A & \text{if } FP=FQ=A\\ 1_B & \text{if } FP=FQ=B\\ \iota_0 & \text{if } FP=A \text{ and } FQ=B,\end{cases}$$

where $\iota_0=<i_0,j_0>$, with $i_0:0\mapsto 1$, $j_0:1,2\mapsto 0$. Note that there is no $\iota:P\to Q$ if $FP=B$ and $FQ=A$. It is not difficult to verify that F is a functor, which is hom-contracting. The following argument shows that it is *not* contracting. Let $C=\{3,4\}$ with $d(3,4)=\frac{1}{2}$, and let $\kappa:A\to C$, with $\kappa=<k,l>$ be defined by $k:0\mapsto 3$ and $l:3,4\mapsto 0$. Then we have $\delta(\kappa)=\frac{1}{2}$, but $F\kappa:FA\to FC$ is $\iota_0:A\to B$ (as defined above), for which $\delta(\iota_0)=1$.

THEOREM 4.4
Let F be a contracting and *hom-contracting functor $F:\mathcal{C}\to\mathcal{C}$. Then F has a unique fixed point up to isometry, that is to say: there exists a $D\in\mathcal{C}$ such that*

(1) $FD\cong D$ and

(2) $\forall D'\in\mathcal{C}\;[FD'\cong D'\Rightarrow D\cong D']$.

PROOF
The proof of this theorem differs from that of theorem 4.2 only in the definition of $\tilde{\gamma}_0$. There we could take for $\tilde{\gamma}_0$ the trivial embedding of D_0 into D', mapping p_0 onto the base-point of D'. Here we have no base-points. But we can use the fact that F is hom-contracting by taking for $\tilde{\gamma}_0$ the unique fixed point of the function $G:(D_0\to^{\mathcal{C}}D')\to(D_0\to^{\mathcal{C}}D')$, that we define by: $G(\tilde{\gamma})=\lambda^{-1}\circ F\tilde{\gamma}\circ\iota_0$, for

$\tilde{\gamma} \in (D_0 \to^{\mathcal{C}} D')$. (Note that G is contracting because F is hom-contracting.) It follows that $\tilde{\gamma}_0$, thus defined, satisfies $\lambda^{-1} \circ F\tilde{\gamma}_0 \circ \iota_0 = \tilde{\gamma}_0$, which serves our purposes.

5. A CLASS OF DOMAIN EQUATIONS WITH UNIQUE SOLUTIONS

In this section we present a class of domain equations over the category $\mathcal{C}$ that have unique solutions. For this purpose we first define a set *Func* of functors on $\mathcal{C}$ and formulate a condition for its elements that implies contractivity and hom-contractivity. It then follows that every domain equation over $\mathcal{C}$ induced by a functor that satisfies this condition, has a unique solution.

DEFINITION 5.1 (Functors)
The class *Func*, with typical elements F, is defined by:

$$F ::= F_M \mid id^{\epsilon} \mid F_1 \to F_2 \mid F_1 \to^{1} F_2 \mid F_1 \overline{\cup} F_2 \mid F_1 \times F_2 \mid \mathcal{P}_{cl}(F) \mid F_1 \circ F_2$$

where M is an arbitrary complete metric space and $\epsilon > 0$. Every $F \in Func$ is to be interpreted as a functor

$$F : \mathcal{C} \to \mathcal{C}$$

as follows. Let $(P, d_P), (Q, d_Q) \in \mathcal{C}$ be complete metric spaces. Let $P \to^{\iota} Q \in \mathcal{C}$, with $\iota = \langle i, j \rangle$. For the definition of each $F \in Func$ we have to specify:

(1) the image of P under F: FP,

(2) the image of d under F: Fd,

(3) the image of ι under F: $F\iota$ $(= \langle Fi, Fj \rangle)$.

(a) $F = F_M$:

(1) $FP = M$,

(2) $Fd = d_M$ (the metric of M),

(3) $F\iota = \langle id_M, id_M \rangle$.

We sometimes use just a set A instead of a metric space M. In this case we provide A with the discrete metric (definition 2.1).

(b) $F = id^{\epsilon}$:

(1) $FP = P$,

(2) $Fd = \lambda(x,y) \cdot \min(1, \epsilon \cdot d(x,y))$,

(3) $F\iota = \iota$.

Next we define functors that are composed. Let $F_1, F_2 \in Func$, such that

(1) $F_1 P = P_1$, $F_2 P = P_2$, $F_1 Q = Q_1$, $F_2 Q = Q_2$,

(2) $F_1 d = d_1$, $F_2 d = d_2$,

(3) $F_1 \iota = \langle i_1, j_1 \rangle$, $F_2 \iota = \langle i_2, j_2 \rangle$.

(c) $F = F_1 \to F_2$:

(1) $FP = P_1 \to P_2$,

(2) $Fd = d_F$ (see definition 2.6(a)),

(3) $F\iota = <\lambda f\cdot(i_2 \circ f \circ j_1),\ \lambda g\cdot(j_2 \circ g \circ i_1)>$.

($F = F_1 \rightarrow^1 F_2$ is defined similarly.)

(d) $F = F_1 \overline{\cup} F_2$:

(1) $FP = P_1 \overline{\cup} P_2$,

(2) $Fd = d_U$ (see definition 2.6(b)),

(3) $F\iota = <\lambda p\cdot$ if $p \in \{0\} \times P_1$ then $i_1((p)_2)$ else $i_2((p)_2)$ fi,

$\lambda q\cdot$ if $q \in \{0\} \times Q_1$ then $j_1((q)_2)$ else $j_2((q)_2)$ fi$>$.

(e) $F = F_1 \times F_2$:

(1) $FP = P_1 \times P_2$,

(2) $Fd = d_P$ (see definition 2.6(c)),

(3) $F\iota = <\lambda<p_1,p_2>\cdot<i_1(p_1),i_2(p_2)>,\lambda<q_1,q_2>\cdot<j_1(q_1),j_2(q_2)>>$.

(f) $F = \mathcal{P}_{cl}(F_1)$:

(1) $FP = \mathcal{P}_{cl}(P_1)$,

(2) $Fd = d_H$ (see definition 2.6(d)),

(3) $F\iota = <\lambda X\cdot\{i_1(x) \mid x \in X\},\lambda Y\cdot$ *closure* $\{j_1(y) \mid y \in Y\}>$.

(g) $F = F_1 \circ F_2$: the usual composition of functors on $\mathcal{C}$.

REMARK

The set *Func* contains elements of various form. We give an example. Let $F_1, F_2 \in Func$. The following functor is an element of the set *Func*, as can be deduced from its definition.

$$F_1 \rightarrow^A F_2 =^{def} id^A \circ (F_1 \rightarrow^1 (id^{\frac{1}{A}} \circ F_2)), \text{ for } A > 0.$$

LEMMA 5.2

For all $F \in Func$ we have: F is a well defined functor on $\mathcal{C}$.

PROOF

We treat only one case by way of example, being (lazy and) confident that it shows the reader how to proceed in the other cases.

Let $F = F_1 \rightarrow^1 F_2$, and suppose F_1 and F_2 are well defined. Let $(P,d_P),(Q,d_Q)$ and $P \rightarrow^\iota Q \in \mathcal{C}$, with $\iota = <i,j>$; furthermore, let for $k = 1,2$:

$$F_k P = P_k, \quad F_k Q = Q_k,$$

$$F_k d_P = d_{P_k}, \quad F_k d_Q = d_{Q_k},$$

$$F_k \iota = <i_k, j_k>.$$

The functor F is defined by

(1) $FP = P_1 \rightarrow^1 P_2$,

(2) $Fd_P = d_F$,

(3) $F\iota = \langle Fi,Fj\rangle = \langle \lambda f\cdot(i_2\circ f\circ j_1),\lambda g\cdot(j_2\circ g\circ i_1)\rangle$.

$$\begin{array}{ccccc} P & & & & P_1\to^1 P_2 \\ i \downarrow \uparrow & j & \xrightarrow[F_1\to^1 F_2]{F=} & \lambda f\cdot(i_2\circ f\circ j_1)=Fi & \downarrow \uparrow \quad Fj=\lambda g\cdot(j_2\circ g\circ i_1) \\ Q & & & & Q_1\to^1 Q_2 \end{array}$$

It follows from proposition 2.7, that $(P_1\to^1 P_2,d_F)$ is a complete metric space, which leaves us to prove:

(a) Fi is isometric,

(b) Fj is NDI and

(c) $Fj\circ Fi = id_{FP}$.

Part (a): Let $f_1,f_2\in P_1\to^1 P_2$. We want to show

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

We have

$$\begin{aligned} \sup_{q\in Q_1}\{d_{Q_2}(i_2\circ f_1\circ j_1(q),\, i_2\circ f_2\circ j_1(q))\} &= \text{[because } i_2 \text{ is isometric]} \\ &\quad \sup_{q\in Q_1}\{d_{P_2}(f_1\circ j_1(q),f_2\circ j_1(q))\} \\ &= \text{[because } j_1 \text{ is surjective]} \\ &\quad \sup_{p\in P_1}\{d_{P_2}(f_1(p),\, f_2(p))\} \\ &= d_{P_1\to P_2}(f_1,f_2). \end{aligned}$$

Part (b): Let $g_1,g_2\in Q_1\to^1 Q_2$. We want to show:

$$d_{FP}(Fj(g_1),Fj(g_2)) \leqslant d_{FQ}(g_1,g_2).$$

Let $p\in P_1$; we have:

$$\begin{aligned} d_{P_2}(Fj(g_1)(p),Fj(g_2)(p)) &= d_{P_2}(j_2\circ g_1\circ i_1(p),j_2\circ g_2\circ i_1(p)) \\ &\leqslant \text{[}j_2 \text{ is NDI]} \\ &\quad d_{Q_2}(g_1\circ i_1(p),g_2\circ i_1(p)) \\ &\leqslant d_{FQ}(g_1,g_2). \end{aligned}$$

Part (c): Let $f\in P_1\to^1 P_2$. We have

$$\begin{aligned} Fj\circ Fi(f) &= j_2\circ i_2\circ f\circ j_1\circ i_1 \\ &= f. \end{aligned}$$

DEFINITION 5.3 (Contraction coefficient)
For each $F\in Func$ we define its so-called *contraction* coefficient (notation: $c(F)$, with $c(F)\in[0,\infty]$), using induction on the complexity of the structure of F.

(*a*) If $F=F_M$, then $c(F)=0$.

(*b*) If $F=id^\epsilon$, then $c(F)=\epsilon$.

Let F_1, $F_2\in Func$, with coefficients $c(F_1)$ and $c(F_2)$. Then we set:

(*c*) If $F = F_1 \to F_2$, then $c(F) = \max\{\infty \cdot c(F_1),\ c(F_2)\}$.

(*d*) If $F = F_1 \to^1 F_2$, then $c(F) = c(F_1) + c(F_2)$.

(If we would restrict ourselves to ultra-metric spaces, we could write $\max\{c(F_1), c(F_2)\}$ here.)

(*e*) If $F = F_1 \overline{\cup} F_2$, then $c(F) = \max\{c(F_1), c(F_2)\}$.

(*f*) If $F = F_1 \times F_2$, then $c(F) = \max\{c(F_1), c(F_2)\}$.

(*g*) If $F = \mathcal{P}_{cl}(F_1)$, then $c(F) = c(F_1)$.

(*h*) If $F = F_1 \circ F_2$, then $c(F) = c(F_1) \cdot c(F_2)$.

(With ∞ we compute as follows: $\infty \cdot 0 = 0 \cdot \infty = 0$, $\infty \cdot c = c \cdot \infty = \infty$, if $c > 0$.)

THEOREM 5.4
For every functor $F \in Func$ we have

$$(1)\ \forall P \to' Q \in \mathcal{C}\,[\delta(F\iota) \leqslant c(F) \cdot \delta(\iota)],$$

$$(2)\ \forall P, Q \in \mathcal{C}\,[F_{P,Q} : (P \to^{\mathcal{C}} Q) \to^{c(F)} (FP \to^{\mathcal{C}} FQ)].$$

PROOF
Let $P, Q \in \mathcal{C}$, $\iota, \iota' \in P \to^{\mathcal{C}} Q$, with $\iota = \langle i, j \rangle, \iota' = \langle i', j' \rangle$.
Case (a) $F = F_M$:
Part (a1)

$$\begin{aligned} \delta(F\iota) &= d_{FQ \to FQ}(Fi \circ Fj, id_M) \\ &= d_{FQ \to FQ}(id_M \circ id_M, id_M) \\ &= 0 = c(F) \cdot \delta(\iota). \end{aligned}$$

part (a2)

$$d_{FP \to^{\mathcal{C}} FQ}(F\iota, F\iota') = d_{M \to^{\mathcal{C}} M}(id, id) = 0 = c(F) \cdot d_{P \to^{\mathcal{C}} Q}(\iota, \iota').$$

Case (b) $F = id^{\epsilon}$:
part (b1)

$$\begin{aligned} \delta(F\iota) &= d_{FQ \to FQ}(Fi \circ Fj, id_{FQ}) \\ &= \sup_{q \in Q}\{d_{FQ}(i \circ j(q), q)\} \\ &= \sup_{q \in Q}\{\epsilon \cdot d_Q(i \circ j(q), q)\} \\ &= \epsilon \cdot \delta(\iota) \\ &= c(F) \cdot \delta(\iota). \end{aligned}$$

Part (b2)

$$\begin{aligned} d_{FP \to^{\mathcal{C}} FQ}(F\iota, \iota') &= \epsilon \cdot d_{P \to^{\mathcal{C}} Q}(\iota, \iota') \\ &= c(F) \cdot d_{P \to^{\mathcal{C}} Q}(\iota, \iota'). \end{aligned}$$

Now let $F_1, F_2 \in$ Func and suppose the theorem holds for these functors. For $k = 1,2$ we use the following notation:

$$\begin{aligned} F_k\iota &= \iota_k\,, \quad F_k\iota' = \iota'_k\,, \quad F_kP = P_k\,, \quad F_kQ = Q_k\,, \\ F_ki &= i_k\,, \quad F_ki' = i'_k\,, \\ F_kj &= j_k\,, \quad F_kj' = j'_k\,, \end{aligned}$$

We only treat the cases that $F = F_1 \to^1 F_2$ and $F = F_1 \times F_2$.
Case (d) $F = F_1 \to^1 F_2$:
Part (d1)

$$\begin{aligned}\delta(F\iota) &= d_{FQ \to FQ}(Fi \circ Fj, id_{FQ}) \\ &= \sup_{g \in FQ}\{d_{FQ}(i_2 \circ j_2 \circ g \circ i_1 \circ j_1, g)\}.\end{aligned}$$

Let $g \in FQ = Q_1 \to^1 Q_2$. For $q_1 \in Q_1$ we have

$$\begin{aligned}d_{Q_2}(i_2 \circ j_2 \circ g \circ i_1 \circ j_1(q_1), g(q_1)) \leqslant\ & d_{Q_2}(i_2 \circ j_2 \circ g \circ i_1 \circ j_1(q_1), g \circ i_1 \circ j_1(q_1)) + \\ & d_{Q_2}(g \circ i_1 \circ j_1(q_1), g(q_1)).\end{aligned}$$

(This "+" could be replaced by "max" in the case of ultra-metric spaces.)
For the first term we have

$$\begin{aligned}d_{Q_2}(i_2 \circ j_2 \circ g \circ i_1 \circ j_1(q_1), g \circ i_1 \circ j_1(q_1)) &\leqslant \sup_{q_2 \in Q_2}\{d_{Q_2}(i_2 \circ j_2(q_2), q_2)\} \\ &= \delta(F_2\iota).\end{aligned}$$

For the second

$$\begin{aligned}d_{Q_2}(g \circ i_1 \circ j_1(q_1), g(q_1)) &\leqslant [\text{because } g \in Q_1 \to^1 Q_2] \\ & \quad d_{Q_1}(i_1 \circ j_1(q_1), q_1) \\ &= \delta(F_1\iota).\end{aligned}$$

We see

$$\begin{aligned}\delta(F\iota) &\leqslant \delta(F_1\iota) + \delta(F_2\iota) \\ &\leqslant [\text{induction}] \\ &\quad (c(F_1) + c(F_2)) \cdot \delta(\iota) \\ &= c(F) \cdot \delta(\iota).\end{aligned}$$

Part (d2)

$$d_{FP \to^e FQ}(F\iota, F\iota') = \max\{d_{FP \to FQ}(Fi, Fi'), d_{FQ \to FP}(Fj, Fj')\}.$$

For the first component, we have

$$d_{FP \to FQ}(Fi, Fi') = \sup_{f \in FP, q \in Q_1}\{d_{Q_2}(Fi(f)(q), Fi'(f)(q))\}.$$

Let $f \in FP, q \in Q_1$. Then

$$\begin{aligned}d_{Q_2}(Fi(f)(q), Fi'(f)(q)) &= d_{Q_2}(i_2 \circ f \circ j_1(q), i_2' \circ f \circ j_1'(q)) \\ &\leqslant d_{Q_2}(i_2 \circ f \circ j_1(q), i_2' \circ f \circ j_1(q)) + d_{Q_2}(i_2' \circ f \circ j_1(q), i_2' \circ f \circ j_1'(q)) \\ &\leqslant d_{P_2 \to Q_2}(i_2, i_2') + d_{Q_2}(i_2' \circ f \circ j_1(q), i_2' \circ f \circ j_1'(q)) \\ &\leqslant [\text{because } i_2' \text{ is isometric }, f \in P_1 \to^1 P_2] \\ &\quad d_{P_2 \to Q_2}(i_2, i_2') + d_{Q_1 \to P_1}(j_1, j_1').\end{aligned}$$

(Again, in the case of ultra-metric spaces, we would have "max" here.)
Likewise, we have for the second component

$$d_{FQ \to FQ}(Fj, Fj') \leqslant d_{P_1 \to Q_1}(i_1, i_1') + d_{Q_2 \to P_2}(j_2, j_2').$$

Together this implies

$$d_{FP\to^{\mathcal{C}}FQ}(F\iota,F\iota') \leqslant d_{P_1\to^{\mathcal{C}}Q_1}(F_1\iota,F_1\iota') + d_{P_2\to^{\mathcal{C}}Q_2}(F_2\iota,F_2\iota')$$
$$\leqslant \text{[induction]}$$
$$(c(F_1)+c(F_2))\cdot d_{P\to^{\mathcal{C}}Q}(\iota,\iota')$$
$$= c(F)\cdot d_{P\to^{\mathcal{C}}Q}(\iota,\iota').$$

Case (f) $F = F_1\times F_2$:
Part (f1)

$$\delta(F\iota) = d_{FQ\to FQ}(Fi\circ Fj, id_{FQ})$$
$$= \sup_{\overline{q}\in FQ}\{d_{FQ}(Fi\circ Fj(\overline{q}),\overline{q})\}$$
$$= \sup_{<q_1,q_2>\in FQ}\{d_{FQ}(<i_1\circ j_1(q_1),i_2\circ j_2(q_2)>,<q_1,q_2>)\}$$
$$= \sup_{<q_1,q_2>\in FQ}\{\max\{d_{Q_1}(i_1\circ j_1(q_1),q_1),d_{Q_2}(i_2\circ j_2(q_2),q_2)\}\}$$
$$= \max\{\sup_{q_1\in Q_1}\{d_{Q_1}(i_1\circ j_1(q_1),q_1)\},\sup_{q_2\in Q_2}\{d_{Q_2}(i_2\circ j_2(q_2),q_2)\}\}$$
$$= \max\{\delta(F_1\iota),\delta(F_2\iota)\}$$
$$\leqslant \text{[induction]}$$
$$(c(F_1)+c(F_2))\cdot\delta(\iota)$$
$$= c(F)\cdot\delta(\iota).$$

Part (f2)

$$d_{FP\to^{\mathcal{C}}FQ}(Fi,Fi') = \sup_{\overline{p}\in FP}\{d_{FQ}(Fi(\overline{p}),Fi'(\overline{p}))\}$$
$$= \sup_{<p_1,p_2>\in FP}\{d_{FQ}(<i_1(p_1),i_2(p_1)>,<i_1'(p_2),i_2'(p_2)>)\}$$
$$= \max\{\sup_{p_1\in P_1}\{d_{Q_1}(i_1(p_1),i_1'(p_1))\},\sup_{p_2\in P_2}\{d_{Q_2}(i_2(p_2),i_2'(p_2))\}\}$$
$$= \max\{d_{P_1\to Q_1}(i_1,i_1'),d_{P_2\to Q_2}(i_2,i_2')\}.$$

Similarly, we have

$$d_{FQ\to FP}(Fj,Fj') = \max\{d_{Q_1\to P_1}(j_1,j_1'),d_{Q_2\to P_2}(j_2,j_2')\}.$$

Thus we obtain

$$d_{FP\to^{\mathcal{C}}FQ}(F\iota,F\iota') = \max\{d_{P_1\to^{\mathcal{C}}Q_1}(F_1\iota,F_1\iota'),d_{Q_2\to^{\mathcal{C}}P_2}(F_2\iota,F_2\iota')\}$$
$$\leqslant \text{[induction]}$$
$$\max\{c(F_1),c(F_2)\}\cdot d_{P\to^{\mathcal{C}}Q}(\iota,\iota')$$
$$= c(F)\cdot d_{P\to^{\mathcal{C}}Q}(\iota,\iota')\cdot$$

COROLLARY 5.5
For every $F\in Func$, with $0\leqslant c(F)<1$, we have

(1) *F is a contracting functor, and*

(2) *F is a hom-contracting functor.*

COROLLARY 5.6
Every reflexive domain equation over $\mathcal{C}$ of the form

$$P \cong FP,$$

for which $F \in Func$ *and* $c(F) < 1$, *has a unique solution (up to isomorphism).*

6. Conclusions

We have presented a technique for constructing fixed points of certain functors over a category of complete metric spaces. This enables us to solve the reflexive domain equations associated with these functors. The technique is an adaptation of the limit construction that was first used in the context of certain partial orders (continuous lattices, complete lattices, complete partial orders). Nevertheless, we have encountered some nice metric phenomena in our metric framework. To begin with, the concept of a converging tower is an analogue to the concept of a Cauchy sequence in a complete metric space, and indeed, both have a limit. Furthermore, a contracting functor on our category of metric spaces is a concept analogous to that of a contracting function on a complete metric space, and both are guaranteed to have a fixed point. If we strengthen our requirements on the functor to include hom-contractivity (also analogous to contractivity of a function), we even know that the fixed point is unique (as is the case with a contracting function). Therefore the whole situation looks very much like Banach's theorem in a category-theoretic disguise.

A few questions remain open, however. We are still looking for a functor that is contracting but not hom-contracting, or even better for a functor that is contracting but has several non-isomorphic fixed points. Another point is what can be said about functors where the argument occurs at the left hand side of a general function space construction (*all* continuous functions, not just the NDI ones).

In any case, the class of functors (and, thus, domain equations) that we can handle is large enough, so that our technique is a useful tool in the construction of domains for the denotational semantics of concurrent programming languages.

Related Work

The subject of solving reflexive domain equations is not new. Various solutions of the kind of equations mentioned above already exist. We shall not try to give an extensive and complete bibliography on this matter and confine ourselves to the following remarks.
We mention the work of Scott ([Sc]), who uses inverse limit constructions for solving domain equations. Our method of generalizing metric notions in terms of category-theoretical notions shows a clear analogy to the work D. Lehmann ([Le]) did in the context of partial orderings. In fact, there is a clear similarity between the metric and the order-theoretic cases: Both are based on theorem 3.14 and in both cases the main part of the work is showing that the premisses of this theorem are satisfied. Of course, the details of these proofs are quite different. It is interesting to notice that in the order-theoretic case one can often prove that there is an *initial* fixed point of the functor: a fixed point that can be embedded in every other fixed point (see, e.g., [SP]), whereas in the metric case we can prove the existence of a *unique* fixed point (up to isomorphism). This is a nice parallel to what happens at the elementary level: in order theory one can prove that certain functions have a *least* fixed point, whereas in complete metric spaces we have *unique* fixed points of contracting functions.
Our work is also related to the general method of solving reflexive equations of Smyth and Plotkin ([SP]). In the terminology used there, we show that our category $\mathcal{C}$ is ω-complete in the limited sense, that all converging towers have direct limits. Further we show that a certain type of ω-continuous functors (called contracting) has a fixed point. (Without having investigated the precise relationship, we also mention here the anology between their notion of an O-category, and the fact that in our

category $\mathcal{C}$ the hom-sets are complete metric spaces.)

7. References

[ABKR] P. America, J. de Bakker, J. Kok, J. Rutten, *A Denotational Semantics of a Parallel Object-Oriented Language,* Technical Report (CS-R8626), Centre for Mathematics and Computer Science, Amsterdam, 1986. (To appear in Information and Computation.)

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

[Du] J. Dugundji, *Topology,* Allyn and Bacon, inc., Boston, 1966.

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

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

[Le] D. Lehmann, *Categories for Mathematical Semantics,* in: Proc. 17th IEEE Symposium on Foundations of Computer Science, 1976.

[ML] S. Mac Lane, *Categories for the Working Mathematician,* Graduate Texts in Mathematics 5, Springer-Verlag, 1971.

[Sc] D.S. Scott, *Continuous Lattices,* in: Toposes, Algebraic Geometry and Logic, Lecture Notes in Mathematics 274, Springer-Verlag, 1972, pp. 97-136.

[SP] M.B. Smyth, G.D. Plotkin, *The Category-Theoretic Solution of Recursive Domain Equations,* SIAM J. Comput, Vol. 11, No. 4, 1982, pp.761-783.

The following paper will appear in *Information and Computation* and is included in this thesis with kind permission of Academic Press, Inc..

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

1. Introduction

In this paper we give a formal semantics of a language called POOL (Parallel Object-Oriented Language). It is a syntactically simplified version of the language POOL-T, which is defined in (America, 1985) and for which (America, 1986) and (America, 1987) give an account of the design considerations. POOL-T was designed in subproject A of ESPRIT project 415 with the purpose of programming a highly parallel machine which is also being developed in this project (see (Odijk, 1987) for an overview). The language provides all the facilities needed to program reasonably large

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

parallel systems and many small and several large applications have been written in it.

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

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

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

An important choice we have made is to use the mathematical framework of *complete metric spaces* for our semantic description. In this we follow and generalize the approach of (De Bakker and Zucker, 1982). (For other applications of this type of semantic framework see (De Bakker et al., 1986).) First, we construct a suitable domain P of *processes*, which is a set of mathematical objects that will be used as meanings. It will satisfy a reflexive domain equation, which will be solved by deriving from it a functor on a certain category of complete metric spaces and then constructing a fixed point for this functor. The mathematical techniques to do this are sketched in section 2 and presented in detail in (America and Rutten, 1988). They are not necessary for an understanding of the rest of the paper.

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

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

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

where

$$Cont_E = Obj \rightarrow P,$$

$$Cont_S = P.$$

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

The environments (elements of the set *Env*) are used to store the meanings of declarations (of classes and methods). With the help of $[\![\cdots]\!]_E$ and $[\![\cdots]\!]_S$ we can define for each unit U a suitable environment γ_U, which contains the meanings of the classes and methods as they are defined in U. It will be constructed as the unique fixed point of a contracting operator on the complete metric space

of environments. The semantic domain *Obj* stands for the set of object names. Its appearance in the defining equations reflects the fact that in POOL each expression or statement is evaluated *by a certain object.* Finally, a continuation will be given as an argument to the semantic functions. This describes what will happen *after* the execution of the current expression or statement. As the continuation of an expression generally depends upon the result of this expression (an object name), its type is $Obj \rightarrow P$, whereas the type of continuations of statements is simply P. This use of continuations makes it possible to define the semantics, especially of object creation, in a convenient and concise way. (For more examples of the use of continuations in semantics, see (De Bruin, 1986) and (Gordon, 1979).)

The denotational semantics proper for POOL is presented in section 4. It first discusses the details of the process domain P. Next, it defines an auxiliary operator for parallel composition, which is used, e.g., in the equation for the creation of a new object. (POOL itself does not have a syntactic operator for parallel execution: parallelism occurs implicitly as a consequence of object creation.) Then the core of the semantic definitions, in terms of the various semantic equations for the respective classes of expressions and statements, is displayed. Once the reader has understood (or taken for granted) the underlying mathematical foundations he will appreciate, we hope, that the framework allows a concise, rigorous, and compositional (the touchstone of a denotational model) definition of intricate notions such as the creation of a new object or the passing of messages leading to the invocation of the appropriate method. Section 4 then continues with the discussion of the standard process p_{ST}, which describes the standard objects (integers, booleans, and *nil*) of the language. Next, the definition of the environment γ_U corresponding to a unit U is given and used to define a process p_U. In a last step we show how the set of all possible sequences of computation steps can be obtained from the process resulting from the parallel composition of p_U and p_{ST}.

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

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

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

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

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 (America and Rutten, 1988) had

not yet been conceived, and continuations did not fit into the process domain). We also wish to thank the following persons for their contribution to the discussions of many of the preliminary ideas on which this report is based: Frank de Boer, Anton Eliëns, Hans Jonkers, Frank van der Linden, John-Jules Meyer, Marly Roncken, and Erik de Vink. Finally we are grateful to the anonymous referees, whose comments on an earlier version of this paper have led to considerable improvements.

2. Metric spaces and domain equations

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

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

2.1. Metric spaces

Definition 2.1 (Metric space)
A *metric space* is a pair (M,d) with M a non-empty set and d a mapping $d:M\times M\rightarrow[0,1]$ (a *metric* or *distance*), 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') $\forall x,y,z\in M\,[d(x,y)\leqslant \max\{d(x,z),d(z,y)\}]$.

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

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

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

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

Definition 2.2
Let (M,d) be a metric space, let $(x_i)_i$ be a sequence in M.
(a) We say that $(x_i)_i$ is a *Cauchy sequence* whenever we have:
$\forall\epsilon>0\ \exists N\in\mathbb{N}\ \forall n,m>N\ [d(x_n,x_m)<\epsilon]$.
(b) Let $x\in M$. We say that $(x_i)_i$ *converges to* x (denoted by $x=\lim_{i\rightarrow\infty}x_i$) 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.
(d) A subset X of a metric space (M,d) is called closed whenever each Cauchy sequence in X converges to an element of X.

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

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

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

(c) Let $\epsilon\geqslant 0$. With $M_1\to^{\epsilon} M_2$ we denote the set of functions f from M_1 to M_2, that satisfy the following property: $\forall x,y\in M_1\ [d_2(f(x),f(y))\leqslant\epsilon\cdot d_1(x,y)]$.
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 *Let $(M_1,d_1),(M_2,d_2)$ be metric spaces. For every $\epsilon\geqslant 0$ and $f\in M_1\to^{\epsilon} M_2$ we have: f is continuous.*

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

(1) $f(x)=x$ *(x is a fixed point of f),*
(2) $\forall y\in M\ [f(y)=y\Rightarrow y=x]$ *(x is unique),*
(3) $\forall x_0\in M\ [\lim_{n\to\infty} f^n(x_0)=x]$, *where $f^{n+1}(x_0)=f(f^n(x_0))$ and $f^0(x_0)=x_0$.*

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

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

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

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

This supremum always exists since the codomain of our metrics is always $[0,1]$. For $\epsilon\geqslant 0$ the set $M_1\to^{\epsilon} M_2$ is a subset of $M_1\to M_2$, and a metric on $M_1\to^{\epsilon} M_2$ can be obtained by taking the restriction of the corresponding d_F.

(b) With $M_1\dot\cup\cdots\dot\cup M_n$ 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 $M_1\dot\cup\cdots\dot\cup M_n$ as follows: For every $x,y\in M_1\dot\cup\cdots\dot\cup M_n$,

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

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

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

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

(d) 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 $\sup\varnothing=0$ and $\inf\varnothing=1$.)

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

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

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

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

(a) $(M_1 \to M_2, d_F)$, $(M_1 \to^\epsilon M_2, d_F)$,

(b) $(M_1 \overline{\cup} \cdots \overline{\cup} M_n, d_U)$,

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

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

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

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

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

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:

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

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

2.2. Solving reflexive domain equations
We shall use as a mathematical domain for our denotational semantics a complete metric space satisfying a so-called *reflexive domain equation* of the following form:

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

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

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

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

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

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

Roughly, their approach amounts to the following: In order to solve $P \cong F(P)$ they define a sequence of complete metric spaces $(P_n)_n$ by: $P_0 = A$ and $P_{n+1} = F(P_n)$, for $n > 0$, such that $P_0 \subseteq P_1 \subseteq \cdots$. Then they take the *metric completion* of the union of these spaces P_n, say $\overline{P}$, and show: $\overline{P} \cong F(\overline{P})$. In this way they are able to solve the equations (1), (2), and (3) above.

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

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

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

In (America and Rutten, 1988) the approach is generalized in order to overcome this problem. The family of complete metric spaces is made into a *category* $\mathcal{C}$ by providing some additional structure. (For an extensive introduction to category theory we refer the reader to (Mac Lane, 1971).) Then the expression F is interpreted as a *functor* $F: \mathcal{C} \to \mathcal{C}$ which is (in a sense) *contracting*. It is proved that a generalized version of Banach's theorem holds, i.e., that contracting functors have a fixed point (up to isometry). Such a fixed point, satisfying $P \cong F(P)$, is a solution of the domain equation.

We shall now give a quick overview of these results, omitting many details and all proofs. For a full treatment we refer the reader to (America and Rutten, 1988).

DEFINITION 2.9 (Category of complete metric spaces)
Let $\mathcal{C}$ denote the category that has complete metric spaces for its objects. The arrows ι in $\mathcal{C}$ are defined as follows: Let M_1, M_2 be complete metric spaces. Then $M_1 \to^{\iota} M_2$ denotes a pair of maps $M_1 \underset{j}{\overset{i}{\rightleftarrows}} M_2$, 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 $\langle i,j \rangle$ for ι.) Composition of the arrows is defined in the obvious way.

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

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

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

This number can be regarded as a measure of the quality with which M_2 is approximated by M_1: the smaller $\delta(\iota)$, the better M_2 is approximated by M_1.
Increasing sequences of metric spaces are generalized in the following

DEFINITION 2.11 (Converging tower)
(a) We call a sequence $(D_n, \iota_n)_n$ of complete metric spaces and arrows a *tower* whenever we have that $\forall n \in \mathbb{N}\ [D_n \to^{\iota_n} D_{n+1} \in \mathcal{C}]$.
(b) The sequence $(D_n, \iota_n)_n$ is called a *converging tower* when furthermore the following condition is satisfied:

$$\forall \epsilon > 0\ \exists N \in \mathbb{N}\ \forall m > n \geq N\ [\delta(\iota_{nm}) < \epsilon], \quad \text{where } \iota_{nm} = \iota_{m-1} \circ \cdots \circ \iota_n \colon D_n \to D_m.$$

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

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

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

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

DEFINITION 2.12 (Cone)
Let $(D_n, \iota_n)_n$ be a tower. Let D be a complete metric space and $(\gamma_n)_n$ a sequence of arrows. We call $(D, (\gamma_n)_n)$ a *cone* for $(D_n, \iota_n)_n$ whenever the following condition holds:

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

DEFINITION 2.13 (Initial cone)
A cone $(D, (\gamma_n)_n)$ for a tower $(D_n, \iota_n)_n$ is called *initial* whenever for every other cone $(D', (\gamma'_n)_n)$ for $(D_n, \iota_n)_n$ there exists a unique arrow $\iota : D \rightarrow D'$ in $\mathcal{C}$ such that:

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

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

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

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

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

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

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

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

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

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

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

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

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

REMARK: In (America and Rutten, 1988) it is shown that contracting functors that are moreover contracting on all *hom-sets* (the sets of arrows in $\mathcal{C}$ between any two given complete metric spaces) have *unique* fixed points (up to isometry). It is also possible to impose certain restrictions upon the category $\mathcal{C}$ such that every contracting functor on $\mathcal{C}$ has a unique fixed point.

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

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

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

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

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

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

(Please note that the definitions of these functors specify, for each metric space (P,d_P), the metric on $F(P)$ *implicitly* (see definition 2.6).) Now it is easily verified that F_1, F_2, F_3, and F_4 are contracting functors on $\mathcal{C}$. Intuitively, this is a consequence of the fact that in the definitions above each occurrence of P is preceded by a factor $id_{1/2}$. Thus these functors have a fixed point, according to theorem 2.18, which is a solution for the corresponding equation.

REMARKS
(1) In (America and Rutten, 1988) it is shown that functors like F_1 through F_4 are also contracting on hom-sets, which guarantees that they have *unique* fixed points (up to isometry).
(2) The results above hold for complete *ultra-metric* spaces too, which can be easily verified. The domain we shall use for our denotational semantics is an ultra-metric space.

3. THE LANGUAGE POOL

3.1 An informal introduction to the language

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

approach quite different from ours.)

A POOL program describes the behaviour of a whole system in terms of its constituents, *objects*. Objects contain some internal data, and some procedures that act on these data (these are called *methods* in the object-oriented jargon). Objects are entities of a dynamic nature: they can be created dynamically, their internal data can be modified, and they have an internal activity of their own. At the same time they are units of protection: the internal data of one object are not directly accessible for other objects.

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

Objects can interact by sending *messages* to each other. A message is a request for the receiver to execute a certain method. Messages are sent and received explicitly. In sending a message, the sender mentions the destination object, the method to be executed, and possibly some parameters (which are again object names) to be passed to this method. After this its activity is suspended. The receiver can specify the set of methods that will be accepted, but it can place no restrictions on the identity of the sender or on the parameters of messages. If a message arrives specifying an appropriate method, the method is executed with the parameters contained in the message. Upon termination, this method delivers a result (an object name), which is returned to the sender of the message. The latter then resumes its own execution. Note that this form of communication strongly resembles the rendez-vous mechanism of Ada (ANSI, 1983).

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

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

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

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

There is a special standard object, *nil*, which can be considered to be an element of every class. Upon the creation of a new object, its instance variables are initialized to *nil*, and when a method is invoked, its temporary variables are also initialized to *nil*. If a message is sent to this object, an error occurs. In general, whenever a run-time error occurs, the whole system will halt immediately.

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

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

3.2 Syntax of POOL

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

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

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

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

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

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

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

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

s ::= $x \leftarrow e$
| $u \leftarrow e$
| **answer** V ($V \subseteq MName,\ V \neq \varnothing$)
| e
| s_1 ; s_2
| **if** e **then** s_1 **else** s_2 **fi**
| **do** e **then** s **od**
| **sel** g_1 **or** $\cdots$ **or** g_n **les**

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

g ::= e **answer** V **then** s ($V \subseteq MName$).

(Note that $V = \varnothing$ is allowed.)

The set *Unit* of units, with typical elements U, . . . :

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

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

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

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

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

3.2.1 Informal explanation

Expressions

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

The next kind of expression is a send expression. Here, e is the destination object, to which the message will be sent, m is the method to be invoked, and e_1 through e_n are the parameters. When a send expression is evaluated, first the destination expression is evaluated, then the parameters are evaluated from left to right and then the message is sent to the destination object. When this object answers the message, the corresponding method is executed, that is, the formal parameters are initialized to the objects names in the message, the temporary variables are initialized to *nil*, and the expression in the method definition is evaluated. The value which results from this evaluation is sent back to the sender of the message and this will be the value of the send expression.

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

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

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

An expression may also be preceded by a statement. In this case the statement is executed before the expression is evaluated.

The expression **self** always results in the name of the object that is executing this expression.

The evaluation of a standard object ϕ results in that object itself. For instance, the value of the expression 23 will be the natural number 23.

Statements

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

The next kind of statement is an answer statement. This indicates that a message is to be answered. The object executing the answer statement waits until a message arrives with a method name that is contained in the set V. Then it executes the method (after initializing the formal parameters and temporary variables). The result of the method is sent back to the sender of the message, and the answer statement terminates.

Next it is indicated that any expression may also occur as a statement. Upon execution, the expression is evaluated and the result is discarded. So only the side effects of the expression evaluation (e.g., the sending of a message) are important.

Sequential composition, conditionals and loops have the usual meaning.

A select statement is executed as follows: First all the expressions (called guards) in the guarded commands are evaluated from left to right. They must all result in an object of class Boolean, otherwise an error occurs and the system is halted immediately. The guarded commands of which the

guards have resulted in *ff* are discarded (they do not play a role in the further execution of the statement). Now one of the remaining guarded commands is chosen. For this there are two possibilities: One possibility is that the (textually) *first* guarded command is chosen in which the answer statement contains no method names (if there such a guarded command). In this case the statement after **then** is executed and the select statement terminates. The second possibility is that a guarded command with a nonempty answer set is chosen. For this the following requirements must be satisfied:
- A message has arrived specifying a method in this answer set.
- This guarded command must be the (textually) *first* one that contains this method in its answer set and for which the guard resulted in *tt*.
- There must be no guarded command with an empty answer set and a true guard occurring before this one.

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

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

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

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

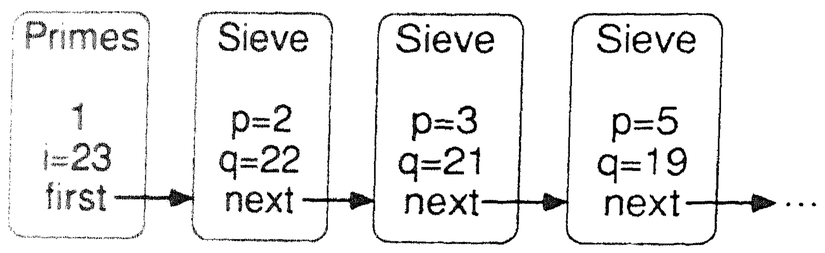
Method definitions
A method definition describes a method. Here u_1 through u_n are the formal parameters and e is the expression to be evaluated when the method is invoked. The set of temporary variables is again implicit: it consists of all the elements of *TVar* that occur in the expression e, with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

3.2.2 Context conditions
For a POOL program to be valid there are a few more conditions to be satisfied. We assume in the semantic treatment that the underlying program is valid.
These context conditions are the following:
- All class names in a unit are different.
- All method names in a class definition are different.
- All parameters in a method definition are different.

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

3.3 An example program

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



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

where

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

with

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

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

$s_{\text{Sieve}} =$

```
answer(input);
p←q;
next←create();
do tt
then answer(input);
     if q!mod(p)!equal(0)!not()
     then next!input(q)
     fi
od.
```

and

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

with

$s_{\text{Primes}} =$

```
first←new(Sieve);
i←2;
do tt
then first!input(i); i←i!add(1)
od
```

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

4. DENOTATIONAL SEMANTICS

This section constitutes the heart of our paper. First, the sets of objects and states are introduced and the mathematical domain P of processes is defined which we use for our denotational semantics. Secondly, an auxiliary semantic operator for parallel composition is defined, followed by the definition of environments. Then the semantics of expressions and statements is defined, with the use of the notion of *continuations*, some familiarity with which may be helpful for the reader. (For an extensive treatment of continuations and so-called expression continuations, which we shall also use, we refer to (Gordon, 1979).) Next, the semantics for the standard objects (integers and booleans) of POOL is given. The section culminates in the definition of the semantics of a unit (a POOL program). This involves in particular the definition of the environment corresponding to it. Finally, the notions of *paths* and *yield* of a process are introduced.

4.1 Domain definitions

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

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

$$\tau: AObj \rightarrow CName,$$

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

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

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

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

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

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

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

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

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

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

REMARKS

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

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

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

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

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

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

with

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

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

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

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

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

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

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

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

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

$$\pi = <\alpha, m, \overline{\beta}, f, p>,$$

with $\alpha \in Obj, m \in MName, \overline{\beta} \in Obj^*$, $f \in (Obj \to P)$, and $p \in P$. The action involved here consists of the registration of an attempt at communication, in which a message is sent to the object α, specifying the method m, together with the parameters $\overline{\beta}$. This is the interpretation of the first three components α, m, and $\overline{\beta}$. The fourth component f, called the *dependent* resumption of this send step, indicates the steps that will be taken after the sender has received the result of the message. These actions will depend on the result, which is modeled by f being a function that yields a process when it is applied to an object name (the result of the message). The last component p, called the *independent*

resumption of this send step, represents the steps to be taken after this send step that need *not* wait for the result of the method execution.

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

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

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

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

DEFINITION 4.4 (Parallel composition)
Let

$$\| : P \times P \rightarrow P$$

be such that it satisfies the following equation:

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

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

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

$$<\alpha,m,\bar{\beta},f,p> \hat{\|} q = <\alpha,m,\bar{\beta},f,p \| q>, \text{ and}$$

$$<\alpha,m,g> \hat{\|} q = <\alpha,m,\lambda\bar{\beta}\cdot\lambda h\cdot(g(\bar{\beta})(h) \| q)>,$$

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

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

REMARKS

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

(2) Since we intend to model parallel composition by interleaving, the merge of two processes p and q consists of three parts. The first part contains all possible first steps of p followed by the parallel composition of their respective resumptions with q. The second part 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. For $\pi \in Step_P$ the definition of $\pi \| q$ is straightforward. The definition of $\pi|_\sigma \rho$ is more involved. It is the empty set if π and ρ do not match. Now suppose they do match, say $\pi = <\alpha,m,\bar{\beta},f,p>$ and $\rho = <\alpha,m,g>$. Then π is a

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

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

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

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

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

Next, *environments* are introduced in the following

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

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

REMARKS

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

(2) When we are going to compute the semantics of a certain unit U, we shall define an environment γ_U such that it contains all information about the definitions that are present in U. It will be needed in the computation of the semantics of U. The first component γ_1 of an environment γ is a function that, supplied with an object name α, gives the process representing the execution of α's *body*. Note that this body depends on the class of α, which can, however, be determined from the object name by applying the function τ. We shall need this first component when we want to define the semantics of a new-expression.
The second component γ_2 gives the meaning of method executions and is used to define the semantics of an answer statement, a method call, and a select statement. When we supply γ_2 with arguments m and α we get the meaning of the execution of the method m by the object α. It depends on the parameters that are passed to the method, so $\bar{\beta}$ is a third argument. The final argument is the expression continuation f, which, applied to the object resulting from the execution of the method, yields a process that represents the steps to be taken next. The result $\gamma_2(m)(\alpha)(\bar{\beta})(f) \in P$ is a process expressing the meaning of the execution of the method m by the object α with parameters $\bar{\beta}$ and expression continuation f.

4.2 Semantics of statements and expressions
In this section we define the semantics of statements by specifying a function $[\![\cdots]\!]_S$ of the following type:

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

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

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

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

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

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

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

The environment γ, the object α and the continuation f serve the same purpose as in the semantics of a statement s. However, there is one important difference: the type of the continuation. The evaluation of expressions always results in a value (an element of *Obj*), upon which the continuation of such an expression generally depends. The function f, when applied to the result β of the expression, will yield the process $f(\beta)$ that is to be executed after the evaluation of the expression.

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

DEFINITION 4.6 (Semantics of expressions)
We define a function

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

where

$$Cont_E = Obj \rightarrow P,$$

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

(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 applied to the

resulting value.

(E2, temporary variable)

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

(E3, send expression)

$$[\![e!m(e_1, \ldots, e_n)]\!]_E(\gamma)(\alpha)(f) =$$
$$[\![e]\!]_E(\gamma)(\alpha)($$
$$\lambda\beta\cdot([\![e_1]\!]_E(\gamma)(\alpha)($$
$$\lambda\beta_1\cdot([\![e_2]\!]_E(\gamma)(\alpha)($$
$$\ldots$$
$$\lambda\beta_{n-1}\cdot([\![e_n]\!]_E(\gamma)(\alpha)($$
$$\lambda\beta_n\cdot\lambda\sigma\cdot\{<\beta, m, \bar{\beta}, f, p_0>\}))\cdots)))))$$

where

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

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. Finally a send step is performed. The object name β refers to the object to which the message is sent. The sequence $<\beta_1, \ldots, \beta_n>$ represents the parameters for the execution of the method m. Besides these values and the method name m the final step $<\beta, m, \bar{\beta}, f, p_0>$ also contains the expression continuation f of the send expression as the dependent resumption. If the attempt at communication succeeds, this continuation will be supplied with the result of the method execution (see section 4.1). The independent resumption of this send step is initialized at p_0.

(E4, method call)

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

where

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

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

(E5, new-expression)

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

where

$\sigma' = <\sigma_1\{\lambda x{\cdot}nil/\beta\},\ \sigma_2,\ \sigma_3 \cup \{\beta\}>$, and

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

A new object of class C is created. It is called $\nu(\sigma_3, C)$: the function ν, supplied with the set of all object names currently in use and the class name C as an argument yields a name of class C that is not yet being used. The state σ is changed by initializing the values of the instance variables of the new object to *nil* and by expanding the set σ_3 with the new name β. The process $\gamma_1(\beta)$, representing the body of the new object, is composed in parallel with the process resulting from the application of the continuation f to β, which is the value of the evaluation of this new-expression. We are able to perform this parallel composition because we know from f what should happen after the evaluation of this new-expression, so here the use of continuations is essential.

(E6, identity checking)

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

(E7, sequential composition)

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

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

(E8, **self**)

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

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

(E9, standard objects)

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

DEFINITION 4.7 (Semantics of statements)
The function

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

where

$$Cont_S = P,$$

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

(S1, assignment to an instance variable)

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

where

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

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

(S2, assignment to a temporary variable)

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

where

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

(S3, answer statement)

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

where for $m \in V$

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

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

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

REMARK

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

$$\pi|_\sigma\rho = \{<\sigma, g(\bar{\beta})(f)\|q>\}.$$

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

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

The continuation of the execution of m is given by $\lambda\beta{\cdot}(f(\beta)\|p)$, the parallel composition of the continuations f and p. This represents the fact that after the rendez-vous, during which the method is executed, the sender and the receiver of the message can proceed in parallel again. (Of course, the independent resumption q may still be executing at this point.) Moreover, the result β of the method execution is passed on to the continuation f of the send expression.

(S4, expressions as statements)

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

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

(S5, sequential composition)

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

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

(S6, conditional)

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

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

(S7, loop statement)

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

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

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

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

(S8, select statement)

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

$\mathit{fi} \cdots)$

$\mathit{fi}),$

where

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

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

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

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

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

LEMMA 4.8
The semantic functions $[\![\cdots]\!]_E$ *and* $[\![\cdots]\!]_S$ *of definitions 4.6 and 4.7 are well defined:*
(a) *For all* $e \in Exp, s \in Stat, \gamma \in Env, \alpha \in A\,Obj$:

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

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

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

4.3 Standard objects

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

$$p_{\mathrm{INT}} = \lambda\sigma\cdot(\{<n, \mathsf{add}, g_n^+>: n \in \mathbb{Z}\} \cup \{<n, \mathsf{sub}, g_n^->: n \in \mathbb{Z}\} \cup \cdots),$$

where

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

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

REMARKS

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

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

(3) The operational intuition behind the definition of p_{INT} is the following: For every $n \in \mathbb{Z}$ the set $p_{\mathrm{INT}}(\sigma)$ contains, among others, two elements, namely $<n, \mathsf{add}, g_n^+>$ and $<n, \mathsf{sub}, g_n^->$. These steps indicate that the integer object n is willing to execute its methods add and sub. If, for example by evaluating $n!\mathsf{add}(n')$, a certain active object sends a request to integer object n to execute the method add with parameter n', then g_n^+, supplied with n' and the continuation f of the active object, is executed. We have that $g_n^+(n')(f)$ is, by definition, the parallel composition of f supplied with the immediate result of the execution of the method add, namely $n+n'$, and the process p_{INT}, which remains unaltered: $g_n^+(n')(f) = f(n+n') \| p_{\mathrm{INT}}$. If, by mistake, a request for the execution of the method add arrives that specifies the wrong type or number of parameters, then $\lambda\sigma\cdot\varnothing$ is the result: the system deadlocks.

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

$$p_{\mathrm{BOOL}} = \lambda\sigma\cdot(\{<b, \mathsf{and}, g_b^{\wedge}>: b \in \{tt, ff\}\} \cup \{<b, \mathsf{or}, g_b^{\vee}>: b \in \{tt, ff\}\} \cup \{<b, \mathsf{not}, g_{\neg b}>: b \in \{tt, ff\}\})$$

where

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

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

$$g_{\neg b} = \lambda\bar{\beta} \in Obj^*\cdot\lambda f \in Obj \to P\cdot (\text{if } \bar{\beta} = <> \text{ then } f(\neg b) \| p_{\mathrm{BOOL}} \text{ else } \lambda\sigma\cdot\varnothing \text{ fi})$$

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

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

$$p_{\mathrm{NIL}} = \lambda\sigma\cdot\{<nil, m, \lambda\bar{\beta}\cdot\lambda f\cdot\lambda\sigma\cdot\varnothing> : m \in MName\}.$$

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

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

$$p_{ST} = p_{INT} \| p_{BOOL} \| p_{NIL},$$

EXAMPLE

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

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

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

$$\begin{aligned}
&[\![x \leftarrow (2!\mathrm{add}(3))]\!]_S(\gamma)(\alpha)(p_0) \\
&\quad = [\![2!\mathrm{add}(3)]\!]_E(\gamma)(\alpha)(f) \\
&\qquad [\text{where } f = \lambda\beta{\cdot}\lambda\sigma'{\cdot}\{<\sigma'', p_0>\} \text{ with } \sigma'' = <\sigma'_1\{(\sigma'_1(\alpha)\{\beta/x\}/\alpha\}, \sigma'_2, \sigma'_3>] \\
&\quad = [\![2]\!]_E(\gamma)(\alpha)(\lambda\beta_1{\cdot}([\![3]\!]_E(\gamma)(\alpha)(\lambda\beta_2{\cdot}\lambda\sigma{\cdot}\{<\beta_1, \mathrm{add}, \beta_2, f, p_0>\}))) \\
&\quad = [\![3]\!]_E(\gamma)(\alpha)(\lambda\beta_2{\cdot}\lambda\sigma{\cdot}\{<2, \mathrm{add}, \beta_2, f, p_0>\}) \\
&\quad = \lambda\sigma{\cdot}\{<2, \mathrm{add}, 3, f, p_0>\}.
\end{aligned}$$

Now the parallel composition:

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

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

4.4 Semantics of a unit

4.5.1 Environments

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

DEFINITION 4.13

Let *Env* be the set of environments as defined in definition 4.5. Thus

$$\begin{aligned}
Env = &(AObj \to P) \\
&\times (MName \to AObj \to Obj^* \to (Obj \to P) \to^1 P).
\end{aligned}$$

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

definition for the class C, then we put

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

where

$$U = <\ldots, C \Leftarrow d, \ldots>, \quad d = <\ldots, s>,$$

otherwise:

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

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

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

where

$$U = <\ldots, C \Leftarrow d, \ldots>,$$

$$d = <\ldots, (\ldots, m \Leftarrow \mu, \ldots), \ldots>,$$

$$\mu = <(u_1, \ldots, u_n), e>,$$

$$\sigma' = <\sigma_1, \sigma_2\{h/\alpha\}, \sigma_3>,$$

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

$$h(u_i) = \beta_i \text{ for } i = 1, \ldots, n,$$

$$h(u) = nil \text{ for } u \notin \{u_1, \ldots, u_n\},$$

$$\bar{\sigma}' = <\bar{\sigma}_1, \bar{\sigma}_2\{\sigma_2(\alpha)/\alpha\}, \bar{\sigma}_3>.$$

Otherwise, we put

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

REMARK

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

LEMMA 4.14

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

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

DEFINITION 4.15

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

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

4.5.2 Semantics of a unit
The execution of a unit U with $U = <C_1 \Leftarrow d_1, \ldots, C_n \Leftarrow d_n>$ consists of the creation of an object of class C_n and the execution of its body.

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

$$\mathcal{D} : Unit \rightarrow P$$

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

$$\mathcal{D}[\![U]\!] = p_U \| p_{ST}$$

where

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

with

$$U = <\ldots, C_n \Leftarrow <\ldots, s>>,$$

and γ_U as given in definition 4.15.

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

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

$$\lambda\sigma \cdot \{<\sigma', \lambda\bar{\sigma} \cdot \{<\bar{\sigma}', p_0>\}>\},$$

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

DEFINITION 4.17 (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) $\forall j \geqslant 1\ [j < length((<\sigma_i, p_i>)_i) \Rightarrow <\sigma_{j+1}, p_{j+1}> \in p_j(\sigma_j)]$
(b) The sequence satisfies one of the following conditions:
  (1) It is infinite. (This represents an infinite computation.)
  (2) The sequence terminates with the pair $<\sigma_n, p_n>$, where $p_n = p_0$. (This represents normal termination of all the objects in the system.)
  (3) The sequence terminates with the pair $<\sigma_n, p_n>$, where $p_n(\sigma_n) = \varnothing$. (This represents abnormal termination.)
  (4) The sequence terminates with the pair $<\sigma_n, p_n>$, where $p_n(\sigma_n) \subset Send_P \cup Answer_P$. (This

represents termination by deadlock.)
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 $<\sigma_i,p_i>$ of a path starting in $<\sigma_1,p_1>$, the state σ_n is passed on to the resumption process p_n.
(2) In general a set $p_i(\sigma_i)$ may contain elements of $Send_P$ or $Answer_P$, besides elements of $\Sigma \times P$. Since we consider paths of only those processes that represent total (POOL) systems that are not expected to communicate with any environment, we view such elements as unsuccessful attempts at communication. Therefore we do not want to incorporate them in our definition of paths. Note that if $p_i(\sigma_i)$ contains only elements of $Send_P$ and $Answer_P$, then the paths ends, and we have the termination by deadlock of case (4) above.
(3) Note that for paths representing the execution of an entire unit case (2) above never arises due to the fact that at least the standard objects are always ready to answer messages. This means that "normal termination" of a POOL program is an instance of case (4) above.

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

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

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

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

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

The state σ_U we start with must be such that

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

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

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

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

5. FAIRNESS

We shall now introduce the notion of *fairness*. A path will be called fair if it does *not* represent a situation in which an object is infinitely often enabled to take a step but never does so.
To determine whether a path is fair or not, for each step that occurs in the path we have to identify the object that takes it. It appears that the semantics of statements as we have defined it offers too little information to make the desired identification. Therefore a small adaptation of our semantic domain P, the merge operator $\|$ and the semantic functions $[\![\cdots]\!]_E$ and $[\![\cdots]\!]_S$ is required.
In our new domain, which we shall still call P, we label every step with the name of the object that takes it. We give the adapted equation that must be satisfied and forget about the details of how to

solve it.

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

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

where

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

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

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

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

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

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

$$p \| q = \begin{cases} p & \text{if } q = p_0 \\ q & \text{if } p = p_0 \\ \lambda\sigma \cdot \; (\{\pi \hat{\|} q : \pi \in p(\sigma) \wedge q(\sigma) \neq \emptyset\} \cup & \text{otherwise.} \\ \quad \{\pi \hat{\|} p : \pi \in q(\sigma) \wedge p(\sigma) \neq \emptyset\} \cup & \\ \quad \bigcup \{\pi|_o \rho : \pi \in p(\sigma), \rho \in q(\sigma)\}) & \end{cases}$$

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

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

The definition of a *path* (as given in definition 4.17) has to be altered straightforwardly: A path now

contains triples $<\kappa_i,\sigma_i,p_i>$. Finally the definition of $[\![\cdots]\!]_E$ and $[\![\cdots]\!]_S$ ought to be changed. We give one example of a clause of the definition of $[\![\cdots]\!]_E$.

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

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

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)

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

(iii)

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

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

agrees to the request.

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

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

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

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

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

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

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

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

6. CONCLUSIONS

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

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

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

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

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

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

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

First, the semantics given here does not provide a clear view of the basic concept of the language, the concept of an object. It would be nice to have a semantics in which the objects appear as building blocks of the system and in which their fundamental properties, e.g. with respect to protection, are already clear from the domain used for their semantics.

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 (America et al., 1986). An equivalence proof would, of course, be very desirable. For a language that is only slightly simpler than POOL (instead of the rendez-vous mechanism it uses simple value transmission) this has already been achieved (see (America and De Bakker, 1988)). Proving the equivalence of the operational and denotational semantics for the full language POOL is the subject of current research.

7. References

America, P. (1985), Definition of the programming language POOL-T, ESPRIT project 415, Doc. No. 0091, Philips Research Laboratories, Eindhoven.

America, P. (1986), Rationale for the design of POOL, ESPRIT project 415, Doc. No. 0053, Philips Research Laboratories, Eindhoven.

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

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

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

America, P., and Rutten, J. J. M. M. (1988), Solving reflexive domain equations in a category of complete metric spaces, *in*: "Proc. Third Workshop on Mathematical Foundations of Programming Language Semantics, New Orleans, 1987" (M. Main, A. Melton, M. Mislove, D. Schmidt, Eds.), LNCS 298, Springer-Verlag, pp. 254-288. (To appear in the Journal of Computer and System Sciences.)

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

Bergstra, J., and Klop, J. W. (1984), Process algebra for synchronous communication, *Information and Control 60*, pp. 109-137.

De Bakker, J. W., Kok, J. N., Meyer, J.-J. Ch., Olderog, E.-R., and Zucker, J. I. (1986), Contrasting themes in the semantics of imperative concurrency, *in*: "Current Trends in Concurrency, Overviews and Tutorials" (J. W. de Bakker, W. P. de Roever, G. Rozenberg, Eds.), Springer-Verlag, LNCS 224, pp. 51-121.

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

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

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

Dugundji, J. (1966), "Topology," Allen and Bacon, Rockleigh, N. J..

Engelking, R. (1977), "General Topology," Polish Scientific Publishers.

Gierz, G., Hofmann, K. H., Keimel, K., Lawson, J. D., Mislove, M., and Scott, D. S. (1980), "A Compendium of Continuous Lattices," Springer-Verlag.

Goldberg, A., and Robson, D. (1983), "Smalltalk-80, The Language and its Implementation," Addison-Wesley.

Gordon, M. J. C. (1979), "The Denotational Description of Programming Languages," Springer-Verlag.

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

Hennessy, M., and Plotkin, G. D. (1979), Full abstraction for a simple parallel programming language, *in*: "Proceedings of the 8th Symposium on the Mathematical Foundations of Computer

Science", LNCS 74, Springer-Verlag, pp. 108-120.
HEWITT, C. (1977), Viewing control structures as patterns of passing messages, *Artificial Intelligence 8*, pp. 323-364.
MAC LANE, S. (1971), "Categories for the Working Mathematician," Springer-Verlag.
ODIJK, E. A. M. (1987), The DOOM system and its applications: a survey of ESPRIT 415 subproject A, *in*: "Parallel Architectures and Languages Europe, Volume I" (J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, Eds.), LNCS 258, Springer-Verlag, pp. 461-479.
VAANDRAGER, F. W. (1986), Process algebra semantics of POOL, Technical Report (CS-R8629), Centre for Mathematics and Computer Science, Amsterdam.

APPENDIX

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

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

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

$$<\sigma',p'>\hat{\odot}\,q\;=\;<\sigma',p'\odot q>,$$

$$<\alpha,m,\bar{\beta},f,p>\hat{\odot}\,q\;=\;<\alpha,m,\bar{\beta},f,p\odot q>,\text{ and}$$

$$<\alpha,m,g>\hat{\odot}\,q\;=\;<\alpha,m,\lambda\bar{\beta}\cdot\lambda h\cdot(g(\bar{\beta})(h)\odot q)>,$$

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

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

LEMMA A.1
(a) *Φ_{PC} is well defined, that is:*
$\forall\odot\in P\times P\to^1 P[\Phi_{PC}(\odot)\in P\times P\to^1 P]$,
(b) *Φ_{PC} is a contraction.*

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

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

where $\tilde{\odot}=\Phi_{PC}(\odot)$.
Let $p_1,p_2,q_1,q_2\in P$. We have (recall that P is an ultra-metric space)

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

It suffices to show that

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

We treat only the first case, the second being symmetric to it.
If one of p_1,q_1,q_2 is equal to p_0, the result is trivial, so suppose $p_1,q_1,q_2 \neq p_0$. Let $\sigma \in \Sigma$ and let for $i=1,2$

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

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

$$\tfrac{1}{2} \cdot d_{\mathcal{P}_a(Step_r)}(X_1 \cup Y_1 \cup Z_1, X_2 \cup Y_2 \cup Z_2) \leq d_P(q_1,q_2).$$

The factor ½ is due to the occurrence of $id_{1/2}$ in the domain equation for P (see definition 4.3). We have

$$d_{\mathcal{P}_a(Step_r)}(X_1 \cup Y_1 \cup Z_1, X_2 \cup Y_2 \cup Z_2) \leq$$
$$\max\{d_{\mathcal{P}_a(Step_r)}(X_1,X_2), d_{\mathcal{P}_a(Step_r)}(Y_1,Y_2), d_{\mathcal{P}_a(Step_r)}(Z_1,Z_2)\}.$$

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

$$d_{\mathcal{P}_a(Step_r)}(Z_1,Z_2) = \max\{\sup_{z_1 \in Z_1}\{d(z_1,Z_2)\}, \sup_{z_2 \in Z_2}\{d(z_2,Z_1)\}\}.$$

We consider only the first supremum:

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

Let $z_1 \in Z_1$. There are several possibilities:
1. Suppose $\{z_1\} = <\alpha,m,\bar{\beta},f,p>|_\sigma <\alpha,m,g_1>$ with $<\alpha,m,\bar{\beta},f,p> \in p_1(\sigma)$, $<\alpha,m,g_1> \in q_1(\sigma)$.
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

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

Then we have

$$\begin{aligned} d_{Step_r}(z_1,z_2) &= d_{Step_r}(<\sigma,g_1(\bar{\beta})(f) \odot p>, <\sigma,g_2(\bar{\beta})(f) \odot p>) \\ &= d_P(g_1(\bar{\beta})(f) \odot p,\ g_2(\bar{\beta})(f) \odot p) \\ &\leq [\text{since } \odot \in P \times P \to^1 P] \\ &\quad d(g_1,g_2) \\ &= d_{Step_r}(<\alpha,m,g_1>, <\alpha,m,g_2>). \end{aligned}$$

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

$$\begin{aligned} d_{Step_r}(<\alpha,m,g_1>, <\alpha,m,g_2>) &\leq d_{\mathcal{P}_a(Step_r)}(q_1(\sigma),q_2(\sigma)) + \epsilon \\ &\leq d_{\Sigma \to \mathcal{P}_a(Step_r)}(q_1,q_2) + \epsilon \\ &\leq 2 \cdot d(q_1,q_2) + \epsilon \end{aligned}$$

Therefore

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

for arbitrary ϵ, so

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

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

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

Therefore

$$d_P(q_1,q_2) \geqslant \tfrac{1}{2}\cdot d_{\mathcal{P}_d(Step_P)}(q_1(\sigma),q_2(\sigma)) \geqslant \tfrac{1}{2}.$$

Now

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

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

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

(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

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

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 $i=1,2$

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

so $p\,\tilde{\odot}_i q(\sigma) = X_i\cup Y_i\cup Z_i$. 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 consider $d_{\mathcal{P}_d(Step_P)}(X_1,X_2)$. By definition of the Hausdorff distance we have

$$d_{\mathcal{P}_d(Step_P)}(X_1,X_2) = \max\{\sup_{\pi_1\in X_1}\{d(\pi_1,X_2)\}, \sup_{\pi_2\in X_2}\{d(\pi_2,X_1)\}\}$$

Let $\pi_1\in X_1$. We show

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

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

$$\inf_{\pi_2\in X_2}\{d_{Step_P}(<\sigma',p'\odot_1 q>,\pi_2)\}\leqslant$$
$$d_{Step_P}(<\sigma',p'\odot_1 q>,<\sigma',p'\odot_2 q>) =$$

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

Thus we have

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

Similarly

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

So

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

And analogously

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

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

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

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

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

and thus

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

LEMMA A.2 (Lemma 4.8)
For every expression e, statement s, environment γ, *and active object* α *we have:*
(i) $[\![e]\!]_E(\gamma)(\alpha)\in(Obj\to P)\to^1 P$
(ii) $[\![s]\!]_S(\gamma)(\alpha)\in P\to^1 P$
(iii) $\forall p\in P[\Phi_{e,s,p}\in P\to^{1/2} P]$
where $\Phi_{e,s,p}: P\to P$ *is defined, for* $q\in P$, *by*

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

PROOF
We prove this lemma using induction on the complexity of the structure of statements and expressions. The proof consists of two parts. Let $\gamma\in Env,\alpha\in AObj$. We show the following:
(a) For all simple (see below) expressions e and statements s we have
$[\![e]\!]_E(\gamma)(\alpha)\in(Obj\to P)\to^1 P$ and $[\![s]\!]_S(\gamma)(\alpha)\in P\to^1 P$.
(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 statement is of the form **answer** V.
Let e be a simple expression. We have to show that

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

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

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

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

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

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

$$d_P([\![\textbf{new}(C)]\!]_E(\gamma)(\alpha)(f_1), [\![\textbf{new}(C)]\!]_E(\gamma)(\alpha)(f_2)) =$$
$$d_p(\lambda\sigma \cdot \{ < \sigma', \gamma_1(\beta) \| f_1(\beta) > \}, \lambda\sigma \cdot \{ < \sigma', \gamma_1(\beta) \| f_2(\beta) > \} =$$
$$\tfrac{1}{2} \cdot \sup_{\sigma \in \Sigma} \{ d_{Step_r}(< \sigma', \gamma_1(\beta) \| f_1(\beta) >, < \sigma', \gamma_1(\beta) \| f_2(\beta) >) \} =$$
$$\tfrac{1}{2} \cdot \sup_{\sigma \in \Sigma} \{ d_P(\gamma_1(\beta) \| f_1(\beta), \gamma_1(\beta) \| f_2(\beta)) \} \leqslant$$
[because $\|$ is NDI]
$$\tfrac{1}{2} \cdot \sup_{\sigma \in \Sigma} \{ d_P(f_1(\beta), f_2(\beta)) \} \leqslant$$
$$\tfrac{1}{2} \cdot d_{Obj \to P}(f_1, f_2).$$

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

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

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

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

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

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

The desired result is straightforward from

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

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

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

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

[because $\|$ is NDI]

$d_P(p_1, p_2)$.

Part (b)

Composite expressions are of the form $e!m(e_1, \ldots, e_n)$, $m(e_1, \ldots, e_n)$, $e_1 \equiv e_2$, or $s;e$. Composite statements are of the form $x \leftarrow e$, $u \leftarrow e$, e, $s_1;s_2$, **if** e **then** s_1 **else** s_2 **fi**, **do** e **then** s **od** or **sel** g_1 **or** $\cdots$ **or** g_n **les**. Suppose that we have proved 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:

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

$d_P([\![e]\!]_E(\gamma)(\alpha)(\cdots \lambda\sigma\cdot\{<\beta, m, \bar{\beta}, f_1, p_0>\} \cdots),$

$\quad [\![e]\!]_E(\gamma)(\alpha)(\cdots \lambda\sigma\cdot\{<\beta, m, \bar{\beta}, f_2, p_0>\} \cdots)) \leqslant$

[by the induction hypothesis for e]

$d(\cdots \lambda\sigma\cdot\{<\beta, m, \bar{\beta}, f_1, p_0>\} \cdots, \cdots \lambda\sigma\cdot\{<\beta, m, \bar{\beta}, f_2, p_0>\} \cdots) \leqslant$

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

$d_P(\lambda\sigma\cdot\{<\beta, m, \bar{\beta}, f_1, p_0>\}, \lambda\sigma\cdot\{<\beta, m, \bar{\beta}, f_2, p_0>\}) \leqslant$

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

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

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

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

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

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]

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

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

[by the induction hypothesis for s]

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

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

$$q_1 =^{def} \text{Fixed Point } (\Phi_{e,s,p_1}),$$

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

We have

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

[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]

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

We see:

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

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

PROOF
We shall show

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

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

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

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

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

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

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

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

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

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

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

The following paper has been reprinted from *Theoretical Computer Science*, Vol. 60, No. 2, 1988, pp. 109–176, with kind permission of Elsevier Science Publishers B. V. (North-Holland).

Theoretical Computer Science 60 (1988) 109-176
North-Holland

DESIGNING EQUIVALENT SEMANTIC MODELS FOR PROCESS CREATION*

Pierre AMERICA

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

Jaco DE BAKKER

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

Communicated by M. Nivat
Received September 1987

Abstract. Operational and denotational semantic models are designed for languages with process creation, and the relationships between the two semantics are investigated. The presentation is organized in four sections dealing with a uniform and static, a uniform and dynamic, a nonuniform and static, and a nonuniform and dynamic language respectively. Here uniform/nonuniform refers to a language with uninterpreted/interpreted elementary actions, and static/dynamic to the distinction between languages with a fixed/growing number of parallel processes. The contrast between uniform and nonuniform is reflected in the use of linear time versus branching time models, the latter employing a version of Plotkin's resumptions. The operational semantics make use of Hennessy's and Plotkin's transition systems. All models are built on metric structures, and involve continuations in an essential way. The languages studied are abstractions of the parallel object-oriented language POOL for which we have designed separate operational and denotational semantics in earlier work. The paper provides a full analysis of the relationship between the two semantics for these abstractions. Technically, a key role is played by a new operator which is able to decide dynamically whether it should act as sequential or parallel composition.

Contents

* Most of this work has been carried out in the context of ESPRIT Project 415: *Parallel Architectures and Languages for Advanced Information Processing: A VLSI-directed Approach.*

1. Introduction

Process creation is an important programming concept which appears in a variety of forms in many contemporary programming styles. In imperative programming one finds it in languages such as Ada [1], NIL [43] and many others. In the context of functional or dataflow languages we refer to [22] for a semantic study dealing with process creation. For logic programming many recent references can be found in [42]. Object-oriented programming (see [5] for a general introduction from a theoretician's point of view) has the family of actor languages (see, e.g., [2, 23, 30]) as examples. The present study was inspired by the language POOL, an acronym for Parallel Object-Oriented Language, described in [3, 4].

In two previous investigations we have developed operational ($\mathcal{O}$) and denotational ($\mathcal{D}$) semantics for POOL [6, 7]. These two semantic models were designed independently of each other, and the investigation reported below constitutes the first step towards the goal of settling the relationship between the two models. For this purpose we concentrate on the programming notion of process creation together with a simple version of process communication, and leave a number of further key notions in POOL for later study. More specifically, we treat communication in the sense—approximately—as exemplified by CSP [31, 32] and do not treat message passing and method invocation—notions which should be situated at the same level as remote procedure call or Ada's rendez-vous. A similar combination of process creation with CSP-like communication was first described in [19], a paper which provides a proof-theoretic treatment of these concepts taken together.

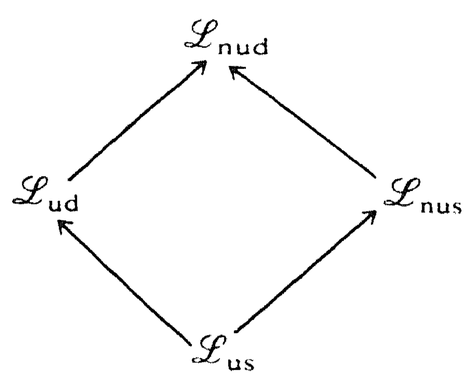
Before going into the characteristics of the languages we shall deal with, let us say something about the terms "operational" and "denotational". *Operational* semantics gives a model of computation by constructing from a given program a kind of "abstract machine" having a set of "states" (which we shall call *configurations*), and describing the *transitions* this abstract machine can make from one state to another. *Denotational* semantics works by assigning a *meaning*, which is a

mathematical entity, to each fragment of a program, in such a way that the meaning of a composite piece of program can be inferred by looking only at the meanings of its parts, not at their internal structure. We say that denotational semantics describes the meaning of programs in a *compositional* way. Fortunately, the technique we use for our operational semantics, transition systems in the style of Hennessy's and Plotkin's Structured Operational Semantics (SOS) [29, 38, 39] describes the abstract machine and its state transitions in a way that is directly related to the syntactic structure of the original program. Due to the explicit presence of this abstract machine, the transition systems employed have, we feel, a strong operational intuition.

The emphasis in our semantics design is very much on a systematic development of the tools for both the operational and denotational models. We have therefore structured the presentation in four sections, dealing with four languages of increasing complexity. Using some terminology which will be explained in a moment, we shall successively present operational and denotational semantics for

(1) a uniform and static language $\mathscr{L}_{us}$;
(2) a uniform and dynamic language $\mathscr{L}_{ud}$;
(3) a nonuniform and static language $\mathscr{L}_{nus}$;
(4) a nonuniform and dynamic language $\mathscr{L}_{nud}$.

These languages are conceptually ordered according to the following diagram:



In this classification, a *uniform* language is one which has uninterpreted elementary actions. In other words, the indivisible or atomic unit of such a language is just a symbol from some alphabet, and the meanings assigned to programs in a uniform language bear strong resemblance to formal languages (here with finite *and infinite* words). A nonuniform language has interpreted elementary actions, in our case assignments and communications. Thus, (individual) variables appear on the scene, and as a consequence we find in our semantics the notion of a *state*, i.e., of a mapping from variables to values. Programs now transform states, and we shall develop a mathematical structure with entities which combine the flavour of state-transforming functions with that of a record of the computational history. In Section 5, we shall provide evidence that the latter notion is necessary in view of the parallel execution operator.

The second distinction in the above diagram concerns that of static versus dynamic languages. In the former, we have a fixed number of parallel processes, in the latter a dynamically growing number of processes: each time a new process is created,

the total number of active processes increases by one. (We shall not investigate in our paper any notion of process destruction, a concept not present in the language POOL.)

The simplest element in the partial order is $\mathscr{L}_{us}$, to be treated in Section 3. It is extended in two directions: one adds the notion of process creation ($\mathscr{L}_{ud}$), dealt with in Section 4, and the other adds the notion of interpreted elementary actions, described in Section 5. Finally, in Section 6, both extensions are brought together, and the full complexity of a nonuniform dynamic language is confronted.

In Sections 3 and 4, the languages are uniform and the semantic models are of the so-called "linear time" variety (see, e.g., [11] or [40]), i.e., they consist of sets of (finite or infinite) sequences over a certain alphabet. The operational semantics is a uniform version of the Structured Operational Semantics (SOS) of Hennessy and Plotkin [29, 38, 39]. The denotational semantics is built on *metric* foundations (apart from the above diagram, no partial order is employed in our paper); this remains true for later (nonuniform) sections. A *distance* between two sequences or sets of sequences is readily defined, and most of the tools of metric topology we use are quite standard. In particular, we shall make heavy use of Banach's fixed point theorem for contracting functions on a complete metric space. Accordingly, our (denotational) semantics will be defined, when dealing with recursive constructs, only when the recursion is *guarded*. In formal languages, one would say that the grammar concerned satisfies a Greibach condition. (In the nonuniform setting we shall take an approach where guardedness is automatically satisfied.)

In each of the Sections 3 to 6 we shall, after having presented the two semantic models, go on to investigate their *equivalence*. In Sections 3 and 4 we actually prove that the two semantics yield the same result, i.e., that for $t \in \mathscr{L}_{us}$ or $t \in \mathscr{L}_{ud}$ we have $\mathscr{O}[\![t]\!] = \mathscr{D}[\![t]\!]$. For $\mathscr{L}_{us}$, this is a result which was already obtained earlier (and presented in [16]). Below, we repeat certain parts of the proof as a first step towards the equivalence theorem for $\mathscr{L}_{ud}$, a result which we believe to be new. In the analysis of $\mathscr{L}_{ud}$ we make essential use of the notion of *continuation*, both of a syntactic and of a semantic kind. Since we develop the semantics of $\mathscr{L}_{us}$ as preparatory for $\mathscr{L}_{ud}$, we have adapted accordingly the treatment of [16], which does not employ continuations. The equivalence proofs for $\mathscr{L}_{us}$ and $\mathscr{L}_{ud}$ have strong similarities. On the other hand, there is also a fundamental difference having to do with the following consequence of process creation: in a statement with a syntactic sequential composition (";"), say $s_1; s_2$, we do not know whether to model the syntactic ";" by semantic concatenation ("$\cdot$") or by parallel execution ("$\|$"). To see this, contrast the statement $a;b$ yielding the singleton set $\{ab\}$ as its meaning, with the statement **new**$(a);b$. The intended meaning of the latter equals that of $a \| b$, which in turn equals the set $\{ab, ba\}$. To overcome this problem we introduce an auxiliary semantic operator ":" which is able, somewhat surprisingly, as it were dynamically to make the decision whether to opt for "$\cdot$" or "$\|$". We consider the introduction of this operator, together with the derivation of its basic technical properties (such as associativity) as a main contribution of our paper.

In Sections 5 and 6 we investigate the nonuniform case. $\mathscr{L}_{\text{nus}}$ has simple communication commands which are syntactic variations on CSP's $P_i?x$ and $P_j!e$ constructs. We stress that our mentioning CSP here is only to indicate the type of communication we have in our language. Partial, let alone full, modelling of CSP is not our aim here. The mathematical structures used to model $\mathscr{L}_{\text{nus}}$ and $\mathscr{L}_{\text{nud}}$ are Plotkin's *resumptions* [37], presented in a fully metric framework as first described in [17] and subsequently extended and put in a category-theoretic perspective in [8]. We use the terminology of *process domains* P, satisfying certain (reflexive) *domain equations* of the form

$$P \cong \mathscr{F}(P)$$

and we shall design the semantics of programs in $\mathscr{L}_{\text{nus}}$ and $\mathscr{L}_{\text{nud}}$ such that the meaning of a program is a *process* $p \in P$. Processes are objects which have a branching structure, and the models for $\mathscr{L}_{\text{nus}}$ and $\mathscr{L}_{\text{nud}}$ are called *branching time* [11, 40].

The operational models for $\mathscr{L}_{\text{nus}}$ and $\mathscr{L}_{\text{nud}}$ once more use SOS style transitions. An important new feature is that, in defining the operational meaning of a program, we collect the information from the induced transition steps into a process. In other words, we assemble the information in successive transition steps into a branching time object. Denotationally, we also use processes as meanings, obtained in the usual manner by a *compositional* system of defining equations. For the nonuniform languages, we do not have that $\mathscr{O}$ and $\mathscr{D}$ yield the same function: In order to allow a compositional definition of $\mathscr{D}$ for the communication constructs, we include in $\mathscr{D}[\![s]\!]$ more information than in $\mathscr{O}[\![s]\!]$ (here s is a nonuniform, static or dynamic, statement). We therefore introduce a natural extension $\mathscr{O}^*$ of $\mathscr{O}$, which preserves one-sided communication information, and then on the one hand establish that $\mathscr{O}^* = \mathscr{D}$, and on the other hand settle the relationship between $\mathscr{O}$ and $\mathscr{O}^*$ in terms of an *abstraction operator abs*, resulting in the equivalence $\mathscr{O} = abs \circ \mathscr{O}^*$.

In Section 6, we combine the techniques designed for $\mathscr{L}_{\text{ud}}$ and $\mathscr{L}_{\text{nus}}$ to deal with all of $\mathscr{L}_{\text{nud}}$. In this way, the reader may obtain a better understanding of this somewhat complicated case: The concepts of process creation and value communication have first been treated in isolation, and now a synthesis of the methods from Sections 4 and 5 is made. In $\mathscr{L}_{\text{nud}}$ we have *classes* (ultimately stemming from Simula [24]), and creation of a process amounts to the creation of a new instance of a class (in the world of object-oriented programming, this instance would be called a (new) object). Such an instance has a name which is (just) another value—in addition to values such as integers or truth-values—and which may be assigned to a variable. In $\mathscr{L}_{\text{nud}}$ we encounter for the first time *expressions* with nontrivial semantics. Consequently, the syntactic and semantic statement continuations used in previous sections are now extended with (syntactic and semantic) *expression continuations.* Operational and denotational semantics for $\mathscr{L}_{\text{nud}}$ are without major surprises once one has digested Sections 4 and 5. At various points, the definitions owe much to similar definitions in [6, 7], though a systematic redesign has been applied in order to allow the final equivalence proof. Again, techniques of Sections 4 and 5 are

brought together, in particular leading to a nonuniform generalization of the ":" operator. Also, an additional argument is necessary to deal with the two forms of recursion now present, one in recursive procedures and the other in recursively defined classes.

This concludes our overview of the contents of the paper. We also mention that in Section 2 we collect some mathematical preliminaries. We list elementary definitions and some useful theorems in metric topology, and provide a brief sketch of the intuition and mathematical basis for (our way of) solving process domain equations.

Detailed semantic models of process creation are scarce in the literature. Semantic studies are reported in a few of the already cited papers [2, 23, 42, 43], but these are all focused on very different problems and techniques. Our work shares with [22] the central role played by continuations. However, that paper investigates process creation in a (deterministic) dataflow setting, and does not address semantic equivalence issues.

Our debt to Plotkin's seminal work in semantics should be clear from the above. To Nivat we are indebted for stimulating our interest in metric techniques going back to his lectures in [35]. Without the detailed semantic analysis of POOL described in [6, 7], the present paper would have been impossible. Many of our semantic definitions can be traced back to concepts and techniques first developed in these two papers.

2. Mathematical preliminaries

2.1. Notation

If X is a set, we denote with $\mathcal{P}(X)$ the power set of X, i.e., the collection of all subsets of X. $\mathcal{P}_\pi(X)$ denotes the collection of all subsets of X which have property π. A sequence $x_0, x_1, \ldots$ of elements of X is usually denoted by $(x_i)_{i=0}^{\infty}$ or, briefly, $(x_i)_i$. The notation $f: X \to Y$ expresses that f is a function with domain X and range Y. We use the notation $f\{y/x\}$, with $x \in X$ and $y \in Y$, for a *variant* of f, i.e., for the function which is defined by

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

If $f: X \to X$ and $f(x) = x$, we call x a *fixed point* of f.

2.2. Metric spaces

Metric spaces are the mathematical structures in which we carry out our semantic work. We give only the facts most needed in this paper. For more details, the reader is referred to [25, 26].

2.1. Definition. A metric space is a pair $\langle M, d\rangle$ where M is a nonempty set and d is a mapping $M \times M \to [0, 1]$ having the following properties:

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

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

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

(d is called a *metric* or *distance*.)

Examples. (1) Let A be an arbitrary set. The *discrete metric* on A is defined as follows: Let $x, y \in A$.

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

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

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

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

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

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

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

(2) 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

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

We call the sequence $(x_i)_i$ *convergent* and write $x = \lim_i x_i$.

(3) $\langle M, d\rangle$ is called *complete* whenever each Cauchy sequence in M converges to an element of M.

2.3. Definition. Let $\langle M_1, d_1\rangle$ and $\langle M_2, d_2\rangle$ be metric spaces.

(1) We say that $\langle M_1, d_1\rangle$ and $\langle M_2, d_2\rangle$ are *isometric* if there is a mapping $f: M_1 \to M_2$ such that

(a) f is a bijection,

(b) $\forall x, y \in M_1\ [d_2(f(x), f(y)) = d_1(x, y)]$.

We then write $M_1 \cong M_2$. If we have a function f satisfying only condition (1)(b), we call it an isometric embedding.

(2) Let $f: M_1 \to M_2$. We call f *continuous* whenever, for each sequence $(x_i)_i$ with limit x in M_1, we have that $\lim_i f(x_i) = f(x)$. We shall denote the set of all continuous functions from M_1 by $M_1 \to^{c} M_2$.

(3) We call a function $f: M_1 \to M_2$ *contracting* if there exists a real number c with $0 \leq c < 1$ such that

$$\forall x, y \in M_1 [d_2(f(x), f(y)) \leq c\, d_1(x, y)].$$

(4) A function $f: M_1 \to M_2$ is called *non-distance-increasing* if

$$\forall x, y \in M_1 [d_2(f(x), f(y)) \leq d_1(x, y)].$$

We shall denote the set of all non-distance-increasing functions from M_1 to M_2 by $M_1 \to^{\mathrm{NDI}} M_2$.

2.4. Lemma. *Let $\langle M_1, d_1\rangle$ and $\langle M_2, d_2\rangle$ be metric spaces, and let $f: M_1 \to M_2$ be a contracting function. Then f is continuous. The same holds for non-distance-increasing functions.*

2.5. Theorem (Banach). *Let $\langle M, d\rangle$ be a complete metric space. Each contracting function $f: M \to M$ has a unique fixed point which equals $\lim_i f^i(x_0)$ for arbitrary $x_0 \in M$. (Here $f^0(x_0) = x_0$ and $f^{i+1}(x_0) = f(f^i(x_0))$.)*

Proof. Since f is contracting, the sequence $(f^i(x_0))_i$ is a Cauchy sequence. By the completeness of $\langle M, d\rangle$, the limit $x = \lim_i f^i(x_0)$ exists. By the continuity of f (Lemma 2.4), $f(x) = f(\lim_i f^i(x_0)) = \lim_i f^{i+1}(x_0) = x$. If, for some $y \in M$, $f(y) = y$ then, by the contractivity of f, $d(x, y) = d(f(x), f(y)) \leq c\, d(x, y)$. Hence, since $c < 1$ we conclude that $d(x, y) = 0$, and $x = y$ follows. □

2.6. Definition. Let $\langle M, d\rangle$ be a metric space.

(1) A subset X of M is called *closed* whenever each converging sequence with elements in X has its limit in X.

(2) A subset X of M is called *compact* whenever each sequence in X has a subsequence which converges to an element of X.

Remarks. (1) The definition of compactness given here is in fact what is called *sequential compactness* in general topology. In a metric space this is equivalent to compactness.

(2) Taking, in Definition 2.6(2), X equal to M defines when the space $\langle M, d\rangle$ is called compact.

(3) In a metric space every compact set is closed.

2.7. Definition. Let $\langle M, d\rangle$, $\langle M_1, d_1\rangle$, and $\langle M_2, d_2\rangle$ 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

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

(2) We define a metric d_{P} on the Cartesian product $M_1 \times M_2$ by

$$d_{\mathrm{P}}(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle) = \max_{i \in \{1,2\}} d_i(x_i, y_i).$$

(3) With $M_1 \sqcup M_2$ we denote the *disjoint union* of M_1 and M_2, which may be defined as $(\{1\} \times M_1) \cup (\{2\} \times M_2)$. We define a metric d_U on $M_1 \sqcup M_2$ as follows:

$$d_U(x, y) = \begin{cases} d_i(x, y) & \text{if } x, y \in \{i\} \times M_i \text{ for } i = 1 \text{ or } i = 2, \\ 1 & \text{otherwise.} \end{cases}$$

In the sequel we shall often write $M_1 \cup M_2$ instead of $M_1 \sqcup M_2$, implicitly assuming that M_1 and M_2 are already disjoint.

(4) Let $\mathcal{P}_{cl}(M) = \{X \mid X \subseteq M, X \text{ closed}\}$. We define a metric d_H on $\mathcal{P}_{cl}(M)$, called the *Hausdorff distance*, as follows:

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

where $d(x, Z) = \inf_{z \in Z} d(x, z)$ (here we use the convention that $\sup \emptyset = 0$ and $\inf \emptyset = 1$, so that the empty set will have distance 1 to every other set).

2.8. Theorem. *Let* $\langle M, d\rangle$, $\langle M_1, d_1\rangle$, $\langle M_2, d_2\rangle$, d_F, d_P, d_U, *and* d_H *be as in Definition* 2.7, *and suppose in addition that* $\langle M, d\rangle$, $\langle M_1, d_1\rangle$, *and* $\langle M_2, d_2\rangle$ *are* complete. *We have that*

(1) $\langle M_1 \to M_2, d_F\rangle$ (*together with* $\langle M_1 \to^c M_2, d_F\rangle$ *and* $\langle M_1 \to^{NDI} M_2, d_F\rangle$),
(2) $\langle M_1 \times M_2, d_P\rangle$,
(3) $\langle M_1 \sqcup M_2, d_U\rangle$,
(4) $\langle \mathcal{P}_{cl}(M), d_H\rangle$

are complete metric spaces. (*Strictly speaking, for the completeness of* $M_1 \to M_2$, *the completeness of* M_1 *is not required.*)

In the sequel we shall often write $M_1 \to M_2$, $M_1 \times M_2$, $M_1 \sqcup M_2$, $\mathcal{P}_{cl}(M)$, etc., when we mean the metric spaces with the metrics just defined.

The proofs of parts (1), (2), and (3) of Theorem 2.8 are straightforward. Part (4) is more involved. It can be proved with the help of the following characterization of completeness of $\langle \mathcal{P}_{cl}(M), d_H\rangle$.

2.9. Theorem. *Let* $\langle \mathcal{P}_{cl}(M), d_H\rangle$ *be as in Definition* 2.7. *Let* $(X_i)_i$ *be a Cauchy sequence in* $\mathcal{P}_{cl}(M)$. *We have*

$$\lim_i X_i = \{\lim_i x_i \mid x_i \in X_i,\ (x_i)_i \text{ a Cauchy sequence in } M\}.$$

Theorem 2.9 is due to Hahn [28]. Proofs of Theorems 2.8 and 2.9 can be found, e.g., in [25] or [26]. The proofs are also repeated in [17].

2.10. Theorem (Metric completion). *Let* M *be an arbitrary metric space. Then there exists a metric space* $\bar{M}$ (*called the* completion *of* M) *together with an isometric embedding* $i : M \to \bar{M}$ *such that*

(1) $\bar{M}$ *is complete,*
(2) *for every complete metric space* M' *and isometric embedding* $j : M \to M'$ *there exists a unique isometric embedding* $\bar{j} : \bar{M} \to M'$ *such that* $\bar{j} \circ i = j$.

Proof. Standard topology. □

Finally, we have the following result from Rounds [41].

2.11. Theorem. *Let $f: M_1 \to M_2$ be an arbitrary function, where M_1 and M_2 are* compact *metric spaces, and define $\hat{f}: \mathcal{P}_{\mathrm{cl}}(M_1) \to \mathcal{P}(M_2)$ by $\hat{f}(X) = \{f(x) \mid x \in X\}$. Then the following statements are equivalent:*

(1) *f is continuous.*

(2) *For every $X \in \mathcal{P}_{\mathrm{cl}}(M_1)$ we have $\hat{f}(X) \in \mathcal{P}_{\mathrm{cl}}(M_2)$, and $\hat{f}$ is continuous with respect to the Hausdorff metrics.*

(3) *For every $X \in \mathcal{P}_{\mathrm{cl}}(M_1)$ we have $\hat{f}(X) \in \mathcal{P}_{\mathrm{cl}}(M_2)$, and, for each decreasing chain $(X_i)_i$ (i.e., $X_i \supseteq X_{i+1}$ for all i) of elements in $\mathcal{P}_{\mathrm{cl}}(M_1)$ we have*

$$\hat{f}\left(\bigcap_i X_i\right) = \bigcap_i \hat{f}(X_i).$$

2.3. Resumptions and domain equations

We begin with a brief intuitive introduction of the notion of *resumption* (due to Plotkin [37]). We use the terminology of *processes* p, q, which are elements of a *process domain* P. We emphasize that we are concerned here with semantics rather than with syntax: processes are elements of mathematical structures rather than (pieces of) program texts. Process domains are obtained as solutions of *domain equations*. In this informal introduction we let A and B stand for arbitrary (fixed) sets (where necessary provided with the discrete metric) and we shall denote by p_0 an arbitrary mathematical object which shall play the role of a *nil* process. A very simple equation is

$$P \cong \{p_0\} \cup (A \times P). \tag{2.1}$$

We can read this equation as follows: a process $p \in P$ is either p_0, which cannot take any action, or it is a pair $\langle a, q \rangle \in A \times P$, where a is the first action taken and q is the *resumption*, describing the rest of p's actions. Clearly, (2.1) has as a solution the set of all finite sequences $\langle a_1, a_2, \ldots, a_n, p_0 \rangle$, with $n \geq 0$ and $a_i \in A$ for all i. The set of all these finite sequences plus all infinite sequences $\langle a_1, a_2, \ldots \rangle$ is another solution.

We next consider

$$P \cong \{p_0\} \cup (A \to (B \times P)). \tag{2.2}$$

This is already a much more interesting equation: each process p is either p_0 or a function which, when supplied with an argument a, yields a pair $p(a) = \langle b, p' \rangle$. We see that p maps a to b, at the same time turning itself into the resumption p'. We can say that p determines its first step b and the resumption p' on the basis of a.

The following equation we consider is

$$P \cong \{p_0\} \cup (A \to \mathcal{P}_{\mathrm{cl}}(B \times P)). \tag{2.3}$$

Now, if we feed a process $p \neq p_0$ with some $a \in A$, a whole set X of possible pairs $\langle b, q \rangle$ results, among which the process can choose freely. For reasons of cardinality, (2.3) has no solution when we take *all* subsets, rather than all closed subsets of $B \times P$. Moreover, we should be more precise about the metrics involved. We should have written (2.3) like this:

$$P \cong \{p_0\} \cup (A \to \mathcal{P}_{\mathrm{cl}}(B \times \mathrm{id}_{1/2}(P))) \tag{2.3'}$$

where, for any positive real number c, id_c maps a metric space $\langle M, d \rangle$ into $\langle M, d' \rangle$ with $d'(x, y) = c\, d(x, y)$. We shall adopt the convention that in domain equations like (2.1), (2.2) and (2.3) every occurrence of the defined space P on the right-hand side is implicitly surrounded by $\mathrm{id}_{1/2}$. (Note that (2.1) and (2.2) can be solved even without this convention, resulting in a set of sequences or trees respectively, with the discrete metric.)

It will turn out that (2.3) is the right type of domain equation for our purposes. We shall, in Sections 5 and 6, specialize A and B to certain sets which have the appropriate semantic connotations. As we shall see later, an important advantage of processes as in (2.3) is that they allow a natural definition of their *merge*, which combines interleaving and communication steps in a way which is quite familiar in concurrency semantics (for one example, see ACP [18]).

We next discuss how one may solve equations as exemplified by (2.1) to (2.3). These equations are special cases of domain equations as studied in depth in the domain theory initiated by Scott and developed further by many researchers (including Plotkin's [37], see, e.g., [27] for a comprehensive reference). We shall here briefly sketch an approach to the solution of such domain equations which is fully couched in the setting of (complete) metric spaces (first described in [17]) and, in this way, avoids any mention of order-theoretic structures. We thus obtain a unified mathematical foundation for our semantics since we exclusively base ourselves on metric techniques. We present a somewhat streamlined version of the results in [17]. There is an important class of domain equations not covered in that paper, viz. equations of the form

$$P \cong \cdots (P \to \cdots) \cdots \tag{2.4}$$

i.e., involving functional domains with the "unknown" domain on the left-hand side of "$\to$". Recently, a fuller treatment of the metric approach has been described by America and Rutten [8]. There, equations $P \cong \mathcal{F}(P)$ are solved in a category of metric spaces, also catering for situations as in (2.4). For the purpose of the present paper, the restricted case to be described below suffices, and we thus avoid the introduction of various category-theoretic notions which are not essential for the applications at hand.

We consider a domain equation

$$P \cong \mathcal{F}(P) \tag{2.5}$$

where $\mathcal{F}$ is a function (technically, a functor on the category of complete metric spaces, but we do not have to be aware of this) which is constructed according to

the following syntax (where c is a real number, $0<c<1$, and M an arbitrary complete metric space with metric d_M):

$$\mathcal{F} ::= \mathcal{F}_M \mid \mathrm{id}_c \mid \mathcal{F}_1 \times \mathcal{F}_2 \mid \mathcal{F}_1 \sqcup \mathcal{F}_2 \mid \mathcal{P}_{\mathrm{cl}}(\mathcal{F}') \mid \mathcal{F}_M \to \mathcal{F}'. \tag{2.6}$$

The above definition of $\mathcal{F}$ should be understood as follows. For each complete metric space $\langle Q, d\rangle$ we define the complete metric space $\langle \mathcal{F}(Q), \mathcal{F}(d)\rangle$ to which $\mathcal{F}$ maps $\langle Q, d\rangle$:

(1) $\mathcal{F}_M(Q) = M$, $\mathcal{F}_M(d) = d_M$. Thus, $\mathcal{F}_M$ is the constant function, yielding $\langle M, d_M\rangle$ for every Q. In various applications, we just give some arbitrary set A and assume for A the discrete metric.

(2) $\mathrm{id}_c(Q) = Q$, $\mathrm{id}_c(d)(x, y) = c\, d(x, y)$.

(3) If $\mathcal{F} = \mathcal{F}_1 \times \mathcal{F}_2$, assume that $\mathcal{F}_i(Q) = Q_i$ and $\mathcal{F}_i(d) = d_i$ for $i = 1, 2$. Then we put $\mathcal{F}(Q) = Q_1 \times Q_2$ and $\mathcal{F}(d) = d_{\mathrm{P}}$ (see Definition 2.7).

(4) If $\mathcal{F} = \mathcal{F}_1 \sqcup F_2$, assume again that $\mathcal{F}_i(Q) = Q_i$ and $\mathcal{F}_i(d) = d_i$ for $i = 1, 2$. Then we put $\mathcal{F}(Q) = Q_1 \sqcup Q_2$ and $\mathcal{F}(d) = d_{\mathrm{U}}$ (see Definition 2.7).

(5) If $\mathcal{F} = \mathcal{P}_{\mathrm{cl}}(\mathcal{F}')$, assume that $\mathcal{F}'(Q) = Q'$ and $\mathcal{F}'(d) = d'$. Now we put $\mathcal{F}(Q) = \mathcal{P}_{\mathrm{cl}}(Q')$ and $\mathcal{F}(d) = (d')_{\mathrm{H}}$ (see Definition 2.7).

(6) If $\mathcal{F} = \mathcal{F}_M \to \mathcal{F}'$, we already know that $\mathcal{F}_M(Q) = M$ and $\mathcal{F}_M(d) = d_M$. Now assume that $\mathcal{F}'(Q) = Q'$ and $\mathcal{F}'(d) = d'$. We put $\mathcal{F}(Q) = M \to Q'$ and $\mathcal{F}(d) = (d')_{\mathrm{F}}$, where $(d')_{\mathrm{F}}$ is the function metric on $M \to Q'$ derived from d' (see Definition 2.7).

According to [17], for $\mathcal{F}$ as just given we can solve (2.5) by the following scheme:

Define inductively

$$P_0 = \langle \{p_0\}, d_0\rangle \quad d_0 \text{ the discrete metric,}$$

$$P_{n+1} = \mathcal{F}(P_n).$$

Observe that—ignoring the obvious identification of P with $\{i\} \times P$ for $i = 1, 2$ in case $\mathcal{F}$ involves a disjoint union—we have for all n

$$P_n \subseteq P_{n+1}. \tag{2.7}$$

Now we put $\langle P_\omega, d_\omega\rangle = \langle \bigcup_n P_n, \bigcup_n d_n\rangle$ (with the obvious interpretation of $\bigcup_n d_n$) and we define $\langle \bar{P}, \bar{d}\rangle$ as the *completion* (see Theorem 2.10) of $\langle P\omega, d_\omega\rangle$. Then we have the following theorem.

2.12. Theorem. *For $\mathcal{F}$ and $\bar{P}$ as above, we have $\bar{P} \cong \mathcal{F}(\bar{P})$.*

Proof. A nonessential variation of the results of [17]. □

Remark. The scope of the techniques applied in the proof of Theorem 2.12 was not fully understood in [17], and substantial clarification was provided by [8]. In addition, [8] brings an essential generalization: The clause $\mathcal{F}_M \to \mathcal{F}'$ in (2.6) is replaced by $\mathcal{F}_1 \to \mathcal{F}_2$, thus dropping the restriction that only constants appear on the left-hand side of "$\to$". A precise analysis is provided of the ensuing situation, involving the notion of *contraction coefficient* $c \geqslant 0$ of a functor $\mathcal{F}$, and culminating

in the result that, for $c < 1$, (2.5) has a unique solution (up to isometry). A key step in this analysis is a generalization of (2.7): in the presence of general functional domains we can no longer gloss over the need for a precise embedding of P_n into P_{n+1}, and a rigorous definition of an *arrow* $\iota : P_n \to P_{n+1}$ is needed. For arbitrary complete metric spaces $\langle M_1, d_1 \rangle$ and $\langle M_2, d_2 \rangle$, such an arrow $\iota : M_1 \to M_2$ is a pair $\langle i, j \rangle$ with $i : M_1 \to M_2$ an isometric embedding and $j : M_2 \to M_1$ a non-distance-increasing function such that $j \circ i$ is equal to the identity function on M_1.

3. A uniform and static language

We begin with a detailed study of $\mathcal{L}_{us}$, a uniform and static language. First we present its syntax, and its operational semantics in the style of Hennessy and Plotkin [29, 38, 39]. Next, we develop the metric framework to define the denotational semantics for $\mathcal{L}_{us}$. Finally, we discuss the relationship between the two semantics and outline an equivalence proof. Most of this section can already be found in [16, Section 2]; we repeat this material here to make the present paper self-contained and to prepare the way for the treatment of the dynamic case in the next section. There are a few new points in the development presented below as well, partly due to the fact that $\mathcal{L}_{us}$ has only one level of parallelism, partly caused by our wish to achieve a smooth transition to the definitions for $\mathcal{L}_{ud}$, the language with dynamic parallelism (a notion not treated in [16]). The latter aim has in particular motivated our use below of the technique of *continuations*.

3.1. Syntax and preliminary definitions

Let A be a finite alphabet of *elementary actions*, with typical elements a, b, c (by this we mean that the letters a, b, and c, possibly adorned with primes or subscripts, will be used to range over elements of A) and let $StmV$ be an infinite set of *statement variables*, with typical elements x, y. Statement variables are used in the syntactic construct for *recursion*, as we shall see in a moment.

3.1. Definition (*Syntax for statements and programs*). (1) The set $\mathcal{S}_{us}$ of (uniform and static) *statements*, with typical element s, is defined by

$$s ::= a \mid x \mid s_1 ; s_2 \mid s_1 \cup s_2 \mid \mu x[s']$$

The prefix μx in the construct $\mu x[s']$ *binds* occurrences of x in s' in the usual way. We call a statement s *closed* if it contains no free occurrences of statement variables.

(2) The set $\mathcal{L}_{us}$ of (uniform and static) *programs*, with typical element t, is defined by

$$t ::= s_1 \| \cdots \| s_n \quad (n \geq 1).$$

Here we require that $s_1, \ldots, s_n$ are all closed (so that programs are always closed).

Examples. (1) Statements: $a;b$, $\mu x[(a;x)\cup b]$, $\mu x[(a;x)\cup(x;b)\cup c]$, $\mu x[(a_1;x;a_2)\cup \mu y[(y;b)\cup c]]$, $a;y;b$ (only the last example is not closed).

(2) Programs: Each of the closed statements listed under (1), and, in addition, $(a;b)\|\mu x[(a;x)\cup b]\|\mu x[(x;b)\cup c]$, $\mu x[a;x]\|\mu y[b;y]$.

A statement s is of one of the following forms:

- an elementary action a,
- the sequential composition $s_1;s_2$ of statements s_1 and s_2,
- the nondeterministic choice $s_1\cup s_2$ (also known as local or internal nondeterminism): $s_1\cup s_2$ is executed by executing either s_1 or s_2, where the choice is made nondeterministically.
- a statement variable x, which is (primarily) used in:
- the recursive construct $\mu x[s]$: its execution amounts to execution of s, where occurrences of x in s are executed by (recursively) executing $\mu x[s]$. For example, with the semantic definitions to be proposed presently, the intended meaning of $\mu x[(a;x)\cup b]$ is the set $a^*\cdot b\cup\{a^\omega\}$.

A program $t=s_1\|\cdots\|s_n$ consists of $n\geq 1$ statements which are to be executed in parallel. Since n remains fixed throughout the execution of t, we call the language $\mathscr{L}_{us}$ *static* to distinguish it from the dynamic language $\mathscr{L}_{ud}$ studied in Section 4.

$\mathscr{L}_{us}$ has no synchronization or communication. The issues which arise when such notions are added to it are studied in detail in (later sections of) [16]. We do not want to complicate our treatment of $\mathscr{L}_{us}$—which plays only a preliminary role in the present context—by including such ramifications.

Substitution of a statement for a statement variable is defined in the familiar way: $s[s'/x]$ denotes the result of substituting s' for all free occurrences of x in s, with the usual precaution of renaming bound variables when necessary to avoid clashes.

In both operational and denotational models we shall use the universe of *streams*, defined as follows.

3.2. Definition (*Streams*, cf. [20, 21]). We assume that $\perp\notin A$. The set A^{st} of all *streams* over A is defined by

$$A^{st}=A^*\cup A^\omega\cup(A^*\times\{\perp\})$$

where A^* (A^ω) is the set of all finite (infinite) words over A.

We shall use u, v, w to range over A^{st} and use ε for the empty stream. Streams of the form $\langle u,\perp\rangle$ will be written as $u\cdot\perp$ or simply $u\perp$. We shall abbreviate $\langle\varepsilon,\perp\rangle$ to $\perp$. The use of $\perp$ is motivated, in an operational setting, by our wish to produce some visible result as the outcome of an infinite computation that does not produce an infinite sequence of elementary actions. For example, we shall organize the definitions such that both $\mu x[x]$ and $\mu x[(x;b)\cup c]$ deliver $\perp$ as an outcome (in the latter case together with cb^*).

We shall use a^ω for the infinite sequence of a's. $length(u)$ yields the number of symbol occurrences (from $A \cup \{\perp\}$) in u. In particular, for $u \in A^\omega$, $length(u) = \infty$, and for $u = u'\perp$, $u' \in A^*$, we have $length(u) = length(u') + 1$. we use "$\leqslant$" for the prefix ordering on A^{st}, i.e., we put $u \leqslant v$ whenever $u = v$ or $u \in A^*$ and, for some $w \in A^{st}$, $u \cdot w = v$ (the reader who wants to see a precise definition of the concatenation "$\cdot$" of streams is referred to Definition 3.12). For example, we have $ab \leqslant abc$, $a^n \leqslant a^\omega$, $ab \leqslant ab\perp$, but $a\perp \not\leqslant ab\perp$. We recall that each $\leqslant$-chain $(u_i)_i$, with $u_i \leqslant u_{i+1}$, $i = 0, 1, \ldots$, has a least upper bound $u = \mathrm{lub}_i\, u_i$ in A^{st}, where $(u_i)_i$ is either infinitely often increasing ($u_i \neq u_{i+1}$ for infinitely many i) and then $u \in A^\omega$, or $(u_i)_i$ *stabilizes* in some u_{i_0} ($u_i = u_{i_0}$ for all $i \geqslant i_0$), and then $u = u_{i_0}$. We conclude this list of definitions with the notation $u(n)$, which denotes the $\leqslant$-prefix of u of length n in case this exists, and which equals u otherwise.

In both this and all subsequent sections we shall make extensive use of so-called *continuations*, both of syntactic and semantic variety. In defining the semantics of a statement, we shall use a continuation to indicate the "actions" which remain to be done *after* this statement. Syntactically, this is done by a piece of program text, a syntactic continuation, to be defined below. Semantic continuations will be introduced in Section 3.3. The use of continuations in the context of $\mathscr{L}_{us}$ is not necessary or especially helpful, but it introduces the techniques which will be applied fruitfully in the following sections.

We shall denote the empty syntactic continuation by E (note that E is not itself a statement) and then define the following sets.

3.3. Definition (*Syntactic continuations*). (1) The set *SyCo* of *syntactic continuations*, with typical element r, is defined by

$$r ::= E \mid s; r'$$

Here we require that each statement s occurring in a syntactic continuation r is closed (so that syntactic continuations are always closed).

(2) We define the set *PSyCo* of *parallel syntactic continuations*, with typical element ρ, as follows:

$$\rho ::= r_1, \ldots, r_n \quad (n \geqslant 1).$$

3.2. Operational semantics

We now proceed with the operational semantics for $\mathscr{S}_{us}$ and $\mathscr{L}_{us}$. We apply the technique of *transition systems*, introduced by Hennessy and Plotkin [29, 38, 39], and proven to be quite fruitful in a variety of concurrency semantics. The particular version employed below is close to the style of definition in [9, 10], though these papers deal in fact with interpreted rather than with uninterpreted languages (cf., for example, the discussion in [12] of the distinction between uniform and nonuniform). In [16] we also discuss the relationships between our version of the transition formalism and other variants one may encounter in the literature.

A configuration is either a pair $\langle \rho, w \rangle$, with $w \in A^* \times \{\perp\}$, or simply a stream w, with $w \in A^*$. A *transition* is a pair of configurations of the form

$$\langle \rho, w \rangle \to \langle \rho', w' \rangle \quad \text{or} \quad \langle \rho, w \rangle \to w''$$

(where $w, w' \in A^* \times \{\perp\}$, $w'' \in A^*$). In order to understand such transitions, we first mention—anticipating later precise definitions—that a program $t = s_1 \| \cdots \| s_n$ will correspond to a parallel continuation $\rho = s_1; E, \ldots, s_n; E$. For each configuration $\langle \rho, w \rangle$, we view ρ as the program currently to be executed, and w as an (unfinished) stream of elementary actions collected so far. The "$\to$" relation as given above either reflects a one-step transition to a new such pair $\langle \rho', w' \rangle$, or a one-step transition to a (finished) stream w''. The *transition system* to be defined in a moment provides the information necessary to *deduce* transitions of the given form. More precisely, we shall define the relation "$\to$" between configurations as the *smallest* (with respect to set inclusion) relation which satisfies the axioms given in the following definition.

3.4. Definition (*Transition system for* $\mathscr{L}_{us}$). The system $\mathscr{T}_{us}$ for $\mathscr{L}_{us}$ consists of the following five *axioms* (in a self-explanatory notation):

$$\langle \ldots, a; r, \ldots, w\perp \rangle \to \langle \ldots, r, \ldots, wa\perp \rangle, \qquad \textbf{Elem}$$

$$\langle \ldots, (s_1; s_2); r, \ldots, w \rangle \to \langle \ldots, s_1; (s_2; r), \ldots, w \rangle, \qquad \textbf{SeqComp}$$

$$\langle \ldots, (s_1 \cup s_2); r, \ldots, w \rangle \to \langle \ldots, s_1; r, \ldots, w \rangle \mid \langle \ldots, s_2; r, \ldots, w \rangle \qquad \textbf{Choice}$$

(here $X \to Y \mid Z$ is short for $X \to Y$ and $X \to Z$),

$$\langle \ldots, \mu x[s]; r, \ldots, w \rangle \to \langle \ldots, s[\mu x[s]/x]; r, \ldots, w \rangle, \qquad \textbf{Rec}$$

$$\langle E, \ldots, E, w\perp \rangle \to w. \qquad \textbf{Term}$$

(Note that, by our conventions, in the first and fifth axiom $w \in A^*$, and in the remaining ones $w \in A^* \times \{\perp\}$.)

Our next step is the definition of a semantic function $\mathscr{O}[\![\cdot]\!]$, yielding, when applied to some ρ, a subset of A^{st}.

3.5. Definition. We define the function

$$\mathscr{O}[\![\cdot]\!] : PSyCo \to \mathscr{P}(A^{st})$$

as follows. Let $\rho \in PSyCo$. We put a stream w into $\mathscr{O}[\![\rho]\!]$ whenever one of the following conditions is satisfied:

(1) There is a finite sequence of configurations $(\langle \rho_i, w_i \rangle)_{i=0}^{n}$ such that $\langle \rho_i, w_i \rangle \to \langle \rho_{i+1}, w_{i+1} \rangle$ for $i = 0, \ldots, n-1$, $\rho_0 = \rho$, $w_0 = \perp$, and $\langle \rho_n, w_n \rangle \to w$.

(2) There is an infinite sequence of configurations $(\langle \rho_i, w_i \rangle)_{i=0}^{\infty}$, such that $\langle \rho_i, w_i \rangle \to \langle \rho_{i+1}, w_{i+1} \rangle$ for $i = 0, 1, \ldots$, $\rho_0 = \rho$, $w_0 = \perp$, $w_i = w_i' \perp$, and $w = (\text{lub}_i\, w_i')\perp$.

Remark. In clause (2) we use the obvious fact that if $\langle \rho, w\perp\rangle \to \langle \rho', w'\perp\rangle$, then $w \leq w'$. Note that, for $(w_i')_i$ infinitely often increasing, $w' =^{\text{def}} \text{lub}_i\, w_i'$ belongs to A^ω, so from the definition $w = w'\perp$ we infer that $w = w'$ (by Definition 3.12, concatenating any stream to the right of some infinite stream has no effect). For $(w_i')_i$ stabilizing in w_{i_0}', we obtain $w = w_{i_0}'\perp$.

Examples. (1) $\mathcal{O}[\![\mu x[(a;x)\cup b];E]\!] = \{a^\omega\}\cup a^*b$, $\mathcal{O}[\![\mu x[(x;a)\cup b];E]\!] = \{\perp\}\cup ba^*$.
(2) $\mathcal{O}[\![(c\cup(a;b));E,d;E]\!] = \{cd, dc, dab, adb, abd\}$.

We conclude the operational semantics definitions with the definition of $\mathcal{O}[\![t]\!]$ for $t \in \mathcal{L}_{\text{us}}$:

3.6. Definition. The mapping $\mathcal{O}[\![\cdot]\!]: \mathcal{L}_{\text{us}} \to \mathcal{P}(A^{\text{st}})$ is defined as follows. Let $t = s_1 \| \cdots \| s_n \in \mathcal{L}_{\text{us}}$. Then

$$\mathcal{O}[\![t]\!] = \mathcal{O}[\![s_1;E,\ldots,s_n;E]\!].$$

Remark. There is a natural connection between the notions discussed above when restricted to programs without parallelism ($t = s_1$) and the languages with finite or infinite words produced by context-free grammars in the sense of, e.g., Nivat [35]. For example, the grammar $X \to aXb|c$ produces $\{a^\omega\}\cup\{a^ncb^n \mid n \geq 1\}$, and so does $\mathcal{O}[\![\mu x[(a;x;b)\cup c]]\!]$. A difference arises in the presence of unguarded recursion (cf. Definition 3.14 below); for example, $\mathcal{O}[\![\mu x[(x;b)\cup c]]\!]$ equals $\{\perp\}\cup cb^*$, whereas $X \to Xb|c$ would, by Nivat's definitions, produce only cb^*. Briefly, the role of $\perp$ in our style(s) of semantics has no counterpart in traditional formal language theory. Fixed point considerations for infinitary languages generated by grammars which may be left recursive (in other words, which do not satisfy the Greibach condition) are discussed for instance by Niwinski [36].

A number of elementary properties of $\mathcal{O}[\![\cdot]\!]$ are collected in the following lemma.

3.7. Lemma. (1) $\mathcal{O}[\![E]\!] = \{\varepsilon\}$.
(2) $\mathcal{O}[\![a;r]\!] = a\cdot\mathcal{O}[\![r]\!]$.
(3) $\mathcal{O}[\![(s_1;s_2);r]\!] = \mathcal{O}[\![s_1;(s_2;r)]\!]$.
(4) $\mathcal{O}[\![(s_1\cup s_2);r]\!] = \mathcal{O}[\![s_1;r]\!]\cup O[\![s_2;r]\!]$.
(5) $\mathcal{O}[\![\mu x[s];r]\!] = \mathcal{O}[\![s[\mu x[s]/x];r]\!]$.

Remark. This lemma presupposes the formal definition of operations on (sets of) streams to be given in Definition 3.12.

Proof of Lemma 3.7. Obvious from the definitions. □

3.3. Denotational semantics

By way of preparation for the denotational semantics for $\mathcal{L}_{\text{us}}$, we present some basic definitions which introduce the metric setting we apply for this purpose.

3.8. Definition. We define the distance $d : A^{st} \times A^{st} \to [0, 1]$ by

$$d(u, v) = 2^{-\sup\{n \mid u(n) = v(n)\}},$$

where $2^{-\infty} = 0$.

Examples. $d(a_1a_2a_3, a_1a_2a_4) = 2^{-2}$; $d(a^n, a^\omega) = 2^{-n}$; $d(\varepsilon, \perp) = 1$.

3.9. Lemma. (1) *(A^{st}, d) is a complete metric space.*

(2) *For finite A, (A^{st}, d) is compact.*

Proof. See, e.g., [35]. □

Let $\mathcal{P}_{nc}(A^{st})$ denote the collection of all nonempty closed subsets of A^{st}. We usually abbreviate $\mathcal{P}_{nc}(A^{st})$ to S_{nc}. Let X, Y range over S_{nc}. We put $X(n) = \{u(n) \mid u \in X\}$. Now we also define a distance $\hat{d}$ on S_{nc}.

3.10. Definition. The distance $\hat{d} : S_{nc} \times S_{nc} \to [0, 1]$ is defined by

$$\hat{d}(X, Y) = 2^{-\sup\{n \mid X(n) = Y(n)\}},$$

where, again, $2^{-\infty} = 0$.

We have the following important theorem.

3.11. Theorem. (1) *$(S_{nc}, \hat{d})$ is a complete metric space, and if A is finite, this space is compact.*

(2) *$\hat{d}$ coincides with the Hausdorff distance (cf. Definition 2.7) induced on S_{nc} by the distance d on streams.*

Proof. Part (2) is easy from the definitions, and part (1) then follows from Theorem 2.8 (together with a theorem that says that compactness also carries over from any M to $\mathcal{P}_{cl}(M)$, see [25, 26]). The omission of the empty subset, which has distance 1 to every other subset does not disturb closedness or compactness. □

Remark. As a consequence of part (1) of Theorem 3.11, each Cauchy sequence $(X_n)_n$ in $(S_{nc}, \hat{d})$ has a limit $\lim_n X_n$ in $(S_{nc}, \hat{d})$, a fact we shall employ several times below.

Next we introduce three semantic operators "$\cdot$", "$\cup$", and "$\|$", which are counterparts of the syntactic operators of sequential composition, choice and parallel execution. The first two are well-known; the $\|$-operator (when applied to two sets) consists of the *shuffle* of all streams in the two operands. As remarked before, no operations involving synchronization or communication are considered for this language. The precise definition of the semantic operators proceeds in stages.

3.12. Definition (*Semantic operators*). (1) We assume as known the operation "·" of prefixing an element $a \in A$ to a finite stream $u \in A^*$, yielding as a result $a \cdot u$ (also written as au). Moreover, we put $a \cdot \langle u, \perp \rangle = \langle au, \perp \rangle$ for $u \in A^*$.

(2) Assume $X, Y \subseteq A^* \cup (A^* \times \{\perp\})$. We define

(a) $a \cdot X = \{au \mid u \in X\}$;

(b) for $u \in A^* \cup (A^* \times \{\perp\})$, we define $u \cdot X$ by induction on the length of u, as follows: $\varepsilon \cdot X = X$, $\perp \cdot X = \{\perp\}$, $(au) \cdot X = a \cdot (u \cdot X)$;

(c) $X \cdot Y = \bigcup \{u \cdot Y \mid u \in X\}$;

(d) $X \cup Y$ is (indeed) the set-theoretic union of X and Y;

(e) $u \mathbin{\lfloor\!\lfloor} W$ (which will be used in (2)(f) is defined by induction on the length of u, as follows: $\varepsilon \mathbin{\lfloor\!\lfloor} X = X$, $\perp \mathbin{\lfloor\!\lfloor} X = \{\perp\}$, $(au) \mathbin{\lfloor\!\lfloor} X = a \cdot (\{u\} \parallel X)$;

(f) $X \parallel Y = (X \mathbin{\lfloor\!\lfloor} Y) \cup (Y \mathbin{\lfloor\!\lfloor} X)$, where $X \mathbin{\lfloor\!\lfloor} Y = \bigcup \{u \mathbin{\lfloor\!\lfloor} X \mid u \in X\}$.

(3) Assume that X and Y are arbitrary elements of $\boldsymbol{S}_{\mathrm{nc}}$, and let $\mathbf{op} \in \{\cdot, \cup, \parallel\}$. Then we put

$$X \ \mathbf{op}\ Y = \lim_n (X(n) \ \mathbf{op}\ Y(n)).$$

3.13. Lemma. (1) *The operators* **op** *from* $\{\cdot, \cup, \parallel\}$ *are well-defined. In particular, for each* $X, Y \in \boldsymbol{S}_{\mathrm{nc}}$, $(X(n) \ \mathbf{op}\ Y(n))_n$ *is a Cauchy sequence.*

(2) *Each* **op** *is a continuous mapping:* $\boldsymbol{S}_{\mathrm{nc}} \times \boldsymbol{S}_{\mathrm{nc}} \to \boldsymbol{S}_{\mathrm{nc}}$.

Proof. Either by combining results from [11] with Rounds's theorem (Theorem 2.11), or by appropriately modifying the proof as given in [17, Appendix B]. □

We need one last step before we can give the definition for the denotational semantic function $\mathscr{D}[\![\cdot]\!]$. We shall restrict the definition of $\mathscr{D}[\![\cdot]\!]$ to statements involving only *guarded* recursion defined as follows.

3.14. Definition. (1) A statement variable x may occur *exposed* in a statement s. This notion is inductively defined as follows:

(a) x occurs exposed in x;

(b) if x occurs exposed in s, then x occurs exposed in $s;s'$, $s \cup s'$, $s' \cup s$, and $\mu y[s]$ for $y \neq x$.

(2) A statement s is called *guarded* when for each of its recursive substatements of the form $\mu x[s']$ we have that x does not occur exposed in s'. A program $t = s_1 \| \cdots \| s_n$ is called guarded if all its constituents s_i are guarded.

Examples. The statements $\mu x[a;x]$ and $\mu x[\mu y[b;y];x]$ are guarded, whereas $\mu x[(x;b) \cup c]$ and $\mu y[\mu x[y];b]$ are unguarded.

Let $\mathscr{S}^{\mathrm{g}}_{\mathrm{us}}$ denote the sets of guarded statements and $\mathscr{L}^{\mathrm{g}}_{\mathrm{us}}$ the set of guarded programs. We shall now define the mappings $\mathscr{D}$:

$$\mathscr{D}[\![\cdot]\!] : \mathscr{S}^{\mathrm{g}}_{\mathrm{us}} \to (\Gamma \to (SeCo \xrightarrow{\mathrm{NDI}} \boldsymbol{S}_{\mathrm{nc}}))$$

and

$$\mathscr{D}[\![\cdot]\!] : \mathscr{L}^{\mathrm{g}}_{\mathrm{us}} \to \boldsymbol{S}_{\mathrm{nc}}$$

where Γ is the set of *environments* and *SeCo* the set of *semantic continuations*, both to be defined below. (Recall from Definition 2.3 that $\to^{\mathrm{NDI}}$ stands for the set of all non-distance-increasing functions.) We take γ to range over Γ and φ to range over $SeCo \to^{\mathrm{NDI}} \boldsymbol{S}_{\mathrm{nc}}$. The type of especially the first $\mathscr{D}$ might require some explanation; it means that we apply the function $\mathscr{D}$ to a guarded statement, an environment, and a continuation in order to get an element from $\boldsymbol{S}_{\mathrm{nc}}$, i.e., a nonempty, closed set of streams.

The definition of the set *SeCo* of semantic continuations is simple: We just take

$$SeCo = \boldsymbol{S}_{\mathrm{nc}},$$

and use X, Y to range over *SeCo* as well. A semantic continuation denotes the semantics of the statements to be executed after the one to which $\mathscr{D}[\![\cdot]\!]$ is applied. To be more precise, when $\mathscr{D}$ is applied to a (guarded) statement s and an environment γ, we get a function $\varphi: SeCo \to^{\mathrm{NDI}} \boldsymbol{S}_{\mathrm{nc}}$. The interpretation of this function is as follows: if $X \in SeCo = \boldsymbol{S}_{\mathrm{nc}}$ is the semantics of a statement, say s', to be executed after s, then the semantics of s and s' together is given by $\varphi(X)$ (this is illustrated very well by part (1)(b) of Definition 3.15 below). At this point continuations may seem a complicated way of doing a simple thing (concatenating sequences), but in later sections we shall see that the technique of continuations enables denotational semantics to do in a simple way things that otherwise require quite an effort.

There are two reasons to require the function φ to be non-distance-increasing: The technical reason is that we want Lemma 3.16 below to hold. The intuitive reason has to do with the fact that such a function φ will not have the opportunity to analyse its argument in detail and make decisive choices based on that analysis, but it will just concatenate the argument to the end of some set of streams, possibly (in later sections) interleaving it with yet another set of streams. This kind of operation will "shift" the argument "to the future", and due to the nature of the metric on $\boldsymbol{S}_{\mathrm{nc}}$, this means that the distance between $\varphi(X)$ and $\varphi(Y)$ will possibly be smaller than the distance between X and Y, but definitely not greater.

For the set of environments we use

$$\Gamma = StmV \to (SeCo \xrightarrow{\mathrm{NDI}} \boldsymbol{S}_{\mathrm{nc}}).$$

An environment gives a meaning to each statement variable. In more conventional languages, which use procedure declarations where we use the μ-construct, the meaning of such a set of declarations would be recorded in an environment γ, which is subsequently used to interpret the procedure calls in the statements after the declarations. Our recursive construct effectively combines a declaration and a call of a "procedure", named with a statement variable. Therefore the statement s within the recursive construct $\mu x[s]$ will be interpreted with respect to an environment different from the one used in interpreting the recursive construct, where the difference lies in the meaning assigned to the statement variable x (see equation (3.1) below).

We are now sufficiently prepared for the following definition.

3.15. Definition (*Denotational semantics for $\mathcal{S}_{us}$ and $\mathcal{L}_{us}$*). (1) Assume that $s \in \mathcal{S}_{us}$ is guarded. We define $\mathcal{D}[\![s]\!]$ by structural induction on s:

(a) $\mathcal{D}[\![a]\!](\gamma)(X) = a \cdot X$,

(b) $\mathcal{D}[\![s_1 ; s_2]\!](\gamma)(X) = \mathcal{D}[\![s_1]\!](\gamma)(\mathcal{D}[\![s_2]\!](\gamma)(X))$,

(c) $\mathcal{D}[\![s_1 \cup s_2]\!](\gamma)(X) = \mathcal{D}[\![s_1]\!](\gamma)(X) \cup \mathcal{D}[\![s_2]\!](\gamma)(X)$,

(d) $\mathcal{D}[\![x]\!](\gamma)(X) = \gamma(x)(X)$,

(e) $\mathcal{D}[\![\mu x[s]]\!](\gamma)(X) = \varphi_\infty(X)$ where φ_∞ is the unique fixed point of the operator $\Phi : (SeCo \to^{NDI} \boldsymbol{S}_{nc}) \to (SeCo \to^{NDI} \boldsymbol{S}_{nc})$ given by $\Phi(\varphi) = \mathcal{D}[\![s]\!](\gamma\{\varphi/x\})$. (We use the variant notation $\gamma\{\varphi/x\}$ introduced in Section 2.1.)

(2) For $t = s_1 \| \cdots \| s_n$, t guarded, we put

$$\mathcal{D}[\![t]\!] = \mathcal{D}[\![s_1]\!](\gamma)(\{\varepsilon\}) \| \cdots \| \mathcal{D}[\![s_n]\!](\gamma)(\{\varepsilon\})$$

where γ is arbitrary (and we assume the obvious associativity of "$\|$").

The definition in clause (1)(e) is justified by the following lemma.

3.16. Lemma. *If s is guarded and x does not occur exposed in s, then we have that the operator Φ defined by $\Phi = \lambda\varphi.\mathcal{D}[\![s]\!](\gamma\{\varphi/x\})$ is* contracting.

Proof. Induction on the complexity of s, using the condition on x. □

By Banach's theorem (Theorem 2.5), the operator Φ in Definition 3.15(1)(e) indeed has a unique fixed point φ_∞. In particular, for the meaning of $\mu x[s]$ we have the familiar fixed point relation (for each γ):

$$\varphi_\infty = \mathcal{D}[\![\mu x[s]]\!](\gamma) = \mathcal{D}[\![s]\!](\gamma\{\varphi_\infty/x\}). \tag{3.1}$$

Note furthermore that $\varphi_\infty = \lim_i \varphi_i$, where φ_0 can be chosen arbitrarily and the rest of the sequence is given by $\varphi_{i+1} = \mathcal{D}[\![s]\!](\gamma\{\varphi_i/x\})$.

3.4. Equivalence of operational and denotational semantics

After having defined both $\mathcal{O}$ and $\mathcal{D}$ for (guarded elements of) $\mathcal{S}_{us}$ and $\mathcal{L}_{us}$, we next discuss the relationship between the two semantics. We shall in fact establish that, for t guarded,

$$\mathcal{O}[\![t]\!] = \mathcal{D}[\![t]\!]. \tag{3.2}$$

We need some technical properties of $\mathcal{O}$ which will play a role in the inductive argument to prove (3.2). A very detailed treatment of variants of these results can be found in [16] (variants stemming from the fact that the latter deals with nested parallelism as well). Therefore, we state the results here without proof.

3.17. Lemma. (1) $\mathcal{O}[\![s;r]\!] = \mathcal{O}[\![s;E]\!] \cdot \mathcal{O}[\![r]\!]$.

(2) $\mathcal{O}[\![r_1, r_2]\!] = \mathcal{O}[\![r_1]\!] \| \mathcal{O}[\![r_2]\!]$.

For the statement of the next theorem we need some further notation: Consider a recursive construct $\mu x[s]$. Let Ω be a new elementary action, i.e., $\Omega \notin A$. (This is

the only place where we find it convenient to distinguish a syntactic elementary action (Ω) from the corresponding semantic one ($\perp$).) Ω will play a role only in connection with Theorem 3.18 below. We first introduce a corresponding axiom (extending the list of transition axioms in Definition 3.4):

$$\langle \ldots, \Omega; r, \ldots, w\rangle \to w. \qquad \textbf{Undef}$$

(Recall that $w \in A^* \times \{\perp\}$. Thus, **Undef** is an axiom which terminates the computation with an unfinished stream.) Moreover, for each $n \geq 0$, s, and x, we introduce the notation $s_x^{(n)}$ given by

$$s_x^0 = \Omega, \qquad s_x^{(n+1)} = s[s_x^{(n)}/x].$$

The following theorem is proved in [16].

3.18. Theorem. *Assume that $\mu x[s]$ is closed and guarded. Then we have*

$$\mathcal{O}[\![\mu x[s]; r]\!] = \lim_n \mathcal{O}[\![s_x^{(n)}; r]\!].$$

Proof. See the argument in [16], which involves an elaborate development of auxiliary tools. □

Theorem 3.18 is in fact crucial for the proof of (3.2). We shall prove (3.2) in a way that anticipates the strategy followed in the next section where we deal with $\mathcal{L}_{\mathrm{ud}}$. Our reason for doing this is our wish to pinpoint the places where the proof of the dynamic case is essentially more involved than that of the static case.

In order to prove (3.2), we first prove a more general result, and then obtain (3.2) as a direct corollary.

3.19. Theorem. *Let s be guarded but not necessarily closed, and let the set of free statement variables of s be contained in $\{x_1, \ldots, x_m\}$, $m \geq 0$. Let $s_1, \ldots, s_m$ be closed and guarded statements, let $\tilde{s} = s[s_i/x_i]_{i=1}^m$, and let, for any r, $\mathcal{O}[\![r]\!]$ be short for $\lambda X.(\mathcal{O}[\![r]\!] \cdot X)$. Let furthermore*

$$\varphi_i = \mathcal{O}[\![s_i; E]\!]$$

for $i = 1, \ldots, m$, and let $\tilde{\gamma} = \gamma\{\varphi_i/x_i\}_{i=1}^m$. Then we have

$$\mathcal{O}[\![\tilde{s}; E]\!] = \mathcal{D}[\![s]\!](\tilde{\gamma}).$$

Proof. Induction on the complexity of s. We treat three representative cases:

Case 1: $s = x_i$. Then $\mathcal{O}[\![\tilde{s}; E]\!] = \mathcal{O}[\![s_i; E]\!] = \varphi_i = \mathcal{D}[\![x_i]\!](\tilde{\gamma})$.

Case 2: $s = s'; s''$. Now the free statement variables of s' and s'' are also among $\{x_1, \ldots, x_m\}$. We can write $\tilde{s}' = s'[s_i/x_i]_{i=1}^m$ and similarly for s''. Then we get

$$\begin{aligned} \mathcal{O}[\![\tilde{s}; E]\!] &= \mathcal{O}[\![(\tilde{s}'; \tilde{s}''); E]\!] \\ &= \mathcal{O}[\![\tilde{s}'; (\tilde{s}''; E)]\!] \qquad \text{(Lemma 3.7)} \end{aligned}$$

$$= \lambda X.\mathcal{O}[\![\tilde{s}';(\tilde{s}'';E)]\!] \cdot X$$

$$= \lambda X.(\mathcal{O}[\![s';E]\!] \cdot (\mathcal{O}[\![\tilde{s}'';E]\!] \cdot X)) \quad \text{(Lemma 3.17 and associativity of "}\cdot\text{")}$$

$$= \lambda X.(\mathcal{O}[\![\tilde{s}';E]\!](\mathcal{O}[\![\tilde{s}'';E]\!](X)))$$

$$= \lambda X.(\mathcal{D}[\![s']\!](\tilde{\gamma})(\mathcal{D}[\![s'']\!](\tilde{\gamma})(X))) \quad \text{(twice the induction hypothesis)}$$

$$= \mathcal{D}[\![s';s'']\!](\tilde{\gamma}).$$

Case 3: $s = \mu y[s']$. Let us first remark that from the conditions on s and $s_1, \ldots, s_m$ it follows that $\tilde{s}$ is guarded. We define $\tilde{s}' = s'[s_i/x_i]_{i=1}^m$ (note that y may still be free in $\tilde{s}'$). Now we have on the one hand

$$\mathcal{O}[\![\tilde{s};E]\!] = \lambda X.(\mathcal{O}[\![\tilde{s};E]\!] \cdot X)$$

$$= \lambda X.\lim_n(\mathcal{O}[\![\tilde{s}_y'^{(n)};E]\!] \cdot X) \quad \text{(Theorem 3.18 and continuity of "}\cdot\text{")}$$

$$= \lim_n(\mathcal{O}[\![\tilde{s}_y'^{(n)};E]\!]).$$

On the other hand, we have $\mathcal{D}[\![s]\!](\tilde{\gamma}) = \lim_n \psi_n$, where ψ_0 can be chosen freely and $\psi_{n+1} = \mathcal{D}[\![s']\!](\tilde{\gamma}\{\psi_n/y\})$. Our choice for ψ_0 will be $\psi_0 = \lambda X.\{\perp\}$. We prove, by induction on n, that

$$\mathcal{O}[\![\tilde{s}_y'^{(n)};E]\!] = \psi_n. \tag{3.3}$$

The case $n = 0$ is clear. Now assume (3.3) as induction hypothesis. Then

$$\mathcal{O}[\![\tilde{s}_y'^{(n+1)};E]\!] = \mathcal{O}[\![s'[s_i/x_i]_{i=1}^m[\tilde{s}_y'^{(n)}/y];E]\!]$$

$$= \mathcal{D}[\![s']\!](\gamma\{\varphi_i/x_i\}_{i=1}^m\{\psi_n/y\}) = \mathcal{D}[\![s']\!](\tilde{\gamma}\{\psi_n/y\}) = \psi_{n+1}.$$

Here we have used the main induction hypothesis with s' replacing s, $m+1$ replacing m, and $s_1, \ldots, s_m, \tilde{s}_y'^{(n)}$ replacing $s_1, \ldots, s_m$. In order for the main induction hypothesis to apply we have to establish that $\mathcal{O}[\![\tilde{s}_y'^{(n)};E]\!] = \psi_n$, which is nothing but our nested induction hypothesis (3.3).

Now that we have proved (3.3) for all n, it is evident that $\mathcal{O}[\![\tilde{s};E]\!] = \mathcal{D}[\![s]\!](\tilde{\gamma})$, which proves the most difficult part of the theorem. □

3.20. Corollary. *For guarded t we have $\mathcal{O}[\![t]\!] = \mathcal{D}[\![t]\!]$.*

Proof. For any closed and guarded s, and any γ, we have, by the previous theorem, that $\mathcal{O}[\![s;E]\!] = \mathcal{D}[\![s]\!](\gamma)$. Hence, $\mathcal{O}[\![s;E]\!] = \mathcal{O}[\![s;E]\!](\{\varepsilon\}) = \mathcal{D}[\![s]\!](\gamma)(\{\varepsilon\})$. If $t = s_1 \| \cdots \| s_n$, we therefore obtain

$$\mathcal{O}[\![t]\!] = \mathcal{O}[\![s_1;E, \ldots, s_n;E]\!] = \mathcal{O}[\![s_1;E]\!] \| \cdots \| \mathcal{O}[\![s_n;E]\!]$$

$$= \mathcal{D}[\![s_1]\!](\gamma)(\{\varepsilon\}) \| \cdots \| \mathcal{D}[\![s_n]\!](\gamma)(\{\varepsilon\}) = \mathcal{D}[\![t]\!]. \qquad \square$$

We conclude this section with a remark on possible other models for $\mathcal{L}_{us}$. Besides the operational and metric denotational (linear time) models for $\mathcal{L}_{us}$, we have also developed several other models which have been described elsewhere:

(1) A denotational semantics based on a cpo structure on (certain) sets of streams equipped with the Smyth order [12, 14, 33, 34].
(2) A denotational semantics based on a cpo structure on (certain) sets of so-called finite observations equipped with the order of reverse set inclusion [12, 14].
(3) A branching time denotational semantics based on a process domain of the kind described in Section 2.3 [11].

The equivalence of the models in (1) and (2) has been established in [14], the equivalence of the model in (1) and the denotational metric model is proved in [13], and the relationship between the branching time model and (any of) the linear time models is settled in [11].

4. A uniform and dynamic language

We now turn our attention to a language with *process creation.* In this section we study the uniform version of this phenomenon as couched in the language $\mathscr{L}_{ud}$. In Section 5 we shall investigate a nonuniform generalization.

A substantial part of the semantic theory for $\mathscr{L}_{us}$ can be carried over to the present case. Thus, we can be much shorter in our definitions. The main equivalence result also closely follows the approach from Section 3, but for one important new problem which requires nontrivial additional analysis.

4.1. Syntax and intuitive explanation

We start with the following definition.

4.1. Definition (*Syntax for statements and programs*). (1) Let s range over the set $\mathscr{S}_{ud}$ of (uniform and dynamic) *statements*:

$$s ::= a \mid x \mid s_1;s_2 \mid s_1 \cup s_2 \mid \mu x[s'] \mid \mathbf{new}(s').$$

(2) Let t range over the set $\mathscr{L}_{ud}$ of (uniform and dynamic) *programs*:

$$t ::= s$$

Here we require again that s is closed. Thus, a program in $\mathscr{L}_{ud}$ is simply a closed statement from $\mathscr{S}_{ud}$.

The intuitive operational semantics for t or s may be described in terms of a dynamically growing number of processes which execute statements in parallel in the following manner:

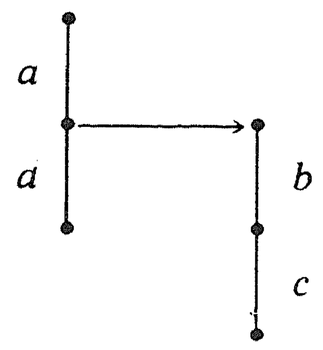
(1) Set an auxiliary variable i to 1, and set s_1 to s, the program to be executed. A process, numbered 1, is created to execute this s_1.

(2) Processes 1 to i are executed in parallel. Process j executes s_j $(1 \leqslant j \leqslant i)$ in the usual way (see Section 3) if s_j begins with an elementary action, sequential composition, choice, or a recursive construct. For example, if s_j begins with an elementary action a, then this a is appended to the output word, and s_j is set to its (syntactic) continuation (the part after this atomic action).

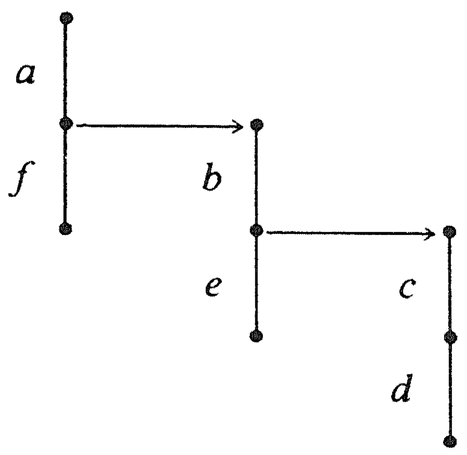
(3) If some process j $(1 \leqslant j \leqslant i)$ has to execute a statement of the form **new**(s'), then the following happens: The variable i is set to $i+1$, then s_i is set to s', and a new process, with number i, is created to execute s_i. Process j will continue to execute the part after the **new**-statement (s_j is set to its continuation). Go back to step (2).

(4) Execution terminates when there is no process left with a nonempty continuation.

Examples. (1) The statement a;**new**$(b;c)$;d determines the execution as suggested by the following picture (where the arrow denotes creation of a new process):



(2) The statement a;**new**$(b$;**new**$(c;d);e);f$ determines the execution as suggested by the diagram:



4.2. Operational and denotational semantics

The above intuitive explanation would clearly benefit from a more formal description, and this will be the main content of the present section.

We first develop the operational semantics for $\mathscr{L}_{\mathrm{ud}}$. We profit from the preparatory work in Section 3, and assume the general framework as described there. Also, configurations $\langle \rho, w \rangle$ or simply w' (with $w \in A^* \times \{\perp\}$, $w' \in A^*$) are as before, except that the statements s in such a parallel syntactic continuation ρ (see Definition 3.3) should now belong to $\mathscr{S}_{\mathrm{ud}}$ instead of $\mathscr{S}_{\mathrm{us}}$. The transition relation "$\rightarrow$" is now defined as the smallest relation satisfying the axioms in the following definition.

4.2. Definition (*Transition system for* $\mathscr{L}_{\mathrm{ud}}$). The transition system $\mathscr{T}_{\mathrm{ud}}$ for $\mathscr{L}_{\mathrm{ud}}$ consists

of all the axioms of Definition 3.4 (i.e., of all of $\mathcal{T}_{us}$), and in addition the axiom

$$\langle \ldots, \textbf{new}(s); r, \ldots, w\rangle \to \langle \ldots, r, \ldots, s; E, w\rangle. \qquad \textbf{New}$$

Here on the left-hand side we have a parallel syntactic continuation ρ with, say, $n \geq 1$ components and $\textbf{new}(s);r$ as the ith component (for some i, $1 \leq i \leq n$). On the right-hand side we have the parallel syntactic continuation ρ' with $n+1$ components, r as the ith component and $s;E$ as the $(n+1)$st component (and no changes with respect to ρ in the remaining components).

The definition of $\mathcal{O}[\![\rho]\!]$ is as before, but now with respect to transition system $\mathcal{T}_{ud}$. Also, since each $t \in \mathcal{L}_{ud}$ equals some $s \in \mathcal{S}_{ud}$, we simply put, for $t = s$, $\mathcal{O}[\![t]\!] = \mathcal{O}[\![s;E]\!]$.

Example. Take $t = a;\textbf{new}(b;\textbf{new}(c);e);f$. Then $\mathcal{O}[\![t]\!] = \{afbce, abfce, abcfe, abcef, afbec, abfec, abefc, abecf\}$.

The elementary properties of $\mathcal{O}$ listed in Lemma 3.7 remain valid. In addition, we have the following lemma.

4.3. Lemma. $\mathcal{O}[\![\textbf{new}(s);r]\!] = \mathcal{O}[\![r, s;E]\!]$.

Proof. Clear from the definitions. □

We proceed with the definitions for the denotational semantics for $\mathcal{S}_{ud}$ and $\mathcal{L}_{ud}$. A complication which arises is that the notion of a statement being guarded has to be refined. A typical case concerns a recursive construct such as $\mu x[\textbf{new}(a);x]$, where the elementary action a does not fulfil the duties of a guard: this construct may choose to start execution with the recursive call x. The precise definition of guardedness requires an amended definition of "x is exposed in s", and this involves, in turn, a notion of generalized **new**-statement.

4.4. Definition. (1) A *generalized* ***new*** *statement* g is defined by

$$g ::= \textbf{new}(s) \mid g_1;g_2 \mid g' \cup s \mid s \cup g' \mid \mu x[g']$$

(2) When a statement variable x occurs *exposed* in a statement $s \in \mathcal{S}_{ud}$ is defined inductively as follows:

(a) x occurs exposed in x;

(b) if x occurs exposed in s, then x occurs exposed in $s;s'$, $s \cup s'$, $s' \cup s$, $\mu y[s]$ (if $y \neq x$), $\textbf{new}(s)$, and in $g;s$.

(3) A statement $s \in \mathcal{S}_{ud}$ is called *guarded* if, for all its recursive substatements of the form $\mu x[s']$, s' contains no exposed occurrences of x.

We shall now give a denotational semantics for $\mathcal{L}_{ud}$ by defining

$$\mathcal{D}[\![\cdot]\!]: \mathcal{S}^{g}_{ud} \to (\Gamma \to (SeCo \xrightarrow{\text{NDI}} \mathbf{S}_{nc})) \quad \text{and} \quad \mathcal{D}[\![\cdot]\!]: \mathcal{L}^{g}_{ud} \to \mathbf{S}_{nc},$$

where we use Γ, *SeCo*, and $\boldsymbol{S}_{\mathrm{nc}}$ as in Section 3.3. (Analogously to Section 3.3, $\mathscr{S}^{\mathrm{g}}_{\mathrm{ud}}$ denotes the set of guarded statements, and $\mathscr{L}^{\mathrm{g}}_{\mathrm{ud}}$ the set of guarded programs.)

4.5. Definition. (1) For guarded $s \in \mathscr{S}_{\mathrm{ud}}$, s not of the form **new**(s'), we take over the clauses from Definition 3.15.

(2) For guarded s of the form **new**(s') we put

$$\mathscr{D}[\![\textbf{new}(s')]\!](\gamma)(X) = \mathscr{D}[\![s']\!](\gamma)(\{\varepsilon\}) \parallel X.$$

(3) For guarded $t \in \mathscr{L}_{\mathrm{ud}}$, $t = s$, we put $\mathscr{D}[\![t]\!] = \mathscr{D}[\![s]\!](\gamma)(\{\varepsilon\})$, where γ is arbitrary.

We see that the meaning of a new-construct **new**(s') in a situation that X remains to be done (i.e., with a semantic continuation X) is given by the result of putting X in parallel with the meaning of s' where nothing remains to be done after it (continuation $\{\varepsilon\}$).

Remark. It has been proved that the expressive power of $\mathscr{L}_{\mathrm{ud}}$ is essentially greater than that of $\mathscr{L}_{\mathrm{us}}$, in the sense that for each $t \in \mathscr{L}_{\mathrm{us}}$ there is a $t' \in \mathscr{L}_{\mathrm{ud}}$ such that $\mathscr{O}[\![t]\!] = \mathscr{O}[\![t']\!]$ (indeed, take $t' = t$), but not the other way around. (IJ.J. Aalbersberg and P. America, personal communication.)

4.3. Equivalence of operational and denotational semantics

We now address the question as to whether, for guarded t, $\mathscr{O}[\![t]\!] = \mathscr{D}[\![t]\!]$. We follow the line of reasoning as in Section 3. First, we again have this lemma.

4.6. Lemma. (1) *For all* $r_1, r_2 \in SyCo$ *we have* $\mathscr{O}[\![r_1, r_2]\!] = \mathscr{O}[\![r_1]\!] \parallel \mathscr{O}[\![r_2]\!]$.

(2) *If* $\mu x[s]$ *is closed and guarded, then* $\mathscr{O}[\![\mu x[s]; r]\!] = \lim_n \mathscr{O}[\![s_x^{(n)}; r]\!]$.

Proof. See the sources given with Lemma 3.17 and Theorem 3.18. □

The next step in the argument concerns the analogue of Lemma 3.17(1) (and, somewhat more hidden, the way in which $\mathscr{O}[\![\cdot]\!]$ is defined, cf. Theorem 3.19). Let us see whether we may expect that $\mathscr{O}[\![s; r]\!] = \mathscr{O}[\![s; E]\!] \cdot \mathscr{O}[\![r]\!]$. It is easy to see that this is not the case by taking, for example, $s =$ **new**(a) and $r = b; E$. Then the left-hand side equals $\{ab, ba\}$ and the right-hand side equals $\{ab\}$. On the other hand, taking $s = a$, $r = b; E$, we see that neither is it true in general that $\mathscr{O}[\![s; r]\!] = \mathscr{O}[\![s; E]\!] \parallel \mathscr{O}[\![r]\!]$. What we need here (and in the definition of $\mathscr{O}[\![\cdot]\!]$) is an operator which, as it were, is able to decide dynamically whether the operation at hand is of a sequential or of a parallel character.

Having pinpointed the problem which distinguishes the situation in the current section from that in Section 3, we develop some additional tools and associated lemmas in such a way that eventually we shall be able to adopt the same style of argument for the main equivalence result as used in Section 3.

We shall introduce the *semantic operator* ":", which should clearly be distinguished from both "·" and "‖". The definition of ":" requires the introduction of an auxiliary elementary action, not belonging to $A \cup \{\perp\}$, and denoted by $\surd$. Its intuitive function is to mark the termination of a local process and (thus) to indicate where a continuation should start. We shall put $A' = A \cup \{\surd\}$, and introduce the extended stream set A^{est} as

$$A^{\text{est}} = A^{\text{st}} \cup \{w_1 \surd w_2 \mid w_1 \in A^*, w_2 \in A^{\text{st}}\}.$$

We now define the operator ":" as follows.

4.7. Definition. We shall put $S'_{\text{nc}} = \mathscr{P}_{\text{nc}}(A^{\text{est}})$ (recall that $S_{\text{nc}} = \mathscr{P}_{\text{nc}}(A^{\text{st}})$).

(1) The operator ":": $A^{\text{est}} \times A^{\text{est}} \to S'_{\text{nc}}$ is given by

$$w:w' = \begin{cases} w_1 \cdot (w_2 \parallel w') & \text{if } w = w_1 \surd w_2, \\ \{w\} & \text{otherwise.} \end{cases}$$

(Note that w' could again contain an occurrence of $\surd$, which will behave as an ordinary elementary action with respect to "‖".)

(2) For $X, Y \in S'_{\text{nc}}$, X and Y with finite streams only, we put

$$X:Y = \bigcup \{u:v \mid u \in X, v \in Y\}.$$

(3) For arbitrary $X, Y \in S'_{\text{nc}}$, we put

$$X:Y = \lim_n (X(n):Y(n)).$$

An important technical lemma concerning the operator ":" is the following one.

4.8. Lemma. (1) *":" is continuous as a mapping* $A^{\text{est}} \times A^{\text{est}} \to S'_{\text{nc}}$ *and as a mapping* $S'_{\text{nc}} \times S'_{\text{nc}} \to S'_{\text{nc}}$.

(2) *Restricting the domain of* ":" *to* $S'_{\text{nc}} \times S_{\text{nc}}$ *will restrict its range to* S_{nc}, *or in other words,* ":": $S'_{\text{nc}} \times S_{\text{nc}} \to S_{\text{nc}}$.

(3) $(X:Y):Z = X:(Y:Z)$, *for* $X, Y, Z \in S'_{\text{nc}}$.

(4) $\{w\surd\}:X = wX$, *for* $w \in A^{\text{st}}$, $X \in S'_{\text{nc}}$.

(5) $(X \cup Y):Z = (X:Z) \cup (Y:Z)$, *for* $X, Y, Z \in S'_{\text{nc}}$.

(6) $(X \parallel Y):Z = X \parallel (Y:Z)$, *for* $X \in S_{\text{nc}}$, $Y, Z \in S'_{\text{nc}}$.

Proof. We only prove part (3). Below, we shall prove that $(u:v):w = u:(v:w)$ for $u, v, w \in A^{\text{est}}$. Then we obtain, for X, Y, Z with finite streams only,

$$(X:Y):Z = \bigcup_{u \in X:Y} \bigcup_{w \in Z} (u:w) = \bigcup_{u_1 \in X} \bigcup_{u_2 \in Y} \bigcup_{w \in Z} ((u_1:u_2):w)$$

$$= \bigcup_{u_1 \in X} \bigcup_{u_2 \in Y} \bigcup_{w \in Z} (u_1:(u_2:w)) = X:(Y:Z).$$

For general X, Y, Z, we take the limit of $X(n):Y(n):Z(n)$.

We now prove that $(u:v):w = u:(v:w)$. If $u \in A^{st}$ (so that u has no occurrence of $\surd$) then $(u:v):w = \{u\} = u:(v:w)$, and if $v \in A^{st}$ then $(u:v):w = u:v = u:(v:w)$. Now suppose that $u = u_1 \surd u_2$ and $v = v_1 \surd v_2$. We prove two inclusions:

(1) $(u:v):w \subseteq u:(v:w)$. We have $u:v = u_1 \cdot (u_2 \| v)$, so $(u:v):w = \bigcup_{w' \in (u_2 \| v)} (u_1 w'):w$. Let $w' \in u_2 \| v$. We distinguish two subcases:

(a) $w' \in A^{st}$. This is only possible (since $v = v_1 \surd v_2$) if $u_2 \in A^\omega \cup (A^* \times \{\perp\})$. Then $w' \in u_2 \| v_1$, so $w' \in u_2 \| (v_1 \cdot (v_2 \| w)) = u_2 \| (v:w)$, and therefore $(u_1 w'):w = \{u_1 w'\} \subseteq u:(v:w)$.

(b) $w' = w_1' \surd w_2'$. Now there are u_{21}, u_{22} such that $u_2 = u_{21} u_{22}$, $w_1' \in u_{21} \| v_1$, $w_2' \in u_{22} \| v_2$. We obtain

$$(u_1 w'):w = u_1 w_1' (w_2' \| w) \subseteq u_1 (u_{21} \| v_1)(u_{22} \| v_2 \| w)$$

$$\subseteq u_1 (u_2 \| (v_1 (v_2 \| w))) = u:(v;w).$$

(2) $u:(v:w) \subseteq (u:v):w$. We have $u:(v:w) = u_1 \cdot (u_2 \| (v:w)) = \bigcup_{u' \in v:w} u_1 \cdot (u_2 \| u') = \bigcup_{v' \in v_2 \| w} u_1 \cdot (u_2 \| (v_1 v'))$. Now let $v' \in v_2 \| w$ and $w' \in u_2 \| (v_1 v')$. There are u_{21}, u_{22}, w_1', and w_2' such that $w' = w_1' w_2'$, $w_1' \in u_{21} \| v_1$, $w_2' \in u_{22} \| v'$. We have that

$$u_1 w_1' \surd (u_{22} \| v_2) \subseteq u_1 \cdot (u_2 \| v_1 \surd v_2) = u:v. \tag{4.1}$$

(The inclusion holds since $u_2 \| v_1 \surd v_2$ contains the set $(u_{21} \| v_1) \surd (u_{22} \| v_2)$, which in turn contains $w_1' \surd (u_{22} \| v_2)$.) We conclude that

$$u_1 w' = u_1 w_1' w_2' \in u_1 w_1' (u_{22} \| v_2 \| w) \subseteq (u:v):w,$$

where the last inclusion follows from (4.1) by postfixing both sides with "$:w$". □

We next show how the new operator ":" solves the problems described after Lemma 4.6. First we extend—for the remainder of this section—the definition of *SyCo* (cf. Definition 3.3), and now put

$$r ::= E \mid \surd \mid s;r'$$

We emphasize that the "elementary action" $\surd$ occurs only in syntactic continuations; the syntax for statements $s \in \mathcal{S}_{ud}$ is not modified. Before we can state and prove the equivalent of Lemma 3.17(1), we discuss the induced amendment of the transition system $\mathcal{T}_{ud}$. Firstly, all axioms of $\mathcal{T}_{ud}$ now refer to r (and ρ) which may involve $\surd$. Secondly, we extend $\mathcal{T}_{ud}$ with an axiom catering for $\surd$. In the present context, we need this axiom only in a restricted version:

$$\langle \ldots, \surd, \ldots, w\perp \rangle \to \langle \ldots, E, \ldots, w\surd\perp \rangle \qquad \textbf{Elem}'$$

where $w \in A^*$ and none of the continuations appearing at the dots (. . .) involves $\surd$. In other words, we restrict attention to parallel syntactic continuations ρ which involve at most one constituent syntactic continuation r ending in $\surd$. This is no real restriction since that property applies to all configurations in transition sequences which interest us: It holds trivially for ρ containing only one component, and it is preserved by applications of the axiom **New**, which creates new components.

We can now state the following lemma, which applies the technique of induction loading to prove Corollary 4.10.

4.9. Lemma. *Let $s \in \mathcal{S}_{ud}$ (not necessarily closed) and suppose that all the free variables in s are in $\{x_1, \ldots, x_k\}$. Now let $s_1, \ldots, s_k$ be closed and guarded and define $\tilde{s} = s[s_i/x_i]_{i=1}^{k}$. Suppose further that for $i = 1, \ldots, k$ and for any r we have*

$$\mathcal{O}[\![s_i;r]\!] = \mathcal{O}[\![s_i;\surd]\!]:\mathcal{O}[\![r]\!]$$

and that $\tilde{s}$ is guarded. Then we have for any r

$$\mathcal{O}[\![\tilde{s};r]\!] = \mathcal{O}[\![\tilde{s};\surd]\!]:\mathcal{O}[\![r]\!].$$

Proof. Induction on the complexity of s. We give full details of the proof, in order to exhibit its dependence on Lemma 4.8.

(1) If $s = a$, then $\tilde{s} = a$, so we get

$$\begin{aligned}
\mathcal{O}[\![\tilde{s};r]\!] = \mathcal{O}[\![a;r]\!] &= a \cdot \mathcal{O}[\![r]\!] && \text{(Lemma 3.7)}\\
&= \{a\surd\}:\mathcal{O}[\![r]\!] && \text{(Lemma 4.8(4))}\\
&= \mathcal{O}[\![a;\surd]\!]:\mathcal{O}[\![r]\!] = \mathcal{O}[\![\tilde{s};\surd]\!]:\mathcal{O}[\![r]\!].
\end{aligned}$$

(2) If $s = x_i$, then $\tilde{s} = s_i$ and the property follows from the assumption about s_i.

(3) If $s = s';s''$, then we get in an obvious way $\tilde{s} = \tilde{s}';\tilde{s}''$, so

$$\begin{aligned}
\mathcal{O}[\![\tilde{s};r]\!] &= \mathcal{O}[\![(\tilde{s}';\tilde{s}'');r]\!]\\
&= \mathcal{O}[\![\tilde{s}';(\tilde{s}'';r)]\!] && \text{(Lemma 3.7)}\\
&= \mathcal{O}[\![\tilde{s}';\surd]\!]:\mathcal{O}[\![\tilde{s}'';r]\!] && \text{(ind. hyp. for } s')\\
&= \mathcal{O}[\![\tilde{s}';\surd]\!]:(\mathcal{O}[\![\tilde{s}'';\surd]\!]:\mathcal{O}[\![r]\!]) && \text{(ind. hyp. for } s'')\\
&= (\mathcal{O}[\![\tilde{s}';\surd]\!]:\mathcal{O}[\![\tilde{s}'';\surd]\!]):\mathcal{O}[\![r]\!] && \text{(Lemma 4.8(3))}\\
&= \mathcal{O}[\![\tilde{s}';(\tilde{s}'';\surd)]\!]:\mathcal{O}[\![r]\!] && \text{(ind. hyp. for } s')\\
&= \mathcal{O}[\![(\tilde{s}';\tilde{s}'');\surd]\!]:\mathcal{O}[\![r]\!]\\
&= \mathcal{O}[\![\tilde{s};\surd]\!]:\mathcal{O}[\![r]\!].
\end{aligned}$$

(4) If $s = s' \cup s''$, then, again, $\tilde{s} = \tilde{s}' \cup \tilde{s}''$ and we get

$$\begin{aligned}
\mathcal{O}[\![s;r]\!] &= \mathcal{O}[\![(\tilde{s}' \cup \tilde{s}'');r]\!]\\
&= \mathcal{O}[\![\tilde{s}';r]\!] \cup \mathcal{O}[\![\tilde{s}'';r]\!] && \text{(Lemma 3.7)}\\
&= (\mathcal{O}[\![\tilde{s}';\surd]\!]:\mathcal{O}[\![r]\!]) \cup (\mathcal{O}[\![\tilde{s}'';\surd]\!]:\mathcal{O}[\![r]\!]) && \text{(ind. hyp. for } s', s'')\\
&= (\mathcal{O}[\![\tilde{s}';\surd]\!] \cup \mathcal{O}[\![\tilde{s}'';\surd]\!]):\mathcal{O}[\![r]\!] && \text{(Lemma 4.8(5)}
\end{aligned}$$

$$= \mathcal{O}[\![(\tilde{s}' \cup \tilde{s}'');\surd]\!]:\mathcal{O}[\![r]\!]$$
$$= \mathcal{O}[\![\tilde{s};\surd]\!]:\mathcal{O}[\![r]\!].$$

(5) If $s = \mathbf{new}(s')$, we get $\tilde{s} = \mathbf{new}(\tilde{s}')$ and then

$$\mathcal{O}[\![\tilde{s};\, r]\!] = \mathcal{O}[\![\mathbf{new}(\tilde{s}');r]\!]$$
$$= \mathcal{O}[\![\tilde{s}';E]\!] \| \mathcal{O}[\![r]\!] \qquad \text{(Lemmas 4.3 and 4.6(1))}$$
$$= (\mathcal{O}[\![\tilde{s}';E]\!] \| \{\surd\}):\mathcal{O}[\![r]\!] \quad (*)$$
$$= (\mathcal{O}[\![\tilde{s}';E]\!] \| \mathcal{O}[\![\surd]\!]):\mathcal{O}[\![r]\!]$$
$$= \mathcal{O}[\![\mathbf{new}(\tilde{s}');\surd]\!]:\mathcal{O}[\![r]\!]$$
$$= \mathcal{O}[\![\tilde{s};\surd]\!]:\mathcal{O}[\![r]\!].$$

Here, at the place marked $(*)$, we have used $(X \| \{\surd\})\colon Z = X \| Z$ if $X \in \boldsymbol{S}_{\mathrm{nc}}$, $Z \in \boldsymbol{S}'_{\mathrm{nc}}$; this is a special case of Lemma 4.8(6) together with Lemma 4.8(4).

(6) Let $s = \mu x[s']$. Suppose (without loss of generality) that $x \notin \{x_1, \ldots, x_k\}$. Put $\tilde{s}' = s'[s_i/x_i]_{i=1}^{k}$, so that $\tilde{s} = \mu x[\tilde{s}']$. Then we have by Lemma 4.6(2)

$$\mathcal{O}[\![\tilde{s};r]\!] = \mathcal{O}[\![\mu x[\tilde{s}'];r]\!] = \lim_n \mathcal{O}[\![\tilde{s}'^{(n)}_x;\, r]\!].$$

Now we shall prove in a minute that

$$\mathcal{O}[\![\tilde{s}'^{(n)}_x;\, r']\!] = \mathcal{O}[\![\tilde{s}'^{(n)}_x;\surd]\!]:\mathcal{O}[\![r']\!] \tag{4.2}$$

for all n and for all r'. Once we have proved this, we can calculate

$$\mathcal{O}[\![\tilde{s};r]\!] = \lim_n \mathcal{O}[\![\tilde{s}'^{(n)}_x;r]\!] \qquad \text{(Lemma 4.6(2))}$$
$$= \lim_n(\mathcal{O}[\![\tilde{s}'^{(n)}_x;\surd]\!]:\mathcal{O}[\![r]\!]) \qquad \text{(property (4.2))}$$
$$= (\lim_n \mathcal{O}[\![\tilde{s}'^{(n)}_x;\surd]\!]):\mathcal{O}[\![r]\!] \qquad \text{(continuity of ":")}$$
$$= \mathcal{O}[\![\tilde{s};\surd]\!]:\mathcal{O}[\![r]\!] \qquad \text{(Lemma 4.6(2)),}$$

which is what we wanted.

We still have to do the proof of property (4.2), which runs by an induction on n (nested within our original induction on the complexity of s). For the case $n = 0$, we have $\tilde{s}'^{(0)}_x = \Omega$, so $\mathcal{O}[\![\tilde{s}'^{(0)}_x;r']\!] = \bot = \bot:\mathcal{O}[\![r']\!] = \mathcal{O}[\![\tilde{s}'^{(0)}_x;\surd]\!]:\mathcal{O}[\![r']\!]$.

For the induction step we assume that property (4.2) holds for a certain value of n. Then we can apply the main induction hypothesis for $k+1$ to s' with $x_1, \ldots, x_{k+1} = x_1, \ldots, x_k, x$ and $s_1, \ldots, s_{k+1} = s_1, \ldots, s_k, \tilde{s}'^{(n)}_x$ in order to get

$$\mathcal{O}[\![\tilde{s}'^{(n+1)}_x;r']\!] = \mathcal{O}[\![\tilde{s}'[\tilde{s}'^{(n)}_x/x];r']\!]$$
$$= \mathcal{O}[\![s'[s_i/x_i]_{i=1}^{k+1};r']\!]$$
$$= \mathcal{O}[\![s'[s_i/x_i]_{i=1}^{k+1};\surd]\!]:\mathcal{O}[\![r']\!]$$
$$= \mathcal{O}[\![\tilde{s}'^{(n+1)}_x;\surd]\!]:\mathcal{O}[\![r']\!]. \qquad \square$$

4.10. Corollary. *For closed and guarded* s, $\mathcal{O}[\![s;r]\!] = \mathcal{O}[\![s;\surd]\!]:\mathcal{O}[\![r]\!]$.

We are, at last, sufficiently prepared for the main theorem of this section.

4.11. Theorem. *Let* $s \in \mathcal{S}_{ud}$, *not necessarily closed, and let the set of free statement variables of* s *be contained in* $\{x_1, \ldots, x_m\}$, $m \geq 0$. *Let* $s_1, \ldots, s_m$ *be closed and guarded statements, let* $\tilde{s} = s[s_i/x_i]_{i=1}^m$, *and define* $\mathcal{O}[\![r]\!]$ *by*

$$\mathcal{O}[\![E]\!] = \mathcal{O}[\![\surd]\!] = \lambda X.X, \qquad \mathcal{O}[\![s';r]\!] = \lambda X.(\mathcal{O}[\![s';\surd]\!]:\mathcal{O}[\![r]\!](X)).$$

Let furthermore $\varphi_i = \mathcal{O}[\![s_i;E]\!]$ *for* $i = 1, \ldots, m$, *and let* $\tilde{\gamma} = \gamma\{\varphi_i/x_i\}_{i=1}^m$. *Now if* $\tilde{s}$ *is also guarded, we have*

$$\mathcal{O}[\![\tilde{s};E]\!] = \mathcal{D}[\![s]\!](\tilde{\gamma}).$$

Proof. Very similar to that of Theorem 3.19. We shall prove two cases of old statements plus the case of the new statement.

Case 1: $s = s';s''$

$$\begin{aligned}
&\mathcal{O}[\![\tilde{s};E]\!] \\
&\quad = \mathcal{O}[\![(\tilde{s}';\tilde{s}'');E]\!] \\
&\quad = \lambda X.(\mathcal{O}[\![(\tilde{s}';\tilde{s}'');\surd]\!]:\mathcal{O}[\![E]\!](X)) \\
&\quad = \lambda X.(\mathcal{O}[\![\tilde{s}';(\tilde{s}'';\surd)]\!]:X) && \text{(Lemma 3.7)} \\
&\quad = \lambda X.(\mathcal{O}[\![\tilde{s}';\surd]\!]:(\mathcal{O}[\![\tilde{s}'';\surd]\!]:X)) && \text{(Corollary 4.10 and Lemma 4.8(3))} \\
&\quad = \lambda X.\mathcal{O}[\![\tilde{s}';E]\!](\mathcal{O}[\![\tilde{s}'';E]\!](X)) \\
&\quad = \lambda X.\mathcal{D}[\![s']\!](\tilde{\gamma})(\mathcal{D}[\![s'']\!](\tilde{\gamma})(X)) && \text{(ind. hyp. for } s' \text{ and } s'') \\
&\quad = \mathcal{D}[\![s';s'']\!](\tilde{\gamma}) = \mathcal{D}[\![s]\!](\tilde{\gamma}).
\end{aligned}$$

Case 2: $s = \mu y[s']$. As in Theorem 3.19, let us define $\tilde{s}' = s'[s_i/x_i]_{i=1}^m$ and calculate

$$\mathcal{O}[\![\tilde{s};E]\!] = \lambda X.(\mathcal{O}[\![\tilde{s};\surd]\!]:X) = \lambda X.\lim_n(\mathcal{O}[\![\tilde{s}'^{(n)}_y;\surd]\!]:X) = \lim_n \mathcal{O}[\![\tilde{s}'^{(n)}_y;E]\!].$$

Here we have used Lemma 4.6(2) and the continuity of ":". From this point on the argument follows exactly the same lines as in Theorem 3.19.

Case 3: $s = \mathbf{new}(s')$.

$$\begin{aligned}
&\mathcal{O}[\![\widetilde{\mathbf{new}(s')};E]\!] \\
&\quad = \lambda X.(\mathcal{O}[\![\surd,\tilde{s}';E]\!]:X) \\
&\quad = \lambda X.((\{\surd\}\|\mathcal{O}[\![\tilde{s}';E]\!]):X) \\
&\quad = \lambda X.(\mathcal{O}[\![\tilde{s}';E]\!]\|X) && \text{(Lemma 4.8, parts (6) and (4))} \\
&\quad = \lambda X.((\mathcal{O}[\![\tilde{s}';\surd]\!]:\mathcal{O}[\![E]\!])\|X) && \text{(Corollary 4.10)} \\
&\quad = \lambda X.(\mathcal{O}[\![\tilde{s}';E]\!](\{\varepsilon\})\|X) && \text{(Lemma 3.7 and def. of } \mathcal{O}) \\
&\quad = \lambda X.(\mathcal{D}[\![s']\!](\tilde{\gamma})(\{\varepsilon\})\|X) && \text{(induction hypothesis)} \\
&\quad = \mathcal{D}[\![\mathbf{new}(s')]\!](\tilde{\gamma}) && \text{(Definition 4.5).} \quad \square
\end{aligned}$$

4.12. Corollary. *For guarded $t \in \mathcal{L}_{ud}$ we have $\mathcal{O}[\![t]\!] = \mathcal{D}[\![t]\!]$.*

Proof. Clear from Theorem 4.11. □

We have thus completed the semantic analysis of $\mathcal{L}_{ud}$, and are now ready for the generalization to the nonuniform case.

5. A nonuniform and static language

This section is devoted to the semantic definitions for a nonuniform and static language. The elementary actions are now interpreted, viz. as assignments and communication actions. However, for the moment we return to a static framework, and leave the treatment of the dynamic case to the next section.

5.1. Syntax

The nonuniform framework involves the introduction of three new syntactic classes:

- The set *IndV* of individual variables, with typical elements x, y. For *IndV* we take an infinite alphabet of variable names.
- The set *Exp* of expressions, with typical element e.
- The set *Test* of conditions, with typical element b.

We shall return to the syntax for expressions and conditions in a moment. Note that we have changed the notation with respect to Sections 3 and 4 in that we now use x, y for individual rather than statement variables. For the latter purpose we here use variables v ranging over *StmV*. (The nonuniform framework has no streams, so we can freely use the letters u, v, w.)

In the static case, a program will again be composed of n components $s_1, \ldots, s_n$. Contrary to the uniform case, we are also interested in the *identity* of, in general, the ith statement (or *process*, in a terminology used, e.g., in CSP [31, 32]), and we introduce for this purpose the set $I = \{1, 2, \ldots\}$ of indices, with i, j, k, l ranging over I. Typically, indices i, j will be used in communication statements of the form $i?x$ or $j!e$, denoting communication of two sorts: The first occurs, in general, in some process k and requires a value for the variable x from process i. The second occurs, say, in a process l and sends the current value α of the expression e to process j. In the case that $k = j$ and $l = i$ and, moreover, the communications *synchronize* in the usual sense, then the "handshake" communication can indeed take place, and the variable x takes the value α. Once more, this informal description requires formal definition, to be elaborated in the sequel.

The last syntactic set we need to introduce is that of (individual) *constants*. We shall not bother to make a distinction between syntactic constants and semantic (basic) values, and use the set V, with typical elements α, β, for both purposes.

We now define the syntax for $\mathcal{S}_{nus}$ and $\mathcal{L}_{nus}$ (and for *Exp*).

5.1. Definition. (1) Let e range over the set *Exp* of expressions:

$$e ::= x \mid \alpha \mid e_1 \textbf{ op } e_2 \mid \textbf{op } e'$$

(Here **op** stands for an arbitrary binary or unary operator. We prefer not to take the trouble to introduce general n-ary function symbols into our language.)

(2) We do not specify a syntax for the elements b of *Test*. We only require that their evaluation terminates and takes place without complications such as side-effects.

(3) Let s range over the set $\mathscr{S}_{\text{nus}}$ of nonuniform and static statements:

$$s ::= x := e \mid s_1 ; s_2 \mid v \mid \mu v[s'] \mid \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} \mid i?x \mid i!e$$

(4) Let t range over the set $\mathscr{L}_{\text{nus}}$ of nonuniform and static programs:

$$t ::= s_1 \| \cdots \| s_n \quad (n \geqslant 1).$$

We require that the statements $s_1, \ldots, s_n$ are closed and furthermore that every index i occurring in t actually corresponds to a component statement, i.e., $i \leqslant n$.

We see $\mathscr{L}_{\text{nus}}$ is similar to (classical) CSP (as in [31]). There are also important differences: the absence (in $\mathscr{L}_{\text{nus}}$) of guarded commands with communication in guards or features such as the distributed termination convention. On the other hand, $\mathscr{L}_{\text{nus}}$ has full recursion rather than only iteration. Compared with $\mathscr{L}_{\text{us}}$, we have simplified $\mathscr{L}_{\text{nus}}$ by dropping the "$\cup$" operator. Extension of the treatment below to cover "$\cup$" is not difficult and we leave it to the reader.

5.2. Operational semantics

We proceed with the development of the framework for the operational semantics for $\mathscr{L}_{\text{nus}}$. Syntactic continuations r are, as before, defined by

$$r ::= E \mid s; r'$$

where s is closed. Instead of parallel syntactic continuations ρ in the form of n-tuples $r_1, \ldots, r_n$, we now let ρ range over sets of the form

$$\{\langle i_1, r_1\rangle, \ldots, \langle i_n, r_n\rangle\} \quad (n \geqslant 1)$$

where all the indices $i_1, \ldots, i_n$ must be different. Thus, in the pair $\langle i, r\rangle$, we make explicit the identity of the component r. We shall not require that every index i occurring in a communication statement $i!e$ or $i?x$ within ρ also occurs as the first component of a pair $\langle i, r\rangle \in \rho$.

We shall often use the notation $\rho \cup \{\langle i, r_i\rangle\}$, with the convention that ρ is supposed not to contain an element of the form $\langle i, r'\rangle$. Such a condition also applies to the notation $\rho \cup \{\langle i, r_i\rangle, \langle j, r_j\rangle\}$: here we suppose that $i \neq j$ and that ρ does not contain an element whose index is i or j.

The next step in the development of the semantic model is the introduction of *states*, and of the meaning or evaluation function for expressions (and conditions).

5.2. Definition. (1) The set of states Σ, with typical element σ, is defined by

$$\Sigma = I \to (IndV \to V).$$

(2) We define the meaning function for expressions,

$$[\![\cdot]\!]: Exp \to (I \to (\Sigma \to V)),$$

as follows:

$$[\![x]\!](i)(\sigma) = \sigma(i)(x), \qquad [\![\alpha]\!](i)(\sigma) = \alpha,$$
$$[\![e_1 \ \mathbf{op} \ e_2]\!](i)(\sigma) = ([\![e_1]\!](i)(\sigma)) \ \mathbf{op}_{\mathrm{sem}}([\![e_2]\!](i)(\sigma)),$$
$$[\![\mathbf{op} \ e]\!](i)(\sigma) = \mathbf{op}_{\mathrm{sem}}([\![e]\!](i)(\sigma)).$$

Here we use $\mathbf{op}_{\mathrm{sem}}$ for the semantic operator corresponding to **op**.

(3) We do not give a detailed definition of $[\![b]\!](i)(\sigma)$, which yields an element of the set of truth values $\{\mathbf{t}, \mathbf{f}\}$.

The operational semantics for $\mathscr{S}_{\mathrm{nus}}$ and $\mathscr{L}_{\mathrm{nus}}$ is again given through a transition system. This time, configurations are of the form $\langle \rho, \sigma \rangle$. Transitions are pairs of configurations written in the form

$$\langle \rho, \sigma \rangle \to \langle \rho', \sigma' \rangle.$$

There is no special role here for (an equivalent of) the $\perp$-action.

Nonuniform transitions involve states rather than streams as the intermediate and final results. Since states are entities which are not naturally amenable to the operation of merging, we shall encounter below the necessity to resort to additional means to formulate results which are counterparts of uniform facts such as $\mathcal{O}[\![r_1, r_2]\!] = \mathcal{O}[\![r_1]\!] \parallel \mathcal{O}[\![r_2]\!]$.

We first give the transition system $\mathscr{T}_{\mathrm{nus}}$ for $\mathscr{L}_{\mathrm{nus}}$. Extending the formalism of the uniform case, we also employ *rules*, written in the format

$$\frac{1 \to 2}{3 \to 4}.$$

The meaning of such a rule is the following: In case a transition $1 \to 2$ is an element of $\mathscr{T}_{\mathrm{nus}}$, then the rule allows us to infer that $3 \to 4$ is a valid transition of $\mathscr{T}_{\mathrm{nus}}$ as well.

Remark. Our framework for the operational semantics gives us quite some freedom, so that we can choose whether to use a rule or an axiom to express the semantics of a certain construct. The intuitive meaning remains the same, but technically an axiom needs a transition to perform a certain transformation, while a rule does not. We could, in fact, formulate the operational semantics for $\mathscr{L}_{\mathrm{nus}}$ in terms of axioms only, but we prefer the version as adopted below. The reason for this is our wish to stay as close as possible to the denotational semantics to be developed subsequently. The denotational framework does not provide so much freedom, mainly because of the necessity to arrive at contracting operators having unique fixed points. We have chosen the denotational semantics with the least possible number of computation steps, and tuned the operational semantics to match it.

5.3. Definition. The transition system $\mathcal{T}_{\text{nus}}$ specifies the relation "$\rightarrow$" between configurations of the form $\langle \rho, \sigma \rangle$ as the *smallest* relation which satisfies the following axioms and rules:

$$\langle \rho \cup \{\langle i, (x := e); r\rangle\}, \sigma\rangle \rightarrow \langle \rho \cup \{\langle i, r\rangle\}, \sigma'\rangle \qquad \textbf{Ass}$$

where $\sigma' = \sigma\{\sigma(i)\{\beta/x\}/i\}$ and $\beta = [\![e]\!](i)(\sigma)$.

$$\frac{\langle \rho \cup \{\langle i, s_1; (s_2; r)\rangle\}, \sigma\rangle \rightarrow \langle \rho', \sigma'\rangle}{\langle \rho \cup \{\langle i, (s_1; s_2); r\rangle\}, \sigma\rangle \rightarrow \langle \rho', \sigma'\rangle}, \qquad \textbf{SeqComp}$$

$$\langle \rho \cup \{\langle i, \mu v[s]; r\rangle\}, \sigma\rangle \rightarrow \langle \rho \cup \{\langle i, s[\mu v[s]/v]; r\rangle\}, \sigma\rangle, \qquad \textbf{Rec}$$

$$\langle \rho \cup \{\langle i, \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}; r\rangle\}, \sigma\rangle \rightarrow \langle \rho \cup \{\langle i, s_1; r\rangle\}, \sigma\rangle \qquad \textbf{Cond}$$

in case $[\![b]\!](i)(\sigma) = \mathbf{t}$, and an analogous axiom for the case $[\![b]\!](i)(\sigma) = \mathbf{f}$.

$$\langle \rho \cup \{\langle i, (j?x); r_1\rangle, \langle j, (i!e); r_2\rangle\}, \sigma\rangle \rightarrow \langle \rho \cup \{\langle i, r_1\rangle, \langle j, r_2\rangle\}, \sigma'\rangle \qquad \textbf{Comm}$$

where $\sigma' = \sigma\{\sigma(i)\{\beta/x\}/i\}$, and $\beta = [\![e]\!](j)(\sigma)$.

Remarks. (1) Observe that no transition is defined for a configuration $\langle \rho \cup \{\langle i, (j?x); r\rangle\}, \sigma\rangle$ in the case that ρ does not contain the matching pair $\langle j, (i!e); r'\rangle$ (and a symmetric observation).

(2) The difference in treatment between **SeqComp** and **Rec**—the first as a rule, the second as an axiom—is motivated by the corresponding definition in the denotational semantics (which will be given in Definition 5.8). In operational terms, replacing $(s_1; s_2); r$ by $s_1; (s_2; r)$ does not take a time step, whereas the replacement of $\mu v[s]$ by $s[\mu v[s]/v]$ *does* take a (silent) time step, (i.e., a step that does not change the state). In a uniform setting, the same effect would be obtained by transforming each recursive construct $\mu x[s]$ into $\mu x[\textbf{skip}; s]$ where **skip** is a special elementary action denoting the silent step. Accordingly, the automatic introduction of silent steps obviates the need for the guardedness restriction.

(3) In the axioms **Ass**, **Cond**, and **Comm** we see how the evaluation of an expression e or condition b is parameterized by the index of the statement which contains the occurrence of the expression or condition involved. Effectively, this means that different components are treated as if they had disjoint sets of variables.

The transition system $\mathcal{T}_{\text{nus}}$ is a natural generalization of the corresponding systems $\mathcal{T}_{\text{us}}$ and $\mathcal{T}_{\text{ud}}$. What is more difficult is the definition of $\mathcal{O}[\![\rho]\!]$ and $\mathcal{O}[\![t]\!]$: a formulation which is a straightforward extension of the uniform approach is not feasible, assuming that we want to express results which are variations on relationships such as

$$\mathcal{O}[\![\rho_1 \cup \rho_2]\!] = \mathcal{O}[\![\rho_1]\!] \,\|\, \mathcal{O}[\![\rho_2]\!]. \tag{5.1}$$

Two problems arise when we consider (5.1). The first concerns the basic question as to well-formedness of (5.1): we have as yet no outcome for $\mathcal{O}[\![\rho]\!]$ which allows

the operation of merging to be applied to two instances of it. The second may be considered as a more "practical" one: In a situation where ρ_1 involves a send and ρ_2 a matching receive communication, $\rho_1 \cup \rho_2$ will allow a matching transition by the **Comm** axiom, whereas the components ρ_1 and ρ_2 separately do not allow the corresponding send and receive actions to proceed. Thus, we expect that neither $\mathcal{O}[\![\rho_1]\!]$ nor $\mathcal{O}[\![\rho_2]\!]$ will contain the necessary information enabling the communication to take place through the semantic operator "$\|$" (in whatever way the latter will be defined).

In order to solve the principal problem, we apply a new method, which might be considered somewhat drastic in an operational context: we choose to deliver a *process*, now taken in the technical sense of Section 2.3, as the outcome of $\mathcal{O}[\![\rho]\!]$. Thus, the outcome of $\mathcal{O}[\![\rho]\!]$ is an element of a certain *process domain* P obtained as the solution of an appropriate recursive domain equation $P \cong \mathcal{F}(P)$, where the form of $\mathcal{F}$ is to be determined in a moment. We intend to show that, by adopting this approach, we achieve two goals: Firstly, we shall be in a position to define "$\|$" as an operation on processes and to apply it to $\mathcal{O}[\![\rho_1]\!]$ and $\mathcal{O}[\![\rho_2]\!]$ above. Secondly, since we shall employ processes as well in our denotational model, we have a much smaller distance to bridge between the operational and denotational definitions.

The domain equation we use to determine the appropriate process domain P exploited below is described in the following definition.

5.4. Definition. (1) Let the set *Comm* of communications, with typical element τ, be given by

$$Comm = I \times (I?IndV \cup I!V).$$

(The delimiters "?" and "!" are used here to underline the connection with statements of the form $i!x$ and $i!e$. Properly speaking, they are cosmetic variants of the Cartesian product operator "×".)

(2) Let the set *Step* of steps, with typical element η, be given by

$$Step = \Sigma \cup Comm.$$

(3) Let the function $\mathcal{F}$ be given by

$$\mathcal{F}(P) = \{p_0\} \cup (\Sigma \to \mathcal{P}_{cl}(Step \times P)).$$

(4) Let P be the process domain solving the equation $P \cong \mathcal{F}(P)$. We shall use p, q to range over P.

(5) Let $P_0 = \{p_0\}$, $P_{n+1} = \mathcal{F}(P_n)$. By the general theory (Section 2.3) we know that each $p \in P$ is either an element of some P_n, in which case we shall call p *finite*, or else p is called *infinite* and there is a Cauchy sequence $(p_n)_n$ with $p_n \in P_n$ such that $p = \lim_n p_n$. For finite p, we call the smallest n such that $p \in P_n$ its *degree*.

(6) We shall use X, Y to range over $\mathcal{P}_{cl}(Step \times P)$ and π to range over $Step \times P$.

Example. We have $\langle\langle i, j?x\rangle, p\rangle \in Step \times P$. Below, we shall always adopt for this the simpler notation $\langle i, j?x, p\rangle$.

We proceed with the semantic definitions for the familiar operators "$\cdot$" and "$\|$", this time defined as mappings $P \times P \to P$. We shall in fact propose two definitions. The first one is probably simpler, and is based on an induction on the degree for finite processes. The second one involves Banach's theorem and is given here to familiarize the reader with its subsequent use in definitions where the simpler inductive definition is less convenient.

5.5. Definition. Let $p, q \in P$. We define $p \cdot q$ and $p \| q$ as follows:

(1) (Definition by induction on the degree of p and q.) We first consider the case that both p and q are finite. We put $p_0 \cdot p = p_0 \| p = p \| p_0 = p$. If p is (or if p and q are) different from p_0, we put

$$p \cdot q = \lambda\sigma.(p(\sigma) \cdot q),$$

$$p \| q = \lambda\sigma.((p(\sigma) \| q) \cup (q(\sigma) \| p) \cup (p(\sigma)|_\sigma q(\sigma)))$$

where $X \cdot q = \{\pi \cdot q \mid \pi \in X\}$, $X \| q = \{\pi \| q \mid \pi \in X\}$, $\langle \eta, p'\rangle \cdot q = \langle \eta, p' \cdot q\rangle$, and $\langle \eta, p'\rangle \| q = \langle \eta, p' \| q\rangle$ (note that, here, the degree of p' is less than the degree of p, or the maximum of the degrees of p and q). Moreover,

$$X|_\sigma Y = \bigcup \{\pi_1|_\sigma \pi_2 \mid \pi_1 \in X, \pi_2 \in Y\},$$

where $\pi_1|_\sigma \pi_2$ is defined by

$$\langle i, j?x, p_1\rangle|_\sigma \langle j, i!\alpha, p_2\rangle = \{\langle \sigma', p_1 \| p_2\rangle\}$$

with $\sigma' = \sigma\{\sigma(i)\{\alpha/x\}/i\}$, together with a symmetric clause, and $\pi_1|_\sigma \pi_2 = \emptyset$ for π_1, π_2 not of the above form.

Finally, for p or q infinite, so that we have $p = \lim_n p_n$ and $q = \lim_n q_n$ with $p_n, q_n \in P_n$, we put $p \cdot q = \lim_n (p_n \cdot q_n)$ and $p \| q = \lim_n (p_n \| q_n)$.

(2) (Definition with Banach's theorem.) We define "$\cdot$" and "$\|$" as the unique fixed points of the contracting (higher-order) functions $\Phi, \Psi : (P \times P \to P) \to (P \times P \to P)$ given in the following manner: Let $\varphi, \psi \in P \times P \to P$ be arbitrary. We now define $\Phi(\varphi)$ and $\Psi(\psi)$. Let us abbreviate $\Phi(\varphi)(p, q)$ to $p \tilde{\varphi} q$ and $\Psi(\psi)(p, q)$ to $p \tilde{\psi} q$. Then we put

$$p \tilde{\varphi} q = \begin{cases} q & \text{if } p = p_0, \\ \lambda\sigma.(p(\sigma) \hat{\varphi} q) & \text{if } p \neq p_0; \end{cases}$$

$$p \tilde{\psi} q = \begin{cases} q & \text{if } p = p_0, \\ p & \text{if } q = p_0, \\ \lambda\sigma.((p(\sigma) \hat{\psi} q) \cup (q(\sigma) \hat{\psi} p) \cup (p(\sigma)|_{\sigma,\psi} q(\sigma))) & \text{otherwise;} \end{cases}$$

where $X \hat{\varphi} q = \{\pi \check{\varphi} q \mid \pi \in X\}$, $X \hat{\psi} q = \{\pi \check{\psi} q \mid \pi \in X\}$, $\langle \eta, p'\rangle \check{\varphi} q = \langle \eta, p' \varphi q\rangle$, $\langle \eta, p'\rangle \check{\psi} q = \langle \eta, p' \psi q\rangle$, and where

$$X|_{\sigma,\psi} Y = \bigcup \{\pi_1|_{\sigma,\psi} \pi_2 \mid \pi_1 \in X, \pi_2 \in Y\}.$$

Here $\pi_1|_{\sigma,\psi}\ \pi_2$ is given by

$$\langle i, j?x, p_1\rangle|_{\sigma,\psi}\ \langle j, i!\alpha, p_2\rangle = \{\langle \sigma', p_1\ \psi\ p_2\rangle\}$$

with $\sigma' = \sigma\{\sigma(i)\{\alpha/x\}/i\}$, together with a symmetric clause, and $\pi_1|_{\sigma,\psi}\ \pi_2 = \emptyset$ for π_1, π_2 not of the above form.

Now we define "$\cdot$" to be the unique fixed point of Φ and "$\|$" as the unique fixed point of Ψ.

It should be clear from these definitions that they are variations on one theme: in the second an appeal to Banach's theorem replaces the inductive argument of the first. We omit the proof that the above definitions are justified (and that they define the same operators). Details of a very similar proof are given in [7].

We are now ready for definition of the operational semantics of $\mathscr{L}_{\text{nus}}$.

5.6. Definition. (1) We define $\mathcal{O}[\![\cdot]\!]: PSyCo \to P$ as follows: Let $\rho \in PSyCo$. If $\rho \subseteq \{\langle 1, E\rangle, \ldots, \langle n, E\rangle\}$, we put $\mathcal{O}[\![\rho]\!] = p_0$. Otherwise,

$$\mathcal{O}[\![\rho]\!] = \lambda\sigma.\{\langle \sigma', \mathcal{O}[\![\rho']\!]\rangle \mid \langle \rho, \sigma\rangle \to \langle \rho', \sigma'\rangle\}$$

where, of course, the transition relation "$\to$" is the one given by $\mathscr{T}_{\text{nus}}$.

(2) The function $\mathcal{O}[\![\cdot]\!]: \mathscr{L}_{\text{nus}} \to P$ is defined as follows. Let $t = s_1 \| \cdots \| s_n$. Then

$$\mathcal{O}[\![t]\!] = \mathcal{O}[\![\{\langle 1, s_1; E\rangle, \ldots, \langle n, s_n; E\rangle\}]\!].$$

It is not difficult to verify that $\mathcal{O}$ as given in part (1) of this definition is well-defined. Once more, we deduce this by the following reasoning: Let the (higher-order) mapping $F: (PSyCo \to P) \to (PSyCo \to P)$ be defined in the following manner:

$$F(\mathcal{M})(\rho) = \begin{cases} p_0 & \text{if } \rho \subseteq \{\langle 1, E\rangle, \ldots, \langle n, E\rangle\}, \\ \lambda\sigma.\{\langle \sigma', \mathcal{M}(\rho')\rangle \mid \langle \rho, \sigma\rangle \to \langle \rho', \sigma'\rangle\} & \text{otherwise.} \end{cases}$$

Then F is a contracting mapping, and $\mathcal{O}$ as given in Definition 5.6(1) is the unique fixed point of F.

Remarks. (1) It is not difficult to establish that, for each $\langle \rho, \sigma\rangle$, there are only finitely many $\langle \rho', \sigma'\rangle$ such that $\langle \rho, \sigma\rangle \to \langle \rho', \sigma'\rangle$. Hence, the set occurring in the $\lambda\sigma.\{\ldots\}$ clause in Definition 5.6(1) is finite and therefore closed.

(2) Note that $\mathcal{O}[\![\rho]\!] = \lambda\sigma.\emptyset$ may well occur. For example, $\mathcal{O}[\![\{\langle 1, (2?x); E\rangle\}]\!] = \lambda\sigma.\emptyset$ since there are no transitions $\langle\{\langle 1, (2?x); E\rangle\}, \sigma\rangle \to \cdots$ defined in $\mathscr{T}_{\text{nus}}$. In general, $\mathcal{O}$ does not preserve information on one-sided attempts at communication.

(3) Processes p which equal $\mathcal{O}[\![\rho]\!]$ for some ρ are in fact elements of a process domain P' which satisfies

$$P' \cong \{p_0\} \cup (\Sigma \to \mathscr{P}_{\text{cl}}(\Sigma \times P')).$$

This is the case since no steps in $Comm \times P$ are delivered by the transition relation "$\to$". The more involved process domain P is exploited in full only in the definitions of $\mathcal{O}^*$ and of the denotational semantics $\mathscr{D}$, both of which we shall discuss presently.

Now that we have given a process interpretation for $\mathcal{O}[\![\rho]\!]$, yielding results in a domain for which "$\|$" is well-defined, we have a well-formed question to ask: is it true that $\mathcal{O}[\![\rho_1 \cup \rho]\!] = \mathcal{O}[\![\rho_1]\!] \| \mathcal{O}[\![\rho_2]\!]$? The answer is negative—for the same reason as already explained earlier. However, a not too far-fetched variation on this property, which does indeed hold, will be presented soon. Rather than immediately getting to this, we first develop the denotational semantics for $\mathcal{L}_{\text{nus}}$. In this way, the reader may acquire some additional appreciation for the way we utilize the process notion in our framework. In fact, a combination of ideas involving:

- the tools of environments and semantic continuations as employed in Section 3,
- the operational semantics of $\mathcal{L}_{\text{nus}}$, and
- the definition(s) of "$\|$"

will altogether provide most of the background to understand the denotational definition.

5.3. Denotational semantics

We introduce semantic continuations and environments in the following definition.

5.7. Definition. (1) The set of semantic continuations is given by $SeCo =^{\text{def}} P$.

(2) We define the set of environments by $\Gamma =^{\text{def}} StmV \to (I \to (SeCo \to^{\text{NDI}} P))$.

We shall use p, q to range over $SeCo$ and γ to range over Γ.

The definition of $\mathcal{D}$ will be given for all $s \in \mathcal{S}_{\text{nus}}$ and all $t \in \mathcal{L}_{\text{nus}}$. Thus, the restriction to statements with only guarded recursion is lifted. As remarked earlier, this is explained by our definition of recursion which involves a treatment of recursive calls such that always at least one initial "silent" step is made upon "procedure entrance". That is, (the equivalent of) a transition is made which does not affect the state but which does take (what may be seen as) one unit of time. For example, execution of $\mu v[v]$ will result in an infinite sequence of such silent steps (rather than in just $\perp$ as in the uniform case). All this is a matter of taste rather than of principle. One may disagree with our feeling that silent steps are more natural in a nonuniform than in a uniform setting.

We now give the definitions of $\mathcal{D}[\![s]\!]$ and of $\mathcal{D}[\![t]\!]$. We shall often suppress parentheses around arguments of functions for easier readability.

5.8. Definition. (1) We define the function

$$\mathcal{D}[\![\cdot]\!] : \mathcal{S}_{\text{nus}} \to (\Gamma \to (I \to (SeCo \xrightarrow{\text{NDI}} P)))$$

as follows:

(a) $\mathcal{D}[\![x := e]\!]\gamma ip = \lambda\sigma.\{\langle \sigma', p\rangle\}$, where $\sigma' = \sigma\{\sigma(i)\{\alpha/x\}/i\}$ and $\alpha = [\![e]\!]i\sigma$;

(b) $\mathcal{D}[\![s_1 ; s_2]\!]\gamma ip = \mathcal{D}[\![s_1]\!]\gamma i(\mathcal{D}[\![s_2]\!]\gamma ip)$;

(c) $\mathcal{D}[\![\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}]\!]\gamma ip$
$= \lambda\sigma.\{\langle \sigma, \textbf{if } [\![b]\!]i\sigma = \textbf{t then } \mathcal{D}[\![s_1]\!]\gamma ip \textbf{ else } \mathcal{D}[\![s_2]\!]\gamma ip \textbf{ fi}\rangle\}$;

(d) $\mathscr{D}[\![v]\!]\gamma ip = \gamma(v)ip$;

(e) $\mathscr{D}[\![\mu v[s]]\!]\gamma ip = \varphi_\infty(i)(p)$ where φ_∞ is the unique fixed point of the operator Φ, which maps the space $I \to (SeCo \to^{\mathrm{NDI}} P)$ to itself, and is given by

$$\Phi(\varphi) = \lambda i.\lambda p.\lambda\sigma.\{\langle\sigma, \mathscr{D}[\![s]\!]\gamma\{\varphi/v\}ip\rangle\};$$

(f) $\mathscr{D}[\![j?x]\!]\gamma ip = \lambda\sigma.\{\langle i, j?x, p\rangle\}$;

$\mathscr{D}[\![j!e]\!]\gamma ip = \lambda\sigma.\{\langle i, j!\alpha, p\rangle\}$, where $\alpha = [\![e]\!]i\sigma$.

(2) We define the function $\mathscr{D}[\![\cdot]\!]: \mathscr{L}_{\mathrm{nus}} \to P$ as follows: Let $t = s_1 \| \cdots \| s_n$ and let γ be arbitrary. Then

$$\mathscr{D}[\![t]\!] = \mathscr{D}[\![s_1]\!]\gamma 1 p_0 \| \cdots \| \mathscr{D}[\![s_n]\!]\gamma n p_0.$$

Remark. The definition in clause (1)(e) above is justified by the fact that the function Φ is contracting. Note that its unique fixed point can again be obtained as $\varphi_\infty = \lim_k \varphi_k$, where φ_0 is arbitrary and

$$\varphi_{k+1} = \Phi(\varphi_k) = \lambda i.\lambda p.\lambda\sigma.\{\langle\sigma, \mathscr{D}[\![s]\!]\gamma\{\varphi_k/v\}ip\rangle\}.$$

Examples. (1) $\mathscr{D}[\![\mu v[v]]\!]\gamma ip = \lambda\sigma.\{\langle\sigma, \lambda\sigma.\{\langle\sigma, \ldots\rangle\}\rangle\}$.

(2) We have

$$\begin{aligned}\mathscr{D}[\![(2?x)\|(1!3)]\!] &= \mathscr{D}[\![2?x]\!]\gamma 1 p_0 \| \mathscr{D}[\![1!3]\!]\gamma 2 p_0 \\ &= \lambda\sigma.\{\langle 1, 2?x, p_0\rangle\} \| \lambda\sigma.\{\langle 2, 1!3, p_0\rangle\} \overset{\mathrm{def}}{=} q_1 \| q_2 \\ &= \lambda\sigma.\{\langle 1, 2?x, q_2\rangle, \langle 2, 1!3, q_1\rangle, \langle\sigma\{\sigma(1)\{3/x\}/1\}, p_0\rangle\}.\end{aligned}$$

The resulting process, say q, contains two steps resulting from one-sided (failing) communication: $\langle 1, \ldots\rangle$ and $\langle 2, \ldots\rangle$. Moreover, there is one step resulting from successful communication: $\langle\sigma\{\ldots\}, p_0\rangle$, where 3 is assigned to x. We recall that the latter step ultimately results from the definition of $\pi_1|_\sigma \pi_2$ (or $\pi_1|_{\sigma,\psi} \pi_2$) given in Definition 5.5. The operation of abstraction, to be introduced in a moment, will simplify the result q to just $\lambda\sigma.\{\langle\sigma\{\ldots\}, p_0\rangle\}$, throwing away the unsuccessful parts $\langle 1, \ldots\rangle$ and $\langle 2, \ldots\rangle$.

5.4. Equivalence of operational and denotational semantics

We return to the question concerning the (non)compositionality of $\mathscr{O}$. We shall introduce an extension of $\mathscr{T}_{\mathrm{nus}}$ to $\mathscr{T}^*_{\mathrm{nus}}$, which induces an associated operational semantics $\mathscr{O}^*$, and we then settle the relationship between $\mathscr{O}$, $\mathscr{O}^*$, and $\mathscr{D}$.

5.9. Definition. (1) We expand the notion of configuration such that it includes pairs of the form $\langle\rho, \eta\rangle$ (recall that η ranges over $Step = \Sigma \cup Comm$). Therefore, in addition to configurations of the form $\langle\rho, \sigma\rangle$, we also consider configurations of the form $\langle\rho, \tau\rangle$. (Actually, the latter ones will only occur on the right-hand side of a transition.)

(2) The transition system $\mathcal{T}^*_{\text{nus}}$ extends the system $\mathcal{T}_{\text{nus}}$ of Definition 5.3 by adding to it the axioms

$$\langle \rho \cup \{\langle i, (j?x);r\rangle\}, \sigma\rangle \to \langle \rho \cup \{\langle i, r\rangle\}, \langle i, j?x\rangle\rangle, \qquad \textbf{IndComm1}$$

$$\langle \rho \cup \{\langle i, (j!e);r\rangle\}, \sigma\rangle \to \langle \rho \cup \{\langle i, r\rangle\}, \langle i, j!\alpha\rangle\rangle \qquad \textbf{IndComm2}$$

where $\alpha = [\![e]\!]i\sigma$. Moreover, the rule **SeqComp** of $\mathcal{T}_{\text{nus}}$:

$$\frac{\langle \rho \cup \{\langle i, s_1;(s_2;r)\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \rho \cup \{\langle i, (s_1;s_2);r\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle}$$

is replaced by

$$\frac{\langle \rho \cup \{\langle i, s_1;(s_2;r)\rangle\}, \sigma\rangle \to \langle \rho', \eta'\rangle}{\langle \rho \cup \{\langle i, (s_1;s_2);r\rangle\}, \sigma\rangle \to \langle \rho', \eta'\}}.$$

(3) The operational meaning $\mathcal{O}^*: PSyCo \to P$ is defined by

$$\mathcal{O}^*[\![\rho]\!] = \begin{cases} \rho_0 & \text{if } \rho \subseteq \{\langle 1, E\rangle, \ldots, \langle n, E\rangle\}, \\ \lambda\sigma.\{\langle \eta', \mathcal{O}^*[\![\rho']\!]\rangle \mid \langle \rho, \sigma\rangle \to \langle \rho', \eta'\rangle\} & \text{otherwise.} \end{cases}$$

(Here we take "$\to$" as determined by $\mathcal{T}^*_{\text{nus}}$.)

(4) The operational meaning $\mathcal{O}^*: \mathcal{L}_{\text{nus}} \to P$ is defined as follows: Let $t = s_1 \| \cdots \| s_n$. Then

$$\mathcal{O}^*[\![t]\!] = \mathcal{O}^*[\![\{\langle 1, s_1;E\rangle, \ldots, \langle n, s_n;E\rangle\}]\!].$$

Following the detailed analysis as in [16], it is not difficult to prove the following theorem.

5.10. Theorem. $\mathcal{O}^*[\![\rho_1 \cup \rho_2]\!] = \mathcal{O}^*[\![\rho_1]\!] \| \mathcal{O}^*[\![\rho_s]\!]$.

For example,

$$\mathcal{O}^*[\![\{\langle 1, (2?x);E\rangle, \langle 2, (1!3);E\rangle\}]\!]$$
$$= \lambda\sigma.\{\langle 1, 2?x, p_1\rangle, \langle 2, 1!3, p_2\rangle, \langle \sigma\{\sigma(1)\{3/x\}/1\}, p_0\rangle\}$$

where $p_1 = \lambda\sigma.\{\langle 2, 1!3, p_0\rangle\}$ and $p_2 = \lambda\sigma.\{\langle 1, 2?x, p_0\rangle\}$. Thus,

$$\mathcal{O}^*[\![\{\langle 1, (2?x);E\rangle, \langle 2, (1!3);E\rangle\}]\!] = \lambda\sigma.\{\langle 1, 2?x, p_0\rangle\} \| \lambda\sigma.\{\langle 2, 1!3, p_0\rangle\}$$
$$= \mathcal{O}^*[\![\{\langle 1, (2?x);E\rangle\}]\!] \| \mathcal{O}^*[\![\{\langle 2, (1!3);E\rangle\}]\!].$$

The relationship between $\mathcal{O}$ and $\mathcal{O}^*$ is settled by the introduction of an *abstraction operator* $abs: P \to P'$ (with P' as given in remark (3) after Definition 5.6). When applied to some $p \in P$, $abs(p)$ deletes from p all pairs $\langle \tau, p'\rangle$ which occur anywhere "inside" p: all unsuccessful attempts at communication disappear, and only the results of successful communications remain, together with the "normal" steps caused by, e.g., assignments. Again (as was the case with any p), $abs(p)$ may have (inner) branches of the form $\lambda\sigma.\emptyset$—a phenomenon which is often called *deadlock*.

The abstraction operator is defined as follows.

5.11. Definition. For finite p we put $abs(p_0) = p_0$, $abs(\lambda\sigma.X) = \lambda\sigma.abs(X)$, and

$$abs(X) = \{\langle\sigma', abs(p')\rangle \mid \langle\sigma', p'\rangle \in X\}.$$

(Note that a pair $\langle\tau, p'\rangle \in X$ will not contribute to $abs(X)$.) For infinite p, with $p = \lim_n p_n$ and $p_n \in P_n$, we take $abs(p) = \lim_n abs(p_n)$.

Again relying on the general results in [16], we have the following theorem.

5.12. Theorem. $\mathcal{O} = abs \circ \mathcal{O}^*$.

The final part of this section is devoted to the proof of the equality of $\mathcal{O}^*$ and $\mathcal{D}$.

5.13. Theorem. *For all* $t \in \mathcal{L}_{\text{nus}}$, $\mathcal{O}^*[\![t]\!] = \mathcal{D}[\![t]\!]$.

The proof closely follows the strategy applied for the uniform version of this result described in Section 3. We first state a simple lemma on $\mathcal{O}^*$ which we need below.

5.14. Lemma. (1) $\mathcal{O}^*[\![\{\langle i, (x := e); r\rangle\}]\!] = \lambda\sigma.\{\langle\sigma', \mathcal{O}^*[\![\{\langle i, r\rangle\}]\!]\rangle\}$, *with* σ' *as usual.*

(2) $\mathcal{O}^*[\![\{\langle i, (s_1; s_2); r\rangle\}]\!] = \mathcal{O}^*[\![\{\langle i, s_1; (s_2; r)\rangle\}]\!]$.

(3) $\mathcal{O}^*[\![\{\langle i, \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}; r\rangle\}]\!]$
$= \lambda\sigma.\{\langle\sigma, \textbf{if } [\![b]\!]i\sigma \textbf{ then } \mathcal{O}^*[\![\{\langle i, s_1; r\rangle\}]\!] \textbf{ else } \mathcal{O}^*[\![\{\langle i, s_2; r\rangle\}]\!] \textbf{fi}\rangle\}$.

(4) $\mathcal{O}^*[\![\{\langle i, (j?x); r\rangle\}]\!] = \lambda\sigma.\{\langle i, j?x, \mathcal{O}^*[\![\{\langle i, r\rangle\}]\!]\rangle\}$.

(5) $\mathcal{O}^*[\![\{\langle i, (j!e); r\rangle\}]\!] = \lambda\sigma.\{\langle i, j!\alpha, \mathcal{O}^*[\![\{\langle i, r\rangle\}]\!]\rangle\}$ *where* $\alpha = [\![e]\!]i\sigma$.

(6) $\mathcal{O}^*[\![\{\langle i, (j?x); r_1\rangle, \langle j, (i!e); r_2\rangle\}]\!] = \lambda\sigma.\{\langle i, j?x, \mathcal{O}^*[\![\{\langle i, r_1\rangle, \langle j, (i!e); r_2\rangle\}]\!]\rangle,$
$\langle j, i!\alpha, \mathcal{O}^*[\![\{\langle i, (j?x); r_1\rangle, \langle j, r_2\rangle\}]\!]\rangle, \langle\sigma', \mathcal{O}^*[\![\{\langle i, r_1\rangle, \langle j, r_2\rangle\}]\!]\rangle\}$ *with* $\alpha = [\![e]\!]i\sigma$ *and* σ' *as usual.*

Proof. Easy from the definitions of $\mathcal{T}^*_{\text{nus}}$ and $\mathcal{O}^*$. □

Remark. Note that part (2) of this lemma would not hold in the form as given if $\mathcal{T}_{\text{nus}}$ contained an axiom for **SeqComp**, rather than a rule. Conversely, part (3) would not hold if we had a rule for **Cond**, instead of an axiom.

The next lemma applies some notation which is a slight variant of the one introduced preceding Theorem 3.18. Let us, temporarily, add the statement **skip** to our language, with an associated transition

$$\langle\rho \cup \{\langle i, \textbf{skip}; r\rangle\}, \sigma\rangle \to \langle\rho \cup \{\langle i, r\rangle\}, \sigma\rangle \qquad \textbf{Skip}$$

(note that we could take **skip** as another name for $x := x$). Let, for given s and v, $s_v^{(n)}$ be defined by $s_v^{(0)} = \textbf{skip}$ and $s_v^{(n+1)} = \textbf{skip}; s[s_v^{(n)}/v]$. We can then prove the following lemma, once more using the framework of [16].

5.15. Lemma. *For closed s*:

$$\mathcal{O}^*[\![\{\langle i, \mu v[s]; r\rangle\}]\!] = \lim_n \mathcal{O}^*[\![\{\langle i, s_v^{(n)}; r\rangle\}]\!].$$

We are now ready for the statement of the main step in the proof of Theorem 5.13.

5.16. Lemma. *Let $s \in \mathcal{S}_{\text{nus}}$ be arbitrary (not necessarily closed) and let the set of free statement variables in s be contained in $\{v_1, \ldots, v_k\}$, $k \geqslant 0$. Let $s_1, \ldots, s_k$ be closed statements, and let $\tilde{s} = s[s_h / v_h]_{h=1}^k$. Let, for any ρ, $\mathcal{O}[\![\rho]\!]$ be short for $\lambda p.(\mathcal{O}^*[\![\rho]\!] \cdot p)$. Let, furthermore, for $h = 1, \ldots, k$,*

$$\varphi_h = \lambda i.\mathcal{O}[\![\{\langle i, s_h; E\rangle\}]\!]$$

and let $\tilde{\gamma} = \gamma\{\varphi_h / v_h\}_{h=1}^k$. We then have, for any i,

$$\mathcal{O}[\![\{\langle i, \tilde{s}; E\rangle\}]\!] = \mathcal{D}[\![s]\!](\tilde{\gamma})(i).$$

Proof. Induction on the complexity of *s*, following the argument as given in the proof of Theorem 3.19, but for the addition of an extra parameter *i*, and replacement of X by p (and using Lemmas 5.14 and 5.15 to deal with the individual cases). □

5.17. Corollary. *For closed s*:

$$\mathcal{O}[\![\{\langle i, s; E\rangle\}]\!] = \mathcal{D}[\![s]\!](\gamma)(i).$$

Now it is easy to prove Theorem 5.13.

Proof of Theorem 5.13. Take any $t = s_1 \| \cdots \| s_n$. Then

$$\begin{aligned}\mathcal{O}^*[\![t]\!] &= \mathcal{O}^*[\![\{\langle 1, s_1; E\rangle, \ldots, \langle n, s_n; E\rangle\}]\!] \\ &= \mathcal{O}^*[\![\{\langle 1, s_1; E\rangle\}]\!] \| \cdots \| \mathcal{O}^*[\![\{\langle n, s_n; E\rangle\}]\!].\end{aligned}$$

By Corollary 5.17, we have for each i that

$$\mathcal{O}^*[\![\{\langle i, s_i; E\rangle\}]\!] = \mathcal{O}^*[\![\{\langle i, s_i; E\rangle\}]\!] \cdot p_0 = \mathcal{O}[\![\{\langle i, s_i; E\rangle\}]\!](p_0) = \mathcal{D}[\![s_i]\!](\gamma)(i)(p_0).$$

Thus,

$$\begin{aligned}\mathcal{O}^*[\![t]\!] &= \mathcal{O}^*[\![\{\langle 1, s_1; E\rangle\}]\!] \| \cdots \| \mathcal{O}^*[\![\{\langle n, s_n; E\rangle\}]\!] \\ &= \mathcal{D}[\![s_1]\!](\gamma)(1)(p_0) \| \cdots \| \mathcal{D}[\![s_n]\!](\gamma)(n)(p_0) = \mathcal{D}[\![t]\!].\end{aligned}$$ □

Remark. Contrary to the situation for the uniform case, we have at present investigated only metric (operational and denotational) models for $\mathcal{L}_{\text{nus}}$. Therefore we have no information on the feasibility of order-theoretic models for this purpose.

6. A nonuniform and dynamic language

We have, at last, arrived at the presentation of the semantic models of a nonuniform and dynamic language. Not surprisingly, it brings a synthesis of the ideas of Sections 4 and 5; for the reader who has understood these sections, the present section contains few surprises. Still, some technical difficulties which are not straightforward from previous considerations remain to be overcome.

6.1. Informal introduction and syntax

As usual, we begin with the syntax. Statements are almost as before, but for the fact that communications $i?x$ or $i!e$ (with static i, $1 \leqslant i \leqslant n$) are now replaced by communications $e?x$ or $e!e'$, in which the value of the expression e is (the name of) a dynamically created process. The expression itself can be, for example, a variable, in which this process name is stored. The syntax of expressions also contains an essential new clause, viz. "**new**(c)". This expresses that a new process (of class c) is to be created. Each program consists of a set of *class declarations* $\langle c_k \Leftarrow s_k \rangle_{k=1}^{n}$, and, assuming that c above equals c_k for some k, the (side-)effect of **new**(c) is the creation of a new process which will execute the statement $s = s_k$. Here we have the counterpart of the construct **new**(s) in Section 4. In addition, this new process is referred to by a (new) name, say α, and the value of the expression e will be this name α. Therefore, in the (common) case that **new**(c) occurs in an assignment $x :=$ **new**(c), the name α of the newly created process is assigned to x. In this way, upon subsequent occurrences of x in, e.g., $x!e$, it is known that the value of e has to be sent to process α.

We now give the formal syntactic definitions. Let *CNam* be the collection of *class names*, with typical element c. Let *IndV* and *StmV* be as before, and let α and β range over the set *Obj* of objects to be defined presently.

6.1. Definition. (1) The set *Exp* of expressions, with typical element e, is defined by

$$e ::= x \mid \alpha \mid e_1 \mathbf{op}\ e_2 \mid \mathbf{op}\ e' \mid \mathbf{new}(c)$$

(Here, again, **op** stands for an arbitrary binary or unary operator.)

(2) We do not give a detailed syntactic definition for the set *Test* of conditions (with typical element b) but we assume, for simplicity, that conditions (unlike expressions) can be evaluated without side-effects.

(3) We define the set $\mathscr{S}_{\mathrm{nud}}$ of statements, with typical element s, by

$$s ::= x := e \mid s_1 ; s_2 \mid v \mid \mu v[s'] \mid \mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi} \mid e?x \mid e!e' \mid ?x \mid !e$$

(4) The set $\mathscr{L}_{\mathrm{nud}}$ of programs, with typical element t is defined by

$$t ::= \langle c_1 \Leftarrow s_1, \ldots, c_n \Leftarrow s_n \rangle \quad (n \geqslant 1).$$

Here we require that all the s_i are closed, that all the c_i are different, and that any class name c occurring in any s_i (in the context **new**(c)) is one of $c_1, \ldots, c_n$.

Remarks. (1) In $\mathscr{S}_{\text{nud}}$ we allow communications of the form $?x$ or $!e$ which do not name a corresponding process (they are, in fact, willing to communicate with *any* other process). However, we shall require, in order that a match be established between a pair of send and receive statements, that at least one of the two explicitly identifies the process in which the other occurs. (Hence, no communication takes place between $?x$ and $!e$.)

(2) By convention, executing a program $t = \langle c_k \Leftarrow s_k \rangle_{k=1}^{n}$ is initiated by executing the statement $x := \mathbf{new}(c_1)$, for some fresh x (i.e., some individual variable not occurring in t). In other words, a process of class c_1 is created implicitly. (Its name is stored nowhere, so this process cannot be addressed explicitly by other processes.)

(3) Note that we now have two forms of recursion, one in constructs of the form $\mu v[s]$ and the other in case of a declaration such as $c \Leftarrow \cdots c \cdots$.

The set *Obj* of objects replaces the set of values v which we encountered in Section 5. It consists firstly of the so-called *standard objects SObj*. Here one may think of the union of the set of values V and the truth-values $\{\mathbf{t}, \mathbf{f}\}$ as employed in Section 5. Moreover, we now also have the set of so-called *active objects AObj*, which consists of the names of processes as mentioned in the introductory paragraph of this section. In fact, we may see *AObj* as the generalization of the set I of Section 5. We define *AObj* as

$$AObj = CNam \times \mathbb{N}$$

where $\mathbb{N}$ is the set of nonnegative integers. At each moment an active object $\langle c, l \rangle$ is the name of the lth process of class c, i.e., the process created by the lth execution of a $\mathbf{new}(c)$ construct.

From now on we shall use the term "object" in the above sense, i.e., for an element of *AObj*, not to confuse it with the technical term "process" in the sense of Section 2.3, the precise meaning of which we shall give in Definition 6.5.

6.2. Operational semantics

We proceed with the preparations for the operational semantics for $\mathscr{L}_{\text{nud}}$. Firstly, we refine the class of syntactic continuations, by distinguishing between statement continuations and expression continuations.

6.2. Definition. (1) The class of syntactic statement continuations *SyStCo*, with typical element r, is defined by

$$r ::= E \mid s; r' \mid e : g$$

where s is closed. (The colon ":" used here should not be confused with the semantic operator ":" as introduced in Definition 4.7. Here it is simply a syntactic symbol, comparable with ";".)

(2) The class of syntactic expression continuations *SyExCo*, with typical element g, is defined by

$$g ::= \lambda z.r$$

where $z \in IndV$. Here z may not occur as the left-hand side of an assignment in r.

(3) The class of parallel syntactic (statement) continuations *PSyCo*, with typical element ρ, is defined as the collection of sets of the form

$$\{\langle \alpha_1, r_1\rangle, \ldots, \langle \alpha_n, r_n\rangle\} \quad (n \geqslant 0)$$

where the α_i are different elements of *AObj*.

The intuitive meaning of a syntactic expression continuation $g = \lambda z.r$ is to describe a computation which depends on some value. The variable z serves as a placeholder for this value in r. When g is given a value, i.e., an object $\alpha \in Obj$, then it delivers a syntactic statement continuation $r[\alpha/z]$ (where the value α is put in the place of z). A syntactic statement continuation r of the form $e{:}g$ is executed by first evaluating the expression e (which may or may not take some time steps or have some side-effect) and then feeding its value into g in the way described above. This yields a syntactic statement continuation which is executed subsequently.

We also extend the class of states by introducing a second component, as follows.

6.3. Definition. We define the set of states by $\Sigma = \Sigma_1 \times \Sigma_2$, with typical element $\sigma = \langle \sigma_{(1)}, \sigma_{(2)}\rangle$. We put $\Sigma_1 = AObj \to (IndV \to Obj)$ and $\Sigma_2 = CNam \to \mathbb{N}$.

A state σ has the following function:

- The first component $\sigma_{(1)}$ is as σ in Section 5, but for the replacement of I by *AObj* and of V by *Obj*. Thus, for any object α and individual variable x, $\sigma_{(1)}(\alpha)(x)$ is the value of α's x-variable.
- The second component $\sigma_{(2)}$ records for each class name c the number $l = \sigma_{(2)}(c)$ of objects of that class that have been created up to this point.

We shall usually suppress indices and simply write σ, also in cases where $\sigma_{(1)}$ or $\sigma_{(2)}$ is meant.

In the transition system to be presented in a moment, we shall take into account the fact that evaluation of expressions may now be more involved since they may contain **new**-constructs. For reasons of simplicity, we shall not include a similar extension in our treatment of conditions. We shall, just as in Section 5, assume that evaluation of a condition b—expressed by the notation $[\![b]\!](\alpha)(\sigma)$—is simple and has no side-effects. (Of course, it is a minor exercise to adapt the treatment below to cover the case of conditions which may include **new**-constructs.)

The operational semantics for $\mathscr{L}_{\text{nud}}$ is given in terms of a transition system $\mathscr{T}_{\text{nud}}$ of axioms and rules for configurations $\langle \rho, \sigma\rangle$. Throughout, $\mathscr{T}_{\text{nud}}$ assumes one fixed program $t = \langle c_k \Leftarrow s_k\rangle_{k=1}^{n}$, and we shall also assume that all class names occurring in any statement are declared in this program t. (We might carry the information

contained in t along as an extra component of the configuration, but we find this too cumbersome.)

6.4. Definition. The transition system $\mathcal{T}_{\text{nud}}$ is given by the following axioms and rules:

$$\langle \rho \cup \{\langle \alpha, (x := \beta); r\rangle\}, \sigma\rangle \to \langle \rho \cup \{\langle \alpha, r\rangle\}, \sigma'\rangle \qquad \textbf{Ass1}$$

where $\sigma' = \sigma\{\sigma(\alpha)\{\beta/x\}/\alpha\}$.

$$\frac{\langle \rho \cup \{\langle \alpha, e : \lambda z.((x := z); r)\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \rho \cup \{\langle \alpha, (x := e); r\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle} \qquad \textbf{Ass2}$$

where z is a fresh variable, i.e., an individual variable not occurring in ρ, e, or r (actually, it is sufficient to require that z does not occur in r). Note that this rule is only useful if e is not itself a constant β.

SeqComp, **Rec**, and **Cond** are as in Definition 5.3 (with α replacing i).

$$\frac{\langle \rho \cup \{\langle \alpha, e : \lambda z.((z?x); r)\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \rho \cup \{\langle \alpha, (e?x); r\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle} \qquad \textbf{Receive1}$$

with z fresh.

$$\frac{\langle \rho \cup \{\langle \alpha, e : \lambda z.(e' : \lambda z'.((z!z'); r))\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \rho \cup \{\langle \alpha, (e!e'); r\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle} \qquad \textbf{Send1}$$

with z and z' fresh.

$$\frac{\langle \rho \cup \{\langle \alpha, e : \lambda z.((!z); r)\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \rho \cup \{\langle \alpha, (!e); r\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle} \qquad \textbf{Send2}$$

with z fresh.

$$\langle \rho \cup \{\langle \alpha, (\beta?x); r_1\rangle, \langle \beta, (\alpha!\alpha'); r_2\rangle\}, \sigma\rangle \to \langle \rho \cup \{\langle \alpha, r_1\rangle, \langle \beta, r_2\rangle\}, \sigma'\rangle \qquad \textbf{Comm1}$$

where $\sigma' = \sigma\{\sigma(\alpha)\{\alpha'/x\}/\alpha\}$.

$$\langle \rho \cup \{\langle \alpha, (\beta?x); r_1\rangle, \langle \beta, (!\alpha'); r_2\rangle\}, \sigma\rangle \to \langle \rho \cup \{\langle \alpha, r_1\rangle, \langle \beta, r_2\rangle\}, \sigma'\rangle \qquad \textbf{Comm2}$$

with σ' as above.

$$\langle \rho \cup \{\langle \alpha, (?x); r_1\rangle, \langle \beta, (\alpha!\alpha'); r_2\rangle\}, \sigma\rangle \to \langle \rho \cup \{\langle \alpha, r_1\rangle, \langle \beta, r_2\rangle\}, \sigma'\rangle \qquad \textbf{Comm3}$$

with σ' as above.

$$\langle \rho \cup \{\langle \alpha, x : g\rangle\}, \sigma\rangle \to \langle \rho \cup \{\langle \alpha, \sigma(\alpha)(x) : g\rangle\}, \sigma\rangle. \qquad \textbf{IndV}$$

$$\frac{\langle \rho \cup \{\langle \alpha, r[\beta/z]\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \rho \cup \{\langle \alpha, \beta : \lambda z.r\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle}. \qquad \textbf{Obj}$$

$$\frac{\langle \rho \cup \{\langle \alpha, (\beta_1 \ \mathbf{op}_{\text{sem}} \ \beta_2) : g\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \rho \cup \{\langle \alpha, (\beta_1 \ \mathbf{op} \ \beta_2) : g\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle}. \qquad \textbf{Binop1}$$

Here, $\beta_1\ \mathbf{op}_{sem}\ \beta_2$ stands for the object β that results if we apply the semantic operator $\mathbf{op}_{sem}$ corresponding to **op** to the objects β_1 and β_2.

$$\frac{\langle \rho \cup \{\langle \alpha, e_1 : \lambda z_1.(e_2 : \lambda z_2.((z_1\ \mathbf{op}\ z_2) : g))\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \rho \cup \{\langle \alpha, (e_1\ \mathbf{op}\ e_2) : g\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle} \qquad \textbf{Binop2}$$

with z_1 and z_2 fresh.

$$\frac{\langle \rho \cup \{\langle \alpha, (\mathbf{op}_{sem}\ \beta) : g\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \rho \cup \{\langle \alpha, (\mathbf{op}\ \beta) : g\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle}. \qquad \textbf{Unop1}$$

Again, $\mathbf{op}_{sem}\ \beta$ stands for the object β' that results if we apply the semantic operator $\mathbf{op}_{sem}$ corresponding to **op** to the object β.

$$\frac{\langle \rho \cup \{\langle \alpha, e : \lambda z.((\mathbf{op}\ z) : g)\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \rho \cup \{\langle \alpha, (\mathbf{op}\ e) : g\rangle\}, \sigma\rangle \to \langle \rho', \sigma'\rangle} \qquad \textbf{Unop2}$$

with z fresh.

$$\langle \rho \cup \{\langle \alpha, \mathbf{new}(c) : g\rangle\}, \sigma\rangle \to \langle \rho \cup \{\langle \alpha, \beta : g\rangle, \langle \beta, s; E\rangle\}, \sigma'\rangle \qquad \textbf{New}$$

where $c \Leftarrow s$ occurs in t, $\beta = \langle c, \sigma(c)+1\rangle$ and $\sigma' = \sigma\{\sigma(c)+1/c\}$.

Remarks. (1) In the **New** axiom, dealing with the case $e = \mathbf{new}(c)$, a new object executing the statement s is created, and the name $\beta = \langle c, \sigma(c)+1\rangle$ is delivered as the resulting value for e. As we already saw, $\langle c, l\rangle$ is the name of the lth object of class c, and, for each c, $\sigma(c)$ stores the currently highest object number. This also explains the update σ' of σ upon object creation.

(2) The general scheme to deal with expression evaluation is the following. If the expression e occurs in a certain context, for example $x := e; r$, then an application of a rule (in our example, **Ass2**) transforms the context to one of the form $e:g$ (in our case, $e : \lambda z.(x := z; r)$), indicating that first e is to be evaluated, after which its value can be used. Because a rule is applied and not an axiom, this does not take any time steps. Now the axioms **IndV** or **New** (which do take a time step) or rules like **Binop1** and **Unop1** (which do not take time) will take care of the evaluation of the expression. If necessary, the rules **Binop2** or **Unop2** will break the expression further apart (again without taking time). After the expression has been evaluated, the rule **Obj** will put the resulting object β back into the original context, and further axioms or rules (in our example, **Ass1**) will deal with this result β in an appropriate way.

The step from $\mathcal{T}_{nud}$ to the corresponding $\mathcal{O}$ is very similar to the one described in Section 5. We first introduce the relevant process domain.

6.5. Definition. (1) The set *Comm* of communications (with typical element τ) is defined by

$$Comm = AObj \times (AObj?IndV \cup ?IndV \cup AObj!Obj \cup !Obj).$$

(2) We define the set *Step* of steps (with typical element η) by

$$Step = \Sigma \cup Comm.$$

(3) The process domain P (typical elements p and q) is the solution of the following domain equation:

$$P \cong \{p_0\} \cup (\Sigma \to \mathcal{P}_{cl}(Step \times P)).$$

6.6. Definition. (1) $\mathcal{O}[\![\cdot]\!]: PSyCo \to P$ is defined by

$$\mathcal{O}[\![\rho]\!] = \begin{cases} p_0 & \text{if } \rho = \{\langle \alpha_1, E\rangle, \ldots, \langle \alpha_n, E\rangle\}, \\ \lambda\sigma.\{\langle \sigma', \mathcal{O}[\![\rho']\!]\rangle \mid \langle \rho, \sigma\rangle \to \langle \rho', \sigma'\rangle\} & \text{otherwise.} \end{cases}$$

(2) $\mathcal{O}[\![\cdot]\!]: \mathcal{L}_{nud} \to P$ is defined as follows. Let $t = \langle c_k \Leftarrow s_k\rangle_{k=1}^{n}$. Then

$$\mathcal{O}[\![t]\!] = \mathcal{O}[\![\{\langle\langle c_1, 1\rangle, s_1; E\rangle\}]\!].$$

Remark. Although not specified here, the process $p = \mathcal{O}[\![t]\!]$ will of course be started in a state σ_0, which satisfies $\sigma_0(c_1) = 1$ and $\sigma_0(c) = 0$ for $c \neq c_1$. The choice of this σ_0 and ρ above amounts to starting the computation with the first object of class c_1, while objects of other classes do not yet exist.

Anticipating the definition of $p \| q$, to be given in Definition 6.7, we again remark that it is not the case that $\mathcal{O}[\![\rho_1 \cup \rho_2]\!] = \mathcal{O}[\![\rho_1]\!] \| \mathcal{O}[\![\rho_2]\!]$. As before, we shall remedy this by extending $\mathcal{T}_{nud}$ to $\mathcal{T}^*_{nud}$, and then introducing a corresponding extension of $\mathcal{O}$ to $\mathcal{O}^*$.

6.3. Denotational semantics

We proceed with the denotational semantic definitions. We first fill in the details of the definition of the merge operator "$\|$" (in this section, we do not use the operator "$\cdot$").

6.7. Definition. Let Ψ, ψ, $\tilde{\psi}$, $\hat{\psi}$, $\check{\psi}$, X, Y, and π be as in Definition 5.5(2), but with P as in Definition 6.5. The only new element in the definition of "$\|$" with respect to Definition 5.5 concerns $\pi_1|_{\sigma,\psi}\, \pi_2$, which is here given by

$$\langle \alpha, \beta\, ?x, p_1\rangle |_{\sigma,\psi} \langle \beta, \alpha\, !\alpha', p_2\rangle = \{\langle \sigma', p_1\, \psi\, p_2\rangle\},$$

$$\langle \alpha, ?x, p_1\rangle |_{\sigma,\psi} \langle \beta, \alpha\, !\alpha', p_2\rangle = \{\langle \sigma', p_1\, \psi\, p_2\rangle\},$$

$$\langle \alpha, \beta\, ?x, p_1\rangle |_{\sigma,\psi} \langle \beta, !\alpha', p_2\rangle = \{\langle \sigma', p_1\, \psi\, p_2\rangle\}$$

with $\sigma' = \sigma\{\sigma(\alpha)\{\alpha'/x\}/\alpha\}$, together with three symmetric clauses, and $\pi_1|_{\sigma,\psi}\, \pi_2 = \emptyset$ for π_1, π_2 not of the above form.

Corresponding to the distinction, for syntactic continuations, between statement continuations r and expression continuations g, we have a similar distinction at the semantic level: We have, besides the set of semantic statement continuations $SeStCo =^{\text{def}} P$ (with typical element p), also a set of semantic expression continuations $SeExCo =^{\text{def}} Obj \to P$, with typical element f.

Furthermore, corresponding to the two types of recursion, we accordingly have two components of an environment, defined as follows.

6.8. Definition. The set of environments is defined by $\Gamma = \Gamma_1 \times \Gamma_2$, with typical element $\gamma = \langle \gamma_{(1)}, \gamma_{(2)} \rangle$, where

$$\Gamma_1 = StmV \to (AObj \to (SeStCo \xrightarrow{\text{NDI}} P)) \quad \text{and} \quad \Gamma_2 = CNam \to (AObj \to P).$$

In an environment $\gamma = \langle \gamma_{(1)}, \gamma_{(2)} \rangle$, the first component $\gamma_{(1)}$ assigns an interpretation to each statement variable, which gives a process after being told which object is to execute the statement and which process is to be activated after this statement variable. This first component corresponds to the environments as used in Section 5.

The second component $\gamma_{(2)}$ is important for the creation of new objects. When given the class c and the name α of the object to be created, $\gamma_{(2)}(c)(\alpha)$ is the process to be activated for it.

Again, we shall often omit the indices in dealing with environments.

We shall define two semantic evaluation functions $\mathscr{D}$ and $\mathscr{E}$, the first for statements and programs, and the second for expressions. Since expressions are now more involved than in Section 5, we consequently need a more complicated definition of their meanings. The relevant types are

$$\mathscr{D}[\![\cdot]\!] : \mathscr{S}_{\text{nud}} \to (\Gamma \to (AObj \to (SeStCo \xrightarrow{\text{NDI}} P))),$$

$$\mathscr{E}[\![\cdot]\!] : Exp \to (\Gamma \to (AObj \to (SeExCo \xrightarrow{\text{NDI}} P)))$$

and, in addition, $\mathscr{D}[\![\cdot]\!] : \mathscr{L}_{\text{nud}} \to P$. We draw attention to the fact that $\mathscr{E}[\![e]\!]$, when supplied with some γ, α, and f, delivers a process $p \in P$ instead of some value $\beta \in Obj$. Values (i.e., objects) which result from evaluating an expression are always passed on to some expression continuation rather than being delivered explicitly by the semantic function.

6.9. Definition. (1) The function $\mathscr{E}$ is defined by

(a) $\mathscr{E}[\![x]\!]\gamma\alpha f = \lambda\sigma.\{\langle \sigma, f(\sigma(\alpha)(x))\rangle\}$;

(b) $\mathscr{E}[\![\beta]\!]\gamma\alpha f = f(\beta)$;

(c) $\mathscr{E}[\![e_1 \ \mathbf{op} \ e_2]\!]\gamma\alpha f = \mathscr{E}[\![e_1]\!]\gamma\alpha(\lambda\beta_1.\mathscr{E}[\![e_2]\!]\gamma\alpha(\lambda\beta_2.f(\beta_1 \ \mathbf{op}_{\text{sem}} \ \beta_2)))$;

(d) $\mathscr{E}[\![\mathbf{op} \ e]\!]\gamma\alpha f = \mathscr{E}[\![e]\!]\gamma\alpha(\lambda\beta.f(\mathbf{op}_{\text{sem}} \ \beta))$;

(e) $\mathscr{E}[\![\mathbf{new}(c)]\!]\gamma\alpha f = \lambda\sigma.\{\langle \sigma', \gamma(c)(\beta) \| f(\beta)\rangle\}$ where $\beta = \langle c, \sigma(c)+1\rangle$ and $\sigma' = \sigma\{\sigma(c)+1/c\}$.

(2) We define the function $\mathscr{D}$ for statements as follows:

(a) $\mathscr{D}[\![x := e]\!]\gamma\alpha p = \mathscr{E}[\![e]\!]\gamma\alpha(\lambda\beta.\lambda\sigma.\{\langle\sigma', p\rangle\})$ where $\sigma' = \sigma\{\sigma(\alpha)\{\beta/x\}/\alpha\}$;

(b) $\mathscr{D}[\![s_1; s_2]\!]\gamma\alpha p = \mathscr{D}[\![s_1]\!]\gamma\alpha(\mathscr{D}[\![s_2]\!]\gamma\alpha p)$;

(c) $\mathscr{D}[\![\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}]\!]\gamma\alpha p$
$= \lambda\sigma.\{\langle\sigma, \textbf{if } [\![b]\!]\alpha\sigma = \textbf{t then } \mathscr{D}[\![s_1]\!]\gamma\alpha p \textbf{ else } \mathscr{D}[\![s_2]\!]\gamma\alpha p \textbf{ fi}\rangle\}$;

(d) $\mathscr{D}[\![v]\!]\gamma\alpha p = \gamma_{(1)}(v)\alpha p$;

(e) $\mathscr{D}[\![\mu v[s]]\!]\gamma\alpha p = \varphi_\infty(\alpha)(p)$, where φ_∞ is the unique fixed point of the function Φ, from the space $AObj \to (SeStCo \to^{NDI} P)$ to itself, which is given by

$$\Phi(\varphi) = \lambda\alpha.\lambda p.\lambda\sigma.\{\langle\sigma, \mathscr{D}[\![s]\!]\gamma\{\varphi/v\}\alpha p\rangle\};$$

(f) $\mathscr{D}[\![e?x]\!]\gamma\alpha p = \mathscr{E}[\![e]\!]\gamma\alpha(\lambda\beta.\lambda\sigma.\{\langle\alpha, \beta?x, p\rangle\})$;

(g) $\mathscr{D}[\![?x]\!]\gamma\alpha p = \lambda\sigma.\{\langle\alpha, ?x, p\rangle\}$;

(h) $\mathscr{D}[\![e!e']\!]\gamma\alpha p = \mathscr{E}[\![e]\!]\gamma\alpha(\lambda\beta.\mathscr{E}[\![e']\!]\gamma\alpha(\lambda\beta'.\lambda\sigma.\{\langle\alpha, \beta!\beta', p\rangle\}))$;

(i) $\mathscr{D}[\![!e]\!]\gamma\alpha p = \mathscr{E}[\![e]\!]\gamma\alpha(\lambda\beta.\lambda\sigma.\{\langle\alpha, !\beta, p\rangle\})$.

(3) Let, for a program t, the mapping $\Psi_t : \Gamma_2 \to \Gamma_2$ be given as follows:

$$\Psi_t(\gamma_2)(c) = \lambda\alpha.\mathscr{D}[\![s]\!](\langle\gamma_1, \gamma_2\rangle)(\alpha)(p_0)$$

where $c \Leftarrow s$ occurs in t, and $\gamma_1 \in \Gamma_1$ is arbitrary (since t is closed, the choice of γ_1 is really immaterial). If c is not declared in t, we can put $\Psi_t(\gamma_2)(c) = \lambda\alpha.p_0$, for example.

Let γ_{2t} be the unique fixed point of Ψ_t (see the remark below). We put $\gamma_t =^{\text{def}} \langle\gamma_1, \gamma_{2t}\rangle$, for arbitrary $\gamma_1 \in \Gamma_1$.

(4) Now we can define the denotational semantics of programs as follows. Let $t = \langle c_1 \Leftarrow s_1, \ldots, c_n \Leftarrow s_n\rangle$. Then

$$\mathscr{D}[\![t]\!] = \mathscr{D}[\![s_1]\!](\gamma_t)(\langle c_1, 1\rangle)(p_0).$$

Remarks. (1) The clause for $\mathscr{E}[\![\textbf{new}(c)]\!]$ uses essentially the same idea as in Section 4 of putting the newly created process $\gamma(c)(\beta)$ in parallel with the (expression) continuation f (supplied with the new name β which is the value of the expression **new**(c)). Here $\gamma(c)(\beta)$—or $\gamma_{(2)}(c)(\beta)$, to be precise—will, in the context of a program $t = \langle c_k \Leftarrow s_k\rangle_{k=1}^n$, contain the relevant information on the class c as a result of the definition of γ_t (to be precise, γ_{2t}) in clause (3). We also observe that due to our requirement that all class names used in a program t must be also be declared in it, the result of γ_t for undeclared classes does not matter (actually, new objects of such classes would execute the process p_0).

(2) Clause (2)(e) is justified by the fact that the mapping Φ is contracting. Again we can obtain its unique fixed point by $\varphi_\infty = \lim_i \varphi_i$, where φ_0 is arbitrary and

$$\varphi_{i+1} = \lambda\alpha.\lambda p.\lambda\sigma.\{\langle\sigma, \mathscr{D}[\![s]\!]\gamma\{\varphi_i/v\}\alpha p\rangle\}.$$

(3) The mapping Ψ_t in clause (3) is contracting since recursive occurrences of c in any s are always constituents of statements which take time steps (specifically in evaluating **new**(c)) before we apply γ to such a recursive occurrence of c.

6.4. *Equivalence of operational and denotational semantics*

We start this section with the promised extension of $\mathscr{T}_{\text{nus}}$ and $\mathscr{O}$.

6.10. Definition. (1) The notion of configuration is expanded so as to include pairs of the form $\langle \rho, \eta \rangle$ (note that η ranges over $Step = \Sigma \cup Comm$).

(2) We obtain the transition system $\mathscr{T}^*_{\text{nud}}$ from $\mathscr{T}_{\text{nud}}$ by adding the axioms

$$\langle \rho \cup \{\langle \alpha, (\beta?x);r\rangle\}, \sigma\rangle \to \langle \rho \cup \{\langle \alpha, r\rangle\}, \langle \alpha, \beta?x\rangle\rangle, \qquad \textbf{Receive2}$$

$$\langle \rho \cup \{\langle \alpha, (?x);r\rangle\}, \sigma\rangle \to \langle \rho \cup \{\langle \alpha, r\rangle\}, \langle \alpha, ?x\rangle\rangle, \qquad \textbf{Receive3}$$

$$\langle \rho \cup \{\langle \alpha, (\beta!\beta');r\rangle\}, \sigma\rangle \to \langle \rho \cup \{\langle \alpha, r\rangle\}, \langle \alpha, \beta!\beta'\rangle\rangle, \qquad \textbf{Send3}$$

$$\langle \rho \cup \{\langle \alpha, (!\beta);r\rangle\}, \sigma\rangle \to \langle \rho \cup \{\langle \alpha, r\rangle\}, \langle \alpha, !\beta\rangle\rangle \qquad \textbf{Send4}$$

and by replacing, in all rules,

$$\frac{\langle \rho_1, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \rho_2, \sigma\rangle \to \langle \rho', \sigma'\rangle}$$

by

$$\frac{\langle \rho_1, \sigma\rangle \to \langle \rho', \eta'\rangle}{\langle \rho_2, \sigma\rangle \to \langle \rho', \eta'\rangle}.$$

(3) Now we define $\mathscr{O}^*[\![\cdot]\!] : PSyCo \to P$ by

$$\mathscr{O}^*[\![\rho]\!] = \begin{cases} p_0 & \text{if } \rho = \{\langle \alpha_1, E\rangle, \ldots, \langle \alpha_n, E\rangle\}, \\ \lambda\sigma.\{\langle \eta', \mathscr{O}^*[\![\rho']\!]\rangle \mid \langle \rho, \sigma\rangle \to \langle \rho', \eta'\rangle\} & \text{otherwise.} \end{cases}$$

(4) $\mathscr{O}^*[\![\cdot]\!] : \mathscr{L}_{\text{nud}} \to P$ is defined as follows. Let $t = \langle c_k \Leftarrow s_k\rangle_{k=1}^n$. Then

$$\mathscr{O}^*[\![t]\!] = \mathscr{O}^*[\![\{\langle\langle c_1, 1\rangle, s_1; E\rangle\}]\!].$$

As in Section 5, we have the following lemma.

6.11. Lemma. $\mathscr{O}^*[\![\rho_1 \cup \rho_2]\!] = \mathscr{O}^*[\![\rho_1]\!] \parallel \mathscr{O}^*[\![\rho_2]\!]$.

The abstraction operator *abs* can be defined as in Definition 5.11 (but now applied to P as in Definition 6.5). Again, we have

6.12. Lemma. $\mathscr{O} = abs \circ \mathscr{O}^*$.

We can now discuss the relationship between $\mathscr{O}^*$ and $\mathscr{D}$. The treatment combines ideas of Sections 4 and 5. We first present a lemma listing various properties of $\mathscr{O}^*$

which are either direct from its definition, or follow as in Section 5 (in turn relying on [16]).

6.13. Lemma. (1) $\mathcal{O}^*[\![\{\langle \alpha, (x := \beta); r\rangle\}]\!] = \lambda\sigma.\{\langle \sigma', \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!]\rangle\}$ *with* σ' *as usual.*

(2) $\mathcal{O}^*[\![\{\langle \alpha, (x := e); r\rangle\}]\!] = \mathcal{O}^*[\![\{\langle \alpha, e : \lambda z.((x := z); r)\rangle\}]\!]$ *where* z *is fresh.*

(3) $\mathcal{O}^*[\![\{\langle \alpha, (s_1; s_2); r\rangle\}]\!] = \mathcal{O}^*[\![\{\langle \alpha, s_1; (s_2; r)\rangle\}]\!]$.

(4) $\mathcal{O}^*[\![\{\langle \alpha,$ **if** b **then** s_1 **else** s_1 **fi**$; r\rangle\}]\!]$

$= \lambda\sigma.\{\langle \sigma,$ **if** $[\![b]\!]\alpha\sigma$ **then** $\mathcal{O}^*[\![\{\langle \alpha, s_1; r\rangle\}]\!]$ **else** $\mathcal{O}^*[\![\{\langle \alpha, s_2; r\rangle\}]\!]$**fi**$\rangle\}$.

(5) $\mathcal{O}^*[\![\{\langle \alpha, \mu v[s]; r\rangle\}]\!] = \lim_n \mathcal{O}^*[\![\{\langle \alpha, s_v^{(n)}; r\rangle\}]\!]$ *where* $s_v^{(0)} =$ **skip** *and* $s_v^{(n+1)} =$ **skip**$; s[s_v^{(n)}/v]$. *Note that here we cannot use* $x := x$ *for* **skip** *any more because* $x := x$ *now costs* two *steps.*

(6) $\mathcal{O}^*[\![\{\langle \alpha, (e?x); r\rangle\}]\!] = \mathcal{O}^*[\![\{\langle \alpha, e : \lambda z.((z?x); r)\rangle\}]\!]$ *with* z *fresh, and similar equations for* $e!e'$ *and* $!e$.

(7) $\mathcal{O}^*[\![\{\langle \alpha, (\beta?x); r\rangle\}]\!] = \lambda\sigma.\{\langle \alpha, \beta?x, \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!]\rangle\}$ *and similar equations for* $?x$, $\beta!\beta'$, *and* $!\beta$.

(8) $\mathcal{O}^*[\![\{\langle \alpha, (\beta?x); r_1\rangle, \langle \beta, (\alpha!\alpha'); r_2\rangle\}]\!] = \lambda\sigma.\{\langle \alpha, \beta?x, \mathcal{O}^*[\![\{\langle \alpha, r_1\rangle, \langle \beta, (\alpha!\alpha'); r_2\rangle\}]\!]\rangle$, $\langle \beta, \alpha!\alpha', \mathcal{O}^*[\![\{\langle \alpha, (\beta?x); r_1\rangle, \langle \beta, r_2\rangle\}]\!]\rangle, \langle \sigma', \mathcal{O}^*[\![\{\langle \alpha, r_1\rangle, \langle \beta, r_2\rangle\}]\!]\rangle\}$ *where* σ' *is as usual, and similar equations for* $?x$ *with* $\alpha!\alpha'$ *and for* $\beta?x$ *with* $!\alpha'$.

(9) $\mathcal{O}^*[\![\{\langle \alpha, x : g\rangle\}]\!] = \lambda\sigma.\{\langle \sigma, \mathcal{O}^*[\![\{\langle \alpha, \sigma(\alpha)(x) : g\rangle\}]\!]\rangle\}$.

(10) $\mathcal{O}^*[\![\{\langle \alpha, \beta : \lambda z.r\rangle\}]\!] = \mathcal{O}^*[\![\{\langle \alpha, r[\beta/z]\rangle\}]\!]$.

(11) $\mathcal{O}^*[\![\{\langle \alpha, (\beta_1 \mathbf{op} \beta_2) : g\rangle\}]\!] = \mathcal{O}^*[\![\{\langle \alpha, (\beta_1 \mathbf{op}_{\mathrm{sem}} \beta_2) : g\rangle\}]\!]$ *and a similar equation for unary operators.*

(12) $\mathcal{O}^*[\![\{\langle \alpha, (e_1 \mathbf{op} e_2) : g\rangle\}]\!] = \mathcal{O}^*[\![\{\langle \alpha, e_1 : \lambda z_1.(e_2 : \lambda z_2.((z_1 \mathbf{op} z_2) : g))\rangle\}]\!]$ *and a similar equation for unary operators.*

(13) $\mathcal{O}^*[\![\{\alpha, \mathbf{new}(c) : g\rangle\}]\!] = \lambda\sigma.\{\langle \sigma', \mathcal{O}^*[\![\{\langle \alpha, \beta : g\rangle, \langle \beta, s; E\rangle\}]\!]\rangle\}$ *where* $c \Leftarrow s$ *occurs in* t *and with* $\sigma' = \sigma\{\sigma(c)+1/c\}$ *and* $\beta = \langle c, \sigma(c)+1\rangle$.

We continue with the analysis which links $\mathcal{O}^*$ with $\mathcal{D}$ and $\mathcal{E}$. Our aim is the proof of the following theorem.

6.14. Theorem. *For a given program* $t = \langle c_k \Leftarrow s_k\rangle_{k=1}^n$, *for closed* s, *arbitrary* r, e *and* g, *and for* γ_t *as in Definition* 6.9(3), *we have*

(1) $\mathcal{O}^*[\![\{\langle \alpha, e : g\rangle\}]\!] = \mathcal{E}[\![e]\!](\gamma_t)(\alpha)(\lambda\beta.\mathcal{O}^*[\![\{\langle \alpha, \beta : g\rangle\}]\!])$,

(2) $\mathcal{O}^*[\![\{\langle \alpha, s; r\rangle\}]\!] = \mathcal{D}[\![s]\!](\gamma_t)(\alpha)(\mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!])$.

In order to prove this theorem, we apply a nonuniform version of the strategy used at the end of Section 4. Since we are concerned with both statements and expressions, we need the nonuniform argument in two forms. Firstly, we introduce the branching time analogues of the constructs $u\surd v$ from Section 4. One form also mentions the $\surd$, the other one is parameterized by objects β from *Obj*, each of which plays a role similar to the one played by $\surd$. For the remainder of this section we introduce *three* domains P, Q, and R with typical elements p, q, and r respectively (the last not to be confused with $r \in SyStCo$).

6.15. Definition. (1) Recall from Definition 6.5 that P is the solution of

$$P \cong \{p_0\} \cup (\Sigma \to \mathcal{P}_{\mathrm{cl}}(Step \times P)).$$

As before, we shall use X to range over $\mathcal{P}_{\mathrm{cl}}(Step \times P)$ and π to range over $Step \times P$.

(2) The domain Q is the solution of the following domain equation

$$Q \cong \{p_0\} \cup (\{\surd\} \times P) \cup (\Sigma \to \mathcal{P}_{\mathrm{cl}}(Step \times Q)).$$

We shall use Y to range over $\mathcal{P}_{\mathrm{cl}}(Step \times Q)$ and ξ to range over $Step \times Q$.

(3) The domain R is defined as the solution of

$$R \cong (Obj \times P) \cup (\Sigma \to P_{\mathrm{cl}}(Step \times R)).$$

We shall use Z to range over $\mathcal{P}_{\mathrm{cl}}(Step \times R)$ and ζ to range over $Step \times R$.

The intuitive interpretation of Q and R is as follows. An element of Q is a process executing a specific *statement* (the "local" one), possibly in parallel with some other processes. Termination of the local statement is explicitly indicated by $\surd$. The idea is that a continuation can start at that point (see the definition of the operator ":" below). More specifically, if $q \in Q$ is of the form $\langle \surd, p \rangle$ this means that the local process terminates immediately, and that the parallel processes continue with p. If in q the local process does not terminate immediately, an ordinary step is possible, after which we come in the same situation again. Because we have also included p_0 in Q, P can be embedded in Q in a canonical way. We shall therefore assume that actually $P \subseteq Q$.

An element of R is evaluating an *expression*, again possibly in parallel with other processes. It will be composed with elements of $Obj \to Q$ or $Obj \to R$ by the operator ":". If the evaluation of the expression terminates, it delivers a value β being the result of this expression, together with an ordinary process p representing the ongoing computation of the other processes (which is to be executed in parallel with the semantic expression continuation).

We shall define four forms of the operator ":" which will take care of the composition of elements of Q and R with appropriate continuations (notice the analogy with Definition 4.7):

6.16. Definition. (1) We define ":": $Q \times Q \to Q$ by the following clauses (which can be completed to a full definition along the lines of Definition 5.5):

(a) $p_0 : q = p_0$;

(b) $\langle \surd, p \rangle : q = p \| q$ (see Definition 6.17 below);

(c) $(\lambda\sigma.Y) : q = \lambda\sigma.(Y : q)$, where $Y : q = \{\xi : q \mid \xi \in Y\}$ and $\langle \eta, q' \rangle : q = \langle \eta, q' : q \rangle$.

(2) We define ":": $Q \times R \to R$ as follows:

(a) $p_0 : r = p_0$;

(b) $\langle \surd, p \rangle : r = p \| r$ (see Definition 6.17);

(c) $(\lambda\sigma.Y) : r = \lambda\sigma.(Y : r)$, where $Y : r = \{\xi : r \mid \xi \in Y\}$ and $\langle \eta, q' \rangle : r = \langle \eta, q' : r \rangle$.

(3) The operator ":": $R \times (Obj \to Q) \to Q$ is given by the following clauses:

(a) $\langle \beta, p \rangle : f = p \| f(\beta)$;

(b) $(\lambda\sigma.Z) : f = \lambda\sigma.(Z : f)$, where $Z : f = \{\zeta : f \mid \zeta \in Z\}$ and $\langle \eta, r \rangle : f = \langle \eta, r : f \rangle$.

(4) Finally, we define the operator ":": $R \times (Obj \to R) \to R$ by the following clauses (we shall use h to range over $Obj \to R$):

(a) $\langle \beta, p \rangle : h = p \| h(\beta)$;

(b) $(\lambda\sigma.Z) : h = \lambda\sigma.(Z : h)$, where $Z : h = \{\zeta : h \mid \zeta \in Z\}$ and $\langle \eta, r \rangle : h = \langle \eta, r : h \rangle$.

Note that if $q \in P$, then $p : q \in P$, so that we also have ":": $Q \times P \to P$. Analogously, if $f \in Obj \to P$, then we get $r : f \in P$, so that we can state ":": $R \times (Obj \to P) \to P$.

We also need the definitions of $p \| q$ and $p \| r$:

6.17. Definition. (1) We define the operator "$\|$": $P \times Q \to Q$ by the following clauses:

(a) $p_0 \| q = q$, $p \| p_0 = p$, $p \| \langle \surd, p' \rangle = \langle \surd, p \| p' \rangle$;

(b) for $p \neq p_0$ and $q \notin \{p_0\} \cup (\{\surd\} \times P)$ we define

$$p \| q = \lambda\sigma.((p(\sigma) \| q) \cup (p \| q(\sigma)) \cup (p(\sigma) |_\sigma q(\sigma)));$$

(c) for $X \in \mathcal{P}_{cl}(Step \times P)$ we put $X \| q = \{\pi \| q \mid \pi \in X\}$, where $\langle \eta, p' \rangle \| q = \langle \eta, p' \| q \rangle$;

(d) for $Y \in \mathcal{P}_{cl}(Step \times Q)$ we put $p \| Y = \{p \| \xi \mid \xi \in Y\}$, where $p \| \langle \eta, q' \rangle = \langle \eta, p \| q' \rangle$;

(e) for X and Y as above, we define

$$X |_\sigma Y = \bigcup \{\pi |_\sigma \xi \mid \pi \in X, \xi \in Y\}$$

where $\langle \eta_1, p' \rangle |_\sigma \langle \eta_2, q' \rangle = \{\langle \sigma', p' \| q' \rangle\}$ with σ' as usual if η_1 and η_2 are matching communications, and $\pi |_\sigma \xi = \emptyset$ otherwise.

Note that restricted to $P \times P$ this coincides with the old operator "$\|$" (see Definition 6.7).

(2) We define the operator "$\|$": $P \times R \to R$ by the following clauses:

(a) $p_0 \| r = r$, $p \| \langle \beta, p' \rangle = \langle \beta, p \| p' \rangle$;

(b) for $p \neq p_0$ and $r \notin Obj \times P$ we define

$$p \| r = \lambda\sigma.((p(\sigma) \| r) \cup (p \| r(\sigma)) \cup (p(\sigma) |_\sigma r(\sigma)));$$

(c) for $X \in \mathcal{P}_{cl}(Step \times P)$ we put $X \| r = \{\pi \| r \mid \pi \in X\}$, where $\langle \eta, p' \rangle \| r = \langle \eta, p' \| r \rangle$;

(d) for $Z \in \mathcal{P}_{cl}(Step \times R)$ we put $p \| Z = \{p \| \zeta \mid \zeta \in Z\}$, where $p \| \langle \eta, r' \rangle = \langle \eta, p \| r' \rangle$;

(e) for X and Z as above, we define

$$X |_\sigma Z = \bigcup \{\pi |_\sigma \zeta \mid \pi \in X, \zeta \in Z\}$$

where $\langle \eta_1, p' \rangle |_\sigma \langle \eta_2, r' \rangle = \{\langle \sigma', p' \| r' \rangle\}$ with σ' as usual if η_1 and η_2 are matching communications, and $\pi |_\sigma \zeta = \emptyset$ otherwise.

Analogous to Lemma 4.8 we have the following important lemma.

6.18. Lemma. (1) *All forms of the mappings* ":" *and* "$\|$" *are continuous.*

(2) *The operators* "$\|$" *are associative*:

(a) $(p_1 \| p_2) \| q = p_1 \| (p_2 \| q)$,

(b) $(p_1 \| p_2) \| r = p_1 \| (p_2 \| r)$.

(3) *The operators* ":" *with the first argument from* Q *are associative*:
(a) $(q_1:q_2):q_3 = q_1:(q_2:q_3)$,
(b) $(q_1:q_2):r = q_1:(q_2:r)$.
(4) *The operators* ":" *with the first argument from* R *have an analogous property* (*let us call it* λ*-associativity*):
(a) $(r:f):q = r:\lambda\beta.(f(\beta):q)$,
(b) $(r:f):r' = r:\lambda\beta.(f(\beta):r')$,
(c) $(r:h):f = r:\lambda\beta.(h(\beta):f)$,
(d) $(r:h):h' = r:\lambda\beta.(h(\beta):h')$.
(5) *Finally, we have a kind of distributivity*:
(a) $(p\|q):q' = p\|(q:q')$,
(b) $(p\|q):r = p\|(q:r)$,
(c) $(p\|r):f = p\|(r:f)$,
(d) $(p\|r):h = p\|(r:h)$.

Proof. Part (1) can be proved by observing that each version of ":" or "$\|$" is the unique fixed point of an appropriate higher-order function that maps continuous operators into continuous operators. Therefore, ":" and "$\|$" are themselves continuous.

For the other parts, one first proves that $p:q = p$ and $p:r = p$ for all $p \in P$, $q \in Q$, and $r \in R$. The rest of the properties are then proved in the order (2)-(5)-(3)-(4), by a metric argument. We illustrate this technique by giving the proof of part (3)(a). (We assume that part (5) has already been proved.) Consider the operators Φ and Ψ, given by $\Phi(q_1, q_2, q_3) = (q_1:q_2):q_3$ and $\Psi(q_1, q_2, q_3) = q_1:(q_2:q_3)$. Both can be seen as elements of the metric space $Q \times Q \times Q \to Q$. We shall show that $\Phi = \Psi$ by proving $d(\Phi, \Psi) = 0$. Let us therefore denote $d(\Phi, \Psi)$ by ε, or in other words,

$$\varepsilon = \sup_{q_1,q_2,q_3 \in Q} d_Q((q_1:q_2):q_3, q_1:(q_2:q_3)). \tag{6.1}$$

Now let $q_1, q_2, q_3 \in Q$ be arbitrary. We show

$$d_Q((q_1:q_2):q_3, q_1:(q_2:q_3)) \leqslant \tfrac{1}{2}\varepsilon. \tag{6.2}$$

Distinguish the following cases:
(1) $q_1 = p_0$. Then $(q_1:q_2):q_3 = p_0:q_3 = p_0 = q_1:(q_2:q_3)$.
(2) $q_1 = \langle\surd, p\rangle$. Then $(q_1:q_2):q_3 = (p\|q_2):q_3 =$ (by part (5)(a)) $p\|(q_2:q_3) = q_1:(q_2:q_3)$.
(3) $q_1 \in \Sigma \to \mathcal{P}_{\mathrm{cl}}(Step \times Q)$. Now by Definition 6.16 we have that $q_1:q_2$, $(q_1:q_2):q_3$, and $q_1:(q_2:q_3)$ are also elements of $\Sigma \to \mathcal{P}_{\mathrm{cl}}(Step \times Q)$. Let $\sigma \in \Sigma$ be arbitrary and set $Y = q_1(\sigma)$. Then we get, by Definition 6.16,

$$(q_1:q_2)(\sigma) = Y:q_2 = \{\xi:q_2 \mid \xi \in Y\} = \{\langle\eta, q':q_2\rangle \mid \langle\eta, q'\rangle \in Y\},$$

$$((q_1:q_2):q_3)(\sigma) = (Y:q_2):q_3 = \{\langle\eta, (q':q_2):q_3\rangle \mid \langle\eta, q'\rangle \in Y\},$$

and

$$(q_1:(q_2:q_3))(\sigma) = Y:(q_2:q_3) = \{\langle\eta, q':(q_2:q_3)\rangle \mid \langle\eta, q'\rangle \in Y\}.$$

Now the time has come to remember our convention from Section 2.3 that, implicitly, every occurrence in the right-hand side of the domain being defined is surrounded by $\mathrm{id}_{1/2}$ (cf. equation (2.3′)). Of course, this also holds for the defining equation for Q in Definition 6.15. From (6.1) it follows that

$$d_Q((q':q_2):q_3, q':(q_2:q_3)) \leqslant \varepsilon.$$

Therefore

$$d_{\mathrm{id}_{1/2}(Q)}((q':q_2):q_3, q':(q_2:q_3)) \leqslant \tfrac{1}{2}\varepsilon.$$

By applying the clauses of Definition 2.7 (and remembering that σ was arbitrary) we can conclude that

$$d_Q((q_1:q_2):q_3, q_1:(q_2:q_3)) \leqslant \tfrac{1}{2}\varepsilon.$$

Because q_1, q_2, and q_3 were arbitrary in (6.2), we can conclude from (6.1) that $\varepsilon \leqslant \frac{1}{2}\varepsilon$, so that $d(\Phi, \Psi) = \varepsilon = 0$ and $\Phi = \Psi$. □

Next, we state the analogues of Lemma 4.9 and Corollary 4.10. By way of preparation we need some extensions to the definitions of *PSyCo* and $\mathcal{O}^*$.

6.19. Definition. (1) We define the set *PSyCo′*, with typical element $\dot{\rho}$, to be the same as *PSyCo*, except that *at most one* of the components has an $\dot{r} \in SyStCo'$, defined (together with $\dot{g} \in SyExCo'$) by

$$\dot{r} ::= \surd \mid s;\dot{r}' \mid e:\dot{g} \qquad \dot{g} ::= \lambda z.\dot{r}$$

with s closed.

(2) The set *PSyCo″*, with typical element $\ddot{\rho}$, is the same as *PsyCo* except that *exactly one* component has an $\ddot{r} \in SyStCo''$, which is defined together with $\ddot{g} \in SyExCo''$ by

$$\ddot{r} ::= s;\ddot{r}' \mid e:\ddot{g} \qquad \ddot{g} ::= \lambda z.\ddot{r} \mid \surd$$

with s closed.

(3) We define the function $\dot{\mathcal{O}}[\![\cdot]\!]: PSyCo' \to Q$ as follows

$$\dot{\mathcal{O}}[\![\dot{\rho}]\!] = \begin{cases} p_0 & \text{if } \dot{\rho} = \{\langle \alpha_1, E\rangle, \ldots, \langle \alpha_k, E\rangle\}, \\ \langle \surd, \mathcal{O}^*[\![\rho']\!]\rangle & \text{if } \dot{\rho} = \{\langle \alpha, \surd\rangle\} \cup \rho', \\ \lambda\sigma.\{\langle \sigma', \dot{\mathcal{O}}[\![\dot{\rho}']\!]\rangle \mid \langle \sigma, \dot{\rho}\rangle \to \langle \sigma', \dot{\rho}'\rangle\} & \text{otherwise.} \end{cases}$$

Here we interpret the transition relation "$\to$" with respect to $\mathcal{T}^*_{\mathrm{nud}}$ (only extended in so far that we declare the existing axioms and rules also applicable to our new parallel syntactic continuations).

(4) We define the function $\ddot{\mathcal{O}}[\![\cdot]\!]: PSyCo'' \to R$ as follows

$$\ddot{\mathcal{O}}[\![\ddot{\rho}]\!] = \begin{cases} \langle \beta, \mathcal{O}^*[\![\rho']\!]\rangle & \text{if } \ddot{\rho} = \{\langle \alpha, \beta:\surd\rangle\} \cup \rho', \\ \lambda\sigma.\{\langle \sigma', \ddot{\mathcal{O}}[\![\ddot{\rho}']\!]\rangle \mid \langle \sigma, \ddot{\rho}\rangle \to \langle \sigma', \ddot{\rho}'\rangle\} & \text{otherwise.} \end{cases}$$

Note that $PsyCo \subseteq PSyCo'$, and that $\dot{\mathcal{O}}$ restricted to *PsyCo* is equal to $\mathcal{O}^*$. Furthermore, Lemma 6.13 also holds for $\dot{\mathcal{O}}$ and $\ddot{\mathcal{O}}$, and we can restate Lemma 6.11 as follows.

6.20. Lemma. (1) $\dot{\mathcal{O}}[\![\rho \cup \dot{\rho}]\!] = \mathcal{O}^*[\![\rho]\!] \| \dot{\mathcal{O}}[\![\dot{\rho}]\!]$.
(2) $\ddot{\mathcal{O}}[\![\rho \cup \ddot{\rho}]\!] = \mathcal{O}^*[\![\rho]\!] \| \ddot{\mathcal{O}}[\![\ddot{\rho}]\!]$.

Now we can state the next lemma.

6.21. Lemma. (1) *For any* $e \in Exp$, $\alpha \in AObj$, *and* $g \in SyExCo$ *we have*

$$\mathcal{O}^*[\![\{\langle \alpha, e{:}g\rangle\}]\!] = \ddot{\mathcal{O}}[\![\{\langle \alpha, e{:}\surd\rangle\}]\!] : (\lambda\beta.\mathcal{O}^*[\![\{\langle \alpha, \beta{:}g\rangle\}]\!])$$

and the same for any $\dot{g}$ *with* $\mathcal{O}^*$ *replaced by* $\dot{\mathcal{O}}$ *and for any* $\ddot{g}$ *with* $\mathcal{O}^*$ *replaced by* $\ddot{\mathcal{O}}$.

(2) *Let* $s \in \mathcal{S}_{\text{nud}}$ (*not necessarily closed*) *and let all free statement variables of* s *be contained in* $\{v_1, \ldots, v_k\}$. *Now let* $s_1, \ldots, s_k$ *be closed statements such that, for any* α *and* r,

$$\mathcal{O}^*[\![\{\langle \alpha, s_i\,; r\rangle\}]\!] = \dot{\mathcal{O}}[\![\{\langle \alpha, s_i\,; \surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!]$$

and for any $\dot{r}$ *the same with* $\mathcal{O}^*$ *replaced by* $\dot{\mathcal{O}}$ *and for any* $\ddot{r}$ *the same with* $\mathcal{O}^*$ *replaced by* $\ddot{\mathcal{O}}$. *If we define* $\tilde{s} = s[s_i / v_i]_{i=1}^k$, *then we have, for any* α *and* r,

$$\mathcal{O}^*[\![\{\langle \alpha, \tilde{s}\,; r\rangle\}]\!] = \dot{\mathcal{O}}[\![\{\langle \alpha, \tilde{s}\,; \surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!]$$

and analogously for any $\dot{r}$ *and for any* $\ddot{r}$.

Proof. Part (1) is proved by induction on the complexity of e. We give some typical cases:

Case 1: $e = \beta$.

$$\begin{aligned}
&\ddot{\mathcal{O}}[\![\{\langle \alpha, \beta{:}\surd\rangle\}]\!] : (\lambda\beta'.\mathcal{O}^*[\![\{\langle \alpha, \beta'{:}g\rangle\}]\!]) \\
&\quad = \langle \beta, p_0\rangle : (\lambda\beta'.\mathcal{O}^*[\![\{\langle \alpha, \beta'{:}g\rangle\}]\!]) && \text{(Definition 6.19)} \\
&\quad = p_0 \| \mathcal{O}^*[\![\{\langle \alpha, \beta{:}g\rangle\}]\!] && \text{(Definition 6.16)} \\
&\quad = \mathcal{O}^*[\![\{\langle \alpha, \beta{:}g\rangle\}]\!] && \text{(Definition 6.7).}
\end{aligned}$$

Exactly the same proof works for $\dot{g}$ with $\dot{\mathcal{O}}$ and for $\ddot{g}$ with $\ddot{\mathcal{O}}$.

Case 2: $e = \mathbf{op}\ e'$.

$$\begin{aligned}
&\mathcal{O}^*[\![\{\langle \alpha, (\mathbf{op}\ e'){:}g\rangle\}]\!] \\
&\quad = \mathcal{O}^*[\![\{\langle \alpha, e'{:}\lambda z.(\mathbf{op}\ z : g)\rangle\}]\!] && \text{(Lemma 6.13(12))} \\
&\quad = \ddot{\mathcal{O}}[\![\{\langle \alpha, e'{:}\surd\rangle\}]\!] : (\lambda\beta'.\mathcal{O}^*[\![\{\langle \alpha, \beta'{:}\lambda z.(\mathbf{op}\ z : g)\rangle\}]\!]) && \text{(ind. hyp.)} \\
&\quad = \ddot{\mathcal{O}}[\![\{\langle \alpha, e'{:}\surd\rangle\}]\!] : (\lambda\beta'.\mathcal{O}^*[\![\{\langle \alpha, \mathbf{op}\ \beta' : g\rangle\}]\!]) && \text{(Lemma 6.13(10))} \\
&\quad = \ddot{\mathcal{O}}[\![\{\langle \alpha, e'{:}\surd\rangle\}]\!] : (\lambda\beta'.\mathcal{O}^*[\![\{\langle \alpha, \mathbf{op}_{\text{sem}}\ \beta' : g\rangle\}]\!]) && \text{(Lemma 6.13(11))} \\
&\quad = \ddot{\mathcal{O}}[\![\{\langle \alpha, e'{:}\surd\rangle\}]\!] : (\lambda\beta'.\ddot{\mathcal{O}}[\![\{\langle \alpha, \mathbf{op}_{\text{sem}}\ \beta' : \surd\rangle\}]\!] : (\lambda\beta.\mathcal{O}^*[\![\{\langle \alpha, \beta{:}g\rangle\}]\!])) \\
&&& \text{(Case 1)} \\
&\quad = (\ddot{\mathcal{O}}[\![\{\langle \alpha, e'{:}\surd\rangle\}]\!] : (\lambda\beta'.\ddot{\mathcal{O}}[\![\{\langle \alpha, \mathbf{op}_{\text{sem}}\ \beta' : \surd\rangle\}]\!])) : (\lambda\beta.\mathcal{O}^*[\![\{\langle \alpha, \beta{:}g\rangle\}]\!]) \\
&&& \text{(Lemma 6.18(4))}
\end{aligned}$$

$= (\ddot{\mathcal{O}}[\![\{\langle \alpha, e' : \surd \rangle\}]\!] : (\lambda \beta'.\ddot{\mathcal{O}}[\![\{\langle \alpha, \beta' : \lambda z.(\mathbf{op}\ z : \surd)\rangle\}]\!])) : (\lambda \beta.\mathcal{O}^*[\![\{\langle \alpha, \beta : g\rangle\}]\!])$
(Lemma 6.13(11, 10))

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, e' : \lambda z.(\mathbf{op}\ z : \surd)\rangle\}]\!] : (\lambda \beta.\mathcal{O}^*[\![\{\langle \alpha, \beta : g\rangle\}]\!])$ (ind. hyp.)

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, \mathbf{op}\ e' : \surd\rangle\}]\!] : (\lambda \beta.\mathcal{O}^*[\![\{\langle \alpha, \beta : g\rangle\}]\!])$. (Lemma 6.13(12))

Again, the proof is also valid for $\dot{g}$ and $\ddot{g}$.

Case 3: $e = \mathbf{new}(c)$.

$\mathcal{O}^*[\![\{\langle \alpha, \mathbf{new}(c) : g\rangle\}]\!]$

$= \lambda \sigma.\{\langle \sigma', \mathcal{O}^*[\![\{\langle \alpha, \beta : g\rangle, \langle \beta, s; E\rangle\}]\!]\rangle\}$
(Lemma 6.13(13), with s, σ', and β as usual)

$= \lambda \sigma.\{\langle \sigma', \mathcal{O}^*[\![\{\langle \beta, s; E\rangle\}]\!] \,\|\, \mathcal{O}^*[\![\{\langle \alpha, \beta : g\rangle\}]\!]\rangle\}$ (Lemma 6.11)

$= \lambda \sigma.\{\langle \sigma', \mathcal{O}^*[\![\{\langle \beta, s; E\rangle\}]\!] \,\|\, (\ddot{\mathcal{O}}[\![\{\langle \alpha, \beta : \surd\rangle\}]\!] : (\lambda \beta'.\mathcal{O}^*[\![\{\langle \alpha, \beta' : g\rangle\}]\!]))\rangle\}$
(Case 1)

$= \lambda \sigma.\{\langle \sigma', (\mathcal{O}^*[\![\{\langle \beta, s; E\rangle\}]\!] \,\|\, \ddot{\mathcal{O}}[\![\{\langle \alpha, \beta : \surd\rangle\}]\!]) : (\lambda \beta'.\mathcal{O}^*[\![\{\langle \alpha, \beta' : g\rangle\}]\!])\rangle\}$
(Lemma 6.18(5))

$= \lambda \sigma.\{\langle \sigma', \mathcal{O}^*[\![\{\langle \beta, s; E\rangle\}]\!] \,\|\, \ddot{\mathcal{O}}[\![\{\langle \alpha, \beta : \surd\rangle\}]\!]\rangle\} : (\lambda \beta'.\mathcal{O}^*[\![\{\langle \alpha, \beta' : g\rangle\}]\!])$
(Definition 6.16)

$= \lambda \sigma.\{\langle \sigma', \ddot{\mathcal{O}}[\![\{\langle \beta, s; E\rangle, \langle \alpha, \beta : \surd\rangle\}]\!]\rangle\} : (\lambda \beta'.\mathcal{O}^*[\![\{\langle \alpha, \beta' : g\rangle\}]\!])$
(Lemma 6.20)

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, \mathbf{new}(c) : \surd\rangle\}]\!] : (\lambda \beta'.\mathcal{O}^*[\![\{\langle \alpha, \beta' : g\rangle\}]\!])$. (Lemma 6.13(13))

Once again, the proof is also valid for $\dot{g}$ and $\ddot{g}$.

Now we can prove part (2) by induction on the complexity of s. Again some typical cases:

Case 4: $s = x := e$ (so $\tilde{s} = s$).

$\mathcal{O}^*[\![\{\langle \alpha, x := e; r\rangle\}]\!]$

$= \mathcal{O}^*[\![\{\langle \alpha, e : \lambda z.(x := z; r)\rangle\}]\!]$ (Lemma 6.13(2))

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, e : \surd\rangle\}]\!] : (\lambda \beta.\mathcal{O}^*[\![\{\langle \alpha, \beta : \lambda z.(x := z; r)\rangle\}]\!])$ (part (1))

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, e : \surd\rangle\}]\!] : (\lambda \beta.\mathcal{O}^*[\![\{\langle \alpha, x := \beta; r\rangle\}]\!])$ (Lemma 6.13(10))

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, e : \surd\rangle\}]\!] : (\lambda \beta.\lambda \sigma.\{\langle \sigma', \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!]\rangle\})$ (Lemma 6.13(1), σ' as usual)

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, e : \surd\rangle\}]\!] : (\lambda \beta.\lambda \sigma.\{\langle \sigma', \dot{\mathcal{O}}[\![\{\langle \alpha, \surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!]\rangle\})$

(because $\dot{\mathcal{O}}[\![\{\langle \alpha, \surd\rangle\}]\!] = \langle \surd, p_0\rangle$ and $\langle \surd, p_0\rangle : q = p_0 \| q = q$)

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, e : \surd\rangle\}]\!] : (\lambda \beta.\lambda \sigma.\{\langle \sigma', \dot{\mathcal{O}}[\![\{\langle \alpha, \surd\rangle\}]\!]\rangle\} : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!])$
(Definition 6.16)

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, e{:}\surd\rangle\}]\!] : (\lambda\beta.\dot{\mathcal{O}}[\![\{\langle \alpha, x := \beta;\surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!])$ (Lemma 6.13(1))

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, e{:}\surd\rangle\}]\!] : (\lambda\beta.\dot{\mathcal{O}}[\![\{\langle \alpha, \beta{:}\lambda z.(x := z;\surd)\rangle\}]\!] : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!])$ (Lemma 6.13(10))

$= (\ddot{\mathcal{O}}[\![\{\langle \alpha, e{:}\surd\rangle\}]\!] : (\lambda\beta.\dot{\mathcal{O}}[\![\{\langle \alpha, \beta{:}\lambda z.(x := z;\surd)\rangle\}]\!])) : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!]$ (Lemma 6.18(4))

$= \dot{\mathcal{O}}[\![\{\langle \alpha, e{:}\lambda z.(x := z;\surd)\rangle\}]\!] : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!]$ (part (1))

$= \dot{\mathcal{O}}[\![\{\langle \alpha, x := e;\surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!].$ (Lemma 6.13(2))

For $\dot{r}$ or $\ddot{r}$ instead of r the proof runs exactly the same.

Case 5: $s = e?x$ (so $\tilde{s} = s$).

$\mathcal{O}^*[\![\{\langle \alpha, e?x;r\rangle\}]\!]$

$= \mathcal{O}^*[\![\{\langle \alpha, e{:}\lambda z.(z?x;r)\rangle\}]\!]$ (Lemma 6.13(6))

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, e{:}\surd\rangle\}]\!] : (\lambda\beta.\mathcal{O}^*[\![\{\langle \alpha, \beta{:}\lambda z.(z?x;r)\rangle\}]\!])$ (part (1))

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, e{:}\surd\rangle\}]\!] : (\lambda\beta.\mathcal{O}^*[\![\{\langle \alpha, \beta?x;r\rangle\}]\!])$ (Lemma 6.13(10))

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, e{:}\surd\rangle\}]\!] : (\lambda\beta.\lambda\sigma.\{\langle \alpha, \beta?x, \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!]\rangle\})$ (Lemma 6.13(7))

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, e{:}\surd\rangle\}]\!] : (\lambda\beta.\lambda\sigma.\{\langle \alpha, \beta?x, \dot{\mathcal{O}}[\![\{\langle \alpha, \surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!]\rangle\})$ (see above)

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, e{:}\surd\rangle\}]\!] : (\lambda\beta.\lambda\sigma.\{\langle \alpha, \beta?x, \dot{\mathcal{O}}[\![\{\langle \alpha, \surd\rangle\}]\!]\rangle\} : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!])$ (Definition 6.16)

$= \ddot{\mathcal{O}}[\![\{\langle \alpha, e{:}\surd\rangle\}]\!] : (\lambda\beta.\dot{\mathcal{O}}[\![\{\langle \alpha, \beta{:}\lambda z.(z?x;\surd)\rangle\}]\!] : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!])$ (Lemma 6.13(7, 10))

$= (\ddot{\mathcal{O}}[\![\{\langle \alpha, e{:}\surd\rangle\}]\!] : (\lambda\beta.\dot{\mathcal{O}}[\![\{\langle \alpha, \beta{:}\lambda z.(z?x;\surd)\rangle\}]\!])) : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!]$ (Lemma 6.18(4))

$= \dot{\mathcal{O}}[\![\{\langle \alpha, e{:}\lambda z.(z?x;\surd)\rangle\}]\!] : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!]$ (part (1))

$= \dot{\mathcal{O}}[\![\{\langle \alpha, e?x;\surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!].$ (Lemma 6.13(6))

Case 6: $s = \mu v[s']$. Without loss of generality we can assume that $v \notin \{v_1, \ldots, v_k\}$. If we define $\tilde{s}' = s'[s_i/v_i]_{i=1}^k$, then we have $\tilde{s} = \mu v[\tilde{s}']$. Now we first prove, by induction on n, that for any α and r (and also for $\dot{r}$),

$$\mathcal{O}^*[\![\{\langle \alpha, \tilde{s}'^{(n)}_v; r\rangle\}]\!] = \dot{\mathcal{O}}[\![\{\langle \alpha, \tilde{s}'^{(n)}_v;\surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle \alpha, r\rangle\}]\!]. \tag{6.3}$$

For $n = 0$, we get $\tilde{s}'^{(0)}_v = \textbf{skip}$ and

$\mathcal{O}^*[\![\{\langle \alpha, \textbf{skip}; r\rangle\}]\!]$

$= \lambda\sigma.\{\langle\sigma, \mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!]\rangle\}$ (definition of **skip**)

$= \lambda\sigma.\{\langle\sigma, \dot{\mathcal{O}}[\![\{\langle\alpha, \surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!]\rangle\}$ (see above)

$= \lambda\sigma.\{\langle\sigma, \dot{\mathcal{O}}[\![\{\langle\alpha, \surd\rangle\}]\!]\rangle\} : \mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!]$ (Definition 6.16)

$= \dot{\mathcal{O}}[\![\{\langle\alpha, \mathbf{skip};\surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!]$ (definition of **skip**).

Now let us assume (6.3) for certain n; then we can apply the outer induction hypothesis for s', with $v_{k+1} = v$ and $s_{k+1} = \tilde{s}'^{(n)}_v$. If we define $\hat{s}'^{(n)}_v = \tilde{s}'[\tilde{s}'^{(n)}_v/v] = s'[s_i/v_i]_{i=1}^{k+1}$, this gives us

$$\mathcal{O}^*[\![\{\langle\alpha, \hat{s}'^{(n)}_v; r\rangle\}]\!] = \dot{\mathcal{O}}[\![\{\langle\alpha, \hat{s}'^{(n)}_v; \surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!]. \tag{6.4}$$

Now we can calculate

$\mathcal{O}^*[\![\{\langle\alpha, \tilde{s}'^{(n+1)}_v; r\rangle\}]\!]$

$= \mathcal{O}^*[\![\{\langle\alpha, (\mathbf{skip};\hat{s}'^{(n)}_v); r\rangle\}]\!]$

$= \mathcal{O}^*[\![\{\langle\alpha, \mathbf{skip};(\hat{s}'^{(n)}_v; r)\rangle\}]\!]$ (Lemma 6.13(3))

$= \lambda\sigma.\{\langle\sigma, \mathcal{O}^*[\![\{\langle\alpha, \hat{s}'^{(n)}_v; r\rangle\}]\!]\rangle\}$ (definition of **skip**)

$= \lambda\sigma.\{\langle\sigma, \dot{\mathcal{O}}[\![\{\langle\alpha, \hat{s}'^{(n)}_v; \surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!]\rangle\}$ (by (6.4))

$= \lambda\sigma.\{\langle\sigma, \dot{\mathcal{O}}[\![\{\langle\alpha, \hat{s}'^{(n)}_v; \surd\rangle\}]\!]\rangle\} : \mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!]$ (Definition 6.16)

$= \dot{\mathcal{O}}[\![\{\langle\alpha, \mathbf{skip};(\hat{s}'^{(n)}_v; \surd)\rangle\}]\!] : \mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!]$ (definition of **skip**)

$= \dot{\mathcal{O}}[\![\{\langle\alpha, (\mathbf{skip};\hat{s}'^{(n)}_v);\surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!]$ (Lemma 6.13(3))

$= \dot{\mathcal{O}}[\![\{\langle\alpha, \tilde{s}'^{(n+1)}_v;\surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!]$

which gives us (6.3) for $n+1$.

Finally, we can compute as follows:

$\mathcal{O}^*[\![\{\langle\alpha, \mu v[\tilde{s}'];r\rangle\}]\!]$

$= \lim_n \mathcal{O}^*[\![\{\langle\alpha, \tilde{s}'^{(n)}_v; r\rangle\}]\!]$ (Lemma 6.13(5))

$= \lim_n(\dot{\mathcal{O}}[\![\{\langle\alpha, \tilde{s}'^{(n)}_v;\surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!])$ (by (6.3))

$= (\lim_n \dot{\mathcal{O}}[\![\{\langle\alpha, \tilde{s}'^{(n)}_v;\surd\rangle\}]\!]) : \mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!]$ (Lemma 6.18(1))

$= \dot{\mathcal{O}}[\![\{\langle\alpha, \mu v[\tilde{s}'];\surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!]$ (Lemma 6.13(5)). □

In order to prove Theorem 6.14, in addition to the reasoning encountered earlier, there is one extra step necessary to deal with the possible recursion in declarations such as $c \Leftarrow \ldots \mathbf{new}(c) \ldots$. This step involves the second component $\gamma_{(2)}$ of an environment γ. For simplicity's sake we again drop the indices.

6.22. Lemma. *Let t be a fixed program. If $\gamma \in \Gamma$ satisfies*

$$\gamma(c) = \lambda\alpha.\mathcal{O}^*[\![\{\langle \alpha, s; E\rangle\}]\!] \tag{6.5}$$

for $c \Leftarrow s$ in t, then we have the following:

(1) *For any $e \in Exp$, $\gamma \in \Gamma$, $\alpha \in AObj$, and $f \in Obj \to P$ we have*

$$\mathcal{E}[\![e]\!]\gamma\alpha f = \ddot{\mathcal{O}}[\![\{\langle \alpha, e{:}\surd\rangle\}]\!]{:}f$$

(2) *Let $s \in \mathcal{S}_{\text{nud}}$ (not necessarily closed) and assume that the free statement variables in s are all in $\{v_1, \ldots, v_k\}$ and let $s_1, \ldots, s_k$ be closed. Put $\tilde{s} = s[s_i/v_i]_{i=1}^k$ and define*

$$\varphi_i = \lambda\alpha.\lambda p.(\dot{\mathcal{O}}[\![\{\langle \alpha, s_i; \surd\rangle\}]\!]{:}p) \quad (i = 1, \ldots, k)$$

and let $\tilde{\gamma} = \gamma\{\varphi_i/v_i\}_{i=1}^k$. Then we have, for any α and p,

$$\mathcal{D}[\![s]\!]\tilde{\gamma}\alpha p = \dot{\mathcal{O}}[\![\{\langle \alpha, \tilde{s}; \surd\rangle\}]\!]{:}p.$$

Proof. The proof follows the same line of argument as in Sections 4 and 5. It runs by induction on the complexity of e and s. We make use of Lemmas 6.13, 6.18, 6.20, and 6.21 and we need the assumption (6.5) to deal with the case $e = \textbf{new}(c)$.

We shall deal with some typical cases here, starting with part (1).

Case 1: $e = \beta$.

$$\begin{aligned}
\mathcal{E}[\![\beta]\!]\gamma\alpha f &= f(\beta) && \text{(Definition 6.9)}\\
&= p_0 \| f(\beta) && \text{(Definition 6.17)}\\
&= \langle \beta, p_0\rangle{:}f && \text{(Definition 6.16)}\\
&= \ddot{\mathcal{O}}[\![\{\langle \alpha, \beta{:}\surd\rangle\}]\!]{:}f && \text{(Definition 6.19).}
\end{aligned}$$

Case 2: $e = \textbf{op}\ e'$.

$$\begin{aligned}
&\mathcal{E}[\![\textbf{op}\ e']\!]\gamma\alpha f\\
&\quad = \mathcal{E}[\![e']\!]\gamma\alpha(\lambda\beta.f(\textbf{op}_{\text{sem}}\,\beta)) && \text{(Definition 6.9)}\\
&\quad = \mathcal{E}[\![e']\!]\gamma\alpha(\lambda\beta.\ddot{\mathcal{O}}[\![\{\langle \alpha, \textbf{op}_{\text{sem}}\,\beta : \surd\rangle\}]\!]{:}f) && \text{(Case 1 for } \textbf{op}_{\text{sem}}\,\beta)\\
&\quad = \mathcal{E}[\![e']\!]\gamma\alpha(\lambda\beta.\ddot{\mathcal{O}}[\![\{\langle \alpha, \beta{:}\lambda z.(\textbf{op}\ z : \surd)\rangle\}]\!]{:}f) && \text{(Lemma 6.13(11, 10))}\\
&\quad = \ddot{\mathcal{O}}[\![\{\langle \alpha, e'{:}\surd\rangle\}]\!] : (\lambda\beta.\ddot{\mathcal{O}}[\![\{\langle \alpha, \beta{:}\lambda z.(\textbf{op}\ z : \surd)\rangle\}]\!]{:}f) && \text{(ind. hyp.)}\\
&\quad = (\ddot{\mathcal{O}}[\![\{\langle \alpha, e'{:}\surd\rangle\}]\!] : (\lambda\beta.\ddot{\mathcal{O}}[\![\{\langle \alpha, \beta{:}\lambda z.(\textbf{op}\ z : \surd)\rangle\}]\!])){:}f && \text{(Lemma 6.18(4))}\\
&\quad = \ddot{\mathcal{O}}[\![\{\langle \alpha, e'{:}\lambda z.(\textbf{op}\ z : \surd)\rangle\}]\!]{:}f && \text{(Lemma 6.21)}\\
&\quad = \ddot{\mathcal{O}}[\![\{\langle \alpha, \textbf{op}\ e' : \surd\rangle\}]\!]{:}f && \text{(Lemma 6.13(12))}
\end{aligned}$$

Case 3: $e = \textbf{new}(c)$.

$$\mathscr{E}[\![\mathbf{new}(c)]\!]\gamma\alpha f$$

$$= \lambda\sigma.\{\langle\sigma', \gamma(c)(\beta)\|f(\beta)\rangle\} \qquad \text{(Definition 6.9, with } \sigma' \text{ and } \beta \text{ as usual)}$$

$$= \lambda\sigma.\{\langle\sigma', \gamma(c)(\beta) \| (\ddot{\mathcal{O}}[\![\{\langle\alpha, \beta{:}\surd\rangle\}]\!]{:}f)\rangle\} \qquad \text{(see Case 1)}$$

$$= \lambda\sigma.\{\langle\sigma', (\gamma(c)(\beta) \| \ddot{\mathcal{O}}[\![\{\langle\alpha, \beta{:}\surd\rangle\}]\!]){:}f\rangle\} \qquad \text{(Lemma 6.18(5))}$$

$$= \lambda\sigma.\{\langle\sigma', (\gamma(c)(\beta) \| \ddot{\mathcal{O}}[\![\{\langle\alpha, \beta{:}\surd\rangle\}]\!])\rangle\}{:}f \qquad \text{(Definition 6.16)}$$

$$= \lambda\sigma.\{\langle\sigma', (\mathcal{O}^*[\![\{\langle\beta, s; E\rangle\}]\!] \| \ddot{\mathcal{O}}[\![\{\langle\alpha, \beta{:}\surd\rangle\}]\!])\rangle\}{:}f \qquad \text{(by (6.5))}$$

$$= \lambda\sigma.\{\langle\sigma', \ddot{\mathcal{O}}[\![\{\langle\beta, s; E\rangle, \langle\alpha, \beta{:}\surd\rangle\}]\!]\rangle\}{:}f \qquad \text{(Lemma 6.21)}$$

$$= \ddot{\mathcal{O}}[\![\{\langle\alpha, \mathbf{new}(c){:}\surd\rangle\}]\!]{:}f. \qquad \text{(Lemma 6.13(13))}$$

And now part (2). Again we deal with a few typical cases.
Case 4: $s = x := e$, so $\tilde{s} = s$.

$$\mathscr{D}[\![x := e]\!]\tilde{\gamma}\alpha p$$

$$= \mathscr{E}[\![e]\!]\tilde{\gamma}\alpha(\lambda\beta.\lambda\sigma.\{\langle\sigma', p\rangle\}) \qquad \text{(Definition 6.9, with } \sigma' \text{ as usual)}$$

$$= \mathscr{E}[\![e]\!]\tilde{\gamma}\alpha(\lambda\beta.\dot{\mathcal{O}}[\![\{\langle\alpha, \beta{:}\lambda z.(x := z;\surd)\rangle\}]\!]{:}p) \qquad \text{(see proof of Lemma 6.21, Case 4)}$$

$$= \ddot{\mathcal{O}}[\![\{\langle\alpha, e{:}\surd\rangle\}]\!] : (\lambda\beta.\dot{\mathcal{O}}[\![\{\langle\alpha, \beta{:}\lambda z.(x := z;\surd)\rangle\}]\!]{:}p) \qquad \text{(part (1))}$$

$$= (\ddot{\mathcal{O}}[\![\{\langle\alpha, e{:}\surd\rangle\}]\!] : (\lambda\beta.\dot{\mathcal{O}}[\![\{\langle\alpha, \beta{:}\lambda z.(x := z;\surd)\rangle\}]\!])){:}p \qquad \text{(Lemma 6.18(4))}$$

$$= \dot{\mathcal{O}}[\![\{\langle\alpha, e{:}\lambda z.(x := z;\surd)\rangle\}]\!]{:}p \qquad \text{(Lemma 6.21)}$$

$$= \dot{\mathcal{O}}[\![\{\langle\alpha, x := e;\surd\rangle\}]\!]{:}p. \qquad \text{(Lemma 6.13(2))}$$

Case 5: $s = \mu v[s']$. Let us assume again that $v \notin \{v_1, \ldots, v_k\}$, so that, if we define $\tilde{s}' = s'[s_i/v_i]_{i=1}^k$, then we have $\tilde{s} = \mu v[\tilde{s}']$. Now, on the one hand, we have, by Lemma 6.13(5) and Lemma 6.18(1), that

$$\dot{\mathcal{O}}[\![\{\langle\alpha, \tilde{s};\surd\rangle\}]\!]{:}p = \lim_n(\dot{\mathcal{O}}[\![\{\langle\alpha, \tilde{s}_v'^{(n)};\surd\rangle\}]\!]{:}p). \tag{6.6}$$

On the other hand, Definition 6.9 says that

$$\mathscr{D}[\![s]\!]\tilde{\gamma}\alpha p = \lim_n \psi_n(\alpha)(p) \tag{6.7}$$

where ψ_0 can be chosen arbitrarily, and

$$\psi_{n+1} = \lambda\alpha.\lambda p.\lambda\sigma.\{\langle\sigma, \mathscr{D}[\![s']\!]\tilde{\gamma}\{\psi_n/v\}\alpha p\rangle\}.$$

Now we make a definite choice for ψ_0, namely

$$\psi_0 = \lambda\alpha.\lambda p.(\dot{\mathcal{O}}[\![\{\langle\alpha, \tilde{s}_v'^{(0)};\surd\rangle\}]\!]{:}p)$$

and we prove, by induction on n, that

$$\psi_n = \lambda\alpha.\lambda p.(\dot{\mathcal{O}}[\![\{\langle\alpha, \tilde{s}_v'^{(n)};\surd\rangle\}]\!]{:}p). \tag{6.8}$$

For $n=0$ this is obvious, so assume (6.8) for some n; then we can apply the outer induction hypothesis to s' with $v_{k+1}=v$ and $s_{k+1}=\tilde{s}'^{(n)}_v$, so our inner induction hypothesis (6.8) says that $\varphi_{k+1}=\psi_n$. We then get (because $s'[s_i/v_i]_{i=1}^{k+1}=\tilde{s}'[\tilde{s}'^{(n)}_v/v]$)

$$\mathscr{D}[\![s']\!]\tilde{\gamma}\{\psi_n/v\}\alpha p=\dot{\mathcal{O}}[\![\{\langle\alpha,\tilde{s}'[\tilde{s}'^{(n)}_v/v];\surd\rangle\}]\!]:p \tag{6.9}$$

and we calculate

$$\begin{aligned}
\psi_{n+1}(\alpha)(p) &= \lambda\sigma.\{\langle\sigma,\mathscr{D}[\![s']\!]\tilde{\gamma}\{\psi_n/v\}\alpha p\rangle\} && \text{(definition of } \psi_{n+1})\\
&= \lambda\sigma.\{\langle\sigma,\dot{\mathcal{O}}[\![\{\langle\alpha,\tilde{s}'[\tilde{s}'^{(n)}_v/v];\surd\rangle\}]\!]:p\rangle\} && \text{(by (6.9))}\\
&= \lambda\sigma.\{\langle\sigma,\dot{\mathcal{O}}[\![\{\langle\alpha,\tilde{s}'[\tilde{s}'^{(n)}_v/v];\surd\rangle\}]\!]\rangle\}:p && \text{(Definition 6.16)}\\
&= \dot{\mathcal{O}}[\![\{\langle\alpha,\mathbf{skip};(\tilde{s}'[\tilde{s}'^{(n)}_v/v];\surd)\rangle\}]\!]:p && \text{(definition of } \mathbf{skip})\\
&= \dot{\mathcal{O}}[\![\{\langle\alpha,(\mathbf{skip};\tilde{s}'[\tilde{s}'^{(n)}_v/v]);\surd\rangle\}]\!]:p && \text{(Lemma 6.13(3))}\\
&= \dot{\mathcal{O}}[\![\{\langle\alpha,\tilde{s}'^{(n+1)}_v;\surd\rangle\}]\!]:p && \text{(definition of } \tilde{s}'^{(n+1)}_v).
\end{aligned}$$

Finally, (6.8) tells us that in (6.6) and (6.7) we are taking the limit of the same sequence, so their respective left-hand sides are equal. □

One more step is necessary before we reach the desired conclusion.

6.23. Lemma. *Let γ_t be as in Definition* 6.9(3). *Then we have that γ_t satisfies* (6.5).

Proof. Choose any γ satisfying (6.5). Then, by the definition of Ψ_t (in Definition 6.9(3)), we have, for $c \Leftarrow s$ in t,

$$\begin{aligned}
\Psi_t(\gamma)(c) &= \lambda\alpha.\mathscr{D}[\![s]\!](\gamma)(\alpha)(p_0)\\
&= \lambda\alpha.(\dot{\mathcal{O}}[\![\{\langle\alpha,s;\surd\rangle\}]\!]:p_0) && \text{(Lemma 6.22)}\\
&= \lambda\alpha.(\dot{\mathcal{O}}[\![\{\langle\alpha,s;\surd\rangle\}]\!]:\dot{\mathcal{O}}[\![\{\langle\alpha,E\rangle\}]\!]) && \text{(Definition 6.19)}\\
&= \lambda\alpha.\mathcal{O}^*[\![\{\langle\alpha,s:E\rangle\}]\!] && \text{(Lemma 6.21)}\\
&= \gamma && \text{(by (6.5))}.
\end{aligned}$$

If we have furthermore that $\gamma(c)=\lambda\alpha.p_0$ for c not declared in t, then we have that γ is a fixed point of Ψ_t, so that $\gamma=\gamma_t$. □

Now we can prove Theorem 6.14:

Proof of Theorem 6.14. For part (1), we calculate as follows:

$$\mathcal{O}^*[\![\{\langle\alpha,e:g\rangle\}]\!]=\ddot{\mathcal{O}}[\![\{\langle\alpha,e:\surd\rangle\}]\!]:(\lambda\beta.\mathcal{O}^*[\![\{\langle\alpha,\beta:g\rangle\}]\!]) \quad \text{(Lemma 6.21)}$$

$$= \mathscr{E}[\![e]\!]\gamma_t\alpha(\lambda\beta.\mathcal{O}^*[\![\{\langle\alpha, \beta : g\rangle\}]\!]) \qquad \text{(Lemma 6.22)}$$

where the application of Lemma 6.22 is allowed by Lemma 6.23.

Now for part (2), we have

$$\mathcal{O}^*[\![\{\langle\alpha, s; r\rangle\}]\!] = \dot{\mathcal{O}}[\![\{\langle\alpha, s; \surd\rangle\}]\!] : \mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!] \quad \text{(Lemma 6.21)}$$

$$= \mathscr{D}[\![s]\!]\gamma_t\alpha(\mathcal{O}^*[\![\{\langle\alpha, r\rangle\}]\!]) \qquad \text{(Lemma 6.22)}$$

where $\tilde{s} = s$ and $\tilde{\gamma}_t = \gamma_t$ because s is closed. Here, again, Lemma 6.23 justifies the application of Lemma 6.22. □

6.24. Corollary. *For any* $t \in \mathscr{L}_{\text{nud}}$, $\mathcal{O}^*[\![t]\!] = \mathscr{D}[\![t]\!]$.

Proof. Let $t = \langle c_i \Leftarrow s_i\rangle_{i=1}^{k}$; then we have

$$\begin{aligned}
\mathcal{O}^*[\![t]\!] &= \mathcal{O}^*[\![\{\langle\langle c_1, 1\rangle, s_1; E\rangle\}]\!] && \text{(Definition 6.10(4))}\\
&= \mathscr{D}[\![s_1]\!](\gamma_t)(\langle c_1, 1\rangle)(\mathcal{O}^*[\![\{\langle\langle c_1, 1\rangle, E\rangle\}]\!]) && \text{(Theorem 6.14(2))}\\
&= \mathscr{D}[\![s_1]\!](\gamma_t)(\langle c_1, 1\rangle)(p_0) && \text{(Definition 6.10(3))}\\
&= \mathscr{D}[\![t]\!] && \text{(Definition 6.9(4)).} \quad \square
\end{aligned}$$

With Corollary 6.24, we have obtained the ultimate goal of our paper: to establish the equivalence of an operational and a denotational semantics for a nonuniform language with process creation.

Acknowledgment

Discusssions with Jeffery Zucker led to a considerably improved way of incorporating syntactic continuations for the uniform case. Moreover, Definition 4.4 is due to him. We are also indebted to Dr. Zucker for pointing out several minor and some major flaws in a draft of this paper which we (hope to) have corrected in the present version.

The contributions of Joost Kok and Jan Rutten to the design of the POOL semantics as reported in [6, 7] were absolutely essential for the present investigation. The idea of assembling transition sequence information into a process is due to Joost Kok. We acknowledge fruitful discussions on our work in the Amsterdam concurrency group, including Frank de Boer, Joost Kok, John-Jules Meyer, Jan Rutten and Erik de Vink.

Finally, we express our thanks to Marisa Venturini Zilli for the opportunity extended to the second author to lecture on the material of this paper in the Advanced

School on Mathematical Models for the Semantics of Parallelism, Rome, September 1986.

References

[1] *The Programming Language Ada Reference Manual*, American National Standards Institute, ANSI/MIL-STD-1815A-1983 (also published as: Lecture Notes in Computer Science **155** (Springer, Berlin, 1983)).

[2] G. Agha, Semantic considerations in the Actor paradigm of concurrent computations, in: S.D. Brookes, A.W. Roscoe and G. Winskel, eds., *Proc. Seminar on Concurrency*, Carnegie-Mellon University, Pittsburgh, PA, July 9-11, 1984, Lecture Notes in Computer Science **197** (Springer, Berlin, 1984) 151-179.

[3] P. America, Definition of the programming language POOL-T, ESPRIT Project 415, Doc. No. 91, Philips Research Laboratories, Eindhoven, The Netherlands, September 1985.

[4] P. America, Rationale for the design of POOL, ESPRIT Project 415, Doc. No. 53, Philips Research Laboratories, Eindhoven, The Netherlands, January 1986.

[5] P. America, Objected-oriented programming: a theoretician's introduction, *EATCS Bull. 29* (June 1986) 69-84.

[6] P. America, J.W. De Bakker, J.N. Kok and J.J.M.M. Rutten, Operational semantics of a parallel object-oriented language, in: *Conf. Rec. 13th Symp. on Principles of Programming Languages*, St. Petersburg, FL (January 13-15, 1986) 194-208.

[7] P. America, J.W. De Bakker, J.N. Kok and J.J.M.M. Rutten, A denotational semantics of a parallel object-oriented language, Report CS-R8626, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, August 1986.

[8] P. America and J.J.M.M. Rutten, Solving reflexive domain equations in a category of complete metric spaces, in: *Proc. Third Workshop on Mathematical Foundations of Programming Language Semantics*, New Orleans, LA, April 8-10, 1987, Lecture Notes in Computer Science **298** (Springer, Berlin, 1988) 254-288.

[9] K.R. Apt, Recursive assertions and parallel programs, *Acta Inform.* **15** (1981) 219-232.

[10] K.R. Apt, Formal justification of a proof system for communicating sequential processes, *J. ACM* **30**(1) (1983) 197-216.

[11] 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, *Theoret. Comput. Sci.* **34** (1984) 135-156.

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

[13] J.W. De Bakker and J.-J.Ch. Meyer, Order and metric in the stream semantics of elemental concurrency, *Acta Inform.* **24** (1987) 491-511.

[14] J.W. De Bakker, J.-J.Ch. Meyer and E.-R. Olderog, Infinite streams and finite observations in the semantics of uniform concurrency, *Theoret. Comput. Sci.* **49**(2, 3) (1987) 87-112.

[15] J.W. De Bakker, J.-J.Ch. Meyer, E.-R. Olderog and J.I. Zucker, Transition systems, infinitary languages and the semantics of uniform concurrency, in: *Proc. 17th ACM Symp. on the Theory of Computing*, Providence, RI (1985) 252-262.

[16] 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 (full version of [15]), *J. Comput. System Sci.* **36** (1988) 158-224..

[17] J.W. De Bakker and J.I. Zucker, Processes and the denotational semantics of concurrency, *Inform. and Control* **54** (1982) 70-120.

[18] J.A. Bergstra and J.W. Klop, Process algebra for synchronous communication, *Inform. and Control* **60** (1984) 109-137.

[19] F.S. De Boer, A proof rule for process creation, in: M. Wirsing, ed., *Formal Description of Programming Concepts III, Proc. Third IFIP WG 2.2 Working Conf.*, Gl. Avernæs, Ebberup, Denmark, August 25-28, 1986 (North-Holland, Amsterdam, 1987) 23-50.
[20] M. Broy, Fixed point theory for communication and concurrency, in: D. Bjørner, ed., *Formal Description of Programming Concepts II* (North-Holland, Amsterdam, 1983) 125-146.
[21] M. Broy, Applicative real-time programming, in: R.E.A. Mason, ed., *Information Processing '83: Proc. IFIP Conference* (North-Holland, Amsterdam, 1983) 259-264.
[22] A. De Bruin and A.P.W. Böhm, The denotational semantics of dynamic networks of processes, *ACM Trans. Programming Languages and Systems* 7(4) (1985) 656-679.
[23] W.D. Clinger, Foundations of actor semantics, Technical Report No. 633, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 1981.
[24] O.-J. Dahl, B. Myhrhaug and K. Nygaard, SIMULA 67, Common Base Language, Norwegian Computing Center, Forskningsvn. lb., Oslo, Norway, 1967.
[25] J. Dugundji, *Topology* (Allyn & Bacon, Newton, MA, 1966).
[26] R. Engelking, *General Topology* (Polish Scientific Publishers, Warsaw, 1977).
[27] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove and D.S. Scott, *A Compendium of Continuous Lattices* (Springer, Berlin, 1980).
[28] H. Hahn, *Reelle Funktionen* (Chelsea, New York, 1948).
[29] M. Hennessy and G.D. Plotkin, Full abstraction for a simple parallel programming language, in: J. Bečvář, ed., *Proc. 8th Symp. on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science **74** (Springer, Berlin, 1979) 108-120.
[30] C. Hewitt, Viewing control structures as patterns of passing messages, *Artificial Intelligence* **8** (1977) 323-364.
[31] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21**(8) (1978) 666-677.
[32] C.A.R. Hoare, *Communicating Sequential Processes* (Prentice-Hall, Englewood Cliffs, NJ, 1985).
[33] J.-J.Ch. Meyer, Merging regular processes by means of fixed point theory, *Theoret. Comput. Sci.* **45** (1986) 193-260.
[34] J.-J.Ch. Meyer and E.P. de Vink, Applications of compactness in the Smyth powerdomain of streams, in: *Proc. TAPSOFT '87, Vol. 1*, Pisa, Italy, March 23-27, 1987, Lecture Notes in Computer Science **249** (Springer, Berlin, 1987) 241-255.
[35] M. Nivat, Infinite words, infinite trees, infinite computations, in: *Foundations of Computer Science III.2*, Mathematical Centre Tracts **109** (1979) 3-52.
[36] D. Niwinski, Fixed point semantics for algebraic (tree) grammars, in: M. Nielsen and E.M. Schmidt, eds., *Proc. 9th Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science **140** (Springer, Berlin, 1982) 384-396.
[37] G.D. Plotkin, A powerdomain construction, *SIAM J. Comput.* **5**(3) (1976) 452-487.
[38] G.D. Plotkin, A structural approach to operational semantics, Report DAIMI FN-19, Computer Science Department, Aarhus University, September 1981.
[39] G.D. Plotkin, An operational semantics for CSP, in: D. Bjørner, ed., *Formal Description of Programming Concepts II* (North-Holland, Amsterdam, 1983) 199-223.
[40] A. Pnueli, Linear and branching structures in the semantics and logics of reactive systems, in: W. Brauer, ed., *Proc. 12th Internat. Coll. on Automata, Languages and Programming*, Nafplion, Greece, July 15-19, 1985, Lecture Notes in Computer Science **194** (Springer, Berlin, 1985) 15-32.
[41] W.C. Rounds, On the relationship between Scott domains, synchronization trees and metric spaces, Report CRL-TR-25-83, University of Michigan, 1983.
[42] V.A. Saraswat, The concurrent logic programming language CP: definition and operational semantics, in: *Conf. Rec. 14th Symp. on Principles of Programming Languages*, München, Fed. Rep. Germany (January 21-23, 1987) 49-62.
[43] S.A. Smolka and R.E. Strom, A CCS semantics for NIL, in: M. Wirsing, ed., *Formal Description of Programming Concepts III, Proc. Third IFIP WG 2.2 Working Conf.*, Gl. Avernæs, Ebberup, Denmark, August 25-28, 1986 (North-Holland, Amsterdam, 1987) 347-373.

Contractions in Comparing Concurrency Semantics

Joost N. Kok

Jan Rutten

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

We define for a number of concurrent imperative languages both operational and denotational semantic models as fixed points of contractions on complete metric spaces. Next, we develop a general method for comparing different semantic models by relating their defining contractions and exploiting the fact that contractions have a unique fixed point.

1980 Mathematical Subject Classification: 68B10, 68C01.
1986 Computing Reviews Categories: D.3.1, F.3.2, F.3.3.
Key words and phrases: concurrency, imperative languages, denotational semantics, operational semantics, metric spaces, contractions, semantic equivalence.
Note: This work was carried out in the context of ESPRIT project 415: Parallel Architectures and Languages for Advanced Information Processing — a VLSI-directed approach.

0. Introduction

We present a study of three concurrent imperative languages, called L_0, L_1, and L_2. For each of them we shall define an *operational semantics* $\mathcal{O}_i$ and a *denotational semantics* $\mathcal{D}_i$, for $i = 0,1,2$, and give a comparison of the two models. (We shall use the terms *semantics* and *semantic model* as synonyms.) This *comparison* is the main subject of our paper, rather than the specific nature of the languages themselves, or the particular properties of their semantics.

The languages L_i have been defined and studied already in much detail in [BMOZ1,2] and [BKMOZ]. We rely heavily on these papers, using many definitions taken from them literally, and others in an adapted version. (The languages L_0, L_1, and L_2 we use here are called L_0, L_2, and L_3 in the papers mentioned.)

Let us try to characterize in a few words the languages under consideration. They all belong to the wide class of *concurrent* (*parallel*) *imperative* programming languages. We shall discuss parallel execution through interleaving (*shuffle*) of elementary actions (in L_0), together with *synchronization* and *communication* (in L_1) and extended with (an elementary form of) *message passing* (in L_2). Imperative concurrency is further characterized by an explicit operator for parallel composition on top of the usual imperative constructs, such as elementary action and sequential composition. Herein it differs from another widely used style, so-called *applicative* concurrency, where the parallelism is implicit. Further, L_0 and L_1 are *uniform* and L_2 is *nonuniform*. In L_0 and L_1 the elementary actions are left atomic, whereas in L_2 an interpretation of these actions is supplied. They consist of assignments, test and send and receive actions. Another important feature is the presence of *local nondeterminacy* (in L_0) and *global nondeterminacy* (in L_1 and L_2). (Sometimes this is called *internal* and *external* nondeterminacy.) The difference between the two has major implications for the different semantic models. (For an extensive discussion of this matter see, e.g., the introduction of [BKMOZ].)

For our semantic definitions we shall use *metric* structures, rather than order-theoretic domains. The metric approach is particularly felicitous for problems where histories, computational traces and tree-

like structures of some kind are essential. Moreover, it allows for the definition of the notion of *contraction*, which we discuss in more detail in a moment. Our operational models Θ_i are based on the transition system technique of Hennessy and Plotkin [HP] and Plotkin [Pl2, Pl3]. They are closely related to the ones defined in [BKMOZ], but there are some differences. We use labeled transitions and (in Θ_1 and Θ_2) communication is treated somewhat differently. Our denotational models D_i are almost exactly the same as in [BKMOZ]. They are defined compositionally, giving the meaning of a compound statement in terms of the meaning of its components, and tackling recursion with the help of fixed points. For D_1 and D_2 we use a reflexive domain, being a solution of some domain equation in the style of Plotkin [Pl1] and Scott [Sc]. We shall not give the details of solving in a metric setting this type of equations, but refer the reader to [BZ], where a solution was presented first, and to [AR], where this metric approach is reformulated and extended in a category-theoretic setting.

Although the semantic models presented here are (roughly) the same as in [BKMOZ], there is one major difference, being the way in which they are defined. In this paper we define both the operational and denotational models as fixed points of contractions.

A *contraction* $f:M\to M$ on a complete metric space M has the useful property that there exists one and only one fixed point $x\in M$ (satisfying $f(x)=x$). This elementary fact is known as Banach's fixed point theorem (see A.4.(b)). Such a fixed point x is entirely determined by the definition of f: any other element $y\in M$ satisfying the same properties as x, that is, satisfying $f(y)=y$, is equal to x. The contractions Φ we use in this paper are always of type

$$\Phi:(M_1\to M_2)\to(M_1\to M_2),$$

that is, they are defined on a complete metric function space $M_1\to M_2$. Then the fixed point of Φ is a function from M_1 to M_2.

The fact that our denotational models can be obtained as fixed points of suitable contractions is not very surprising, fixed points playing traditionally an important role in denotational semantics. It is interesting, however, to observe that the same method applies to the definition of *operational* models. One might wonder whether the models thus obtained still deserve to be called operational. That this is the case follows from the fact that they equal the models defined in the usual manner, without the use of fixed points (see lemma 1.12).

The main advantage of this style of defining semantic models as fixed points is that it enables us to compare them more easily. This brings us to the discussion of what has been announced above to be the main subject of this paper: the *comparison* of operational and denotational semantic models, which we shall also call the study of their *semantic equivalence*. About the question why this would be an interesting problem we want to be brief. Different semantic models of a given language can be regarded as different views of the same object. So they are in some way related. Their precise relationship we want to capture in some formal statement.

Let us now sketch the way we use contractions in our study of semantic equivalences. Let L be a language. Suppose an operational model Θ for L is given as the fixed point of a contraction

$$\Phi:(L\to M)\to(L\to M),$$

where M is a complete metric space. Suppose furthermore that we have a denotational model $\mathcal{D}$ for L of the same type as Θ, that is, with $\mathcal{D}:L\to M$, for which we can prove $\Phi(\mathcal{D})=\mathcal{D}$. Then it follows from the uniqueness of the fixed point of Φ that $\Theta=\mathcal{D}$.

In the context of *complete partial ordering structures* similar approaches exist (see, e.g., [HP] and [AP]). There, the operational semantics Θ can be characterized as the (with respect to the pointwise ordering)

smallest function $\mathcal{F}$ satisfying $\Phi(\mathcal{F})=\mathcal{F}$, for some continuous function Φ. Then it follows from $\Phi(\mathcal{D})=\mathcal{D}$ that Θ is smaller than $\mathcal{D}$. In order to establish $\Theta=\mathcal{D}$ it is proved that Θ satisfies the defining equations for $\mathcal{D}$, from which it follows that $\mathcal{D}$ is smaller than Θ. Please note that within the metric setting we can omit the second part of the proof.

In general Θ and $\mathcal{D}$ have different types, that is, they are mappings from L to different mathematical domains. In the languages we consider, this difference is caused by the fact that recursion is treated in the denotational and operational semantics *with* and *without* the use of so-called environments, respectively. Therefore, Θ and $\mathcal{D}$ cannot be fixed points of the same contraction. Now suppose Θ and $\mathcal{D}$ are defined as fixed points of

$$\Phi:(L\to M_1)\to(L\to M_1), \text{ and } \Psi:(L\to M_2)\to(L\to M_2)$$

respectively, where M_1 and M_2 are different complete metric spaces. Then we can relate Θ and $\mathcal{D}$ by defining an intermediate semantic model for L as the fixed point of a contraction

$$\Phi':(L\to M')\to(L\to M'),$$

and by relating Φ , Φ' and Ψ as follows. If we define

$$f_1:(L\to M_1)\to(L\to M'), \text{ and } f_2:(L\to M_2)\to(L\to M'),$$

and we next succeed in proving the commutativity (indicated by *) of the next diagram:

$$\begin{array}{ccc} L\to M_1 & \overset{\Phi}{\to} & L\to M_1 \\ f_1\downarrow & *_1 & \downarrow f_1 \\ L\to M' & \overset{\Phi'}{\to} & L\to M' \\ f_2\uparrow & *_2 & \uparrow f_2 \\ L\to M_2 & \overset{\Psi}{\to} & L\to M_2 \end{array}$$

then we are able to deduce the following relation between Θ and $\mathcal{D}$:

$$f_2(\mathcal{D})=f_1(\Theta).$$

It is straightforward from $*_1$ and $*_2$, and the fact that Φ, Φ', and Ψ are contractions.

This will be the procedure we follow for the models Θ_0 and $\mathcal{D}_0$ of L_0 in section 1. There f_1 and f_2 are such, that for closed statements (i.e., containing no free statement variables) $s\in L_0$, we have: $\Theta(s)=\mathcal{D}(s)$. Once this result has been achieved for L_0, it is straightforward to adapt the definitions, lemmas and theorems involved so as to deduce a similar result for L_1 and L_2. (For the latter languages there is one slight complication. It appears to be convenient to relate $L\to M_1$ and $L\to M_2$ via *two* intermediate types, $L\to M'$ and $L\to M''$.) In [BMOZ1,2] and [BKMOZ] there have already been given proofs for the semantic equivalence of operational and denotational models for L_0 and L_1. These proofs, however, are quite complicated and not so easy to understand. Furthermore, the proof for L_1 is much more complex than that for L_0, involving an intermediate ready-set domain.

The method of proving semantic equivalence as described above is general in the sense that it is applicable to very different languages, such as L_0, L_1 and L_2.

This paper has seven sections. You are now reading section 0, the introduction. It is followed by the treatment of L_0, L_1 and L_2 in sections 1, 2, and 3 respectively. Then, in section 4, some conclusions and remarks about future research are formulated. Section 5 gives the references and section 6, the appendix, gives the basic definitions of metric topology.

Acknowledgements
We are much indebted to Jaco de Bakker, John-Jules Meijer, Ernst-Rudiger Olderog, and Jeffrey Zucker, authors and co-authors of the papers [BMOZ1,2] and [BKMOZ], respectively, on which we have relied heavily. We are also grateful to Jaco de Bakker for his many comments and suggestions made during our work on this subject. We thank Pierre America for pointing out an error in the definition of guardedness (which caused us considerable trouble and therefore increased our insights). We acknowledge fruitful discussions on our work in the Amsterdam concurrency group, including Jaco de Bakker, Frank de Boer, Arie de Bruin, John-Jules Meijer, and Erik de Vink. Finally, we express our thanks to Dini Verloop, who has expertly typed this document.

1. A SIMPLE LANGUAGE (L_0)

1.1 *Syntax*

For the definition of the first language studied in this paper, we need two sets of basic elements. Let A, with typical elements $a,b, \ldots$, be the set of *elementary actions*. For A we take an arbitrary, possibly infinite, set. Further, let *Stmv*, with typical elements $x,y, \ldots$, be the set of statement variables. For *Stmv* we take some infinite set of symbols.

DEFINITION 1.1 (Syntax for L_0)
We define the set of *statements* L_0, with typical elements $s,t, \ldots$, by the following syntax:

$$s ::= a \mid s_1;s_2 \mid s_1 \cup s_2 \mid s_1 \| s_2 \mid x \mid \mu x[t]$$

where $t \in L_0^x$, the set of statements which are *guarded for* x, to be defined below.

A statement s is of one of the following six forms:
- an *elementary action* a.
- the *sequential composition* $s_1;s_2$ of statements s_1 and s_2.
- the *nondeterministic choice* $s_1 \cup s_2$, also known as *local nondeterminism* [FHLR]: $s_1 \cup s_2$ is executed by executing either s_1 or s_2 chosen nondeterministically.
- the *concurrent execution* $s_1 \| s_2$, modeled by the arbitrary interleaving *(shuffle)* of the elementary actions of s_1 and s_2.
- a statement variable x, which is (normally) used in
- the recursive construct $\mu x[t]$: its execution amounts to execution of t where occurrences of x in t are executed by (recursively) executing $\mu x[t]$. For example, with the definition to be proposed presently, the intended meaning of $\mu x[(a;x) \cup b]$ is the set $a^* \cdot b \cup \{a^\omega\}$.

An important restriction of our language is that we consider only recursive constructs $\mu x[t]$, for which t is guarded for x: $t \in L_0^x$. Intuitively, a statement t is guarded for x when all occurrences of x in t are preceded by some statement. More formally:

DEFINITION 1.2 (Syntax for L_0^x)
The set L_0^x of statements which are guarded for x is given by

$$\begin{aligned} t ::= \quad & a \\ \mid \quad & t;s, \quad \text{for } s \in L_0 \\ \mid \quad & t_1 \cup t_2 \mid t_1 \| t_2 \\ \mid \quad & y \quad , \quad \text{for } y \neq x \\ \mid \quad & \mu x[t] \end{aligned}$$

$\mid \quad \mu y[t'], \quad \text{for } y \neq x,\ t' \in L_0^x \cap L_0^y.$

REMARK
In order to avoid possible confusion about the definitions of L_0 and L_0^x, let us give a more extensive definition, for which the ones given above are shorthand. We define L_0 and, for every $x \in Stmv$, L_0^x simultaneously and in stages:

Stage 0:

$$L_0(0) = A \cup Stmv, \quad L_0^x(0) = A \cup (Stmv \setminus \{x\})$$

Stage (n+1):

$$\begin{aligned} L_0(n+1) = L_0(n) \cup\ & \{s_1;s_2 \mid s_1,s_2 \in L_0(n)\} \\ \cup\ & \{s_1 \cup s_2 \mid s_1,s_2 \in L_0(n)\} \\ \cup\ & \{s_1 \| s_2 \mid s_1,s_2 \in L_0(n)\} \\ \cup\ & \{\mu x[t] \mid t \in L_0^x(n)\}. \end{aligned}$$

$$\begin{aligned} L_0^x(n+1) = L_0^x(n) \cup\ & \{t;s \mid t \in L_0^x(n),\ s \in L_0(n)\} \\ \cup\ & \{t_1 \cup t_2 \mid t_1,t_2 \in L_0^x(n)\} \\ \cup\ & \{t_1 \| t_2 \mid t_1,t_2 \in L_0^x(n)\} \\ \cup\ & \{\mu x[t] \mid t \in L_0^x(n)\} \\ \cup\ & \{\mu y[t] \mid y \neq x \wedge t \in L_0^x(n) \cap L_0^y(n)\}. \end{aligned}$$

We define

$$L_0 = \bigcup_{n \in \mathbb{N}} L_0(n), \quad L_0^x = \bigcup_{n \in \mathbb{N}} L_o^x(n).$$

REMARK (Empty statement)
It appears to be useful to have the languages under consideration contain a special element, denoted by E, which can be regarded as the empty statement. From now on E is considered to be an element of L_0, and L_0^x. We shall still write L_0 for $L_0 \cup \{E\}$ and L_0^x for $L_0^x \cup \{E\}$. Please note that syntactic constructs like $s;E$ or $E \| s$ are *not* in L_0.

Now that we have formulated the notion of guardedness for x, we can easily generalize this:

DEFINITION 1.3 (Guarded statements)
The set L_0^g of guarded statements (guarded for all x) is defined as

$$L_0^g = \cap_{x \in Stmv} L_0^x.$$

As L_0 and L_0^x, also L_0^g has a simple inductive structure.

LEMMA 1.4 *The set L_0^g can be given by the following syntax:*

$$t ::= a \mid t;s \mid t_1 \cup t_2 \mid t_1 \| t_2 \mid \mu x[t]$$

where $s \in L_0$.

We need yet another notion of syntactic nature, that is, the notion of closedness.

DEFINITION 1.5 (Free variables, closed statements)
For every statement $s \in L_0$ we define the set $FV(s)$ of all statement variables that occur freely in s as usual:

$$FV(a) = \varnothing,\ FV(x) = \{x\},\ FV(\mu x[s]) = FV(s) \setminus \{x\},$$
$$FV(s_1 op s_2) = FV(s_1) \cup FV(s_2), \text{ for } op = ;, \cup, \|.$$

We call a statement s *closed* (notation: closed (s)), whenever $FV(s) = \varnothing$. Finally, we define for $L = L_0, L_0^{\star}$, and L_0^{δ}:

$$L^{cl} = \{s \mid s \in L \mid \text{closed } (s)\}$$

We have: $(L_0)^{cl} = (L_0^{\star})^{cl} = (L_0^{\delta})^{cl}$.

We expect that the reader may benefit from a few

EXAMPLES
First we observe that $L_0^{\delta} \subseteq L_0^{\star} \subseteq L_0$. Further we have that

$$x \in L_0,\ x \notin L_0^{\star}$$
$$y;x \in L_0^{\star},\ y;x \notin L_0^{\delta}$$
$$\mu x[y;x] \in L_0,\ \mu y[y;x] \notin L_0$$
$$a;\mu x[y;x] \in L_0^{\star} \cap L_0^{\delta}$$
$$\mu y[a;\mu x[y;x]] \in L_0$$

1.2 *Operational semantics*

We first introduce a semantic universe for both the operational and the denotational semantics for L_0.

DEFINITION 1.6 (Semantic universe P_0)
Let A^{∞}, the set of finite and infinite words over A, be given by

$$A^{\infty} = A^{*} \cup A^{\omega}.$$

For the empty word we use the special symbol ϵ. Let $d_{A^{\infty}}$ denote the usual metric on A^{∞} (see example A.1.1). We define

$$P_0 = \mathcal{P}_{nc}(A^{\infty}),$$

with typical elements $p,q, \ldots$, the set of all non-empty, compact subsets of A^{∞}. As a metric on P_0 we take $d_{P_0} = (d_{A^{\infty}})_H$, the Hausdorff distance induced by $d_{A^{\infty}}$. According to proposition A.7 we have that P_0 together with the metric d_{P_0} is a complete metric space.

The operational semantics for L_0 is based on the notion of a *transition relation.*

DEFINITION 1.7 (Transition relation for L_0^{δ})
We define a transition relation

$$\rightarrow \subseteq L_0^{\delta} \times A \times L_0$$

(writing $s \xrightarrow{a} s'$ for $(s,a,s') \in \rightarrow$) as the smallest relation satisfying
(i) $a \rightarrow E$ (for all $a \in A$)
(ii) for all $a \in A$, $s,t \in L_0^{\delta}$, $s',\bar{s} \in L_0$: if $s' \neq E$, then:

$$s \xrightarrow{a} s' \Rightarrow (s;\bar{s} \xrightarrow{a} s';\bar{s}$$
$$\wedge\ s \cup t \xrightarrow{a} s' \wedge t \cup s \xrightarrow{a} s'$$
$$\wedge\ s \| t \xrightarrow{a} s' \| t \wedge t \| s \xrightarrow{a} t \| s'$$
$$\wedge\ \mu x[s] \xrightarrow{a} s'[\mu x[s]/x]),$$

where the latter statement is obtained by replacing all free occurrences of x in s by $\mu x[s]$; and if $s' = E$, then:

$$s \xrightarrow{a} E \Rightarrow (s;\bar{s} \xrightarrow{a} \bar{s}$$
$$\wedge\ s \cup t \xrightarrow{a} E \wedge t \cup s \xrightarrow{a} E$$
$$\wedge\ s \| t \xrightarrow{a} t \wedge t \| s \xrightarrow{a} t$$
$$\wedge\ \mu x[s] \xrightarrow{a} E).$$

Intuitively, $s \xrightarrow{a} s'$ tells us that s can do the elementary action a as a first step, resulting in the statement s'. We now give the definition of Θ_0, the operational semantics for L_0^{cl}. (It is defined on *closed* statements only, because we do not want to give an operational meaning to, e.g., $a;x$: what should it be?) It will be the fixed point of the following contraction.

DEFINITION 1.8 (Φ_0)
Let $\Phi_0 : (L_0^{cl} \to P_0) \to (L_0^{cl} \to P_0)$ be given by

$$\Phi_0(F)(s) = \begin{cases} \{\epsilon\} & \text{if } s = E \\ \bigcup \{a \cdot F(s') \mid s' \in L_0^{cl} \wedge a \in A \wedge s \xrightarrow{a} s'\} & \text{if } s \neq E \end{cases}$$

for $F \in L_0^{cl} \to P_0$ and $s \in L_0^{cl}$.

REMARKS 1.9
(1) It is straightforward to prove that Φ_0 is contracting.
(2) Please note that closed (s) and $s \xrightarrow{a} s'$ imply closed (s').
(3) We have that $\Phi_0(F)(s)$ is a non-empty, compact subset of A^∞, because $\{a \mid \exists s' \in L_0^{cl}[s \xrightarrow{a} s']\}$ is finite and non-empty (this follows from lemma 1.14 below) and $F(s')$ is compact for every $s' \in L_0^{cl}$. This implies that $\Phi_0(F) \in L_0^{cl} \to P_0$.

DEFINITION 1.10 (Θ_0): $\Theta_0 = \text{Fixed Point}(\Phi_0)$

REMARK: We use open brackets to denote application of Θ_0 to an argument s: $\Theta_0[\![s]\!]$.

In [BKMOZ] another, seemingly more operational, definition of Θ_0 is given. We shall repeat a slightly different version of it here and show that it is equivalent to this fixed-point definition.

DEFINITION 1.11 (Θ_0^*)
Let $s \in L_0^{cl}$, $s \neq E$. We define $\Theta_0^* : L_0^{cl} \to P_0$ by putting $w \in A^\infty$ in $\Theta_0^*[\![s]\!]$ if and only if one of the following two conditions is satisfied:

(i) $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} s_n \wedge s_n = E \wedge w = a_1 \cdots a_n$

(ii) $s \to s_1 \to s_2 \to \cdots \to s_n \to \cdots \wedge w = a_1 \cdots a_n a_{n+1} \cdots$

(where $s \xrightarrow{a_1} s' \xrightarrow{a_2} s''$ abbreviates $s \xrightarrow{a_1} s' \wedge s' \xrightarrow{a_2} s''$). If $s = E$, then $\Theta_0^*[\![s]\!] = \{\epsilon\}$.

LEMMA 1.12: $\Theta_0 = \Theta_0^*$

PROOF
Let $w \in A^\infty$, $s \in L_0^{\prime}$, with $s \neq E$. We have

$$w \in \Theta_0^*[\![s]\!] \Leftrightarrow \text{[definition } \Theta_0^*\text{]}$$

$$\exists a \in A\, \exists s' \in L_0^{\prime}\, \exists w' \in A^\infty\ [s \xrightarrow{a} s' \wedge w = a \cdot w' \wedge w' \in \Theta_0^*[\![s]\!]]$$

$$\Leftrightarrow \text{[definition } \Phi_0\text{]}$$

$$w \in \Phi_0(\Theta_0^*)(s).$$

Since $\Theta_0^* \in L_0^{\prime} \to P_0$, it follows that $\Theta_0^* = \Phi(\Theta_0^*)$. Thus $\Theta_0^* = \Theta_0$.

We give yet another characterization of Θ_0. It is based on the following definition and will be the one we use in proving semantic equivalence.

DEFINITION 1.13 (Initial steps)
We define a function

$$I : L_0^{\delta} \to \mathcal{P}_{fin}(A \times L_0)$$

(where $\mathcal{P}_{fin}(X) = \{Y \mid Y \subseteq X \wedge \text{finite } (Y)\}$) by induction on L_0^{δ}:
(i) $I(E) = \varnothing$, and $I(a) = \{(a, E)\}$
(ii) Suppose $I(s) = \{(a_i, s_i)\}$, $I(t) = \{(b_j, t_j)\}$ for $s,t \in L_0^{\delta}$, $a_i, b_j \in A$, $s_i, t_j \in L_0$. (The variables i and j range over some finite sets of indices, which we have omitted.) Then

$$I(s;\bar{s}) = \{(a_i, s_i;\bar{s})\} \text{ (for } \bar{s} \in L_0)$$

$$I(s \cup t) = I(s) \cup I(t)$$

$$I(s \| t) = \{(a_i, s_i \| t)\} \cup \{(b_j, s \| t_j)\}$$

$$I(\mu x[s]) = \{(a_i, s_i[\mu x[s]/x])\}.$$

REMARK: Please note that for all $s \neq E$ the set $I(s)$ is finite and non-empty.

This definition is motivated by the following lemma, which can be easily proved.

LEMMA 1.14: $\forall a \in A \forall s \in L_0^{\delta} \forall s' \in L_0\ [s \xrightarrow{a} s' \Leftrightarrow (a, s') \in I(s)]$

COROLLARY 1.15: $\Phi_0(F)(s) = \bigcup \{a \cdot F(s') \mid (a,s') \in I(s)\}$, for $F : L_0^{\prime} \to P_0$, $s \in L_0^{\prime} \setminus \{E\}$.

1.3 *Denotational semantics*

The second semantic function we define for L_0 will be *denotational:* We call a semantic function $F : L_0 \to M$ (where M is some mathematical domain) denotational if it is compositionally defined and tackles recursion with the help of fixed points. The first condition is satisfied if for every syntactic operator *op* in L_0 we can define a corresponding semantic operator $\tilde{op} : M \times M \to M$ (assuming *op* to be binary) such that

$$F(s_1 op s_2) = F(s_1)\tilde{op}F(s_2).$$

As semantic domain for the denotational semantics of L_0 we take again P_0. The semantic operators corresponding with ; , $\cup$, and $\|$, the syntactic operators in L_0, will be of type $P_0 \times P_0 \to P_0$.

DEFINITION 1.16 (Semantic operators)
The operators $\tilde{;}$, $\tilde{\cup}$, $\tilde{\|}$: $P_0 \times P_0 \to P_0$ are defined as follows. Let $p, q \in P_0$, then

$$(i) \quad p\tilde{;}q = \begin{cases} q & \text{if } p = \{\epsilon\} \\ \bigcup \{a \cdot (p_a \tilde{;} q) \mid p_a \neq \varnothing\} & \text{otherwise} \end{cases}$$

$$(ii) \quad p \tilde{\cup} q = p \cup q \quad \text{(set-theoretic union)}$$

$$(iii) \quad p \tilde{\|} q = \begin{cases} p & \text{if } q = \{\epsilon\} \\ q & \text{if } p = \{\epsilon\} \\ \bigcup \{a \cdot (p_a \tilde{\|} q) \mid p_a \neq \varnothing\} \cup \bigcup \{a \cdot (p \tilde{\|} q_a) \mid q_a \neq \varnothing\} & \text{otherwise,} \end{cases}$$

where, for every $p \in P_0$ and $a \in A$, we define:

$$p_a = \{w \mid w \in A^\infty \wedge a \cdot w \in p\}.$$

(We often write op rather than $\tilde{op}$ if no confusion is possible.)

REMARKS 1.17
(1) This definition is self-referential and needs some justification. We shall give it for $\tilde{;}$ and leave the case of $\|$ to the reader. We define a mapping: $\Phi: (P_0 \times P_0 \to P_0) \to (P_0 \times P_0 \to P_0)$ by

$$\Phi(F)(p,q) = \begin{cases} q & \text{if } p = \{\epsilon\} \\ \bigcup \{a \cdot F(p_a, q) \mid p_a \neq \varnothing\} & \text{otherwise.} \end{cases}$$

It is not difficult to show that Φ is contracting. Then we define: $\tilde{;} = \text{Fixed Point}(\Phi)$, which satisfies the equation of definition 1.16 above.

(2) If we define the *left-merge* operator $\lfloor\!\lfloor$ by

$$p \lfloor\!\lfloor q = \begin{cases} \varnothing & \text{if } p = \{\epsilon\} \\ \bigcup \{a \cdot (p_a \| q) \mid p_a \neq \varnothing\} & \text{otherwise,} \end{cases}$$

then we have that

$$p \| q = p \lfloor\!\lfloor q \cup q \lfloor\!\lfloor p$$

(using the fact that $p' \| q' = q' \| p'$, for all p' and q'). This abbreviation will be helpful in some future proofs.

We need the following properties, which are easily verified:

LEMMA 1.18
(a) For $op = \tilde{;}$, $\tilde{\cup}$, and $\tilde{\|}$ we have

$$\forall p,p',q,q' \in P_0 \; [d_{P_0}(p \; op \; q, p' \; op \; q') \leqslant \max\{d_{P_0}(p,p'), d_{P_0}(q,q')\}].$$

(b) For $p,p' \in P_0$ with $\epsilon \notin p$, $\epsilon \notin p'$, and $q,q' \in P_0$ we have

$$d_{P_0}(p\tilde{;}q, p'\tilde{;}q') \leqslant \max\{d_{P_0}(p,p'), \tfrac{1}{2} \cdot d_{P_0}(q,q')\}.$$

(c) The operators $\tilde{;}$, $\tilde{\cup}$, and $\tilde{\|}$ preserve compactness.

We shall treat recursion with the help of environments, which are used to store and retrieve meanings of statement variables. They are defined in

DEFINITION 1.19 (Semantic environments)
The set Γ of *semantic environments*, with typical elements γ, is given by

$$\Gamma = Stmv \rightarrow^{fin} P_0.$$

We write $\gamma\{p/x\}$ for a *variant* of γ which is like γ but with $\gamma\{p/x\}(x)=p$.

Now we have defined everything we need to introduce the denotational semantics for L_0.

DEFINITION 1.20 (Ψ_0, D_0)
We shall define D_0 as the fixed point of

$$\Psi_0 : (L_0 \rightarrow \Gamma \rightarrow^1 P_0) \rightarrow (L_0 \rightarrow \Gamma \rightarrow^1 P_0)$$

which is given by induction on L_0. (Here $\Gamma \rightarrow^1 P_0$ denotes the set of non-distance-increasing functions (see A.3.(c)).) Let $F \in L_0 \rightarrow \Gamma \rightarrow^1 P_0$, then:

(*i*) $\Psi_0(F)(a)(\gamma)=\{a\}$, $\Psi_0(F)(x)(\gamma)=\gamma(x)$, $\Psi_0(F)(E)(\gamma)=\{\epsilon\}$

(*ii*) $\Psi_0(F)(s\ op\ t)(\gamma) = \Psi_0(F)(s)(\gamma)\ \tilde{op}\ \Psi_0(F)(t)(\gamma)$

(*iii*) $\Psi_0(F)(\mu x[s])(\gamma) = \Psi_0(F)(s)(\gamma\{F(\mu x[s])(\gamma)/x\})$ for $s \in L_0^x$,

for $op = ;, \cup, \|$, and $\tilde{op}$ as in definition 1.16. (We define $\Psi_0(F)$ only for those s and γ, such that $FV(s) \subseteq dom(\gamma)$.) Now we set

$$D_0 = \text{Fixed Point}(\Psi_0).$$

REMARK: We have: $D_0[\![\mu x[s]]\!](\gamma) = D_0[\![s]\!](\gamma\{D_0[\![\mu x[s]]\!](\gamma)/x\})$. (As for $\mathcal{O}_0$, we also use open brackets for $\mathcal{P}_0$.)

It is not obvious that Ψ_0 is contracting. The fact that we consider only *guarded* recursion is essential for proving it.

LEMMA 1.21
(a) *If* $F \in L_0 \rightarrow \Gamma \rightarrow^1 P_0$, *then* $\Psi_0(F) \in L_0 \rightarrow \Gamma \rightarrow^1 P_0$.
(b) *If* $F \in L_0 \rightarrow \Gamma \rightarrow^1 P_0$, *then for all* $\gamma_1, \gamma_2 \in \Gamma$, $s \in L_0$:
(*) $\forall y \in Stmv[s \notin L_0^y \Rightarrow \gamma_1(y)=\gamma_2(y)]$
$\Rightarrow$
(**) $d_{P_0}(\Psi_0(F)(s)(\gamma_1), \Psi_0(F)(s)(\gamma_2)) \leqslant \frac{1}{2} \cdot d_\Gamma(\gamma_1, \gamma_2)$.
(c) Ψ_0 *is contracting on* $L_0 \rightarrow \Gamma \rightarrow^1 P_0$.

PROOF
(a) The proof of (a) goes along the lines of (b), which is more interesting.
(b) Let $F \in L_0 \rightarrow^1 P_0$, let $\gamma_1, \gamma_2 \in \Gamma$. We use induction on L_0.
(i) For $s=a$ we have: $d_{P_0}(\Psi_0(F)(a)(\gamma_1), \Psi_0(F)(a)(\gamma_2))=0$. Let $s=x$, with $x \in Stmv$. Suppose (*) holds for x. Then

$$\begin{aligned} d_{P_0}(\Psi_0(F)(x)(\gamma_1), \Psi_0(F)(x)(\gamma_2)) &= d_{P_0}(\gamma_1(x), \gamma_2(x)) \\ &= 0 \ \text{[because of (*)]}. \end{aligned}$$

(ii) We only treat sequential composition and recursion. Let $s = s_1; s_2$, with $s_1, s_2 \in L_0$. Suppose (b)

holds for s_1 and s_2. Suppose (*) holds for $s_1;s_2$. This implies that (*) holds for s_1. Thus we have (**) for s_1. Now:

$$
\begin{aligned}
& d_{P_0}(\Psi_0(F)(s_1;s_2)(\gamma_1),\ \Psi_0(F)(s_1;s_2)(\gamma_2)) \\
&= d_{P_0}(\Psi_0(F)(s_1)(\gamma_1)\tilde{;}\,\Psi_0(F)(s_2)(\gamma_1),\ \Psi_0(F)(s_1)(\gamma_2)\tilde{;}\,\Psi_0(F)(s_2)(\gamma_2)) \\
&\leqslant [\text{for all } s\in L_0\setminus\{E\},\ F \text{ and } \gamma \text{ we have: } \epsilon\notin\Psi_0(F)(s)(\gamma);\ \text{thus lemma 1.18}(b)\ \text{applies}] \\
&\quad \max\{d_{P_0}(\Psi_0(F)(s_1)(\gamma_1),\ \Psi_0(F)(s_1)(\gamma_2)),\ \tfrac{1}{2}\cdot d_{P_0}(\Psi_0(F)(s_2)(\gamma_1),\ \Psi_0(F)(s_2)(\gamma_2))\} \\
&\leqslant [(**) \text{ for } s_1;\ (a) \text{ for } s_2] \\
&\quad \max\{\tfrac{1}{2}\cdot d_\Gamma(\gamma_1,\gamma_2),\ \tfrac{1}{2}\cdot d_\Gamma(\gamma_1,\gamma_2)\} \\
&= \tfrac{1}{2}\cdot d_\Gamma(\gamma_1,\gamma_2).
\end{aligned}
$$

(The proof for $s_1 \cup s_2$ and $s_1 \| s_2$ is similar.) Next we treat recursion. Let $s_1\in L_0$ and suppose that $\mu x[s_1]$ satisfies (*). Then s_1 satisfies it. Thus we have (**) for s_1. Now

$$
\begin{aligned}
& d_{P_0}(\Psi_0(F)(\mu x[s_1])(\gamma_1),\ \Psi_0(F)(\mu x[s_1])(\gamma_2)) \\
&= d_{P_0}(\Psi_0(F)(s)(\gamma_1\{F(\mu x[s_1])(\gamma_1)/x\}),\ \Psi_0(F)(s)(\gamma_2\{F(\mu x[s_1])(\gamma_2)/x\})) \\
&\leqslant [(*) \text{ holds for } s_1,\ \text{also w.r.t. } \gamma_i\{F(\mu x[s_1])(\gamma_i)/x\},\ \text{for } i=1,2,\ \text{thus so does}(**)] \\
&\quad \tfrac{1}{2}\cdot d_\Gamma(\gamma_1\{F(\mu x[s_1])(\gamma_1)/x\},\ \gamma_2\{F(\mu x[s_1])(\gamma_2)/x\}) \\
&\leqslant \tfrac{1}{2}\cdot\max\{d_\Gamma(\gamma_1,\gamma_2),\ d_{P_0}(F(\mu x[s_1])(\gamma_1),\ F(\mu x[s_1])(\gamma_2))\} \\
&\leqslant [(a) \text{ for } \mu x[s_1]] \\
&\quad \tfrac{1}{2}\cdot d_\Gamma(\gamma_1,\gamma_2).
\end{aligned}
$$

(c) Let $F_1, F_2\in L_0\to\Gamma\to{}^1P_0$. We only treat recursion. Suppose $d_{P_0}(\Psi_0(F_1)(s)(\gamma),\ \Psi_0(F_2)(s)(\gamma)) \leqslant \frac{1}{2}\cdot d(F_1,F_2)$, for some $s\in L_0^{\surd}$, all $\gamma\in\Gamma$. Then

$$
\begin{aligned}
& d_{P_0}(\Psi_0(F_1)(\mu x[s])(\gamma),\ \Psi_0(F_2)(\mu x[s])(\gamma)) \\
&= [\gamma_i = \gamma\{F_i(\mu x[s])(\gamma)/x\},\ i=1,2] \\
&\quad d_{P_0}(\Psi_0(F_1)(s)(\gamma_1),\ \Psi_0(F_2)(s)(\gamma_2)) \\
&\leqslant \max\{d_{P_0}(\Psi_0(F_1)(s)(\gamma_1),\Psi_0(F_2)(s)(\gamma_1)),\ d_{P_0}(\Psi_0(F_2)(s)(\gamma_1),\Psi_0(F_2)(s)(\gamma_2))\} \\
&\leqslant [\text{induction, } (b)] \\
&\quad \max\{\tfrac{1}{2}\cdot d(F_1,F_2),\ \tfrac{1}{2}\cdot d_\Gamma(\gamma_1,\gamma_2)\} \\
&= \max\{\tfrac{1}{2}\cdot d(F_1,F_2),\ \tfrac{1}{2}\cdot d_{P_0}(F_1(\mu x[s])(\gamma),\ F_2(\mu x[s])(\gamma))\} \\
&= \tfrac{1}{2}\cdot d(F_1,F_2).
\end{aligned}
$$

1.4 *Semantic equivalence of* $\mathcal{O}_0$ and $\mathcal{D}_0$

An important difference between $\mathcal{D}_0$ and $\mathcal{O}_0$ is that recursion is treated with and without semantic environments, respectively. We have

$$\mathcal{D}_0[\![\mu x[s]]\!](\gamma) = \mathcal{D}_0[\![s]]\!](\gamma\{\mathcal{D}_0[\![\mu x[s]]\!](\gamma)/x\})$$

and

$$\mathcal{O}_0[\![\mu x[s]]\!] = \mathcal{O}_0[\![s[\mu x[s]/x]]\!].$$

In the latter case the statement $\mu x[s]$ is *syntactically* substituted for all free statement variables x in s, whereas in the first case the environment γ is changed by setting x to the *semantic* value of $\mu x[s]$. We shall compare $\mathcal{O}_0$ and $\mathcal{D}_0$ by relating both to an intermediate semantic function $\mathcal{O}_0'$, which takes *syntactic* instead of *semantic* environments as arguments. It will be defined such that for syntactic environments δ:

$$\mathcal{O}_0'[\![\mu x[s]]\!](\delta) = \mathcal{O}_0'[\![s]\!](\delta\{\mu x[s]/x\}).$$

Here δ is changed, the new value of x is the statement $\mu x[s]$. By first comparing $\mathcal{O}_0$ and $\mathcal{O}_0'$ and next $\mathcal{O}_0'$ and $\mathcal{D}_0$ we are able to prove the main result of this section: $\mathcal{O}_0[\![s]\!] = D_0[\![s]\!](\gamma)$, for all $s \in L_0^I$ and arbitrary $\gamma \in \Gamma$. For the definition of $\mathcal{O}_0'$, we need

DEFINITION 1.22 (Syntactic environments)
The set Δ of *syntactic* environments, with typical elements δ, is defined by

$$\Delta = \{\delta \mid \delta \in (Stmv \rightarrow^{fin} L_0) \wedge (\delta \text{ is normal})\},$$

where the notion of *normal* environments is given in:

DEFINITION 1.23 (Normal environments)
A syntactic environment δ is called *normal*, whenever

(i) $\forall x \in dom(\delta)\ [\delta(x) \in L_0^x]$

(ii) $\forall s \in L_0\ [FV(s) \subseteq dom(\delta) \Rightarrow \exists k \geqslant 0\ [s[\delta]^k \in L_0^{cl}]]$,

where $s[\delta]^0 = s$, $s[\delta]^1 = s[\delta(x_1)/x_1, \ldots, \delta(x_n)/x_n]$ (with $FV(s) = \{x_1, \ldots, x_n\}$) and $s[\delta]^{n+1} = (s[\delta])[\delta]^n$. For δ normal and $s \in L_0$, with $FV(s) \subseteq dom(\delta)$, we define

$$s\langle\delta\rangle = s[\delta]^k,$$

where $k = \min\{m \mid s[\delta]^m \in L_0^{cl}\}$.

REMARKS
(1) From now on we shall assume whenever we consider $s \in L_0$ and $\delta \in \Delta$ together (as two arguments for a function, or as a pair) that $FV(s) \subseteq dom(\delta)$.
(2) Let $\delta \in Stmv \rightarrow^{fin} L_0$ be such that for $x,y \in Stmv$: $\delta(x) = y$ and $\delta(y) = x$. Such an environment is *not* normal. It does not give us any useful information about the values of x and y.
(3) It would be too restrictive to require for all $\delta \in Stmv \rightarrow^{fin} L_0$ that $\forall x \in dom(\delta)\ [x[\delta] \in L_0^{cl}]$. An example may illustrate this. Let δ be defined such that $dom(\delta) = \{x,y\}$, and

$$\delta(y) = \mu y[b;x;y],\quad \delta(x) = \mu x[a;\mu y[b;x;y]].$$

Such an environment we shall encounter when computing $\mathcal{O}_0'[\![\mu x[a;\mu y[b;x;y]]]\!]$. Now $y[\delta] = \delta(y) \notin L_0^{cl}$, but $y[\delta]^2 \in L_0^{cl}$.

Now that we have introduced syntactic environments, we can formulate a principle of induction for the set $L_0 \times \Delta$, which we shall heavily use in the sequel.

THEOREM 1.24 (Induction principle for $L_0 \times \Delta$)
Let $\Xi \subseteq L_0 \times \Delta$. *If:*

(1) $A \times \Delta \subseteq \Xi$

(2) $\{s,t\} \times \Delta \subseteq \Xi \Rightarrow \{s;\bar{s}, s \cup t, s \| t\} \times \Delta \subseteq \Xi$ *for* $s,t,\bar{s} \in L_0$

(3) $\{s\}\times\Delta\subseteq\Xi \Rightarrow \{\mu x[s]\}\times\Delta\subseteq\Xi$ for $s\in L_0^{\delta}$

(4) $(\delta(x),\delta)\in\Xi \Rightarrow (x,\delta)\in\Xi$ *for* $x\in Stmv$ *and* $\delta\in\Delta$,

then:

$$\Xi = L_0\times\Delta$$

PROOF

Let $\Xi\subseteq L_0\times\Delta$, suppose Ξ satisfies (1) through (4). We first prove fact (a) and fact (b) given below, and next show that (a) and (b) imply: $\Xi=L_0\times\Delta$. So we have

fact (a): $L_0^{\delta}\times\Delta\subseteq\Xi$

fact (b): $\forall S\subseteq L_0\times\Delta\,[S\subseteq\Xi\Rightarrow S'\subseteq\Xi]$, where

$$S' = \{(s,\delta)\mid(s,\delta)\in L_0\times\Delta\wedge\forall x\in FV(s)\,[s\notin L_0^{\delta}\Rightarrow(\delta(x),\delta)\in S]\}.$$

To show that (a) holds, we use (1), (2), and (3), and induction on the structure of L_0^{δ}. We proceed with (b). Let $S\subseteq L_0\times\Delta$ and suppose $S\subseteq\Xi$. Let S' be as above. We use (1) through (4) and induction on L_0 to show that $S'\subseteq\Xi$. Let $(s,\delta)\in S'$, for $s\in L_0$, $\delta\in\Delta$.

(i) $s\equiv a$: $(a,\delta)\in\Xi$, because (1).

(ii) $s\equiv s_1 ops_2$: Suppose that if $(s_i,\delta)\in S'$, then $(s_i,\delta)\in\Xi$, for $i=1,2$. If $(s,\delta)\in S'$, then also (s_1,δ) and $(s_2,\delta)\in S'$. Thus $(s_1,\delta),(s_2,\delta)\in\Xi$. By (2) we have: $(s_1\ op\ s_2,\delta)\in\Xi$.

(iii) $s\equiv\mu x[s_1]$, for $s_1\in L_0^{\delta}$: Suppose that $(s_1,\delta)\in S'$ implies $(s_1,\delta)\in\Xi$. Because $s_1\in L_0^{\delta}$ we have: $(s_1,\delta)\in S'\Leftrightarrow(\mu x[s_1],\delta)\in S'$. Because $(\mu x[s_1],\delta)\in S'$ we have $(s_1,\delta)\in\Xi$. Thus, using (3), we have $(\mu x[s_1],\delta)\in\Xi$.

(iv) $s\equiv x$: If $(x,\delta)\in S'$, then $(\delta(x),\delta)\in S$, thus (because $S\subseteq\Xi$) $(\delta(x),\delta)\in\Xi$. Because of (4), we then have that $(x,\delta)\in\Xi$.

Thus facts (a) and (b) hold. Next we show that $\Xi=L_0\times\Delta$. For this purpose we define, for all $n\in\mathbb{N}$:

$$V_0 = L_0^{\delta}\times\Delta,$$

$$V_{n+1} = \{(s,\delta)\mid(s,\delta)\in L_0\times\Delta\wedge\forall x\in FV(s)\,[s\notin L_0^{\delta}\Rightarrow(\delta(x),\delta)\in V_n]\}.$$

Then we have:

$$(*)\quad \forall s\in L_0\forall\delta\in\Delta\exists n\in\mathbb{N}\,[s[\delta]^n\in L_0^{\delta}\Rightarrow(s,\delta)\in V_n],$$

which we prove with induction on $n\in\mathbb{N}$. Let $s\in L_0$ and $\delta\in\Delta$. If $s[\delta]^0\in L_0^{\delta}$, then $s\in L_0^{\delta}\subseteq L_0^{\delta}$. Thus $(s,\delta)\in V_0$. Now suppose (*) holds for $n\in\mathbb{N}$, and suppose $s[\delta]^{n+1}\in L_0^{\delta}$. Then $(s[\delta])[\delta]^n\in L_0^{\delta}$, thus by induction $(s[\delta],\delta)\in V_n$. This implies $(s,\delta)\in V_{n+1}$, which proves (*) for $n+1$.

Because all $\delta\in\Delta$ are normal we have

$$\forall(s,\delta)\in L_0\times\Delta\exists n\in\mathbb{N}\,[s[\delta]^n\in L_0^{\delta}].$$

Together with (*) this implies:

$$\forall(s,\delta)\in L_0\times\Delta\exists n\in\mathbb{N}\,[(s,\delta)\in V_n].$$

Since $V_n\subseteq L_0\times\Delta$, for all $n\in\mathbb{N}$, it follows that $L_0\times\Delta=\cup_{n\in\mathbb{N}}V_n$. Now $V_0\subseteq\Xi$ because of (a), and $V_n\subseteq\Xi\Rightarrow V_{n+1}\subseteq\Xi$ because of (b), so we conclude: $\Xi=L_0\times\Delta$.

□

REMARK

We cannot reason about a free statement variable x unless we know what statement it is bound to. Therefore, we consider non-closed statements together with syntactic environments, which give information about the free variables they contain. This explains why we have formulated an induction principle for $L_0\times\Delta$ instead of L_0 only.

Now let $\Xi \subseteq L_0 \times \Delta$. The first three conditions of the principle suffice to prove that $L_0^g \times \Delta \subseteq \Xi$, since they express exactly the syntactic structure of L_0^g (see lemma 1.4). (We have chosen L_0^g here instead of L_0^d, because the latter set has no simple inductive structure.) Thus also $L_0^d \times \Delta$ ($\subseteq L_0^g \times \Delta$) $\subseteq \Xi$. Adding condition (4) enables us to prove $L_0 \times \Delta \subseteq \Xi$. This may be motivated by the following. For every statement $s \in L_0$ and normal environment $\delta \in \Delta$ there exists an $l \in \mathbb{N}$ such that $s[\delta]^l \in L_0^d \subseteq L_0^g$. Let us call $k \in \mathbb{N}$ with $k = \min\{l \mid s[\delta]^l \in L_0^d\}$ the *degree of closedness* of s with respect to δ. Please note that every $s \in L_0^d$ has degree 0, and arbitrary $s \in L_0$ has, for arbitrary δ, a finite degree. Therefore, this degree can be used as a measure for the complexity of statements. Our induction principle is indeed a principle of induction on the degree of closedness. Conditions (1), (2), and (3) are sufficient to prove Ξ for all (s, δ) with degree 0. They form, so to speak, the basis of the principle. Condition (4) expresses the "step part": if Ξ holds for $(\delta(x), \delta)$, which has degree k, say, then Ξ holds for (x, δ), which then has degree $k+1$.

We now proceed with the definition of Θ_0'. It will be of type

$$\Theta_0' : L_0 \to \Delta \to P_0,$$

which could be called intermediate between

$$\Theta_0 : L_0^d \to P_0, \text{ and } D_0 : L_0 \to \Gamma \to P_0.$$

Instead of basing the definition of Θ_0' on some transition relation (as in definition 1.8) we use a variant of the initial step function (definition 1.13).

DEFINITION 1.25 (Initial steps with syntactic environments)
We define a function

$$I' : L_0 \to \Delta \to \mathcal{P}_{fin}(A \times L_0 \times \Delta),$$

using the induction principle for $L_0 \times \Delta$. The predicate $\Xi \subseteq L_0 \times \Delta$ we use is defined as:

$$\Xi(s, \delta) \equiv I'(s)(\delta) \text{ is defined.}$$

We shall define I' such that Ξ satisfies the induction conditions. Thus we ensure that I' is defined for every $s \in L_0$ and $\delta \in \Delta$ (with $FV(s) \subseteq dom(\delta)$).

(1) $I'(E)(\delta) = \varnothing$, and $I'(a)(\delta) = \{(a, E, \delta)\}$, for all $a \in A$, $\delta \in \Delta$.

(2) Suppose $I'(s) = \lambda\delta.\{(a_i, s_i, \delta_i)\}$, $I'(t) = \lambda\delta.\{(b_j, t_j, \delta_j)\}$ for $s, t, s_i, t_j \in L_0$, $a_i, b_j \in A$, and $\delta_i, \delta_j \in \Delta$. (The variables i and j range over some finite sets of indices, which are omitted.) Then:

$$I'(s\,;\overline{s})(\delta) = \{(a_i,\ s_i;\overline{s},\ \delta_i)\} \quad (\textit{for } \overline{s} \in L_0)$$

$$I'(s \cup t)(\delta) = I'(s)(\delta) \cup I'(t)(\delta)$$

$$I'(s \| t)(\delta) = \{(a_i,\ s_i \| t,\ \delta_i)\} \cup \{(b_j,\ s \| t_j,\ \delta_j)\}$$

(3) For the definition of $I'(\mu x[s])$ we have to consider possible clashes of variables. Therefore, we distinguish between two cases (supposing that $I'(s)$ has already been defined):

$$I'(\mu x[s])(\delta) = \begin{cases} I'(s)(\delta\{\mu x[s]/x\}) & \text{if } x \notin dom(\delta) \\ I'(\overline{s})(\delta\{\mu\overline{x}[\overline{s}]/\overline{x}\}) & \text{if } x \in dom(\delta), \end{cases}$$

where $\overline{x}$ is some fresh variable with $\overline{x} \notin dom(\delta)$ and $\overline{s} = s[\overline{x}/x]$.

(4) Suppose $I'(\delta(x))(\delta)$ has already been defined. We set:

$$I'(x)(\delta) = I'(\delta(x))(\delta).$$

REMARKS
(1) We have: if $I'(s)(\delta) = \{(a_i, s_i, \delta_i)\}$, then *normal* (δ_i), and thus $\delta_i \in \Delta$, for all i.

(2) The definition of $I'(\mu x[s])(\delta)$, with $x \in dom(\delta)$, is correct, because s and $\overline{s}$ have the same complexity.
(3) If $I'(s)(\delta)=\{(a_i,s_i,\delta_i)\}$, then for all i: $\forall x \in Stmv[x \in dom(\delta) \cap dom(\delta_i) \Rightarrow \delta(x) = \delta_i(x)]$.

DEFINITION 1.26 (Φ_0')
We define $\Phi_0':(L_0 \to \Delta \to P_0) \to (L_0 \to \Delta \to P_0)$ by

$$\Phi_0'(F)(s)(\delta) = \begin{cases} \{\epsilon\} & \text{if } s=E \\ \bigcup \{a \cdot F(s')(\delta') \mid (a,s',\delta') \in I'(s)(\delta)\} & \text{otherwise} \end{cases}$$

for $F \in L_0 \to \Delta \to P_0$, $s \in L_0$, and $\delta \in \Delta$ with $FV(s) \subseteq dom(\delta)$.

DEFINITION 1.27: $\mathcal{O}_0' = \text{Fixed Point}(\Phi_0')$

Next, we compare $\mathcal{O}_0$ and $\mathcal{O}_0'$. We can do this by relating I and I', since we have:

$$\mathcal{O}_0[\![s]\!] = \bigcup \{a \cdot \mathcal{O}_0[\![s']\!] \mid (a,s') \in I(s)\}, \text{ for } s \in L_0^{\mathcal{I}},\ s \neq E$$

$$\mathcal{O}_0'[\![s]\!](\delta) = \bigcup \{a \cdot \mathcal{O}_0'[\![s']\!](\delta') \mid (a,s',\delta') \in I'(s)(\delta)\}, \text{ for } s \in L_0,\ s \neq E,\ \delta \in \Delta$$

THEOREM 1.28 (Relating I and I')
For all $s \in L_0$ and $\delta \in \Delta$, with $FV(s) \subseteq dom(\delta)$, we have:

$$\forall a \in A \forall s' \in L_0 \forall \delta' \in \Delta\ [(a,s',\delta') \in I'(s)(\delta) \Leftrightarrow (a,s'<\delta'>) \in I(s<\delta>)].$$

(For the definition of $s<\delta>$ see 1.23.)

PROOF
We define

$$\Xi(s,\delta) \equiv \forall a \in A \forall s' \in L_0 \forall \delta' \in \Delta\ [(a,s',\delta') \in I'(s)(\delta) \Leftrightarrow (a,s'<\delta'>) \in I(s<\delta>)]$$

and use the induction principle for $L_0 \times \Delta$ to show that $\Xi = L_0 \times \Delta$. We only treat the case of recursion. Suppose $s \in L_0^{\chi}$ such that $\{s\} \times \Delta \subseteq \Xi$. We have to show that $\{\mu x[s]\} \times \Delta \subseteq \Xi$. Let $\delta \in \Delta$ and assume (without loss of generality) that $x \notin dom(\delta)$. Then

$$I'(\mu x[s])(\delta) = I'(s)(\delta')$$

where $\delta' = \delta\{\mu x[s]/x\}$ (by the definition of I'). On the other hand, we have

$$\begin{aligned} I(\mu x[s]<\delta>) &= [x \notin dom(\delta)] \\ &\quad I(\mu x[s<\delta>]) \\ &= I(s<\delta>[\mu x[s<\delta>]/x]) \end{aligned}$$

(the latter equality following from:

$$\forall t \in L_0^{\mathcal{I}}\ [I(\mu x[t]) = I(t[\mu x[t]/x])]).$$

We take a quick (but deep) breath and proceed as follows:

$$\begin{aligned} & s<\delta>[\mu x[s<\delta>]/x] = [\text{definition } s<\delta>] \\ & \quad s[\delta]<\delta>[\mu x[s<\delta>]/x] \\ &= [x \notin dom(\delta),\ \forall y \in dom(\delta)\ [x \notin FV(\delta(y))]] \\ & \quad s[\delta][\mu x[s<\delta>]/x]<\delta> \\ &= s[\delta][\mu x[s]/x]<\delta> \end{aligned}$$

$$= [\delta' = \delta\{\mu x[s]/x\}]$$
$$s[\delta']<\delta>$$
$$= [x \notin FV(s[\delta'])]$$
$$s[\delta']<\delta'>$$
$$= s<\delta'>.$$

Thus we have $I(\mu x[s]<\delta>)=I(s<\delta'>)$. Combining this with $I'(\mu x[s])(\delta)=I'(s)(\delta')$, which we saw above, yields:

$$\Xi(\mu x[s],\delta) \Leftrightarrow \Xi(s,\delta').$$

Because $\{s\}\times\Delta\subseteq\Xi$ we may conclude: $\Xi(\mu x[s],\delta)$.

□

We formulate the relation of Θ_0 and Θ_0' in terms of their defining contractions Φ_0 and Φ_0'. This can be elegantly done using the following

DEFINITION 1.29
We define $<>:(L_0^{\prime}\to P_0) \to (L_0\to\Delta\to P_0)$, for every $F\in L_0^{\prime}\to P_0$, by

$$<>(F) = F^{<>} \text{ (notation)}$$
$$= \lambda s\in L_0\cdot\lambda\delta\in\Delta\cdot F(s<\delta>).$$

REMARK
This mapping links two kinds of semantic functions, one using syntactic environments, and the other one not using environments. If $F\in L_0^{\prime}\to P_0$, then $F^{<>}$ is a in a sense *extended* version of F: it can take as an argument also statements $s\in L_0$ that are *not* closed, provided it is supplied with a syntactic environment, which is to give the (syntactic) values for the free variables in s.

THEOREM 1.30 *(Relating Φ_0 and Φ_0')*: $\forall F\in L_0^{\prime}\to P_0\ [\Phi_0'(F^{<>}) = (\Phi_0(F))^{<>}]$

PROOF
The theorem is an immediate consequence of theorem 1.28. Let $F\in L_0^{\prime}\to P_0$, let $s\in L_0$, $s\neq E$.

$$\Phi_0'(F^{<>})(s)(\delta) = \bigcup\{a\cdot F^{<>}(s')(\delta')\mid(a,s',\delta')\in I'(s)(\delta)\}$$
$$= \bigcup\{a\cdot F(s'<\delta'>)\mid(a,s',\delta')\in I'(s)(\delta)\}$$
$$= [\text{theorem 1.28}]$$
$$\bigcup\{a\cdot F(s'<\delta'>)\mid(a,s'<\delta'>)\in I(s<\delta>)\}$$
$$= \Phi_0(F)(s<\delta>)$$
$$= (\Phi_0(F))^{<>}(s)(\delta).$$

Because Φ_0 and Φ_0' are contractions with Θ_0 and Θ_0' as their respective fixed points, we have:

COROLLARY 1.31 ($\Theta_0'=\Theta_0^{<>}$): $\forall s\in L_0\ \forall\delta\in\Delta\ [\Theta_0'[\![s]\!](\delta) = \Theta_0[\![s<\delta>]\!]]$.

Finally we relate

Θ_0':$L_0 \to \Delta \to P_0$ and $\mathfrak{D}_0$:$L_0 \to \Gamma \to P_0$.
For this purpose we define the following mapping.

DEFINITION 1.32
We define $\sim$:$(L_0 \to \Gamma \to P_0) \to (L_0 \to \Delta \to P_0)$ by:

$$\sim(F) = \tilde{F} \text{ (notation)}$$
$$= \lambda s \in L_0 \cdot \lambda \delta \in \Delta \cdot F(s)(\tilde{\delta}^F)$$

for $F \in L_0 \to \Gamma \to P_0$, where $\tilde{\delta}^F$ is given by $\tilde{\delta}^F = \lambda x \in dom(\delta) \cdot F(\delta(x))(\tilde{\delta}^F)$. (We often write $\tilde{\delta}$ rather than $\tilde{\delta}^F$ if from the context it is clear which F should be taken.)

REMARKS
(1) We have to justify the self-referential definition of $\tilde{\delta}^F$. For this purpose we define

$$\Xi(s, \delta) \equiv \forall x \in FV(s)\ [s \notin L_0^x \to (\tilde{\delta}^F(x) \text{ is well defined})],$$

for $s \in L_0$ and $\delta \in \Delta$, and use the induction principle to prove: $\Xi = L_0 \times \Delta$. Then it follows for all $x \in Stmv$ with $x \in dom(\delta)$ that $\tilde{\delta}^F(x)$ is well defined. Conditions (1) through (3) of the induction principle are trivially fulfilled. We prove condition (4). Suppose $(\delta(x), \delta) \in \Xi$. Thus $\tilde{\delta}^F(y)$ is well defined for all $y \in FV(\delta(x))$. This implies that $\tilde{\delta}^F(x)$ is well defined, since

$$\tilde{\delta}^F(x) = F(\delta(x))(\tilde{\delta}^F).$$

(2) In the same way as $<>$, also $\sim$ links two different kinds of semantic functions, one using *syntactic*, and the other using *semantic* environments. Again F is an extended version of F in the sense that it takes syntactic environments as an argument instead of semantic ones. In the definition above a syntactic environment $\delta \in \Delta$ is changed into a *semantic version* (according to the semantic function F) $\tilde{\delta}^F$ of it, which then is supplied as an argument to F.

Next, we come to the main theorem of this chapter. It relates the denotational semantics $\mathfrak{D}_0$ and the operational semantics $\mathcal{O}'_0$, which is a fixed point of Φ'_0, by stating that also $\tilde{\mathfrak{D}}_0$ is a fixed point of Φ'_0. From this it follows that $\Theta_0' = \tilde{\mathfrak{D}}_0$.

THEOREM 1.33: $\Phi_0'(\tilde{\mathfrak{D}}_0) = \tilde{\mathfrak{D}}_0$

PROOF
Let $\Xi \subseteq L_0 \times \Delta$ be defined by

$$\Xi(s, \delta) \equiv \Phi_0'(\tilde{\mathfrak{D}}_0)(s)(\delta) = \tilde{\mathfrak{D}}_0(s)(\delta)$$

for $(s, \delta) \in L_0 \times \Delta$. We use the induction principle for $L_0 \times \Delta$ to show that $\Xi = L_0 \times \Delta$. Let $\delta \in \Delta$.
(1) For $a \in A$ we have $\Phi_0'(\tilde{\mathfrak{D}}_0)(a)(\delta) = \{a\} = \tilde{\mathfrak{D}}_0(a)(\delta)$, so $A \times \Delta \subseteq \Xi$.
(2) Let $s, \bar{s} \in L_0$ and suppose $\Xi(s, \delta)$. We show: $\Xi(s;\bar{s}, \delta)$.

$\Phi_0'(\tilde{\mathfrak{D}}_0)(s;\bar{s})(\delta) =$ [definition Φ_0' and $I'(s;\bar{s})$]

$$\bigcup \{a' \cdot \tilde{\mathfrak{D}}_0(s';\bar{s})(\delta') \mid (a', s', \delta') \in I'(s)(\delta)\}$$
$$= \bigcup \{a' \cdot (\tilde{\mathfrak{D}}_0(s')(\delta');\ \tilde{\mathfrak{D}}_0(\bar{s})(\delta')) \mid (a', s', \delta') \in I'(s)(\delta)\}$$
$$= \text{[see remark (3) after definition 1.25]}$$
$$\bigcup \{a' \cdot (\tilde{\mathfrak{D}}_0(s')(\delta');\ \tilde{\mathfrak{D}}_0(\bar{s})(\delta)) \mid (a', s', \delta') \in I'(s)(\delta)\}$$
$$= \text{[definition ;]}$$

$(\bigcup\{a'\cdot\tilde{\mathfrak{D}}_0(s')(\delta')\mid(a',s',\delta')\in I'(s)(\delta)\});\ \tilde{\mathfrak{D}}_0(\overline{s})(\delta)$

$=$ [definition Φ_0']

$\Phi_0'(\tilde{\mathfrak{D}}_0)(s)(\delta);\ \tilde{\mathfrak{D}}_0(\overline{s})(\delta)$

$=$ [because $\Xi(s,\delta)$]

$\tilde{\mathfrak{D}}_0(s)(\delta);\ \tilde{\mathfrak{D}}_0(\overline{s})(\delta)$

$=\tilde{\mathfrak{D}}_0(s;\overline{s})(\delta)$.

This proves $\Xi(s;\overline{s},\delta)$. Now let $s,t\in L_0$ and suppose $\Xi(s,\delta)$ and $\Xi(t,\delta)$. We show: $\Xi(s\|t,\delta)$.

$\Phi_0'(\tilde{\mathfrak{D}}_0)(s\|t)(\delta)$ $=$ [definition Φ_0' and $I'(s\|t)$]

$\bigcup\{a'\cdot\tilde{\mathfrak{D}}_0(s'\|t)(\delta')\mid(a',s',\delta')\in I'(s)(\delta)\}\cup$

$\bigcup\{b'\cdot\tilde{\mathfrak{D}}_0(s\|t')(\delta')\mid(b',t',\delta')\in I'(t)(\delta)\}$

$=\bigcup\{a'\cdot(\tilde{\mathfrak{D}}_0(s')(\delta')\|\tilde{\mathfrak{D}}_0(t)(\delta'))\mid(a',s',\delta')\in I'(s)(\delta)\}\cup$

$\bigcup\{b'\cdot(\tilde{\mathfrak{D}}_0(s)(\delta')\|\tilde{\mathfrak{D}}_0(t')(\delta'))\mid(b',t',\delta')\in I'(t)(\delta)\}$

$=$ [see remark (3) after definition 1.25]

$\bigcup\{a'\cdot(\tilde{\mathfrak{D}}_0(s')(\delta')\|\tilde{\mathfrak{D}}_0(t)(\delta))\mid(a',s',\delta')\in I'(s)(\delta)\}\cup$

$\bigcup\{b'\cdot(\tilde{\mathfrak{D}}_0(s)(\delta)\|\tilde{\mathfrak{D}}_0(t')(\delta'))\mid(b',t',\delta')\in I'(t)(\delta)\}$

$=$ [definition $\perp\!\!\!\perp$ (see remark 1.17(2))]

$((\bigcup\{a'\cdot\tilde{\mathfrak{D}}_0(s')(\delta')\mid(a',s',\delta')\in I'(s)(\delta)\})\perp\!\!\!\perp\tilde{\mathfrak{D}}_0(t)(\delta))\ \cup$

$((\bigcup\{b'\cdot\tilde{\mathfrak{D}}_0(t')(\delta')\mid(b',t',\delta')\in I'(t)(\delta)\})\perp\!\!\!\perp\tilde{\mathfrak{D}}_0(s)(\delta))$

$=$ [definition Φ_0']

$(\Phi_0'(\tilde{\mathfrak{D}}_0)(s)(\delta)\perp\!\!\!\perp\tilde{\mathfrak{D}}_0(t)(\delta))\ \cup$

$(\Phi_0'(\tilde{\mathfrak{D}}_0)(t)(\delta)\perp\!\!\!\perp\tilde{\mathfrak{D}}_0(s)(\delta))$

$=$ [we have $\Xi(s,\delta)$ and $\Xi(t,\delta)$]

$(\tilde{\mathfrak{D}}_0(s)(\delta)\perp\!\!\!\perp\tilde{\mathfrak{D}}_0(t)(\delta))\ \cup$

$(\tilde{\mathfrak{D}}_0(t)(\delta)\perp\!\!\!\perp\tilde{\mathfrak{D}}_0(s)(\delta))$

$=\tilde{\mathfrak{D}}_0(s)(\delta)\|\tilde{\mathfrak{D}}_0(t)(\delta)$

$=\tilde{\mathfrak{D}}_0(s\|t)(\delta)$.

This proves $\Xi(s\|t,\delta)$. The case $\Xi(s\cup t,\delta)$ is simple.

(3) Let $s\in L_0^\delta$ and suppose $\{s\}\times\Delta\subseteq\Xi$. We show: $\Xi(\mu x[s],\delta)$. Assume (without loss of generality) that $x\notin dom(\delta)$. Then

$\Phi_0'(\tilde{\mathfrak{D}}_0)(\mu x[s])(\delta)$ $=$ [definition Φ_0' and $I'(\mu x[s])(\delta)$; let $\delta'=\delta\{\mu x[s]/x\}$]

$\bigcup\{a'\cdot\tilde{\mathfrak{D}}_0(s')(\delta')\mid(a',s',\delta')\in I'(s)(\delta')\}$

$=\Phi_0'(\tilde{\mathfrak{D}}_0)(s)(\delta')$

$=$ [we have $\Xi(s,\delta')$]

$\tilde{\mathfrak{D}}_0(s)(\delta')$

$= \mathcal{D}_0[\![s]\!](\tilde{\delta}')$

$=$ [definition $\tilde{\delta}'$]

$\mathcal{D}_0[\![s]\!](\tilde{\delta}\{\mathcal{D}_0[\![\mu x[s]]\!](\tilde{\delta})/x\})$

$=$ [definition $\mathcal{D}_0$]

$\mathcal{D}_0[\![\mu x[s]]\!](\tilde{\delta})$

$= \tilde{\mathcal{D}}_0(\mu x[s])(\delta)$

This proves $\Xi(\mu x[s],\delta)$.

(4) Let $x \in Stmv$, suppose $\Xi(\delta(x),\delta)$. Now

$\Phi_0'(\tilde{\mathcal{D}}_0)(x)(\delta) =$ [definition Φ_0' and $I'(x)(\delta)$]

$\Phi_0'(\tilde{\mathcal{D}}_0)(\delta(x))(\delta)$

$=$ [because $\Xi(\delta(x),\delta)$]

$\tilde{\mathcal{D}}_0(\delta(x))(\delta)$

$= \mathcal{D}_0[\![\delta(x)]\!](\tilde{\delta})$

$=$ [definition $\tilde{\delta}$]

$\tilde{\delta}(x)$

$= \mathcal{D}_0[\![x]\!](\delta)$

$= \tilde{\mathcal{D}}_0(x)(\delta)$.

Thus $\Xi(x,\delta)$.

The induction principle now implies: $\Xi = L_0 \times \Delta$.

□

As an immediate consequence of this theorem, we have

COROLLARY 1.34 ($\mathcal{O}_0' = \tilde{\mathcal{D}}_0$): $\forall s \in L_0\ \forall \delta \in \Delta\ [\mathcal{O}_0'[\![s]\!](\delta) = \mathcal{D}_0[\![s]\!](\tilde{\delta})]$.

Now combining corollaries 1.31 and 1.34 yields the main theorem of this section.

THEOREM 1.35 ($\mathcal{O}_0^{<>} = \tilde{\mathcal{D}}_0$): $\forall s \in L_0\ \forall \delta \in \Delta\ [\mathcal{O}_0[\![s<\delta>]\!] = \mathcal{D}_0[\![s]\!](\tilde{\delta})]$.

COROLLARY 1.36: *For all* $s \in L_0^l$, *and arbitrary* $\gamma \in \Gamma$: $\mathcal{O}_0[\![s]\!] = \mathcal{D}_0[\![s]\!](\gamma)$.

1.5 *Summary of section* 1

It may be useful to give a short overview of this section because we shall follow the same approach of proving semantic equivalence in the next sections. We have defined an operational semantics $\mathcal{O}_0$ for L_0 as the fixed point of Φ_0, and a denotational semantics $\mathcal{D}_0$ as the fixed point of Ψ_0. We have related $\mathcal{O}_0$ and $\mathcal{D}_0$ via an intermediate semantic function $\mathcal{O}_0'$, defined as the fixed point of Φ_0'. To be more precise, we have related Φ_0, Ψ_0, and Φ_0' using mappings $<>$ and $\sim$, for which we have proved some properties, schematically represented by the following diagram:

$$
\begin{array}{ccc}
L_0^{\langle\rangle}\to P_0 & \overset{\Phi_0}{\to} & L_0^{\langle\rangle}\to P_0 \\
\langle\rangle\downarrow & * & \downarrow\langle\rangle \\
L_0\to\Delta\to P_0 & \overset{\Phi_0'}{\to} & L_0\to\Delta\to P_0 \\
\sim\uparrow & *_{fix} & \uparrow\sim \\
L_0\to\Gamma\to P_0 & \overset{\Psi_0}{\to} & L_0\to\Gamma\to P_0
\end{array}
$$

The * in the upper rectangle indicates that it commutes, the symbol $*_{fix}$ in the lower rectangle indicates that it commutes only for the fixed point of Ψ_0 (that is, $\mathcal{D}_0$). Please note that * has been formulated as theorem 1.30, and $*_{fix}$ as theorem 1.33. The main result of section 1 (theorem 1.35) follows from this diagram, because * implies: $\Theta_0^{\langle\rangle} = \Theta_0'$ and $*_{fix}$ implies: $\Theta_0' = \tilde{\mathcal{D}}_0$.

REMARK
The lower rectangle does *not* commute for arbitrary $F\in L_0\to\Gamma\to P_0$. As an example take $F=\lambda s\cdot\lambda\gamma\cdot\{\epsilon\}$. Then, for given $a,b\in A$ and $\delta\in\Delta$:

$$
\begin{aligned}
\tilde{\Psi_0}(F)(a;b)(\delta) &= \Psi_0(F)(a;b)(\tilde{\delta}^{\Psi_0(F)}) \\
&= \Psi_0(F)(a)(\tilde{\delta}^{\Psi_0(F)})\,\tilde{;}\,\Psi_0(F)(b)(\tilde{\delta}^{\Psi_0(F)}) \\
&= \{a\}\tilde{;}\{b\} \\
&= \{ab\},
\end{aligned}
$$

whereas

$$
\begin{aligned}
\Phi_0'(\tilde{F})(a;b)(\delta) &= \{a\cdot\tilde{F}(b)(\delta)\} \\
&= \{a\cdot F(b)(\tilde{\delta}^F)\} \\
&= \{a\}.
\end{aligned}
$$

2. A LANGUAGE WITH COMMUNICATION AND GLOBAL NONDETERMINISM (L_1)

2.1 *Syntax*

For L_1 we introduce some structure to the (possibly infinite) alphabet A of elementary actions. Let $C\subseteq A$ be a subset of so-called *communications*. From now on let c range over C and a,b over A. Similarly to CCS [Mil] or CSP [Ho] we stipulate a bijection $^-:C\to C$ with $^-\circ^- = id_C$. It yields for every $c\in C$ a *matching* communication $^-(c)$, which will be denoted by $\bar{c}$. In $A\setminus C$ we have a special element τ denoting a successful communication. Let *Stmv*, with typical elements $x,y,\ldots$, be again the set of statement variables.

DEFINITION 2.1 (Syntax for L_1)
The set L_1, with typical elements $s,t,\ldots$, is given by

$$s ::= a \mid s_1;s_2 \mid s_1+s_2 \mid s_1\|s_2 \mid x \mid \mu x[t]$$

where $t\in L_1^x$, which is defined below. Please note that $a\in A\supseteq C$.

DEFINITION 2.2 (Syntax for L_1^x)
The set L_1^x of statements which are guarded for x is given by

$$
\begin{aligned}
t ::= \; & a \\
| \; & t;s, \text{ for } s \in L_1 \\
| \; & t_1 + t_2 \;|\; t_1 \| t_2 \\
| \; & y, \text{ for } y \neq x \\
| \; & \mu x[t] \\
| \; & \mu y[t'], \text{ for } y \neq x,\ t' \in L_1^x \cap L_1^y
\end{aligned}
$$

DEFINITION 2.3 (Syntax for L_1^g)
The set L_1^g of statements which are guarded for all $x \in Stmv$ is defined by

$$t ::= a \mid t;s \mid t_1 + t_2 \mid t_1 \| t_2 \mid \mu x[t],$$

where $s \in L_1$.

REMARK
We extend L_1, L_1^x, and L_1^g with the empty statement E (see the remark following definition 1.2).

The definitions of $FV(s)$ (free variables of s) and of (syntactically) closed statements are as in section 1. The language L_1 differs from L_0 in two respects. First, the presence of communication actions entails a more sophisticated interpretation of $s_1 \| s_2$. Secondly, the operators of global nondeterminism $s_1 + s_2$ and of local nondeterminism $s_1 \cup s_2$ of L_0 are differently interpreted. For an extensive discussion of L_1 we refer the reader to [BKMOZ] (where, for obvious reasons, it is called L_2). After we have defined an operational semantics for L_1, we shall briefly discuss the intuitive meaning of L_1.

2.2 *Operational semantics*

DEFINITION 2.4 (Semantic universe P_1)
Let, as in definition 1.7, the set A^∞ be defined as $A^\infty = A^* \cup A^\omega$. We extend this set by allowing as the last element of a finite sequence a special element ∂, which will be used to denote *deadlock*:

$$A_\partial^\infty = A^* \cup A^* \cdot \{\partial\} \cup A^\omega.$$

Now we define a complete metric space P_1, with typical elements $p, q, \ldots$, as

$$P_1 = \mathcal{P}_{nc}(A_\partial^\infty),$$

the set of all non-empty, compact subsets of A_∂^∞. As a metric on P_1 we take $(d_{A_\partial^\infty})_H$ (see A.6(d)). We shall use P_1 as the semantic universe for the operational semantics of L_1, which will again (as for L_0) be based on a transition relation:

DEFINITION 2.5 (Transition relation for L_1^g)
We define a transition relation

$$\rightarrow \subseteq L_1^g \times A \times L_1$$

as the smallest relation satisfying

(i) $a \xrightarrow{a} E$, for $a \in A$. (Please note that it is also possible that $a \in C$!)

(ii) for all $a \in A$, $s, t \in L_1^g$ and $s', \bar{s} \in L_1$: if $s' \neq E$, then:

$$s \xrightarrow{a} s' \Rightarrow (s;\bar{s} \xrightarrow{a} s';\bar{s}$$

$$\wedge\ s + t \xrightarrow{a} s' \ \wedge\ t + s \xrightarrow{a} s'$$

$$\wedge\ s\|t \xrightarrow{a} s'\|t\ \wedge\ t\|s \xrightarrow{a} t\|s'$$

$$\wedge\ \mu x[s] \xrightarrow{a} s'[\mu x[s]/x]);$$

and if $s' = E$, then:

$$s \xrightarrow{a} E \Rightarrow (s;\bar{s} \xrightarrow{a} \bar{s}$$

$$\wedge\ s+t \xrightarrow{a} E\ \wedge\ t+s \xrightarrow{a} E$$

$$\wedge\ s\|t \xrightarrow{a} t\ \wedge\ t\|s \xrightarrow{a} t$$

$$\wedge\ \mu x[s] \xrightarrow{a} E).$$

(iii) for all $c \in C$, $s,t \in L_1^q$, $s',t' \in L_1$: if $s' \neq E \neq t'$, then:

$$(s \xrightarrow{c} s'\ \wedge\ t \xrightarrow{\bar{c}} t') \Rightarrow s\|t \xrightarrow{\tau} s'\|t',$$

and if $s' = E$, then:

$$(s \xrightarrow{c} E\ \wedge\ t \xrightarrow{\bar{c}} t') \Rightarrow s\|t \xrightarrow{\tau} t'.$$

DEFINITION 2.6 (Φ_1)
Let $\Phi_1 : (L_1^q \to P_1) \to (L_1^q \to P_1)$ be given by

$$\Phi_1(F)(s) = \begin{cases} \{\epsilon\} & \text{if } s = E \\ \{\partial\} & \text{if } \{a \mid \exists s'[s \xrightarrow{a} s'] \wedge a \notin C\} = \varnothing \\ \bigcup \{a \cdot F(s') \mid s \xrightarrow{a} s' \wedge a \notin C\} & \text{otherwise,} \end{cases}$$

for $F \in L_1^q \to P_1$ and $s \in L_1^q$.

DEFINITION 2.7: $\theta_1 =$ Fixed Point(Φ_1)

EXAMPLES
The following examples illustrate the intended meaning of L_1:

$$\theta_1[\![x]\!] = \{\partial\}$$

$$\theta_1[\![c\|\bar{c}]\!] = \{\tau\}$$

$$\theta_1[\![(a;c)\|(b;\bar{c})]\!] = \{ab\tau, ba\tau\}$$

$$\theta_1[\![(a;b) + (a;c)]\!] = \{ab, a\partial\}$$

$$\theta_1[\![a;(b+c)]\!] = \{ab\},$$

for $c \in C$, $a,b \in A \setminus C$.

Thus with global nondeterminacy + the statements $s_1 = (a;b)+(a;c)$ and $s_2 = a;(b+c)$ get different meanings under θ_1. This difference can be understood as follows: If s_1 performs the elementary action a, the remaining statement is either the elementary action b or the communication c. In case of c, a deadlock occurs since no matching communication is available. However, if s_2 performs a, the remaining statement is $b+c$, which cannot deadlock because the action b is possible. Thus

communications c create deadlock only if neither a matching communication $\overline{c}$ nor an alternative elementary action b is available.
We again characterize the operational semantics by defining for each statement s a set of pairs of which the first element denotes a possible first step of s.

DEFINITION 2.8 (Initial steps)
We define a function $I: L_1^{\delta} \to \mathcal{P}_{fin}(A \times L_1)$ by induction on L_1^{δ}.
(i) $I(E) = \emptyset$ and $I(a) = \{(a, E)\}$
(ii) Suppose $I(s) = \{(a_i, s_i)\}$, $I(t) = \{(b_j, t_j)\}$ for $s,t \in L_1^{\delta}$, $a_i, b_j \in A$, and $s_i, t_j \in L_1$. (The variables i and j range over some finite sets of indices, which we have omitted.) Then

$$I(s;\overline{s}) = \{(a_i, s_i;\overline{s})\} \text{ (for } \overline{s} \in L_1)$$
$$I(s+t) = I(s) \cup I(t)$$
$$I(s \| t) = \{(a_i, s_i \| t)\} \cup \{(b_j, s \| t_j)\} \cup \{(\tau, s_i \| t_j) \mid a_i = \overline{b}_j\}$$
$$I(\mu x[s]) = \{(a_i, s_i[\mu x[s]/x])\}.$$

LEMMA 2.9: $\forall a \in A \ \forall s \in L_1^{\delta} \ \forall s' \in L_1 \ [s \xrightarrow{a} s' \Leftrightarrow (a,s') \in I(s)]$

COROLLARY 2.10: *For* $F \in L_1^{cl} \to P_1$ *and* $s \in L_1^{cl}$, *such that* $\{a \mid \exists s'[s \xrightarrow{a} s']\} \neq \emptyset$, *we have:*

$$\Phi_1(F)(s) = \bigcup \{a \cdot F(s') \mid (a,s') \in I(s) \wedge a \notin C\}.$$

2.3 *Denotational semantics*

We follow [BKMOZ] in introducing a *branching* time semantics for L_1. First we have to define a suitable semantic universe. It is obtained as a solution of the following *domain equation*:

$$\overline{P} \cong \{p_0\} \cup \mathcal{P}_{co}(A \times \overline{P}). \qquad (*)$$

Such a solution we call a *domain*, and its elements are called *processes*. We can read the equation as follows: a process $p \in \overline{P}$ is either p_0, the so-called *nil* process indicating termination, or it is a (compact) set X of pairs $<a,q>$, where a is the first action taken and q is the *resumption*, describing the rest of p's actions. If X is the empty set, it indicates deadlock (as does ∂ in the operational semantics). For reasons of cardinality (*) has no solution when we take *all* subsets, rather than all *compact* subsets of $A \times \overline{P}$. Moreover, we should be more precise about the metrics involved. We should have written (*) like this:

DEFINITION 2.11 (Semantic universe $\overline{P}_1$)
Let $(\overline{P}_1, d)$ be a complete metric space satisfying the following reflexive domain equation:

$$\overline{P} \cong \{p_0\} \,\overline{\cup}\, \mathcal{P}_{co}(A \times id_{1/2}(\overline{P})),$$

where, for any positive real number c, id_c maps a metric space (M,d) onto (M,d') with $d'(x,y) = c \cdot d(x,y)$, and $\overline{\cup}$ denotes the *disjoint* union (see definition A.6). (For a formal definition of the metric on $\overline{P}$ we refer the reader to the appendix.) Typical elements of $\overline{P}_1$ are p and q, and are called *processes*.

We shall not go into the details of solving this equation. In [BZ] it was first described how to solve this type of equations in a metric setting. In [AR] this approach is reformulated and extended in a category-theoretic setting.

As in definition 1.16 we define a number of operators on $\bar{P}_1$.

DEFINITION 2.12 (Semantic operators)
The operators $\tilde{;}$, $\tilde{+}$, $\tilde{\|}$: $\bar{P}_1 \times \bar{P}_1 \to \bar{P}_1$ are defined as follows. Let $p,q \in \bar{P}_1$, then:

$$(i) \quad p\tilde{;}q = \begin{cases} q & \text{if } p = p_0 \\ \{<a, p'\tilde{;} q> \mid <a,p'> \in p\} & \text{otherwise} \end{cases}$$

$$(ii) \quad p\tilde{+}q = \begin{cases} p & \text{if } q = p_0 \\ q & \text{if } p = p_0 \\ p \cup q & \text{otherwise} \end{cases}$$

$$(iii) \quad p\tilde{\|}q = \begin{cases} p & \text{if } q = p_0 \\ q & \text{if } p = p_0 \\ \{<a, p'\tilde{\|}q \mid <a, p'> \in p\} \cup & \\ \{<a, p\tilde{\|}q' \mid <a, q'> \in q\} \cup & \\ \{<\tau, p'\tilde{\|}q'> \mid <c,p'> \in p \wedge <\bar{c},q'> \in q\} & \text{otherwise.} \end{cases}$$

(We often write op rather than $\tilde{o}p$ if no confusion is possible.) For a justification of these definitions see remark 1.17.

DEFINITION 2.13 (Semantic environments)
We use Γ to denote the set of semantic environments (as in definition 1.19), with typical elements γ, given by

$$\Gamma = Stmv \to^{fin} \bar{P}_1.$$

DEFINITION 2.14 ($\Psi_1, \mathcal{D}_1$)
We define the denotational semantics $\mathcal{D}_1$ of L_1 as

$$\mathcal{D}_1 = \text{Fixed Point}(\Psi_1),$$

where $\Psi_1 : L_1 \to \Gamma \to \bar{P}_1$ is defined exactly as Ψ_0 in definition 1.20 but for the following two clauses:

$$\Psi_1(F)(a)(\gamma) = \{<a,p_0>\}$$

$$\Psi_1(F)(E)(\gamma) = p_0.$$

We realize that it must be difficult for the reader who sees this type of denotational semantics for the first time to understand and appreciate it. Nevertheless, we consider it for our purposes preferable to refer the reader to [BKMOZ], where he can find an extensive explanation. In this paper, we want to stress the technique of proving semantic equivalences, with which we now proceed.

2.4 *Semantic equivalence of* $\mathcal{O}_1$ and $\mathcal{D}_1$

It is quite obvious that the result of the previous section, as formulated in corollary 1.36, namely that

$$\forall s \in L_0^d \ \forall \gamma \in \Gamma \ [\mathcal{O}_0[\![s]\!] = \mathcal{D}_0[\![s]\!](\gamma)],$$

does not hold for the semantic functions $\mathcal{O}_1$ and $\mathcal{D}_1$. The semantic universe P_1 of $\mathcal{O}_1$ is a set of sets of streams, whereas $\bar{P}_1$, the semantic universe for $\mathcal{D}_1$, is a set of tree-like, branching processes. Thus, when comparing the types of $\mathcal{O}_1 : L_1 \to P_1$ and $\mathcal{D}_1 : L_1 \to \Gamma \to \bar{P}_1$, we observe that besides the fact that $\mathcal{D}_1$ takes a statement as an argument as well as an environment, which $\mathcal{O}_1$ does not (as is the case with $\mathcal{D}_0$ and $\mathcal{O}_0$), there is a second difference between $\mathcal{O}_1$ and $\mathcal{D}_1$. That is, they have different co-domains:

$P_1 \neq \overline{P}_1$ (which is *not* the case in the previous section). The strategy we shall follow to relate Θ_1 and $\mathcal{D}_1$ is to define functions

$$\Theta_1' : L_1 \to \Delta \to P_1$$

(where Δ will again be a set of syntactic environments) and

$$\mathcal{D}_1' : L_1 \to \Delta \to \overline{P}_1,$$

and then relate Θ_1 and Θ_1' (similarly as with Θ_0 and Θ_0'), next $\mathcal{D}_1'$ and $\mathcal{D}_1$ (similarly as with Θ_0' and $\mathcal{D}_0$), and finally compare Θ_1' and $\mathcal{D}_1'$ by using a suitable abstraction operator $\alpha : \overline{P}_1 \to P_1$. Like we did in the previous section we define Θ_1' (and $\mathcal{D}_1'$) as fixed point of a contraction.
We start with the comparison of Θ_1 and Θ_1'.

DEFINITION 2.15 (Syntactic environments)
The set Δ of syntactic environments, with typical elements δ, is given by

$$\Delta = \{\delta \mid \delta \in (Stmv \to^{\text{fin}} L_1) \wedge (\delta \text{ is normal})\}.$$

(For the notion of *normal* see definition 1.23.)

We formulate an induction principle for $L_1 \times \Delta$, as in 1.24.

THEOREM 2.16 (Induction principle for $L_1 \times \Delta$)
Let $\Xi \subseteq L_1 \times \Delta$. *If*

(1) $A \times \Delta \subseteq \Xi$

(2) $\{s,t\} \times \Delta \subseteq \Xi \Rightarrow \{s;\overline{s},\ s+t,\ s \| t\} \times \Delta \subseteq \Xi$, for $s,t,\overline{s} \in L_1$

(3) $\{s\} \times \Delta \subseteq \Xi \Rightarrow \{\mu x[s]\} \times \Delta \subseteq \Xi$, for $s \in L_0^x$

(4) $(\delta(x),\delta) \in \Xi \Rightarrow (x,\delta) \in \Xi$, for $x \in Stmv$, and $\delta \in \Delta$

then:

$$\Xi = L_1 \times \Delta$$

PROOF: See theorem 1.24.

DEFINITION 2.17 (Initial steps with syntactic environments)
As in definition 1.25 we use the induction principle to define a function

$$I' : L_1 \to \Delta \to \mathcal{P}_{\text{fin}}(A \times L_1 \times \Delta).$$

(1) $I'(E)(\delta) = \varnothing$, and $I'(a)(\delta) = \{(a,E,\delta)\}$ for all $a \in A$, $\delta \in \Delta$.
(2) Suppose $I'(s) = \lambda\delta.\{(a_i,s_i,\delta_i)\}$ and $I'(t) = \lambda\delta.\{(b_j,t_j,\delta_j)\}$ for $s,t \in L_1$, $a_i,b_j \in A$, and $\delta_i,\delta_j \in \Delta$. Then:

$$I'(s;\overline{s})(\delta) = \{(a_i,\ s_i;\overline{s},\ \delta_i)\} \text{ (for all } \overline{s} \in L_1)$$

$$I'(s+t)(\delta) = I'(s)(\delta) \cup I'(t)(\delta)$$

$$I'(s \| t)(\delta) = \{(a_i,\ s_i \| t,\ \delta_i)\} \cup \{(b_j,\ s \| t_j,\ \delta_j)\} \cup \{(\tau,\ s_i \| t_j,\ \delta_i \cup \delta_j) \mid \overline{a}_i = b_j\}$$

(3), (4): as in definition 1.25.

REMARK
In the clause for $s \| t$ in the above definition we take the union of two environments, δ_i and δ_j. This we can always do, if we impose the restriction upon all δ_i's and δ_j's that:

$$\text{if } \overline{a}_i = b_j, \text{ then } (dom(\delta_i) \setminus dom(\delta)) \cap (dom(\delta_j) \setminus dom(\delta)) = \varnothing.$$

If this condition is not satisfied (and in general it is not) a suitable renaming of variables should be applied. An example of a statement for which this should happen is: $\mu x[c;x]\|\mu x[\overline{c};x]$.

DEFINITION 2.18 (Φ_1')
We define $\Phi_1':(L_1\to\Delta\to P_1)\to(L_1\to\Delta\to P_1)$ by

$$\Phi_1'(F)(s)(\delta) = \begin{cases} \{\epsilon\} & \text{if } s = E \\ \{\partial\} & \text{if } \{(a,s',\delta')\in I'(s)(\delta)\,|\,a\notin C\} = \varnothing \\ \bigcup\{a\cdot F(s')(\delta')|\ & (a,s',\delta')\in I'(s)(\delta)\wedge a\notin C\} \text{ otherwise} \end{cases}$$

for $F\in L_1\to\Delta\to P_1$, $s\in L_1$, and $\delta\in\Delta$.

DEFINITION 2.19: Θ_1' = Fixed Point(Φ_1')

THEOREM 2.20 *(Relating I and I')*

$$\forall s\in L_1\ \forall\delta\in\Delta\ [I'(s)(\delta) = \{(a_i,s_i,\delta_i)\} \Leftrightarrow I(s<\delta>) = \{(a_i,s_i<\delta_i>)\}]$$

PROOF: See theorem 1.28.

DEFINITION 2.21: We define $<>:(L_1^{cl}\to P_1)\to(L_1\to\Delta\to P_1)$ by

$$<>F = F^{<>}$$
$$= \lambda s\in L_1\cdot\lambda\delta\in\Delta\cdot F(s<\delta>)$$

for $F\in L_1^{cl}\to P_1$.

THEOREM 2.22 (Relating Φ_1 and Φ_1'): $\forall F\in L_1^{cl}\to P_1\ [\Phi_1'(F^{<>}) = (\Phi_1(F))^{<>}]$

PROOF: See theorem 1.30.

COROLLARY 2.23 ($\Theta_1' = \Theta_1^{<>}$): $\forall s\in L_1\ \forall\delta\in\Delta\ [\Theta_1'[\![s]\!](\delta) = \Theta_1[\![s<\delta>]\!]]$

Next we define $\mathfrak{D}_1':L_1\to\Delta\to\overline{P}_1$ as the fixed point of the contraction below and compare $\mathfrak{D}_1$ and $\mathfrak{D}_1'$.

DEFINITION 2.24 (Ψ_1')
We define $\Psi_1':(L_1\to\Delta\to\overline{P}_1)\to(L_1\to\Delta\to\overline{P}_1)$ by

$$\Psi_1'(F)(s)(\delta) = \begin{cases} \{\epsilon\} & \text{if } s = E \\ \{<a,F(s')(\delta')>\,|\,(a,s',\delta')\in I'(s)(\delta)\} & \text{otherwise,} \end{cases}$$

for $F\in L_1\to\Delta\to\overline{P}_1$, $s\in L_1$, and $\delta\in\Delta$.

DEFINITION 2.25: $\mathfrak{D}_1'$ = Fixed Point(Ψ_1')

REMARK
As Θ_1' also $\mathfrak{D}_1'$ takes *syntactic* environments as arguments. Their co-domains, however, are different: $P_1\neq\overline{P}_1$. One could call $\mathfrak{D}_1'$ a branching variant of Θ_1'. Another difference is that $\Theta_1'(c)(\delta)=\{\partial\}$, whereas $\mathfrak{D}_1'(c)(\delta)=\{<c,p_0>\}$, for $c\in C$ and $\delta\in\Delta$.

In order to relate $\mathfrak{D}_1':L_1\to\Delta\to\overline{P}_1$ and $\mathfrak{D}_1:L_1\to\Gamma\to\overline{P}_1$ we use the following

DEFINITION 2.26

Let $\sim:(L_1 \to \Gamma \to \overline{P}_1) \to (L_1 \to \Delta \to \overline{P}_1)$ be given by

$$\sim(F) = \tilde{F}$$
$$= \lambda s \in L_1 \cdot \lambda \delta \in \Delta \cdot F(s)(\tilde{\delta}^F)$$

for $F \in L_1 \to \Gamma \to \overline{P}_1$, where $\tilde{\delta}^F$ is defined as $\tilde{\delta}^F = \lambda x \in dom(\delta) \cdot F(\delta(x))(\tilde{\delta}^F)$. (For a justification of the definition of $\tilde{\delta}^F$ see remark (1) following definition 1.31.)

THEOREM 2.27: $\Phi_1'(\tilde{\mathcal{D}}_1) = \tilde{\mathcal{D}}_1$

PROOF: This theorem can be proved in essentially the same way as theorem 1.33.

COROLLARY 2.28: $\mathcal{D}_1' = \tilde{\mathcal{D}}_1$

Finally we provide the only missing link in the chain that is to connect Θ_1 with $\mathcal{D}_1$: the comparison of

$$\Theta_1' : L_1 \to \Delta \to P_1 \text{ and } \mathcal{D}_1' : L_1 \to \Delta \to \overline{P}_1.$$

We relate their different semantic universes P_1 and $\overline{P}_1$ in the following

DEFINITION 2.29 (Abstraction operator α)
We define an abstraction operator $\alpha : \overline{P}_1 \to P_1$ by:

$$\alpha = streams \circ restr,$$

where *restr* (for *restriction*) and *streams* are recursively defined:

$$(i) \quad restr : P_1 \to P_1$$

$$p \mapsto \begin{cases} p_0 & \text{if } p = p_0 \\ \{<a, restr(p')> \mid <a,p'> \in p \wedge a \notin C\} & \text{otherwise} \end{cases}$$

$$(ii) \quad streams : \overline{P}_1 \to P_1$$

$$p \mapsto \begin{cases} \{\epsilon\} & \text{if } p = p_0 \\ \{\partial\} & \text{if } p = \varnothing \\ \bigcup \{a \cdot streams(p') \mid <a,p'> \in p\} & \text{otherwise.} \end{cases}$$

REMARKS

(1) Since the definition of *restr* and *streams* is recursive, we have to verify that it is well formed. It suffices to note that these functions can be defined as fixed points of contracting functions (cf. remark 1.17).

(2) The abstraction operator α transforms a (branching) process $p \in \overline{P}_1$ into an element $\alpha(p) \in P_1$ in two steps. First it cuts off all branches (all subprocesses) of p_1 that are labeled with an element of C: these c's can be regarded as failed (individual) attempts at communication. This is what *restr* does. Then *streams* takes all paths (streams) of the result of *restr* (p), putting a ∂ symbol (denoting deadlock) at the end of all paths ending in the empty process. This can be understood as follows: When a path in *restr*(p) ends in the empty process this means that the operation *restr* has cut off everything at the end of the corresponding path in p. By definition of *restr* only c's could have been present. Thus this path in p should be interpreted as indicating a situation in which only individual communication steps can be taken. Operationally, we consider this to be a case of deadlock. Therefore, we replace this empty process by ∂. This is what *streams* does.

Now that we have defined a mapping $\alpha:\overline{P}_1 \to P_1$, we extend it in the following way.

DEFINITION 2.30
Let $\alpha:(L_1 \to \Delta \to \overline{P}_1) \to (L_1 \to \Delta \to P_1)$ be defined by

$$\begin{aligned} \alpha(F) &= F^\alpha \text{ (notation)} \\ &= \lambda s \in L_1 \cdot \lambda \delta \in \Delta \cdot \alpha(F(s)(\delta)) \end{aligned}$$

for $F \in L_1 \to \Delta \to \overline{P}_1$. (Please note that we use again the symbol α. We trust that no confusion will arise from this slight abuse of language.)

THEOREM 2.31 (Relating Ψ_1' and Φ_1'): $\forall F \in L_1 \to \Delta \to \overline{P}_1\ [\Phi_1'(F^\alpha) = (\Psi_1'(F))^\alpha]$

PROOF
Let $F \in L_1 \to \Delta \to \overline{P}_1$, let $s \in L_1$ and $\delta \in \Delta$ be such that $\{(a,s',\delta') \in I'(s)(\delta) \mid a \notin C\} \neq \varnothing$. Then:

$$\begin{aligned} \Phi_1'(F^\alpha)(s)(\delta) &= \bigcup \{a \cdot F^\alpha(s')(\delta') \mid (a,s',\delta') \in I'(s)(\delta) \wedge a \notin C\} \\ &= \bigcup \{a \cdot (\alpha(F(s')(\delta')) \mid (a,s',\delta') \in I'(s)(\delta) \wedge a \notin C\} \\ &= \mathit{streams}(\{<a, \mathit{restr}(F(s')(\delta'))> \mid (a,s',\delta') \in I'(s)(\delta) \wedge a \notin C\}) \\ &= \mathit{streams} \circ \mathit{restr}(\{<a, F(s')(\delta')> \mid (a,s,\delta') \in I'(s)(\delta)\}) \\ &= \alpha(\Psi_1'(F)(s)(\delta)) \\ &= (\Psi_1'(F))^\alpha(s)(\delta). \end{aligned}$$

If $s \in L_1$ and $\delta \in \Delta$ are such that $\{(a,s',\delta') \in I'(s)(\delta) \mid a \notin C\} = \varnothing$, then

$$\begin{aligned} \Phi_1'(F^\alpha)(s)(\delta) &= \{\partial\} \\ &= \mathit{streams}(\varnothing) \\ &= \mathit{streams} \circ \mathit{restr}(\{<a, F(s')(\delta')> \mid (a,s',\delta') \in I'(s)(\delta)\}) \\ &= (\Psi_1'(F))^\alpha(s)(\delta). \end{aligned}$$

COROLLARY 2.32 ($(\mathcal{D}_1')^\alpha = \Theta_1'$): $\forall s \in L_1\ \forall \delta \in \Delta\ [\alpha(\mathcal{D}_1'[\![s]\!](\delta) = \Theta_1'[\![s]\!](\delta)]$

Combining corollaries 2.23, 2.28 and 2.32, which state:

(2.23) $\Theta_1^{<>} = \Theta_1'$

(2.32) $\Theta_1' = (\mathcal{D}_1')^\alpha$

(2.28) $\mathcal{D}_1' = \tilde{\mathcal{D}}_1$,

now yields the main theorem of this section:

THEOREM 2.33 ($\Theta_1^{<>} = (\tilde{\mathcal{D}}_1)^\alpha$): $\forall s \in L_1\ \forall \delta \in \Delta\ [\Theta_1[\![s<\delta>]\!] = \alpha(\mathcal{D}_1[\![s]\!](\tilde{\delta}))]$

COROLLARY 2.34: For all $s \in L_1^d$ and arbitrary $\gamma \in \Gamma$: $\Theta_1[\![s]\!] = \alpha(\mathcal{D}_1[\![s]\!](\gamma))$.

2.5 *Summary of section* 2

We can again give a quick overview of the main theorems of this section by drawing a diagram as follows:

$$
\begin{array}{lcll}
 & \overset{\Phi_1}{} & & \\
L_1^{cl} \to P_1 & \to & L_1^{cl} \to P_1 & \\
<> \downarrow & * & \downarrow <> & \text{(theorem 2.22)} \\
 & \overset{\Phi_1'}{} & & \\
L_1 \to \Delta \to P_1 & \to & L_1 \to \Delta \to P_1 & \\
\alpha \uparrow & * & \uparrow \alpha & \text{(theorem 2.31)} \\
 & \overset{\Psi_1'}{} & & \\
L_1 \to \Delta \to \bar{P}_1 & \to & L_1 \to \Delta \to \bar{P}_1 & \\
\sim \uparrow & *_{fix} & \uparrow \sim & \text{(theorem2.27)} \\
 & \overset{\Psi_1}{} & & \\
L_1 \to \Gamma \to \bar{P}_1 & \to & L_1 \to \Gamma \to \bar{P}_1 &
\end{array}
$$

where (as in subsection 1.5) * indicates commutativity and $*_{fix}$ indicates commutativity with respect to the fixed point of Ψ_1 (that is, $\mathfrak{D}_1$). Please note that if we could identify P_1 and $\bar{P}_1$, we could identify the second and the third horizontal lines of this diagram, leaving out the mapping α. This would yield a diagram of exactly the same shape as that of subsection 1.5. This is just a way of rephrasing what has already been said above: The only new thing about proving semantic equivalence for L_1, compared with L_0, is the presence of a difference between the semantic universes P_1 and $\bar{P}_1$ of $\mathfrak{O}_1$ and $\mathfrak{D}_1$, which made the introduction of α necessary. Theorems 2.22 and 2.27 are just (slightly) modified versions of theorems already present in section 1 (namely, theorems 1.30 and 1.33).

3. A NONUNIFORM LANGUAGE WITH VALUE PASSING (L_2)

We devote the third section of our paper to the discussion of semantic equivalence for a *nonuniform* language. Elementary actions are no longer uninterpreted but taken as either assignments or tests. Communication actions c and $\bar{c}$ are refined to actions $c?v$ and $c!e$ (with v variable and e an expression), and successful communication now involves two effects: both *synchronization* (as in the language L_1) and *value passing*: the (current) value of e is assigned to v. Thus, we have here the synchronous handshaking variety of message passing in the sense of CCS or CSP.

We shall introduce a language L_2 which embodies these features and present its operational and denotational semantics $\mathfrak{O}_2$ and $\mathfrak{D}_2$. Nonuniformity of L_2 calls for the notion of *state* in both semantic models: They now deliver sets of streams, or processes, over state transformations, not over uninterpreted actions as in the previous sections. The main goal of this section is to provide the reader with yet another example of a language to which the method for proving semantic equivalence, as developed in section 1 and 2, applies. Although L_2 will be in some sense more complex than L_1 and accordingly $\mathfrak{O}_2$ and $\mathfrak{D}_2$ more intricate than $\mathfrak{O}_1$ and $\mathfrak{D}_1$, the proof of the equivalence of operational and denotational semantics will essentially be the same. Because of this emphasis on proving semantic equivalence, we shall not give very much explanation when defining the semantics. For this we refer the reader again to [BKMOZ], which we (roughly) follow in our definition of $\mathfrak{O}_2$ and $\mathfrak{D}_2$. Nor shall we give any proofs, because all of them can be obtained by straightforwardly modifying a corresponding one from section 2.

3.1 *Syntax*

We now present the syntax of L_2. We use three new syntactic categories, viz.

- the set *Var*, with elements v,w, of *individual variables*
- the set *Exp*, with elements e, of *expressions*
- the set *Bexp*, with elements b, of *boolean expressions*.

We shall not specify a syntax for *Exp* and *Bexp*. We assume that (boolean) expressions are of an

elementary kind; in particular, they have no side effects and their evaluation always terminates. Statement variables $x,y,\ldots$ are as before, as are the communications $c \in C$. The latter now appear syntactically as part of value passing communication actions $c?v$ or $c!e$.

DEFINITION 3.1 (Syntax for L_2)

$$s ::= v:=e \mid b \mid c?v \mid c!e \mid s_1;s_2 \mid s_1 + s_2 \mid s_1 \| s_2 \mid x \mid \mu x[t]$$

where $t \in L_2^x$, defined in

DEFINITION 3.2 (Syntax for L_2^x)
The set L_2^x of statements which are guarded for x is given by

$$\begin{aligned} t ::= {} & v:=e \mid b \mid c?v \mid c!e \\ & \mid t;s, \text{ for } s \in L_2 \\ & \mid t_1 + t_2 \mid t_1 \| t_2 \\ & \mid y, \text{ for } y \neq x \\ & \mid \mu x[t] \\ & \mid \mu y[t'], \text{ for } y \neq x,\ t' \in L_2^x \cap L_2^y \end{aligned}$$

DEFINITION 3.3 (Syntax for L_2^δ)
The set L_2^δ of statements which are guarded for all $x \in Stmv$ is defined by

$$t ::= v:=e \mid b \mid c?v \mid c!e \mid t;s \mid t_1 + t_2 \mid t_1 \| t_2 \mid \mu x[t],$$

where $s \in L_2$.

REMARK: The sets L_2, L_2^x, and L_2^δ are extended with the empty statement E (cf. the remark preceding definition 1.3).

It will be useful to unite assignments $v:=e$, tests b and communications $c?v$ and $c!e$ into one set of *basic* steps.

DEFINITION 3.4 (Basic steps)
We define the set *Bsteps* of basic steps, with typical element a, by

$$BStep = Comm \cup Bexp \cup Asg,$$

where the set *Comm* of communications is defined by

$$Comm = \{c?v \mid c \in C, v \in Var\} \cup \{c!e \mid c \in C, e \in Exp\},$$

and the set *Asg*, of assignments, is defined by

$$Asg = \{v:=e \mid v \in Var, e \in Exp\}.$$

The sets *BSteps* and *Comm* can be regarded as the nonuniform equivalents of the sets A of atomic actions and C of communications of the previous section.

3.2 *Operational semantics*

DEFINITION 3.5 (Transition relation for L_2^δ)
We define $\rightarrow \subseteq L_2^\delta \times BStep \times L_2$ as the smallest relation satisfying

(i) $a \xrightarrow{a} E$, for all $a \in BStep$. (Please note that it is also possible that $a \in Comm$!)

(ii) for all $a \in BStep$, $s, t \in L_2^\delta$ and $s', \bar{s} \in L_2$: if $s' \neq E$, then:

$$s \xrightarrow{a} s' \Rightarrow (s;\bar{s} \xrightarrow{a} s';\bar{s}$$
$$\wedge\ s+t \xrightarrow{a} s' \wedge t+s \xrightarrow{a} s'$$
$$\wedge\ s\|t \xrightarrow{a} s'\|t \wedge t\|s \xrightarrow{a} t\|s'$$
$$\wedge\ \mu x[s] \xrightarrow{a} s'[\mu x[s]/x]);$$

and if $s' = E$, then:

$$s \xrightarrow{a} E \Rightarrow (s;\bar{s} \xrightarrow{a} \bar{s}$$
$$\wedge\ s+t \xrightarrow{a} E \wedge t+s \xrightarrow{a} E$$
$$\wedge\ s\|t \xrightarrow{a} t \wedge t\|s \xrightarrow{a} t$$
$$\wedge\ \mu x[s] \xrightarrow{a} E).$$

(iii) for all $s,t \in L_2^\delta$, $s',t' \in L_2$, and $c?v, c!e \in Comm$: if $s' \neq E \neq t'$, then:

$$(s \xrightarrow{c!e} s' \wedge t \xrightarrow{c?v} t') \Rightarrow (s\|t \xrightarrow{v:=e} s'\|t' \wedge t\|s \xrightarrow{v:=e} t'\|s'),$$

and if $s' = E$, then:

$$(s \xrightarrow{c!e} E \wedge t \xrightarrow{c?v} t') \Rightarrow (s\|t \xrightarrow{v:=e} t' \wedge t\|s \xrightarrow{v:=e} t').$$

For both operational and denotational models the notion of *state* is fundamental. Elements v,w in *Var* will have values in a set *Val*. A state is a function that maps variables to their (current) values. Accordingly, we define

DEFINITION 3.6 (States)
The set Σ of *states*, with typical element σ, is defined as

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

We shall also employ a special failure state ∂, with $\partial \notin \Sigma$, and define

$$\Sigma_\partial^\infty = \Sigma^* \cup \Sigma^* \cdot \{\partial\} \cup \Sigma^\omega.$$

Elements of Σ_∂^∞ will be denoted by finite or infinite tuples $<\sigma_1,\sigma_2, \ldots>$. The empty tuple will be denoted by ϵ. We shall write σ for $<\sigma>$. Concatenation is defined as usual.

For expressions $e \in Exp$ and $b \in BExp$ we postulate a simple semantic evaluation function, details of which we do not bother to provide. The values of e and b in state σ will be denoted simply by

$$[\![e]\!]\sigma\ (\in Val) \text{ and } [\![b]\!]\sigma\ (\in \{tt,ff\}).$$

DEFINITION 3.7 (Semantic universe P_2)
We define the semantic universe P_2 by

$$P_2 = \Sigma \to \mathcal{P}_{nc}(\Sigma_\partial^\infty),$$

where $\mathcal{P}_{nc}(\Sigma_\partial^\infty)$ is the set of all non-empty and compact subsets of Σ_∂^∞.

DEFINITION 3.8 (Φ_2)
Let $\Phi_2: (L_2^{cl} \to P_2) \to (L_2^{cl} \to P_2)$ be defined by

$$\Phi_2(F)(E) = \{\epsilon\};$$

if $\{a \mid \exists s'[s \xrightarrow{a} s'] \wedge (a \in Asg \vee (a \in BExp \wedge [\![a]\!]\sigma = tt))\} = \varnothing$, then

$$\Phi_2(F)(s) = \{\partial\};$$

otherwise

$$\Phi_2(F)(s) = \bigcup \{\sigma \cdot F(s')(\sigma) \mid s \xrightarrow{b} s' \wedge [\![b]\!]\sigma = tt\} \cup \bigcup \{\sigma_{v:=e} \cdot F(s')(\sigma_{v:=e}) \mid s \xrightarrow{v:=e} s'\},$$

for $F \in L_2^{cl} \to P_2$ and $s \in L_2$, and with

$$\sigma_{v:=e} = \sigma\{[\![e]\!]\sigma/v\}.$$

(The notation $\sigma_{v:=e}$ will also be used in the sequel.)

DEFINITION 3.9: $\mathcal{O}_2$ = Fixed Point(Φ_2)

EXAMPLES

$$\mathcal{O}_2[\![v:=0]\!] = \lambda\sigma \cdot \{<\sigma\{0/v\}>\}.$$

$$\mathcal{O}_2[\![v:=0 \| (v:=1;\ v:=v+1)]\!] = \lambda\sigma \cdot \{<\sigma\{0/v\}, \sigma\{1/v\}, \sigma\{2/v\}>, <\sigma\{1/v\}, \sigma\{0/v\}, \sigma\{1/v\}>, <\sigma\{1/v\}, \sigma\{2/v\}, \sigma\{0/v\}>\}$$

$$\mathcal{O}_2[\![v:=0;\ \mu x[v:=v+1;\ x]]\!] = \lambda\sigma \cdot \{<\sigma\{0/v\}, \sigma\{1/v\}, \sigma\{2/v\}, \ \ldots >\}$$

$$\mathcal{O}_2[\![v:=0; v<0]\!] = \lambda\sigma \cdot \{<\sigma\{0/v\}, \partial>\}$$

$$\mathcal{O}_2[\![c?v]\!] = \lambda\sigma \cdot \{<\partial>\}$$

$$\mathcal{O}_2[\![c?v \| c!3]\!] = \lambda\sigma \cdot \{<\sigma\{3/v\}>\}$$

We can again characterize the operational model using an initial step function.

DEFINITION 3.10 (Initial steps)
Let $I: L_2^{\S} \to \mathcal{P}_{fin}(BStep \times L_2)$ be defined by

(i) $I(E) = \varnothing$, $I(a) = \{(a,E)\}$, for $a \in BStep$

(ii) Suppose $I(s) = \{(a_i, s_i)\}$, $I(t) = \{(b_j, t_j)\}$ for $s,t \in L_2^{\S}$, $a_i, b_j \in BStep$, and $s_i, t_j \in L_2$. Then

$$I(s;\overline{s}) = \{(a_i, s_i;\overline{s})\}, \text{ for } \overline{s} \in L_2$$

$$I(s+t) = I(s) \cup I(t)$$

$$I(s \| t) = \{(a_i, s_i \| t)\} \cup \{(b_j, s \| t_j)\} \cup \{(v:=e,\ s_i \| t_j) \mid (a_i = c?v \wedge b_j = c!e) \vee (a_i = c!e \wedge b_j = c?v)\}$$

$$I(\mu x[s]) = \{(a_i, s_i[\mu x[s]/x])\}.$$

LEMMA 3.11: $\forall a \in BStep \forall s \in L_2^{\S}\ \forall s \in L_2\ [s \xrightarrow{a} s' \Leftrightarrow (a,s') \in I(s)]$

COROLLARY 3.12

For $F \in L_2^{cl} \to P_2$, $s \in L_2^{cl}$ *and* $\sigma \in \Sigma$ *with* $\{(a,s') \in I(s) \mid a \in Asg \vee (a \in BExp \wedge [\![a]\!]\sigma = tt)\} \neq \emptyset$:

$$\Phi_2(F)(s)(\sigma) = \bigcup \{\sigma \cdot F(s')(\sigma) \mid (b,s') \in I(s) \wedge [\![b]\!]\sigma = tt))\} \cup$$
$$\bigcup \{\sigma_{v:=e} \cdot F(s')(\sigma_{v:=e}) \mid (v:=e,\ s') \in I(s)\}.$$

3.3 *Denotational semantics*

As in section 2.3 we start with the definition of a suitable semantic universe. It will be a process domain that is obtained as a solution of the following domain equation:

$$\overline{P} \cong \{p_0\} \cup \mathcal{P}_{co}(SSteps \times \overline{P}),$$

where the set *SSteps* of *semantic steps*, with typical elements κ, is given by

$$\begin{aligned} SSteps = {} & (\Sigma \to \Sigma) \\ & \cup (\Sigma \to \{tt, ff\}) \\ & \cup (C \times Var) \\ & \cup (C \times (\Sigma \to Val)). \end{aligned}$$

We can read this equation as follows: a process $p \in \overline{P}$ is either p_0, the nil process, or it is a (compact) set X of semantic steps $\kappa \in SSteps$. Such a semantic step can have one out of four forms. First it can be a state transformation. These will be used to give a semantics to assignments. Then it can be a mapping from states to the set of truth values, corresponding with boolean expressions. Next, it can be a pair $<c,v>$, corresponding with an input statement $c?v$. And finally it can be a pair $<c,f>$, corresponding with an output statement $c!e$. Here, f is used to denote the value of e (that is, $[\![e]\!] \in \Sigma \to Val$).

As in section 2.3 we should be more precise about the metrics involved. We give a formal definition below and refer the reader to section 2.3 for further explanation and references.

DEFINITION 3.13 (Semantic universe $\overline{P}_2$)

Let $(\overline{P}_2, d)$ be a complete metric space such that it satisfies the following domain equation:

$$\overline{P} \cong \{p_0\} \overline{\cup} \mathcal{P}_{co}(SSteps \times id_{1/2}(\overline{P})),$$

with *SSteps* as above. Typical elements of $\overline{P}_2$ will be p and q.

DEFINITION 3.14 (Semantic operators)

The operators $\tilde{;}$, $\tilde{+}$, and $\tilde{\|}: \overline{P}_2 \times \overline{P}_2 \to \overline{P}_2$ are defined as follows. Let $p,q \in \overline{P}_2$, $\kappa \in SSteps$, $c \in C$, $v \in Var$, and $f \in \Sigma \to Val$. Then:

(i)

$$p \tilde{;} q = \begin{cases} q & \text{if } p = p_0 \\ \{<\kappa,\ p' \tilde{;} q> \mid <\kappa, p'> \in p\} & \text{if } p \neq p_0 \end{cases}$$

(ii)

$$p \tilde{+} q = \begin{cases} p & \text{if } q = p_0 \\ q & \text{if } p = p_0 \\ p \cup q & \text{otherwise} \end{cases}$$

(iii) If $p = p_0$, then $p \tilde{\|} q = q \tilde{\|} p = q$. If $p \neq p_0$ and $q \neq p_0$, then:

$$
\begin{aligned}
p \tilde{\|} q \;=\; & \{<\kappa, p' \| q> \mid <\kappa, p'> \in p\} \cup \\
& \{<\kappa, p \| q'> \mid <\kappa, q'> \in q\} \cup \\
& \{<\lambda\sigma\cdot\sigma\{f(\sigma)/v\}, p' \| q'> \mid (<<c,v>, p'> \in p \wedge <<c,f>, q'> \in q) \vee \\
& \qquad (<<c,f>, p'> \in p \wedge <<c,v>, q'> \in q)\}.
\end{aligned}
$$

For a justification of these self-referential definitions see remark 1.17.

DEFINITION 3.15 (Semantic environments): $\Gamma = Stmv \xrightarrow{fin} \overline{P}_2$ (typical elements are γ).

DEFINITION 3.16 $(\Psi_2, \mathfrak{D}_2)$
We define the denotational semantics $\mathfrak{D}_2$ of L_2 as

$$\mathfrak{D}_2 \;=\; \text{Fixed Point}(\Psi_2),$$

where $\Psi_2 : (L_2 \to \Gamma \to \overline{P}_2) \to (L_2 \to \Gamma \to \overline{P}_2)$ is given, for $F \in L_2 \to \Gamma \to \overline{P}_2$, by:

(*i*) $\Psi_2(F)(a)(\gamma) = \{<\kappa_a, p_0>\}$, and $\Psi_2(F)(E)(\gamma) = p_0$,

with

$$\kappa_a \;=\; \begin{cases} \lambda\sigma\cdot\sigma_{v:=e} & \text{if } a = v:=e \\ \lambda\sigma\cdot[\![a]\!]\sigma & \text{if } a \in BExp \\ <c, v> & \text{if } a = c?v \\ <c, \lambda\sigma\cdot[\![e]\!]\sigma> & \text{if } a = c!e. \end{cases}$$

(*ii*) $\Psi_2(F)(s \; op \; t)(\gamma) \;=\; \Psi_2(F)(s)(\gamma) \; \tilde{op} \; \Psi_2(F)(t)(\gamma)$ for $op = ;, +, \|$.

(*iii*) $\Psi_2(F)(\mu x[s])(\gamma) \;=\; \Psi_2(F)(s)(\gamma\{F(\mu x[s](\gamma)/x\})$.

Similarly to lemma 1.21 we have that Ψ_2 is contracting.

EXAMPLES

$$\mathfrak{D}_2[\![v:=0]\!](\gamma) \;=\; \{<\lambda\sigma\cdot\sigma\{0/v\}, p_0>\}$$

$$\mathfrak{D}_2[\![v:=1; v:=v+1]\!](\gamma) \;=\; \{<\lambda\sigma\cdot\sigma\{1/v\}, \{<\lambda\sigma'\cdot\sigma'\{\sigma'(v)+1/v\}, p_0>\}>\}$$

$$\begin{aligned}
\mathfrak{D}_2[\![c?v \,\|\, c!3]\!](\gamma) \;=\; & \{<<c,v>, \{<<c, \lambda\sigma\cdot 3>, p_0>\}>, \\
& \quad <<c, \lambda\sigma\cdot 3>, \{<<c,v>, p_0>\}>, \\
& \quad <\lambda\sigma\cdot\sigma\{3/v\}, p_0>\}
\end{aligned}$$

$\mathfrak{D}_2[\![v:=0;\ \mu x[v:=v+1;\ x]]\!] \;=\; \{<\lambda\sigma\cdot\sigma\{0/v\}, p>\}$, where $p \in \overline{P}_2$ satisfies

$p \;=\; \{<\lambda\sigma\cdot\sigma\{\sigma(v)+1/v\}, p>\}$.

3.4 ***Semantic equivalence of*** $\mathfrak{O}_2$ **and** $\mathfrak{D}_2$

The proof of the semantic equivalence of $\mathfrak{O}_2$ and $\mathfrak{D}_2$ is essentially the same as in the previous section. Therefore, we only give a brief outline of how to proceed, leaving out the details of some definitions, omitting all proofs, and stressing the (small) differences. We define

$$\mathfrak{O}_2' \;=\; \text{Fixed Point}(\Phi_2') \text{ and } \mathfrak{D}_2' \;=\; \text{Fixed Point}(\Psi_2')$$

with Φ_2' and Ψ_2' defined as follows. Let $\Phi_2' : (L_2 \to \Delta \to P_2) \to (L_2 \to \Delta \to P_2)$ be given by

$$\Phi_2'(F)(E)(\delta) \;=\; \{\epsilon\};$$

if $\{(a,s',\delta')\in I'(s)(\delta) \mid a\in Asg \vee (a\in BExp \wedge [\![a]\!]\sigma = tt)\} = \emptyset$, then

$$\Phi_2'(F)(s)(\delta) = \{\partial\};$$

otherwise

$$\Phi_2'(F)(s)(\delta) = \bigcup\{\sigma\cdot F(s')(\sigma)(\delta') \mid (b,s',\delta')\in I'(s)(\delta)\wedge [\![b]\!]\sigma = tt\} \cup$$
$$\bigcup\{\sigma_{v:=e}\cdot F(s')(\delta')(\sigma_{v:=e}) \mid (v:=e,s',\delta')\in I'(s)(\delta)\},$$

for $F\in L_2\to\Delta\to P_2$, $s\in L_2$ and $\delta\in\Delta$ (Δ and I' can be defined similarly to definitions 2.5 and 2.17). Let $\Psi_2':(L_2\to\Delta\to P_2)\to(L_2\to\Delta\to\overline{P}_2)$ be defined by

$$\Psi_2'(F)(s)(\delta) = \begin{cases} p_0 & \text{if } s = E \\ \{<\kappa_a, F(s')(\delta')> \mid (a,s',\delta')\in I'(s)(\delta)\} & \text{otherwise,} \end{cases}$$

(with κ_a as in definition 3.16) for $F\in L_2\to\Delta\to\overline{P}_2$, $s\in L_2$, and $\delta\in\Delta$.

The definitions of Φ_2' and Ψ_2' are somewhat more involved than their counterparts from section 2. What is different here is that a syntactic basic step does not literally coincide with the semantic step that represents its meaning. In the previous section we had elementary actions a and c both as syntactic and semantic entities. Here we have syntactic basic steps $v:=e$, b, $c!e$, and $c?v$, all of which are semantically represented in a different way.
Similarly to the definitions 2.21 and 2.26 we can define mappings

$$<>:(L_2^{cl}\to P_2)\to(L_2\to\Delta\to P_2) \quad \text{and}$$
$$\sim:(L_2\to\Gamma\to P_2)\to(L_2\to\Delta\to P_2),$$

and prove

$$\mathcal{O}_2' = \mathcal{O}_2^{<>} \text{ and } \mathcal{D}_2' = \tilde{\mathcal{D}}_2.$$

Finally, we can compare $\mathcal{O}_2'$ and $\mathcal{D}_2'$ by recursively defining a suitable abstraction operator $\alpha:\overline{P}_2\to P_2$ by

$$\alpha(p_0)(\sigma) = \{\epsilon\},$$

and, for $p\neq p_0$, by

$$\alpha(p)(\sigma) = \bigcup\{f(\sigma)\cdot\alpha(p')(f(\sigma)) \mid <f,p'>\in p \wedge f\in\Sigma\to\Sigma\} \cup$$
$$\bigcup\{\sigma\cdot\alpha(p')(\sigma) \mid <f,p'>\in p \wedge (f\in\Sigma\to\{ff,tt\}) \wedge f(\sigma) = tt\},$$

if $\{<f,p'> \mid <f,p'>\in p \wedge (f\in\Sigma\to\Sigma \vee (f\in\Sigma\to\{ff,tt\}\wedge f(\sigma)=tt))\}\neq\emptyset$, and by

$$\alpha(p)(\sigma) = \{\partial\}, \text{ otherwise.}$$

(For a justification of this self-referential definition see remark 1.17.) In $\alpha(p)(\sigma)$ all pairs $<\kappa,p'>\in p$ with $\kappa\in\Sigma\to\{tt,ff\}$ and $\kappa(\sigma)=ff$, or $\kappa\in C\times Var$, or $\kappa\in C\times(\Sigma\to Val)$, are neglected. This corresponds with the restriction operator of definition 2.29. A second effect of applying α is that it transforms a (branching) process $p\in\overline{P}_2$ into a function $\alpha(p)\in P_2=\Sigma\to\mathcal{P}_{nc}(A_\delta^\infty)$, which yields, when supplied with an argument σ, a set of streams (in a sense the *paths* of p). In this respect α is similar to the operator *streams* of definition 2.29. Applying α has yet another effect. If $f\in\Sigma\to\Sigma$ and $<f,p'>\in p$, then $f(\sigma)\cdot\alpha(p')(f(\sigma))\in\alpha(p)(\sigma)$: the state transformation f is applied to the current state σ, and the resulting state $f(\sigma)$ is concatenated with $\alpha(p')(f(\sigma))$, in which $f(\sigma)$, being the new state, is passed through to α applied to p', the resumption of f. In this way, the effect of different state transformations occurring subsequently in p is accumulated. A simple example may illustrate this. Consider

$$p = \mathcal{D}_2[\![v:=1;v:=v+1]\!]$$

$$= \{<\lambda\sigma\cdot\sigma_{v:=1}, \{<\lambda\sigma'\cdot\sigma'_{v:=\sigma'(v)+1}, p_0>\}>\}.$$

Then

$$\begin{aligned}\alpha(p)(\sigma) &= \{<\sigma_{v:=1}, \alpha(\{<\lambda\sigma'\cdot\sigma'_{v:=\sigma'(v)+1}, p_0>\})(\sigma_{v:=1})>\}\\ &= \{<\sigma_{v:=1}, \sigma_{v:=2}, \alpha(p_0)(\sigma_{v:=2})>\}\\ &= \{<\sigma_{v:=1}, \sigma_{v:=2}>\}.\end{aligned}$$

Next, we extend α to a mapping $\alpha:(L_2\to\Delta\to\bar{P}_2)\to(L_2\to\Delta\to P_2)$ by putting for $F\in L_2\to\Delta\to\bar{P}_2$:

$$\begin{aligned}\alpha(F) &= F^\alpha\\ &= \lambda s\cdot\lambda\delta\cdot\alpha(F(s)(\delta)).\end{aligned}$$

We shall prove that

$$\forall F\in L_2\to\Delta\to\bar{P}_2\ [\Phi_2'(F^\alpha) = (\Psi_2'(F))^\alpha].$$

Let $F\in L_2\to\Delta\to\bar{P}_2$, $s\in L_2$, $\delta\in\Delta$, and $\sigma\in\Sigma$ be such that

$$\{(a,s',\delta')\in I'(s)(\delta)\mid a\in Asg\vee(a\in BExp\wedge[\![a]\!]\sigma = tt)\}\neq\varnothing.$$

Then

$$\begin{aligned}&\Phi_2'(F^\alpha)(s)(\delta)(\sigma)\\ &= \bigcup\{\sigma\cdot F^\alpha(s')(\delta')(\sigma)\mid (b,s',\delta')\in I'(s)(\delta)\wedge[\![b]\!]\sigma=tt\}\cup\\ &\quad\bigcup\{\sigma_{v:=e}\cdot F^\alpha(s')(\delta')(\sigma_{v:=e})\mid (v:=e,s',\delta')\in I'(s)(\delta)\}\\ &= \bigcup\{\sigma\cdot(\alpha(F'(s')(\delta'))(\sigma))\mid (b,s',\delta')\in I'(s)(\delta')\wedge[\![b]\!]\sigma=tt\}\cup\\ &\quad\bigcup\{\sigma_{v:=e}\cdot(\alpha(F'(s')(\delta'))(\sigma_{v:=e}))\mid (v:=e,s',\delta')\in I'(s)(\delta')\}\\ &= \alpha(\{<\kappa_a, F'(s')(\delta')>\mid (a,s',\delta')\in I'(s)(\delta')\})(\sigma)\\ &\quad[\text{with } \kappa_a \text{ as above}]\\ &= \alpha(\Psi_2'(F)(s)(\delta))(\sigma)\\ &= (\Psi_2'(F))^\alpha(s)(\delta)(\sigma).\end{aligned}$$

The case that $\Phi_2'(F^\alpha)(s)(\delta)(\sigma)=\{\partial\}$ goes similarly. This proves

$$\forall F\in L_2\to\Delta\to\bar{P}_2\ [\Phi_2'(F^\alpha) = (\Psi_2'(F))^\alpha].$$

Now it follows that

$$(\mathfrak{D}_2')^\alpha = \Theta_2'.$$

Collecting the results from above, we see:

$$\Theta_2^{<>} = (\tilde{\mathfrak{D}}_2)^\alpha, \text{ or, equivalently}$$

$$\forall s\in L_2\ \forall\delta\in\Delta\ [\Theta_2[\![s<\delta>]\!] = \alpha(\mathfrak{D}_2[\![s]\!](\tilde{\delta}))],$$

with the obvious corollary, that

$$\forall s\in L_2^{cl}\ \forall\gamma\in\Gamma\ [\Theta_2[\![s]\!] = \alpha(\mathfrak{D}_2[\![s]\!](\gamma))].$$

4. Conclusions

We have developed a uniform method of comparing different semantic models for imperative concurrent programming languages. We have defined operational and denotational semantic models for such languages as fixed points of contractions on complete metric spaces, and have related them by relating their corresponding contractions. Here, we benefit from the metric structure of the underlying mathematical domains, which ensures the uniqueness of the fixed point of such contractions (Banach's theorem). It turns out that once this method has been applied to a certain (simple) language (L_0), it can be easily generalized for more complex languages (L_1 and L_2). This we consider to be the strength of this approach. Currently, we are investigating possible extensions of this method to deal with yet other languages, containing, e.g., program constructs for process creation.

Our investigations are related to the question of *full abstraction*, which at the same time is a topic for further research. If L is a language with semantics $\mathcal{O}$ and $\mathcal{D}$, then we call $\mathcal{D}$ *fully abstract with respect to* $\mathcal{O}$ if

$$\forall s \in L \forall t \in L\ [\mathcal{D}[\![s]\!] = \mathcal{D}[\![t]\!] \Leftrightarrow \forall C(\cdot)\ [\mathcal{O}[\![C(s)]\!] = \mathcal{O}[\![C(t)]\!]],$$

where $C(\cdot)$ ranges over the set of *contexts* for L, that is, the set of statements in L containing one or more *holes*. An example would be s;(·), where (·) denotes the hole. Given such a context $C(\cdot)$ and a statement s the statement $C(s)$ is obtained by substituting s for all the holes in $C(\cdot)$. The issue of full abstraction is mostly raised with respect to a model $\mathcal{O}$ that is *operational*, expressing a notion of observability, and a model $\mathcal{D}$ that is *compositional*. Then it follows from a relation between $\mathcal{O}$ and $\mathcal{D}$ of the form $\mathcal{O} = \alpha \circ \mathcal{D}$ that for all s and $t \in L$:

$$\mathcal{D}[\![s]\!] = \mathcal{D}[\![t]\!] \Rightarrow \forall C(\cdot)\ [\mathcal{O}[\![C(s)]\!] = \mathcal{O}[\![C(t)]\!]].$$

(This property is sometimes called: *correctness* of $\mathcal{D}$ with respect to $\mathcal{O}$.) Thus, our result of proving $\mathcal{O} = \alpha \circ \mathcal{D}$ partly solves the problem of full abstraction. The reversed arrow is still an issue for further research.

5. References

[AP] K. Apt, G. Plotkin, *Countable nondeterminism and random assignment,* Journal of the Association for Computing Machinery, Vol. 33, No. 4, October 1986, pp. 724-767.

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

[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 (J.W. de Bakker, W.P. de Roever, G. Rozenberg, eds.), Lecture Notes in Computer Science 224, Springer-Verlag, 1986, pp. 51-121.

[BMOZ1] J.W. de Bakker, J.-J. Ch. Meyer, E.-R. Olderog, J.I. Zucker, *Transition systems, infinitary languages and the semantics of uniform concurrency,* in: Proceedings 17th ACM STOC, Providence, R.I. (1985) 252-262.

[BMOZ2] J.W. de Bakker, J.-J. Ch. Meyer, E.-R. Olderog, J.I. Zucker, *Transition systems, metric spaces and ready sets in the semantics of uniform concurrency,* Report CS-R8601, Centre for

Mathematics and Computer Science, Amsterdam, January 1986. (To appear in: Journal of Computer and System Sciences.)

[BZ] J.W. DE BAKKER, J.I. ZUCKER, *Processes and the denotational semantics of concurrency,* Information and Control 54 (1982) 70-120.

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

[En] E. ENGELKING, *General topology,* Polish Scientific Publishers, 1977.

[FHLR] N. FRANCEZ, C.A.R. HOARE, D.J. LEHMANN, W.P. DE ROEVER, *Semantics of nondeterminism, concurrency and communication,* J. CSS 19 (1979) 290-308.

[HP] M. HENNESSY, G.D. PLOTKIN, *Full abstraction for a simple parallel programming language,* in: Proceedings 8th MFCS (J. Bečvař ed.), Lecture Notes in Computer Science 74 Springer-Verlag (1979) 108-120.

[Ho] C.A.R. HOARE, *Communicating sequential processes,* Prentice Hall International, 1985.

[Mic] E. MICHAEL, *Topologies on spaces of subsets,* in: Trans. AMS 71 (1951), pp. 152-182.

[Mil] R. MILNER, *A Calculus of communicating systems,* Lecture Notes in Computer Science 92, Springer-Verlag, 1980.

[Pl1] G.D. PLOTKIN, *A powerdomain construction,* SIAM J. Comp. 5 (1976) 452-487.

[Pl2] G.D. PLOTKIN, *A structural approach to operational semantics,* Report DAIMI FN-19, Comp. Sci. Dept., Aarhus Univ. 1981.

[Pl3] G.D. PLOTKIN, *An operational semantics for CSP,* in: Formal Description of Programming Concepts II (D. Bjørner ed.) North-Holland, Amsterdam (1983) 199-223.

[Sc] D.S. SCOTT, *Domains for denotational semantics,* Proc. 9th ICALP (M. Nielsen, E.M. Schmidt, eds.), Lecture Notes in Computer Science 140, Springer-Verlag, 1982, pp. 577-613.

6. APPENDIX: MATHEMATICAL DEFINITIONS

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

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

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

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

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

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

DEFINITION A.2
Let (M,d) be a metric space, let $(x_i)_i$ be a sequence in M.
(a) We say that $(x_i)_i$ is a *Cauchy sequence* whenever we have:
$\forall \epsilon > 0\ \exists N \in \mathbb{N}\ \forall n,m > N\ [d(x_n,x_m) < \epsilon]$.
(b) Let $x \in M$. We say that $(x_i)_i$ *converges to* x and call x the *limit* of $(x_i)_i$ whenever we have:
$\forall \epsilon > 0\ \exists N \in \mathbb{N}\ \forall n > N\ [d(x,x_n) < \epsilon]$.
Such a sequence we call *convergent*. Notation: $\lim_{i\to\infty} x_i = x$.
(c) The metric space (M,d) is called *complete* whenever each Cauchy sequence converges to an element of M.

DEFINITION A.3
Let $(M_1,d_1),(M_2,d_2)$ be metric spaces.
(a) We say that (M_1,d_1) and (M_2,d_2) are *isometric* if there exists a bijection $f:M_1 \to M_2$ such that:
$\forall x,y \in M_1\ [d_2(f(x),f(y)) = d_1(x,y)]$. We then write $M_1 \cong M_2$. When f is not a bijection (but only an injection), we call it an *isometric embedding*.
(b) Let $f:M_1 \to M_2$ be a function. We call f *continuous* whenever for each sequence $(x_i)_i$ with limit x in M_1 we have that $\lim_{i\to\infty} f(x_i) = f(x)$.
(c) Let $A \geqslant 0$. With $M_1 \to^A M_2$ we denote the set of functions f from M_1 to M_2 that satisfy the following property:
$\forall x,y \in M_1\ [d_2(f(x),f(y)) \leqslant A \cdot d_1(x,y)]$.
Functions f in $M_1 \to^1 M_2$ we call *non-distance-increasing* (NDI), functions f in $M_1 \to^\epsilon M_2$ with $0 \leqslant \epsilon < 1$ we call *contracting*.

PROPOSITION A.4
(a) *Let $(M_1,d_1),(M_2,d_2)$ be metric spaces. For every $A \geqslant 0$ and $f \in M_1 \to^A M_2$ we have: f is continuous.*
(b) *(Banach's fixed-point theorem)*
Let (M,d) be a complete metric space and $f:M \to M$ a contracting function. Then there exists an $x \in M$ such that the following holds:
(1) *$f(x) = x$ (x is a fixed point of f),*
(2) *$\forall y \in M\ [f(y) = y \Rightarrow y = x]$ (x is unique),*
(3) *$\forall x_0 \in M\ [\lim_{n\to\infty} f^{(n)}(x_0) = x]$, where $f^{(n+1)}(x_0) = f(f^{(n)}(x_0))$ and $f^{(0)}(x_0) = x_0$.*

DEFINITION A.5 (Compact subsets)
A subset X of a metric space (M,d) is called *compact* whenever each sequence in X has a subsequence that converges to an element of X.

DEFINITION A.6
Let $(M,d),(M_1,d_1),\ldots,(M_n,d_n)$ be metric spaces.
(a) With $M_1 \to M_2$ we denote the set of all continuous functions from M_1 to M_2. We define a metric d_F on $M_1 \to M_2$ as follows. For every $f_1,f_2 \in M_1 \to M_2$

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

For $A \geqslant 0$ the set $M_1 \to^A M_2$ is a subset of $M_1 \to M_2$, and a metric on $M_1 \to^A M_2$ can be obtained by taking the restriction of the corresponding d_F.
(b) With $M_1 \cup \cdots \cup M_n$ we denote the *disjoint union* of $M_1,\ldots,M_n$, which can be defined as $\{1\} \times M_1 \cup \cdots \cup \{n\} \times M_n$. We define a metric d_U on $M_1 \cup \cdots \cup M_n$ as follows. For every $x,y \in M_1 \cup \cdots \cup M_n$

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

(c) We define a metric d_P on $M_1 \times \cdots \times M_n$ by the following clause.

For every $(x_1, \ldots, x_n), (y_1, \ldots, y_n) \in M_1 \times \cdots \times M_n$

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

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

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

where $d(x, Z) =^{def} \inf_{z \in Z}\{d(x,z)\}$ for every $Z \subseteq M$, $x \in M$.
In $\mathcal{P}_{co}(M) =^{def} \{X | X \subseteq M \wedge X$ is compact$\}$ we also have the empty set as an element. We define d_H on $\mathcal{P}_{co}(M)$ as above but extended with the following case. If $X \neq \varnothing$, then

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

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

PROPOSITION A.7
Let (M,d), $(M_1,d_1), \ldots, (M_n,d_n)$, d_F, d_U, d_P and d_H be as in definition A.6 and suppose that (M,d), $(M_1,d_1), \ldots, (M_n,d_n)$ are complete. We have that
(a) $(M_1 \underline{\rightarrow} M_2, d_F)$, $(M_1 \rightarrow^A M_2, d_F)$,
(b) $(M_1 \cup \cdots \cup M_n, d_U)$,
(c) $(M_1 \times \cdots \times M_n, d_P)$,
(d) $(\mathcal{P}_{nc}(M), d_H)$, and $(\mathcal{P}_{co}(M), d_H)$
are complete metric spaces. If (M,d) and (M_i,d_i) are all ultra-metric spaces these composed spaces are again ultra-metric. (Strictly spoken, for the completeness of $M_1 \rightarrow M_2$ and $M_1 \rightarrow^A M_2$ we do not need the completeness of M_1. The same holds for the ultra-metric property.)

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

PROPOSITION A.8
Let $(\mathcal{P}_{co}(M), d_H)$ be as in definition A.6. Let $(X_i)_i$ be a Cauchy sequence in $\mathcal{P}_{co}(M)$. We have:

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

The proof of proposition A.8 can be found in [Mic] as a generalization of a similar result (for *closed* subsets) in [Du] and [En].

Semantic Correctness for a Parallel Object-Oriented Language

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

Different semantic models are studied for a language called POOL: a parallel object-oriented language. It is a simplified version of POOL-T, a language that is actually used to write programs for a parallel machine. The most important aspect of this language is that it describes a system as a collection of communicating objects that all have internal activities which are executed in parallel. For POOL, an operational and a denotational semantics have been developed previously. The former semantics aims at the intuitive operational meaning of the language, whereas the main characteristic of the latter is compositionality. In this paper, we relate both models, which are quite different, and prove the semantic correctness of the denotational semantics with respect to the operational semantics. Our semantic investigations take place in the mathematical framework of complete metric spaces. For the operational semantics we use a simple space of functions from states to compact sets of streams (which are sequences of states); for the denotational semantics, a domain of processes is used, which is the solution of a reflexive domain equation over a category of complete metric spaces. The main mathematical tool we use is Banach's theorem, which states that contractions on complete metric spaces have unique fixed points. Both the operational and the denotational semantics are reformulated and are presented, as well as many operators on the semantic domains, as the fixed point of a suitably defined contraction. In this way, we are able to establish a formal equivalence between both models. For this purpose, we introduce an intermediate domain, which first is compared to the operational model by means of an abstraction operator. This function takes processes, which are tree-like structures, as arguments and yields sets of streams as results. Next, it is shown that both the intermediate and the denotational model are fixed points of the same contraction, from which their equality follows. From both facts, the main result of our study follows: The operational meaning of a POOL program is equal to the denotational meaning to which the abstraction operator is applied. In this manner, the correctness of the denotational semantics with respect to the operational semantics is established.

1980 Mathematical Subject Classification: 68B10, 68C01.
1986 Computing Reviews Categories: D.3.1, F.3.2, F.3.3.
Key words and phrases: operational semantics, denotational semantics, process creation, object-oriented programming, semantic correctness, complete metric spaces, contractions.

1. Introduction

We study different semantic models for a language called POOL: parallel object-oriented language. Although the theoretical foundations of object-oriented programming in general, and of *parallel* object-oriented programming in particular, have not been paid much attention to, this language has been extensively studied in a formal semantic context: In [ABKR86(a)] and [ABKR86(b)], an operational and a denotational semantics of POOL have been developed. The main goal of this paper is to compare the two models, which are quite different, by proving some formal relation between them, which at the same time will establish the correctness of the denotational semantics with respect to the operational semantics. Before we explain in some detail the language POOL and the contents of this paper, we first give a short explanation of the notion of semantic correctness and the way it can be proved.

A semantics for a programming language $\mathfrak{L}$ is a mapping $\mathfrak{M}:\mathfrak{L}\rightarrow D$, where D is some mathematical domain (a set, a complete partial ordering, a complete metric space), which we call the semantic

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

universe of $\mathfrak{M}$. Sometimes $\mathfrak{M}$ is called a model for $\mathfrak{L}$. Traditionally, two main types of semantics are distinguished: *operational* semantics and *denotational* semantics. Without wanting to get involved in a discussion about the precise definitions, we state that in our view the main characteristic of the former is that its definition is based on a *transition relation* ([HP79], [Pl81], [Pl83]); a denotational semantics is characterised by the fact that it is defined in a *compositional* manner: the denotational semantics of a composite statement is given in terms of the denotational semantics of its components. (As a second distinctive property one often considers the way in which recursion is treated: The usual view is that an operational semantics treats recursion by means of so-called *syntactic environments* (or body replacement) whereas a denotational semantic uses *semantic environments*, in combination with some fixed-point argument.)

Now consider an operational semantics $\mathcal{O}:\mathfrak{L}\to D$ and a denotational semantics $\mathfrak{D}:\mathfrak{L}\to D'$. A natural question is whether $\mathfrak{D}$ is *correct* with respect to $\mathcal{O}$, that is, whether $\mathfrak{D}$ makes *at least* the same distinctions as $\mathcal{O}$ does. (Often, $\mathfrak{D}$ makes more; see [KR88] for a simple example.) If we define for a semantics $\mathfrak{M}:\mathfrak{L}\to D''$ an equivalence relation $\equiv_{\mathfrak{M}}$ by

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

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

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

One way to prove the correctness of $\mathfrak{D}$ is to introduce a so-called *abstraction* operator $\alpha:D'\to D$, which (is in general not injective and) relates the denotational semantic universe with the operational one. If one can prove that

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

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

As a mathematical framework for our semantic descriptions we have chosen *complete metric spaces*. (For the basic definitions of topology see [Du66] or [En77].) In this we follow and generalize [BZ82]. (For other applications of this type of semantic framework see [BKMOZ86].) We follow [KR88] in using contractions on complete metric spaces as our main mathematical tool, exploring the fact that contractions have *unique* fixed points (Banach's theorem). We shall define both operators on our semantic universes and the semantic models themselves as fixed points of suitably defined contractions. In this way, we are able to use a general method for proving semantic correctness: Suppose we have defined $\mathcal{O}$ as the fixed point of a contraction

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

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

It is the approach sketched above that will be applied to the language POOL. Before doing so, we start in section 2 with a toy language that is extremely simple but has with POOL in common a construct for process creation. This section can be seen as a prolongation of the introduction and tries to give the reader some feeling for the techniques used. Since no definitions or results of this section are used in the other sections it can be skipped without any problem.

The language POOL is described in detail in section 3. It is a simplified version of the language POOL-T, which is defined in [Am85] and for which [Am86] and [Am87] give an account of the design considerations. POOL-T was designed in subproject A of ESPRIT project 415 with the purpose of programming a highly parallel machine which is also being developed in this project (see [Od87] for an overview). The language provides all the facilities needed to program reasonably large parallel systems and several large applications and many small ones have been written in it.

In POOL, a system is viewed as a collection of *objects*. These are dynamic entities containing *data* (stored in *variables*) and *methods* (a kind of procedures). Objects can be created dynamically during the execution of a program and each of them has an internal activity (its *body*) in which it can execute

expressions and statements. While inside an object everything proceeds sequentially, the concurrent execution of the bodies of all the objects can give rise to a large amount of parallelism. Objects can interact by sending *messages* to each other. Acceptance of a message gives rise to a rendez-vous between sender and receiver, during which an appropriate method is executed.

In section 4, we follow [ABKR86(a)] in defining an operational semantics for POOL. It is based on a transition relation and is given, and here we differ from [ABKR86(a)], as the fixed point of a contraction. The semantic domain used is a complete metric space of (functions from states to) compact sets of streams, which are sequences of states.

In section 5, we present a denotational semantics for POOL, very similar to the model given in [ABKR86(b)]. We define a mapping from the set of POOL programs (called *units*) to some reflexive domain of processes $\overline{P}$ (cf. [Pl76]), which is a complete metric space with tree-like structures for its elements. It satisfies a reflexive domain equation, which is solved by deriving from it a functor on a category of complete metric spaces and then taking the fixed point of this functor. The mathematical techniques to do so are sketched in section 2 of [ABKR86(b)] and presented in detail in [AR88]. Before we assign a semantic value to the unit as a whole, we first define the semantics of expressions and statements, which will be given by functions of the following type:

$$\mathcal{D}_E: L_E \to AObj \to Cont_E \to \overline{P}, \quad \text{and} \quad \mathcal{D}_S: L_S \to AObj \to Cont_S \to \overline{P},$$

where L_E and L_S are the sets of expressions and statements and

$$Cont_E = Obj \to \overline{P}, \quad Cont_S = \overline{P}.$$

The semantic domain $AObj$ stands for the set of (active) object names. Its appearance in the semantics of expressions and statements reflects the fact that in POOL each expression or statement is evaluated *by a certain object*. Further, a *continuation* will be given as an argument to the semantic functions. This describes what will happen *after* the execution of the current expression or statement. As the continuation of an expression generally depends upon the result of this expression (an object name), its type is $Obj \to \overline{P}$, whereas the type of continuations of statements is simply $\overline{P}$. The use of continuations makes it possible to define the semantics, especially of object creation, in a convenient and concise way. (For more examples of the use of continuations in semantics, see [Br86] and [Go79].)

After having defined an operational and a denotational semantics for POOL, we come to the main subject of our paper: The comparison of both models. This constitutes a non-trivial problem, mainly because, first, the respective semantic domains are very different and, secondly, because the denotational semantics is defined in terms of continuations, whereas the operational semantics is direct, that is, does not use continuations. Moreover, the communication mechanism of POOL (consisting of message passing with method invocation) is dealt with quite differently by the two models. The solution that we propose consists of the introduction of an intermediate semantic model, in section 6, which has in common with the operational semantics that it is direct (without continuations) and that it is based on the same transition relation, but which has for its range the same reflexive domain of processes as the denotational model has. Then, in section 7, this intermediate model is related to the operational semantics by means of an abstraction operator which takes processes as arguments and yields sets of streams. Next, it is connected with (an extended version of) the denotational semantics by the observation that both models are fixed point of the same contraction. As a result, it follows that the operational semantics of a unit equals its denotational meaning to which the abstraction operator is applied.

Section 8, which contains the references, is followed by three appendices. Appendix I gives the mathematical definitions we use; in appendix II, the abstraction operator that is used in the proof of the semantic correctness for POOL is defined in all formal detail. Finally, appendix III shows how the language POOL can be extended with so-called *standard* objects and how the definitions and proofs can be adapted in order to obtain a similar correctness result for the extended language.

Semantic treatments of parallel object-oriented languages in general are scarce; we only know

[Cl81], which gives a detailed mathematical model for an actor language. This is done by defining a set of so-called augmented actor event diagrams, each of which is a fairly complicated structure representing (the beginning of) a single computation. In order to deal with nondeterminism, a novel power domain construction is used. As to the comparison of operational and denotational semantics for languages with process creation, we only know of [AB88], where some simplified versions of POOL are studied. None of these languages, however, contains the original POOL-T constructs for communication (for message passing with method invocation), the treatment of which, in the correctness proof, we consider to be an essential part of this paper.

ACKNOWLEDGEMENTS: We wish to thank Pierre America for his detailed and constructive comments on preliminary versions of this paper. Discussions with Jaco de Bakker are gratefully acknowledged, as well as the contributions of the Amsterdam Concurrency Group: Jaco de Bakker, Frank de Boer, Arie de Bruin, Joost Kok, John-Jules Meyer and Erik de Vink. We thank Mini Middelberg for the expert typing of this document.

2. A VERY SIMPLE LANGUAGE WITH PROCESS CREATION

Before we tackle the main problem of this paper, we would like to start with a much simpler case: We introduce a very small "toy" language L_T and present an operational and a denotational semantics for it. Next, we shall compare these two models. All this can be regarded as a little exercise, a "warming up" so to speak, aiming at a better understanding of what follows in the next section: It turns out that for both the languages L_T and POOL (to be introduced in the next section) the operational and denotational semantics can be compared in very much the same way.

For the definition of L_T we need a set $(a,b \in)A$ of *elementary actions*. (Throughout this paper, we shall use the notation $(x, y \in)X$ for the introduction of a set X with typical elements x and y.) For A we take an arbitrary, possibly infinite, set. It will contain a subset $(c \in)C \subseteq A$ of so-called *communications*. Similarly to CCS ([Mil80]), we define a bijection $^- : C \to C$ with $^- \circ ^- = id_C$. It yields for every $c \in C$ a matching communication $\overline{c}$. In $A \setminus C$ we have a special element τ denoting successful communication.

DEFINITION 2.1 (Syntax for L_T)
The set of statements $(s,t \in)L_T$ is given by

$$s ::= a \mid s_1;s_2 \mid \mathbf{new}(s).$$

Note that $a \in A \supseteq C$. To L_T we add a special element E, denoting the *empty* statement. Note that syntactic constructs like s; E and $\mathbf{new}(E)$ are *not* in L_T.

A statement is of one of the following forms: First, it can be an elementary action a. Here elementary means that it is an uninterpreted action. Examples of possible interpretations are assignments, or read and write actions. Secondly, a statement s can be the sequential composition s_1; s_2 of statements s_1 and s_2. Finally, it may be a new-statement $\mathbf{new}(s)$, the execution of which amounts to the creation of a new process which executes s. A more detailed explanation will follow below.

The operational semantics will be formulated using the notion of *parallel statements*. A parallel statement is a finite sequence of statements which are to be executed in parallel.

DEFINITION 2.2 (Parallel statements)
Let $(\rho,\pi \in)Par$ be given by $Par = (L_T)^*$, the set of finite sequences of statements. Typical elements will also be indicated by $<s_1, \ldots, s_n>$, for $n \geq 1$. For $\rho = <s_1, \ldots, s_n>$ and $\pi = <t_1, \ldots, t_m>$ we define $\rho^\wedge\pi = <s_1, \ldots, s_n, t_1, \ldots, t_m>$.

Next we define the operational semantics of parallel statements. It is based on the well known

notion of a *transition relation* (in the style of Hennessy and Plotkin ([HP79, Pl81, Pl83])).

DEFINITION 2.3 (Transition relation for *Par*)
Let $\rightarrow \subseteq Par \times A \times Par$ be the smallest relation (writing $\rho - a \rightarrow \rho'$ for $(\rho, a, \rho') \in \rightarrow$) satisfying:

(1) $<a> - a \rightarrow <E>$, $<a;s> - a \rightarrow <s>$

(2) if $<s> - a \rightarrow \rho$, then $<\mathbf{new}(s)> - a \rightarrow \rho$

(3) if $<s,t> - a \rightarrow \rho$, then $<\mathbf{new}(s);t> - a \rightarrow \rho$

(4) if $<s_1;(s_2;s_3)> - a \rightarrow \rho$, then $<(s_1;s_2);s_3> - a \rightarrow \rho$

(5) if $\rho - a \rightarrow \rho'$, then $\rho^\wedge \pi - a \rightarrow \rho'^\wedge \pi$ and $\pi^\wedge \rho - a \rightarrow \pi^\wedge \rho'$

(6) if $\rho - c \rightarrow \rho'$ and $\pi - \overline{c} \rightarrow \pi'$, then $\rho^\wedge \pi - \tau \rightarrow \rho'^\wedge \pi'$,

for $a \in A$, $c \in C$, $s,t,s_1,s_2,s_3 \in L_T$, and $\rho,\rho',\pi,\pi' \in Par$.

Intuitively, $\rho - a \rightarrow \rho'$ tells us that starting in the parallel statement ρ the elementary action a can be performed, resulting in the parallel statement ρ'. Interesting in the definition above are (3), (5) and (6). According to (3), the parallel statements $<s,t>$ and $<\mathbf{new}(s);t>$ can perform the same elementary actions. In other words, evaluating $<\mathbf{new}(s);t>$ results in a parallel statement $<s,t>$. Thus we see that the length of a parallel statement increases when $\mathbf{new}(s)$ is evaluated. Operationally, this can be viewed as the creation of a process that starts evaluating s, while statement t is being executed in parallel. According to (5), a composite parallel statement $\rho^\wedge \pi$ can perform all the elementary actions that can be performed by either ρ or π. In (6) it is expressed that if ρ can perform a communication action c and π can perform a matching communication action $\overline{c}$, then $\rho^\wedge \pi$, the parallel statement composed of ρ and π, can perform a τ action, denoting a successful communication.

EXAMPLE: $<\mathbf{new}(c); a; \mathbf{new}(\overline{c}); b> - a \rightarrow <c, \mathbf{new}(\overline{c}); b> - b \rightarrow <c, \overline{c}, E> - \tau \rightarrow <E, E, E>$.

Before we give the definition of the operational semantics of parallel statements, we introduce its semantic universe P.

DEFINITION 2.4 (Semantic universe P)
Let A^* denote the set of finite sequences or *words* of elements of A; let ϵ denote the empty word. We extend this set by allowing as the last element of a finite sequence a special element ∂, which denotes *deadlock*:

$$(w \in) A_\partial = A^* \cup A^* \cdot \{\partial\}.$$

Now we define $(p,q \in) P = \mathcal{P}_{nf}(A_\partial^*)$, the set of all non-empty, finite subsets of A_∂^*. Let $d_{A_\partial^*}$ denote the usual metric on A_∂^* (see the definition in A.1.1). We take $d_P = (d_{A_\partial^*})_H$, the Hausdorff metric induced by $d_{A_\partial^*}$, as a metric on P. According to proposition A.7, we have that (P,d_P) is a complete metric space.

DEFINITION 2.5 (Operational semantics Θ)
Let $\Theta =$ Fixed Point (Φ), where $\Phi:(Par \rightarrow P) \rightarrow (Par \rightarrow P)$ is given, for $F \in Par \rightarrow P$, and $\rho \in Par$, by

$$\Phi(F)(\rho) = \begin{cases} \{\epsilon\} & \text{if } \rho = <E, \ldots, E> \\ \{\partial\} & \text{if } \forall a \forall \rho' [\rho - a \rightarrow \rho' \Rightarrow a \in C] \wedge \rho \neq <E, \ldots, E> \\ \bigcup \{a \cdot F(\rho') : \rho - a \rightarrow \rho' \wedge a \notin C\} & \text{otherwise.} \end{cases}$$

It is straightforward to show that Φ is a contraction and thus has a unique fixed point.

Since our language does not contain any constructs for recursion, we need not be able to describe infinite behavior. Therefore, it is not really necessary to define Θ using a contraction on a complete metric space. It would have been sufficient to take P as an ordinary set without any metric, and define Θ with an easy induction on the structure of statements. Our motivation for nevertheless exploiting metric structures here is given by the fact that in the next section we *will* deal with recursion and infinite behavior. There the use of some mathematical structure which can handle these, such as complete metric spaces, is obligatory. Our use of complete metric spaces at this stage can be seen as part of the introductory function of this section.

The operational semantics Θ can be best explained by giving a few

EXAMPLES:

$$\Theta[\![<a>]\!] = a\cdot\Theta[\![<E>]\!] = a\cdot\{\epsilon\} = \{a\}$$

$$\Theta[\![<\textbf{new}\ (a)>]\!] = \{a\}$$

$$\Theta[\![<c>]\!] = \{\partial\}$$

$$\Theta[\![<c,\bar{c}>]\!] = \{\tau\}$$

$$\Theta[\![<a;b>]\!] = a\cdot\Theta[\![<b>]\!] = \{ab\}$$

$$\Theta[\![<\textbf{new}\ (a);b>]\!] = \{a\cdot\Theta[\![<E,b>]\!],\ b\cdot\{\Theta[\![<a,E>]\!]\} = \{ab, ba\}$$

Note that a single communication $<c>$, without a matching communication $\bar{c}$ in parallel, creates a deadlock.

Such an operational semantics is nice, because it is intuitively very clear. However, it is not *compositional* with respect to the binary syntactic operator ;, that is, there is no semantic operator $\tilde{;}: P\times P\to P$, corresponding to ;, such that for all s and t:

$$\Theta[\![<s;t>]\!] = \Theta[\![<s>]\!]\ \tilde{;}\ \Theta[\![<t>]\!].$$

This can be easily seen by the following argument. Suppose there *is* such an operator $\tilde{;}$. Then:

$$\begin{aligned}\Theta[\![<\textbf{new}\ (a);b>]\!] &= \Theta[\![<\textbf{new}\ (a)>]\!]\ \tilde{;}\ \Theta[\![<b>]\!]\\ &= [\text{since } \Theta[\![<\textbf{new}\ (a)>]\!] = \Theta[\![<a>]\!]]\\ &\quad\ \Theta[\![<a>]\!]\ \tilde{;}\ \Theta[\![<b>]\!]\\ &= \Theta[\![<a;b>]\!],\end{aligned}$$

which yields a contradiction, as can be seen from the examples above.

The denotational semantics to be defined in a moment has the property that it is compositional with respect to the syntactic operators in L_T.

First, we define a suitable semantic universe.

DEFINITION 2.6 (Semantic universe $\bar{P}$)
We define a complete metric space $(p,q\in)\bar{P}$ by $\bar{P} = \mathcal{P}_{nf}(A^*)$, the set of non-empty finite subsets of A^*. Let d_{A^*} be the usual metric on A^*; we define $d_P = (d_{A^*})_H$.

The only difference between P and $\bar{P}$ is that the latter does not contain finite sequences ending in ∂.

DEFINITION 2.7 (Denotational semantics $\mathcal{D}$)
Let $\mathcal{D}$: $L_T\to$ Cont $\to\bar{P}$, where Cont$=\bar{P}$ denotes the set of *continuations*, be given by

$$\mathcal{D}[\![a]\!](p) = a\cdot p,\ \mathcal{D}[\![E]\!](p) = p$$

$$\mathfrak{D}[\![\mathbf{new}(s)]\!](p) = p \| \mathfrak{D}[\![s]\!](\{\epsilon\})$$
$$\mathfrak{D}[\![s\,;t]\!](p) = \mathfrak{D}[\![s]\!](\mathfrak{D}[\![t]\!](p)),$$

with $\|\colon P \times P \to P$ as defined below.

A continuation $p \in \mathrm{Cont}$ denotes the semantics of the statement to be executed after the one to which $\mathfrak{D}$ is applied. The meaning of a new-construct **new** (s) with continuation p is determined as follows: The meaning of s is computed with the empty continuation $\{\epsilon\}$, which indicates that after s nothing remains to be done. Since s is to be executed in parallel with everything that follows, the result is composed in parallel with p, which indicates the remainder of the program after s.

DEFINITION 2.8 (Parallel composition $\|$)
Let $\|\colon P \times P \to P$ be such that it satisfies, for $p,q \in P$,

$$p \| q = p \mathbin{\mathbb{L}} q \cup q \mathbin{\mathbb{L}} p \cup p \mid q,$$

where

$$p \mathbin{\mathbb{L}} q = \bigcup \{a \cdot (p_a \| q) \colon p_a \neq \emptyset\} \cup \{q : \epsilon \in p\},$$
$$p \mid q = \bigcup \{\tau \cdot (p_c \| q_{\bar{c}}) \colon p_c \neq \emptyset \neq q_{\bar{c}}\},$$

with $p_a = \{w \colon a \cdot w \in p\}$, the set containing all the postfixes of a in p.

The above definition is self-referential and needs some justification. Formally, we can define $\|$ as the fixed point of a contraction $\Psi\colon (\overline{P} \times \overline{P} \to \overline{P}) \to (\overline{P} \times \overline{P} \to \overline{P})$ given, for $f \in \overline{P} \times \overline{P} \to \overline{P}$, by

$$\Psi(f)(p,q) = p \mathbin{\mathbb{L}}_f q \cup q \mathbin{\mathbb{L}}_f p \cup p \mid_f q,$$

where

$$p \mathbin{\mathbb{L}}_f q = \bigcup \{a \cdot f(p_a, q) \colon p_a \neq \emptyset\} \cup \{q : \epsilon \in p\},$$
$$p \mid_f q = \bigcup \{\tau \cdot (f(p_c, q_{\bar{c}})) \colon p_c \neq \emptyset \neq q_{\bar{c}}\}.$$

Note that $\mathfrak{D}$ is compositional with respect to ";". The corresponding semantic operator $\tilde{;}\colon ((\overline{P} \to \overline{P}) \times (\overline{P} \to \overline{P})) \to (\overline{P} \to \overline{P})$ is not expressed explicitly in the definition of $\mathfrak{D}$. For completeness sake, we give its definition. We have, for $f, g \in \overline{P} \to \overline{P}$:

$$f \,\tilde{;}\, g = \lambda p \cdot f(g(p)).$$

Semantic equivalence of Θ and $\mathfrak{D}$
After having defined Θ and $\mathfrak{D}$ for *Par* and L_T, we next discuss the relationship between the two semantics. We shall compare Θ and $\mathfrak{D}$ by relating both to an intermediate semantics $\Theta'\colon Par \to P$, given in

DEFINITION 2.9 (Intermediate semantics Θ')
Let $\Theta' =$ Fixed Point (Φ'), where $\Phi'\colon (Par \to \overline{P}) \to (Par \to \overline{P})$ is given, for $F \in Par \to \overline{P}$ and $\rho \in Par$, by

$$\Phi'(F)(\rho) = \begin{cases} \{\epsilon\} & \text{if } \rho = <E, \ldots, E> \\ \bigcup \{a \cdot F(\rho') \colon \rho - a \to \rho'\} & \text{otherwise.} \end{cases}$$

Note that in Φ', as opposed to Φ, single-sided communication steps $a \in C$ are allowed. The difference between Θ and Θ' can be illustrated by giving a few examples:

$$\Theta[\![<c>]\!] = \{\partial\}, \quad \Theta[\![<c,\bar{c}>]\!] = \{\tau\},$$
$$\Theta'[\![<c>]\!] = \{c\}, \quad \Theta'[\![<c,\bar{c}>]\!] = \{c\bar{c},\ \bar{c}c,\ \tau\}.$$

The relationship between θ and Θ' will be expressed using the following abstraction operation.

DEFINITION 2.10 (Abstraction operator α)
We define an abstraction operator $\alpha: \bar{P} \to P$ by

$$\alpha(p) = \begin{cases} \{\partial\} & \text{if } \forall a[p_a \neq \emptyset \Rightarrow a \in C] \\ \bigcup \{a \cdot (\alpha(p_a)) : a \notin C \wedge p_a \neq \emptyset\} \cup \{\epsilon : \epsilon \in p\} & \text{otherwise,} \end{cases}$$

with p_a as in definition 2.8. (For a justification of this self-referential definition see the remark following definition 2.8.)

The definition of α can be understood as follows: If all the words $w \in p$ begin with a communication action $a \in C$, we have operationally a deadlock, since no single communication action is allowed. Therefore, we then have: $\alpha(p) = \{\partial\}$. In the last case, $\alpha(p)$ contains all the words in p that begin with a non-communication action $a \in A \setminus C$, with α recursively applied to p_a, the set of postfixes of a; additionally, $\alpha(p)$ contains ϵ if $\epsilon \in p$.

The following theorem can be proved straightforwardly.

THEOREM 2.11: $\forall F \in Par \to \bar{P}\,[\Phi(\alpha \circ F) = \alpha \circ \Phi'(F)]$

Since Φ and Φ' are contractions and thus have unique fixed points, it follows that

COROLLARY 2.12: $\Theta = \alpha \circ \Theta'$

PROOF
We have: $\alpha \circ \Theta' = \alpha \circ \Phi'(\Theta') = \Phi(\alpha \circ \Theta')$. Thus both $\alpha \circ \Theta'$ and Θ are fixed points of Φ which implies that they are equal.

The relationship between Θ' and $\mathfrak{D}$ can be elegantly expressed using the following mapping.

DEFINITION 2.13
We define $\sim: (L_T \to Cont \to \bar{P}) \to (Par \to \bar{P})$ as follows. We denote, for $F \in L_T \to Cont \to \bar{P}$, $\sim(F)$ by $\tilde{F}$ and put

$$\tilde{F} = \lambda \rho \in Par \cdot (F(s_1)(\{\epsilon\}) \| \cdots \| F(s_n)(\{\epsilon\})),$$

with $\rho = \langle s_1, \ldots, s_n \rangle$.

A simple consequence, using the associativity of $\|$, of this definition is: $\tilde{F}(\rho \hat{} \tau) = \tilde{F}(\rho) \| \tilde{F}(\tau)$. If the function $\tilde{F}$ takes a parallel statement $\langle s_1, \ldots, s_n \rangle$ as an argument, then the F values of all the substatements s_i supplied with the empty continuation $\{\epsilon\}$ are computed and next composed in parallel.
Now we can prove that $\Theta' = \tilde{\mathfrak{D}}$. It is a corollary of the following

THEOREM 2.14: $\Phi'(\tilde{\mathfrak{D}}) = \tilde{\mathfrak{D}}$

PROOF
The proof uses induction on the structure of parallel statements. We treat one typical case, leaving the other ones to the reader. Consider $\rho \hat{} \pi \in Par$ and suppose $\rho \neq \langle E, \ldots, E \rangle$ and $\pi \neq \langle E, \ldots, E \rangle$. Suppose we already know that $\Phi'(\tilde{\mathfrak{D}})(\rho) = \tilde{\mathfrak{D}}(\rho)$ and $\Phi'(\mathfrak{D})(\pi) = \tilde{\mathfrak{D}}(\pi)$. We show: $\Phi'(\tilde{\mathfrak{D}})(\rho \hat{} \pi) = \tilde{\mathfrak{D}}(\rho \hat{} \pi)$.

$$\Phi'(\tilde{\mathfrak{D}})(\rho \hat{} \pi) = \bigcup \{a \cdot \tilde{\mathfrak{D}}(\rho') : \rho \hat{} \pi - a \to \rho'\}$$

$= $ [definition of $\rightarrow$ (2.3 (5) and (6))]

$\bigcup \{a \cdot \tilde{\mathcal{D}}(\rho' \hat{} \pi) : \rho - a \rightarrow \rho'\} \cup \bigcup \{a \cdot \tilde{\mathcal{D}}(\rho \hat{} \pi') : \pi - a \rightarrow \pi'\} \cup$

$\bigcup \{\tau \cdot \tilde{\mathcal{D}}(\rho' \hat{} \pi') : \rho - c \rightarrow \rho' \wedge \pi - \bar{c} \rightarrow \pi'\}$

$= $ [definition $\sim$]

$\bigcup \{a \cdot (\tilde{\mathcal{D}}(\rho') \| \tilde{\mathcal{D}}(\pi)) : \rho - a \rightarrow \rho'\} \cup \bigcup \{a \cdot (\tilde{\mathcal{D}}(\rho) \| \tilde{\mathcal{D}}(\pi')) : \pi - a \rightarrow \pi'\} \cup$

$\bigcup \{\tau \cdot (\tilde{\mathcal{D}}(\rho') \| \tilde{\mathcal{D}}(\pi')) : \rho - c \rightarrow \rho' \wedge \pi - \bar{c} \rightarrow \pi'\}$

$= $ [definitions $\perp\!\!\!\perp$ and $|$]

$(\bigcup \{a \cdot \tilde{\mathcal{D}}(\rho') : \rho - a \rightarrow \rho'\} \perp\!\!\!\perp \tilde{\mathcal{D}}(\pi)) \cup (\bigcup \{a \cdot \tilde{\mathcal{D}}(\pi') : \pi - a \rightarrow \pi'\} \perp\!\!\!\perp \tilde{\mathcal{D}}(\rho)) \cup$

$(\bigcup \{c \cdot \tilde{\mathcal{D}}(\rho') : \rho - c \rightarrow \rho'\} \mid \bigcup \{\bar{c} \cdot \tilde{\mathcal{D}}(\pi') : \pi - \bar{c} \rightarrow \pi'\})$

$= (\Phi'(\tilde{\mathcal{D}})(\rho) \perp\!\!\!\perp \tilde{\mathcal{D}}(\pi)) \cup (\Phi'(\tilde{\mathcal{D}})(\pi) \perp\!\!\!\perp \tilde{\mathcal{D}}(\rho)) \cup (\Phi'(\tilde{\mathcal{D}})(\rho) \mid \Phi'(\tilde{\mathcal{D}})(\pi))$

$= $ [induction]

$(\tilde{\mathcal{D}}(\rho) \perp\!\!\!\perp \tilde{\mathcal{D}}(\pi)) \cup (\tilde{\mathcal{D}}(\pi) \perp\!\!\!\perp \tilde{\mathcal{D}}(\rho) \cup (\tilde{\mathcal{D}}(\rho) \mid \tilde{\mathcal{D}}(\pi))$

$= \tilde{\mathcal{D}}(\rho) \| \tilde{\mathcal{D}}(\pi)$

$= \tilde{\mathcal{D}}(\rho \hat{} \pi)$ □

COROLLARY 2.15: $\mathcal{O}' = \tilde{\mathcal{D}}$

Combining Corollaries 2.12 and 2.15 now yields the main theorem of this section.

MAIN THEOREM 2.16: $\mathcal{O} = \alpha \circ \tilde{\mathcal{D}}$

COROLLARY 2.17: $\forall s \in L_T [\mathcal{O}[\![<s>]\!] = \alpha(\mathcal{D}[\![s]\!](\{\epsilon\}))]$.

3. THE LANGUAGE POOL

In this paper, we compare different semantic models of a language that we call POOL: Parallel Object-Oriented Language. It is a simplified version of a language called POOL-T, which is defined in [Am85]. (For an account of the design considerations for POOL-T see [Am86] and [Am87].) The simplification is two-fold. First, we omitted certain language constructs from POOL-T (such as the select statement and the method call) as well as some of the protection mechanisms offered by the definition of classes (such as different classes having different (instances of) variables and method definitions). We have done this in order to make life somewhat easier: the semantic definitions are shorter and so are the proofs of the theorems. We feel justified in doing so, since it is straightforward to extend the approach of this paper to the full language. Secondly, we give an abstract syntactic description of POOL which is a simplified version of the formal description of POOL-T.

A POOL program describes the behavior of a whole system in terms of its constituents, *objects*. Objects contain some internal data, and some procedures that act on these data (these are called *methods* in the object-oriented jargon). Objects are entities of a dynamic nature: they can be created dynamically, their internal data can be modified, and they have an internal activity of their own. At the same time they are units of protection: the internal data of one object are not directly accessible for other objects.

An object uses *variables* (more specifically: instance variables) to store its internal data. Each

variable can contain the *name* of an object (another object, or, possibly, the object under consideration itself). An assignment to a variable can make it refer to an object different from the object referred to before. The variables of one object cannot be accessed directly by other objects. They can only be read and changed by the object itself.

Objects can interact by sending *messages* to each other. A message is a request for the receiver to execute a certain method. Messages are sent and received explicitly. In sending a message, the sender mentions the destination object, the method to be executed, and possibly a parameter (which is again an object name) to be passed to this method. After this, its activity is suspended. The receiver can specify the set of methods that will be accepted, but it can place no restrictions on the identity of the sender or on the parameters of messages. If a message arrives specifying an appropriate method, the method is executed with the parameters contained in the message. Upon termination, this method delivers a result (an object name), which is returned to the sender of the message. The latter then resumes its own execution. Note that this form of communication strongly resembles the rendez-vous mechanism of Ada ([ANSI83]).

A method can access the variables of the object by which it is executed (the receiver of a message). Furthermore, it has a formal parameter, which is initialized to the actual parameter specified in the message.

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

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

At this point, it might be useful to emphasize the distinction between an object and its name. Objects are intuitive entities as described above. In this paper, there will appear no mathematical construction that directly models a single object with all its dynamic properties (although it would be interesting to see a semantics which does this). Object names, on the other hand, are modeled explicitly as elements of some abstract set *Obj*. Object names are only *references* to objects. On its own, an object name gives little information about the object it refers to. In fact, object names are just sufficient to distinguish the individual objects from each other. Note that variables and parameters contain object names, and that expressions result in object names, not objects. If in the sequel we speak, for example, of "the object α", we hope the reader will understand that the object with name α is meant.

Now we describe the (abstract) syntax of the language POOL. We assume that the following sets of syntactic elements are given:

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

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

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

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

DEFINITION 3.1 (Expressions, statements, units)
We define the set of expressions $(e \in) L_E$ and the set of statements $(s \in) L_S$ by:

$$e ::= x \mid u \mid e_1!m(e_2) \mid \textbf{new}(C) \mid s;e \mid \textbf{self}$$

$$s ::= x \leftarrow e \mid u \leftarrow e \mid \textbf{answer}\ m \mid s_1;s_2 \mid \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi} \mid \textbf{do}\ e\ \textbf{then}\ s\ \textbf{od}.$$

The set $(U \in) Unit$ of units is defined by

$$U ::= < (C_1 \Leftarrow s_1, \ldots, C_n \Leftarrow s_n), (m_1 \Leftarrow <u_1, e_1>, \ldots, m_k \Leftarrow <u_m, e_k>) >.$$

We write $C \Leftarrow s \in U$ if there exists an i such that $C_i = C$ and $s_i = s$. Similarly, we write $m \Leftarrow < u, e > \in U$.

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

The next kind of expression is a send expression. Here, e_1 is the destination object, to which the message will be sent, m is the method to be invoked, and e_2 is the parameter. When a send expression is evaluated, the destination expression and the parameter expression are evaluated successively. Next, the message is sent to the destination object. When this object answers the message, the corresponding method is executed, that is, the formal parameter is initialized to the name of the object in the message, and the expression in the method definition is evaluated. The value which results from this evaluation is sent back to the sender of the message and this will be the value of the send expression.

A new-expression indicates that a new object is to be created, an instance of the indicated class. Its 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 name of the object that is executing this expression.

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

An answer statement indicates that a message is to be answered. The object executing the answer statement waits until a message arrives with a method name that is specified by the answer statement. Then it executes the method (after initializing the formal parameter). The result of the method is sent back to the sender of the message, and the answer statement terminates.

Sequential composition, conditionals and loops have the usual meaning.

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

The relationship between POOL and POOL-T is the following: POOL is obtained from POOL-T via two successive simplifications. First, certain language constructs from POOL-T are omitted (like the select statement) as well as some of the protection mechanisms in POOL-T, which are offered by the definition of classes (such as different classes having different variables and method definitions). Secondly, some syntactical simplifications are performed and some context information is omitted (POOL-T is a statically typed language whereas POOL is not). The reason for making the first simplification is simply lack of space, to which should be added the consideration that it would be straightforward to extend our results to the full language. The sole reason for making the second simplification is that POOL-T is a practical programming language, for which readability, among others, is more important than syntactic simplicity. Therefore, it is convenient to take a simplified language, POOL, as the semantic core of POOL-T.

If one compares the version of POOL described in this paper with the one given in [ABKR86(a)] and [ABKR86(b)], some minor differences can be observed. (For example, in the send expression of definition 3.1 above only one parameter can be specified whereas in the definitions of the papers mentioned an arbitrary number of parameters is allowed.) However, it can easily be seen that it is straightforward to adapt the definitions and proofs given in this paper such that they apply to the version of POOL occurring in [ABKR86(a)] and [ABKR86(b)].

4. An operational semantics for POOL

In this section we give the definition of an operational semantics for POOL, which is a modified version of the one given in [ABKR86(a)]. (At the end of this section, we shall compare both models in some detail.) It is based on a *transition relation* and will be defined as the fixed point of a suitable contraction. For this purpose, we introduce a number of syntactic and semantic notions.

First of all, we introduce the set of objects.

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

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

such that $\nu(X) \notin X$, for every finite $X \subseteq AObj$. Given a set X of object names, the function ν yields a new name not in X.
Further we define

$$Obj = AObj \cup SObj,$$

where $SObj$ is the set of so-called *standard* objects, to be introduced in Appendix III.

A possible example of such a set $AObj$ and function ν could be obtained by setting:

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

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

In POOL, a few standard classes, the instances of which are called standard objects, are predefined; examples are the classes of booleans and integers. The semantic treatment of these standard objects is somewhat different from the way the active objects (which are created during the execution of a POOL program) are treated. Because we want to formulate our semantic models as concisely as possible in order to focus on the correctness proof, the standard objects are treated in an appendix (III).

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

Definition 4.2 ($L_{E'}$, $L_{S'}$)
Let $(e \in) L_{E'}$ and $(s \in) L_{S'}$ be defined by

$$e ::= x \mid u \mid e_1!m(e_2) \mid \textbf{new}(C) \mid s;e \mid \textbf{self} \mid \alpha \mid (e, \phi)$$

$$s ::= x \leftarrow e \mid u \leftarrow e \mid \textbf{answer}\ m \mid s_1;s_2 \mid \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi} \mid \textbf{do}\ e\ \textbf{then}\ s\ \textbf{od} \mid$$
$$\textbf{release}(\beta, s) \mid (e, \psi)$$

with $\alpha, \beta \in AObj$, $\phi \in L_{PE}$ and $\psi \in L_{PS}$. Here the sets of *parameterized expressions* $(\phi \in) L_{PE}$ and *parameterized statements* $(\psi \in) L_{PS}$ are given by

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

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

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

Let us explain the new syntactic constructs. In addition to what we already had in L_E, an expression $e \in L_{E'}$ can be an *active* object α or a pair (e, ϕ) of an expression e and a parameterized

expression ϕ. The latter will be executed as follows: First the expression e is evaluated, then the result β is substituted in ϕ and $\phi(\beta)$ is executed. As new statements we have release statements **release**(β,s) and parameterized statements (e,ϕ). If the statement **release**(β,s) is executed, the active object β will start executing the statement s (in parallel to the objects that are already executing). The release statement will be used in the description of the communication between two objects (see definition 4.8 below). The interpretation of (e,ψ) is similar to that of (e,ϕ).

DEFINITION 4.3 (Empty statement)
The set $L_{S'}$, as given in the definition above, is extended with a special element E, denoting the *empty statement*. This extended set is again called $L_{S'}$. Note that we do *not* have elements like $s;E$ or **do** e **then** E **od** in $L_{S'}$. (There is, however, one exception: We *do* allow E in **if** e **then** s **else** E **fi**, which is needed in definition 4.8(A8) below.)

DEFINITION 4.4 (States)
The set of states $(\sigma\in)\Sigma$ is defined by

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

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

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

DEFINITION 4.5 (Labelled statements)
The set of *labelled statements* $((\alpha, s)\in)LStat$ is given by $LStat=AObj\times L_{S'}$.

A labelled statement (α, s) should be interpreted as a statement s which is going to be executed by the active object α.

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

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

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

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

DEFINITION 4.6 (Configurations)
The set of configurations $(\rho\in)Conf$ is given by

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

We also introduce:

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

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

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

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

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

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

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

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

DEFINITION 4.7 (Transition labels)
The set of *transition labels* $(\lambda \in)\Lambda$ is given by

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

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

Now we are ready to define a transition relation for POOL.

DEFINITION 4.8 (Transition relation)
Let $U \in Unit$. We define a *labelled transition relation*

$$-U\rightarrow \subseteq \text{Conf} \times \Lambda \times \text{Conf}'.$$

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

$$\rho_1\ -U, \lambda\rightarrow \rho_2.$$

Such a transition reflects a possible execution step of type λ of the configuration ρ_1, yielding a new configuration ρ_2. The relation $-U\rightarrow$ is defined as the smallest relation satisfying the following properties:

Axioms

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

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

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

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

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

(A6) $<\{(\alpha, \textbf{answer}\ m)\}, \sigma> -U, (\alpha ? m)\rightarrow <\{(\alpha, E)\}, \sigma>$

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

Rules

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

then $<\{(\alpha, z \leftarrow e)\},\sigma> \; -U, \lambda \rightarrow \rho$, for $z \in IVar \cup TVar$.

(R2) If $<\{(\alpha, s)\},\sigma> \; -U, \lambda \rightarrow <\{(\alpha, s')\} \cup X,\sigma'>$,

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

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

If $<\{(\alpha, s)\},\sigma> \; -U, \lambda \rightarrow <\{(\alpha, \psi)\} \cup X,\sigma'>$,

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

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

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

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

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

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

then $<\{(\alpha,$ **if** e **then** s_1 **else** s_2 **fi** $)\},\sigma> \; -U, \lambda \rightarrow \rho$.

(Here s_2 is allowed to be E.)

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

then $<\{(\alpha, (e_1 ! m(e_2), \psi))\},\sigma> -U, \lambda \rightarrow \rho$.

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

(R8) If $<\{(\alpha, (e, \lambda u \cdot (\phi(u), \psi)))\},\sigma> \; -U, \lambda \rightarrow \rho$, then $<\{(\alpha, ((e,\phi),\psi))\},\sigma> \; -U, \lambda \rightarrow \rho$.

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

If $<\{(\alpha, \psi(\alpha))\},\sigma> \; -U, \lambda \rightarrow \rho$, then $<\{(\alpha,$ **(self**$,\psi))\},\sigma> \; -U, \lambda \rightarrow \rho$.

(R10) If $<X,\sigma> \; -U, \lambda \rightarrow <X',\sigma'>$, then $<X \cup Y,\sigma> \; -U, \lambda \rightarrow <X' \cup Y,\sigma'>$.

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

$<Y,\sigma> \; -U, \beta_1 ? m \rightarrow <\{(\beta_1, s)\} \cup Y',\sigma>$,

then $<X \cup Y,\sigma> \; -U, \tau \rightarrow$

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

where $\sigma' = \sigma\{\beta_2 / \beta_1, u_m\}$, and $m \Leftarrow\!= <u_m, e_m> \in U$.

(End of definition.)

The general scheme for the evaluation of an expression is very similar to the approach taken in [AB88]. Expressions always occur in the context of a (possibly parameterized) statement, such as $x \leftarrow e$. A statement containing e as a subexpression is transformed into a pair (e,ψ) of the expression e and a parameterized statement ψ by application of one of the rules. (In our example, $x \leftarrow e$ becomes $(x, \lambda u \cdot x \leftarrow u)$ by an application of (R1).) Then e is evaluated, using the axioms and rules, and results in some value $\beta' \in Obj$. (Applying (A1) transforms $(x, \lambda u \cdot x \leftarrow u)$ of our example into $(\beta', \lambda u \cdot x \leftarrow u)$, for some $\beta' \in Obj$.) Next, an application of (R9) will put the resulting object β' back into the original context ψ (yielding $x \leftarrow \beta'$ in our example). Finally, the statement $\psi(\beta')$ is further evaluated by using the axioms and the rules. (The evaluation of $x \leftarrow \beta'$ results, by using (A6), in a transformation of the state.)

Let us briefly explain some of the axioms and rules above.

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

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

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

(Note that we have not incorporated any transitions for the standard objects; this is done in Appendix III.)

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

DEFINITION 4.9 (Semantic universe P)
Let $(w\in)\Sigma_\partial^\infty = \Sigma^* \cup \Sigma^\omega \cup \Sigma^*\cdot\{\partial\}$, the set of *streams*. We define

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

where $\mathcal{P}_{ncompact}(\Sigma_\partial^\infty)$ is the set of all non-empty compact subsets of Σ_∂^∞, and the symbol ∂ denotes deadlock. The set P is a complete metric space when supplied with the usual metric (see definition A.6).

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

DEFINITION 4.10 (Operational semantics for POOL)
We define the operational semantics of finite subsets of labelled statements. Let, for a unit $U\in Unit$, the function

$$\Phi_U\colon (\mathcal{P}_{fin}(LStat)\to P)\to(\mathcal{P}_{fin}(LStat)\to P)$$

be given, for $F\in\mathcal{P}_{fin}(LStat)\to P$ and $X\in\mathcal{P}_{fin}(LStat)$, by:

$$\Phi_U(F)(X)=\lambda\sigma\cdot\begin{cases}\{\epsilon\} & \text{if } \forall\alpha\forall s\ [(\alpha,s)\in X\Longrightarrow s=E]\\ \{\partial\} & \text{if } \neg<X,\sigma>\ -U,\ \tau\to \text{ and } \exists\alpha\exists s\ [s\neq E\wedge(\alpha,s)\in X]\\ \bigcup\{\sigma'\cdot F(X')(\sigma')\colon <X,\sigma>-U,\ \tau\to <X',\sigma'>\} & \text{otherwise,}\end{cases}$$

where

$$<X,\sigma> \ -U, \tau\rightarrow \ = \exists X' \exists \sigma' \ [<X,\sigma> \ -U, \tau\rightarrow <X',\sigma'>].$$

Now the operational semantics Θ_U: $\mathcal{P}_{fin}(LStat)\rightarrow P$ is given as

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

It is straightforward to prove that Φ_U is a contraction and thus has a unique fixed point.

The definition of Φ_U is very similar to the definition of Φ in the previous section (definition 2.5). If, for a given $X\in\mathcal{P}_{fin}(LStat)$ and $\sigma\in\Sigma$, we have that $\neg<X,\sigma> \ -U, \tau\rightarrow$, then no computation steps, which are indicated by τ, are possible from $<X,\sigma>$. The transitions that *are* possible are of the form

$$<X,\sigma> \ -U, (\alpha,\beta_1!m(\beta_2)) \rightarrow\rho \ , \ \text{ or } <X,\sigma> \ -U, (\alpha?m) \rightarrow\rho' \ ,$$

denoting attempts of a single object α to perform a communication action without any matching object being present. This is an instance of deadlock and therefore we here have: $\Theta_U[\![X]\!](\sigma)=\{\partial\}$. On the other hand, for every transition

$$<X,\sigma> \ -U, \tau \rightarrow <X',\sigma'>$$

the set $\Theta_U[\![X]\!](\sigma)$ includes the set $\sigma'\cdot \Theta_U[\![X']\!](\sigma')$, in which the transformed state σ' is concatenated with the operational meaning of X' in state σ'.

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

DEFINITION 4.11 (Operational semantics of a unit)
Let $[\![\ \cdots \ ]\!]_\Theta$: $Unit\rightarrow P$ be given, for a unit $U=<(\ldots, C_n\Leftarrow s_n), \ldots >$, by

$$[\![U]\!]_\Theta = \Theta_U[\![\{(\nu(\varnothing), s_n)\}]\!].$$

The execution of a unit $U=<(\ldots, C_n\Leftarrow s_n), \ldots >$ consists of the creation of an object of class C_n and the execution of its body. Its name is given by $\nu(\varnothing)$, the name of the first object.

Comparison with [ABKR86(a)]

In [ABKR86(a)], an operational semantics for POOL is defined which differs from Θ_U in a number of respects: There, a transition relation without labels is used whereas we have a labelled transition relation here; further, in [ABKR86(a)] communication is modeled by means of a so-called *wait* statement as opposed to the release statement we use here; also our use of parameterized expressions and statements is new. All these differences can be seen as minor variations of the semantic definitions and are motivated by the main goal of this paper, which is to relate the operational semantics with the denotational one. There is one major difference, however, which we shall treat in some detail: In definition 4.10 of this paper, Θ_U is given as the fixed point of a contraction, whereas in [ABKR86(a)] the operational semantics is defined in terms of finite and infinite sequences of transitions. In order to show the equivalence of both approaches, we now define an operational semantics Θ^*_U in the style of [ABKR86(a)], for which we next shall prove that it equals Θ_U.

DEFINITION 4.12 (Alternative operational semantics)
Let, for a $U\in Unit$, the function

$$\Theta^*_U: \mathcal{P}_{fin}(LStat)\rightarrow P$$

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

$$w\in\Theta^*_U[\![X]\!](\sigma)$$

if and only if one of the following conditions is satisfied:
(1) $w=\sigma_1\cdots\sigma_n$ and there exist $X_1, \ldots, X_n$ such that

$$<X,\sigma> -U,\tau\rightarrow <X_1,\sigma_1> -U,\tau\rightarrow \cdots -U,\tau\rightarrow <X_n,\sigma_n> \text{ and } \forall(\alpha,s)\in X_n\,[s=E]$$

(2) $w=\sigma_1\sigma_2\cdots$ and there exist $X_1,X_2,\ldots$ such that

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

(3) $w=\sigma_1\cdots\sigma_n\cdot\partial$ and there exist $X_1,\ldots,X_n$ such that

$$<X,\sigma> -U,\tau\rightarrow <X_1,\sigma_1> -U,\tau\rightarrow \cdots -U,\tau\rightarrow <X_n,\sigma_n>$$

and $\exists(\alpha,s)\in X_n\,[s\neq E]$ and $\neg<X_n,\sigma_n>-U,\tau\rightarrow$

It is not straightforward that the sets $\Theta^*_U[\![X]\!](\sigma)$ are in P, that is, that they are compact; we prove this fact in the following

LEMMA 4.13 *(Compactness of Θ^*_U): For every $X\in\mathcal{P}_{fin}(LStat)$ and $\sigma\in\Sigma$: $\Theta^*_U[\![X]\!](\sigma)$ is compact.*

PROOF

Let $(w_i)_i$ be a sequence of words in $\Theta^*_U[\![X]\!](\sigma)$ ($\subseteq\Sigma^\infty_\partial$), say

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

We show that $(w_i)_i$ has a converging subsequence with its limit in $\Theta^*_U[\![X]\!](\sigma)$. Assume for simplicity that all words w_i are infinite. Since $w_i\in\Theta^*_U[\![X]\!](\sigma)$, for every i, there exist infinite transition sequences such that

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

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

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

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

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

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

$$<X^1_{f(i)},\ \sigma^1_{f(i)}> = <X_1,\sigma_1>.$$

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

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

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

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

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

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

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

Then we have

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

Thus we have constructed a converging subsequence of $(w_i)_i$ with its limit in $\Theta^*_U[\![X]\!](\sigma)$. (In case the

words w_i are not all infinite a similar argument can be given.)

It is not difficult to show that $\Theta_U = \Theta_U^*$:

THEOREM 4.14: $\Theta_U = \Theta_U^*$

PROOF

We prove that Θ_U^* is also a fixed point of Φ_U, from which the equality follows. Let $X \in \mathcal{P}_{fin}(LStat)$ such that $\exists(\alpha,s) \in X\ [s \neq E]$, let $\sigma \in \Sigma$ and let $w \in \Sigma_\partial^\infty$. If $w = \partial$ then

$$w \in \Phi_U(\Theta_U^*)(X)(\sigma) \Leftrightarrow w \in \Theta_U^*[\![X]\!](\sigma).$$

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

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

So we see: $\Theta_U^* = \Phi_U(\Theta_U^*)$.

5. A DENOTATIONAL SEMANTICS FOR POOL

The denotational semantics that is defined in this section was already presented (in a slightly different form) in [ABKR86(b)]. (For a comparison of the two models we refer the reader to the end of this section.)

Our denotational model has a so-called *domain* (a solution of a reflexive domain equation) for its semantic universe. In [BZ82] it was first described how to solve these equations in a metric setting. Then, in [AR88], this approach was generalized in order to deal with equations of the form: $P \cong \cdots P \rightarrow \cdots$, a case that was not covered by [BZ82]. For a quick overview of the main results of [AR88], the reader might want to read section 2 of [ABKR86(b)].

Further, our model is based on the use of *continuations*. For an extensive treatment of continuations and expression continuations, which we shall use as well, we refer to [Go79].

We start with the definition of a domain $\bar{P}$, the elements of which we shall call *processes* from now on.

DEFINITION 5.1 (Semantic process domain $\bar{P}$)

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

$$P \cong \{p_0\} \cup id_{1/2}(\Sigma \rightarrow \mathcal{P}_{compact}(Step_P)),$$

where $(\pi,\rho \in)Step_P$ is

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

with

$$\begin{aligned} Comp_P &= \Sigma \times P, \\ Send_P &= Obj \times MName \times Obj \times (Obj \rightarrow P) \times P, \\ Answer_P &= Obj \times MName \times (Obj \rightarrow (Obj \rightarrow P) \rightarrow^1 P). \end{aligned}$$

(The sets $\{p_0\}$, Σ, Obj, and $MName$ become complete ultra-metric spaces by supplying them with the discrete metric.)

In [AR88], it is described how to find for such an equation a solution which is unique up to isomorphy. Let us try to explain intuitively the intended interpretation of the domain P. First, we observe that in the equation above the subexpression $id_{\frac{1}{2}}$ is necessary only to guarantee that the equation is solvable by defining a contracting functor on $\mathcal{C}$, the category of complete metric spaces (see Appendix I). For a, say, more operational understanding of the equation it does not matter.

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

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

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

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

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

$$\pi = <\alpha, m, \beta, f, p>,$$

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

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

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

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

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

DEFINITION 5.2 (Parallel composition)
Let $\| : \overline{P} \times \overline{P} \rightarrow \overline{P}$ be such that it satisfies the following equation:

$$p \| q = \lambda\sigma \cdot ((p(\sigma) \mathbin{\mathbb{L}} q) \cup (q(\sigma) \mathbin{\mathbb{L}} p) \cup (p(\sigma) \mid_\sigma q(\sigma))),$$

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

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

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

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

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

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

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

We observe that this definition is self-referential, since the merge operator occurs at the right hand side of the definition. For a formal justification of this definition see the appendix of [ABKR86(b)], where the merge operator is given as the unique fixed point of a contraction on $\overline{P} \times \overline{P} \to^1 \overline{P}$.

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

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

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

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

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

$$\mathcal{D}_E : L_E \to AObj \to Cont_E \to^1 \overline{P},$$

$$\mathcal{D}_S : L_S \to AObj \to Cont_S \to^1 \overline{P}$$

where

$$Cont_E = Obj \to \overline{P} \quad \text{and} \quad Cont_S = \overline{P}.$$

Let $s \in L_S$, $\alpha \in AObj$, and $p \in \overline{P}$. The semantic value of the statement s is given by

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

The object name α represents the object that executes s. Secondly, the semantic value of s depends on its so-called *continuation* p: 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.

The semantic value of an expression $e \in L_E$, for an object α and an expression continuation $f \in Cont_E$, is given by

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

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

Please note the difference between the notions of *resumption* and *continuation*. A resumption is a part of a semantic step $\pi \in Step_{\bar{P}}$, indicating the remaining steps to be taken after the current one. A continuation, on the other hand, is an argument to a semantic function. It may appear as a resumption in the result. A good example of this is the definition of $\hat{F}_S(x \leftarrow e)$ (in definition 5.3(S1)) below.

DEFINITION 5.3 (Semantics of expressions and statements)
Let

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

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

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

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

where

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

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

EXPRESSIONS

(E1, instance variable)

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

The value of the instance variable x is 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 then applied to the resulting value.

(E2, temporary variable)

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

(E3, send expression)

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

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

(E4, new-expression)

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

where

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

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

A new object of class C is created. It is called $\nu(\sigma_3)$: the function ν supplied with the set of all object names currently in use yields a name that is not yet being used. The state σ is changed by expanding the set σ_3 with the new name β. The process $F_S(s_C)(\beta)(p_0)$ is the meaning of the body of the new object β with p_0 as a nil continuation. It is composed in parallel with $f(\beta)$, the process resulting from the application of the continuation f to β, the result of the evaluation of this new-expression. We are able to perform this parallel composition because we know from f what should happen after the evaluation of this new-expression, so here the use of continuations is essential.

(E5, sequential composition)

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

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

(E6, **self**)

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

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

STATEMENTS

(S1, assignment to an instance variable)

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

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

(S2, assignment to a temporary variable)

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

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

(S3, answer statement)

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

where

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

with

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

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

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

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

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

Now that we have defined the semantics of send expressions and answer statements let us briefly return to the definition of $\pi|_\sigma\rho$ (definition 5.2). Let $\pi=<\alpha,m,\beta,f,q>$ (the result from the elaboration of a send expression) and $\rho=<\alpha,m,g>$ (resulting from an answer statement). Then $\pi|_\sigma\rho$ is defined as

$$\pi|_\sigma\rho=\{<\sigma,g(\beta)(f)\|q>\}.$$

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

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

The continuation of the execution of m is given by $\lambda\beta'\cdot\lambda\bar{\sigma}\cdot\{<\bar{\sigma}',f(\beta')\|p>\}$, which consists of a state transformation followed by the parallel composition of the continuations f and p. This represents the fact that after the rendez-vous, during which the method is executed, the sender and the receiver of the message can proceed in parallel again. (Of course, the independent resumption q may still be executing at this point.) Moreover, the result β' of the method execution is passed on to the continuation f of the send expression.

(S4, sequential composition)

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

(S5, conditional)

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

(S6, loop statement)

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

(End of definition 5.3.)

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

DEFINITION 5.4 (Denotational semantics of a unit)

We define $[\![\cdots]\!]_{\mathcal{D}}$: $Unit \to \bar{P}$. For a unit $U \in Unit$, with $U = <(\ldots, C_n \Leftarrow s_n), \ldots >$, we set

$$[\![U]\!]_{\mathcal{D}} = \mathcal{D}_S[\![s_n]\!](\nu(\varnothing))(p_0).$$

The execution of a unit always starts with the creation of an object of class C_n and the execution of its body. Therefore, the meaning of a unit U is given by the denotational meaning of s_C, the body of class C_n, supplied with $\nu(\varnothing)$, denoting the name of the first active object, and with p_0, the empty continuation.

Comparison with [ABKR86(b)]

There are some differences between the denotational semantics $<\mathcal{D}_E, \mathcal{D}_S>$ presented here and the denotational semantics given in [ABKR86(b)]: The former model is given as the fixed point of a contraction Ψ_U and does not use so-called *environments* to deal with process creation (**new**(C)) and the meaning of the execution of a method (**answer** m); the latter model is defined without the use of a contraction and *does* use environments. In [ABKR86(b)], the semantics of a unit U is given with the help of a special environment γ_U, which contains information about the class and method definitions in U and is obtained as the fixed point of a suitably defined contraction. Another difference is the way the loop statement is treated: In this paper, the definition of its semantics fits smoothly in the definition of $<\mathcal{D}_E, \mathcal{D}_S>$ as a fixed point. In [ABKR86(b)], a contraction is defined especially for this case.

Another way to express these differences is that the three constructs for recursion present in POOL (i.e., the new expression, the answer and the loop statement) are treated here by means of one fixed point definition, whereas in [ABKR86(b)], environments are used for the first two forms of recursion and a specially defined contraction for the last one. However, we state (without proof) that the two definitions are equivalent: it is straightforward how to translate the one approach into the other.

An additional difference between the denotational semantics of a unit given here and the one presented in [ABKR86(b)] is the presence of a semantic representation of the standard objects in the latter, whereas these are not treated in this section. As mentioned before, we do not treat standard objects now because we want to concentrate on the correctness proof. In order to show, however, that our proof (to be presented in section 7) can also deal with standard objects, we shall extend, in Appendix III, both our operational and our denotational semantics with a semantic representation of standard objects, and prove that the correctness result still holds for these extended models.

6. An intermediate semantics

After having defined an intermediate semantics Θ_U for $\mathcal{P}_{fin}(LStat)$ and a denotational semantics $\mathcal{D}_U$ for L_E and L_S we shall, in the next section, discuss the relationship between the two. As we did in section 2, we shall compare Θ_U and $\mathcal{D}_U$ by relating both to an intermediate semantics Θ_U': $\mathcal{P}_{fin}(LStat) \to \bar{P}$, the definition of which is the subject of this section.

Definition. 6.1 (Intermediate semantics Θ_U')
Let $U \in Unit$. Let Θ_U': $\mathcal{P}_{fin}(LStat) \to \bar{P}$ be given by

$$\Theta_U' = Fixed\ Point\ (\Phi_U'),$$

where

$$\Phi_U'\text{: } (\mathcal{P}_{fin}(LStat) \to \bar{P}) \to (\mathcal{P}_{fin}(LStat) \to \bar{P})$$

is defined, for $F \in \mathcal{P}_{fin}(LStat) \to \bar{P}$ and $X \in \mathcal{P}_{fin}(LStat)$, as follows.

If $\forall\alpha\forall s\ [(\alpha,s)\in X\Rightarrow s=E]$, then $\Phi_U'(F)(X)=p_0$. Otherwise we have

$$\Phi_U'(F)(X)=\lambda\sigma\cdot(C_F\cup S_F\cup A_F)$$

where

$$C_F = \{<\sigma',F(X')>:\ <X,\sigma>\ -U,\ \tau\rightarrow\ <X',\sigma'>\},$$

$$S_F = \{<\beta_1,m,\beta_2,\lambda\beta\cdot F(\{(\alpha,\psi(\beta))\}),\ F(X')>:$$
$$<X,\sigma>\ -U,\ (\alpha,\beta_1!m(\beta_2))\rightarrow\ <\{(\alpha,\psi)\}\cup X',\sigma>\},$$

$$A_F = \{<\alpha,m,g_m>:\ <X,\sigma>\ -U,\ (\alpha?m)\rightarrow\ <\{(\alpha,s)\}\cup X',\sigma>\}$$

with

$$g_m = \lambda\beta\cdot\lambda f\cdot(\lambda\overline{\sigma}\cdot\{<\overline{\sigma}',\mathcal{D}_E[\![e_m]\!](\alpha)(\lambda\beta'\cdot\lambda\hat{\sigma}\cdot\{<\hat{\sigma}',f(\beta')\|F(\{(\alpha,s)\})>\})>\}\ \|\ F(X')),$$

and

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

$$\hat{\sigma}' = \hat{\sigma}\{\overline{\sigma}_2(\alpha)(u_m)/\alpha,u_m\},$$

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

(It is straightforward to show that Φ_U' is a contraction.)

The function Θ_U' differs from the operational semantics Θ_U in two ways. First, its range is the semantic universe $\overline{P}$, which is used for the denotational semantics $\mathcal{D}_U$, instead of P, the semantic universe of Θ_U: For every set $X\in\mathcal{P}_{fin}(LStat)$ the function Θ_U' yields a *process* $\Theta_U'(X)\in\overline{P}$, rather than a function from states to sets of streams of states. Secondly, in addition to the computation steps (indicated by the set C_F above) single-sided communication steps are present in $\Theta_U'(X)$ (indicated by S_F and A_F, for send and answer steps), whereas $\Theta_U(X)$ contains only computation steps. On the other hand, the similarity between the definitions of Θ_U and Θ_U' is obvious: both are based on the transition relation $-U\rightarrow$ for $\mathcal{P}_{fin}(LStat)$.

At first sight, two facts regarding the relation between $\mathcal{O}'_U$ and $\mathcal{D}_U$ can be mentioned. First, they have the same range, that is, the semantic universe $\overline{P}$ of processes, in which single-sided communication actions are visible. Secondly, $\mathcal{D}_U$ is defined compositionally with the use of semantic operators (like the merge $\|$), whereas the definition of Θ_U' is based, as was mentioned above, on the transition relation $-U\rightarrow$.

In the next section the relationship between Θ_U, Θ_U' and $\mathcal{D}_U$ will be formally expressed. Let us, for the time being, try to elucidate the definition of Θ_U' above by explaining what communication steps are present in $\Theta_U'(X)$.

Corresponding with every send transition of the form

$$<X,\sigma>\ -U,\ (\alpha,\beta_1!m(\beta_2))\rightarrow\ <\{(\alpha,\psi)\})\cup X',\sigma>$$

the set $\Theta_U'(X)(\sigma)$, for a state $\sigma\in\Sigma$, contains a send step of the form

$$<\beta_1,\ m,\ \beta_2,\ \lambda\beta\cdot\Theta_U'(\{(\alpha,\psi(\beta))\}),\ \Theta_U'(X')>.$$

Here β_1, m and β_2 indicate that a message specifying the method m with parameter β_2 is sent to the object β_1. The dependent resumption of this send step is $\lambda\beta\cdot\Theta_U'(\{(\alpha,\psi(\beta))\})$: the meaning of the statement that will be executed by α as soon as it receives the result β of the message. The last component of this send step, the independent resumption, consists of $\Theta_U'(X')$, which is the meaning of all the statements executed by objects other than α. Thus it is reflected that these objects need not wait till the message is answered; they may proceed in parallel.

Next, $\Theta_U'(X)(\sigma)$ can contain some answer steps. For every answer transition

$$<X,\sigma> -U,(\alpha?m)\rightarrow <\{(\alpha,s)\}\cup X',\sigma>$$

the set $\Theta_U'(X)(\sigma)$ includes an answer step

$$<\alpha,m,g_m>,$$

with g_m as in the definition above. It indicates that the object α is willing to answer a message specifying the method m, while the resumption g_m indicates what should happen when an appropriate message arrives. This function g_m, when supplied with a parameter β and a dependent resumption f (both to be received from the sending object), consists of the parallel composition of the process $\Theta_U'(X')$ together with the process

$$\lambda\bar{\sigma}\cdot\{<\bar{\sigma}',\ \mathcal{D}_E[\![e_m]\!](\alpha)(\lambda\beta'\cdot\lambda\hat{\sigma}\cdot\{<\hat{\sigma}',\ f(\beta')\|\Theta_U'(\{(\alpha,s)\})>\})>\}\ .$$

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

7. Semantic correctness

We are now ready to establish the main result of this paper. We shall relate the operational semantics Θ_U and the denotational semantics $\mathcal{D}_U$ by first comparing Θ_U and Θ_U', the intermediate semantics defined in the previous section, and next comparing Θ_U' and $\mathcal{D}_U$. These relationships will be formally expressed by means of suitably defined abstraction operations. From this we shall deduce the fact that

$$[\![U]\!]_\Theta = abstr([\![U]\!]_\mathcal{D}),$$

where abstr: $\bar{P}\rightarrow P$ is such an abstraction operation.

Part 1: Comparing Θ_U and Θ_U'

We start with the definition of abstr: $\bar{P}\rightarrow P$, which relates the semantic universes P and $\bar{P}$ of Θ_U and Θ_U'.

Definition 7.1 (Abstraction operation *abstr*)
Let *abstr*: $\bar{P}\rightarrow P$ be defined as follows. We set $abstr(p_0)=\{\epsilon\}$. If $p\in\bar{P}\setminus\{p_0\}$, then

$$abstr(p)=\lambda\sigma\cdot\begin{cases}\{\partial\} & \text{if } p(\sigma)\cap Comp_{\bar{P}}=\varnothing\\ \bigcup\{\sigma'\cdot abstr(p')(\sigma'):\ <\sigma',p'>\in p(\sigma)\} & \text{otherwise,}\end{cases}$$

where $Comp_{\bar{P}}=\Sigma\times\bar{P}$. (Formally, we can define this operation correctly by giving it as the fixed point of a suitably defined contraction on $\bar{P}\rightarrow P$: See Appendix II for an extensive formal treatment of the function *abstr*.)

The function *abstr* transforms a process $p\in\bar{P}$ into a function $abstr(p)\in P=\Sigma\rightarrow\mathcal{P}_{nc}(\Sigma_\partial^\infty)$, which yields for every $\sigma\in\Sigma$ a set $abstr(p)(\sigma)$ of streams. (If one regards the process p as a tree-like structure, these streams can be considered the branches of p.) If $p(\sigma)\cap Comp_{\bar{P}}=\varnothing$, that is, if $p(\sigma)$ is empty or contains only single-sided communication steps, then we have a case of deadlock because, operationally, single-sided communication is not possible. Therefore we then have: $abstr(p)(\sigma)=\{\partial\}$. If, however, $p(\sigma)$ does contain a computation step $<\sigma',p'>$, then we have: $\sigma'\cdot abstr\ (p')(\sigma')\subseteq abstr\ (p)(\sigma)$.

The changed state σ' is concatenated with $abstr(p')(\sigma')$, in which σ' is passed through to *abstr* applied to p', the resumption of $<\sigma',p'>$. Thus the effect of different state transformations occurring subsequently in p is accumulated.

Next, we use the operation *abstr* to relate Φ_U and Φ_U'.

THEOREM 7.2 *(Relating Φ_U and Φ_U'):* $\forall F \in \mathcal{P}_{fin}(LStat) \to \bar{P}\ [\Phi_U(abstr \circ F) = abstr \circ (\Phi_U'(F))]$

PROOF
Let $F \in \mathcal{P}_{fin}(LStat) \to \bar{P}$, $X \in \mathcal{P}_{fin}(LStat)$ and $\sigma \in \Sigma$. Suppose $\neg \forall \alpha\, \forall s\ [(\alpha,s) \in X \Rightarrow s = E]$. If $\neg <X,\sigma> -U, \tau \to$, then

$$\Phi_U(abstr \circ F)(X)(\sigma) = \{\partial\}$$
$$= abstr(\Phi_U'(F)(X))(\sigma)$$

since $\Phi_U'(F)(X)(\sigma) \cap Comp_{\bar{P}} = \varnothing$. (Recall that $Comp_{\bar{P}} = \Sigma \times \bar{P}$.) If $<X,\sigma> -U, \tau \to$ we have

$$\Phi_U(abstr \circ F)(X)(\sigma) = \bigcup \{\sigma' \cdot (abstr \circ F)(X')(\sigma') : <X,\sigma> -U, \tau \to <X',\sigma'>\}$$
$$= \bigcup \{\sigma' \cdot (abstr(F)(X'))(\sigma') : <X,\sigma> -U, \tau \to <X',\sigma'>\}$$
$$= [\text{see definition 6.1}]$$
$$abstr(\lambda\sigma \cdot C_F)(\sigma)$$
$$= abstr(\lambda\sigma \cdot (C_F \cup S_F \cup A_F))(\sigma)$$
$$= abstr(\Phi_U'(F)(X))(\sigma)$$
$$= (abstr \circ \Phi_U'(F))(X)(\sigma).$$

Since Φ_U and Φ_U' are contractions and thus have unique fixed points, the following corollary is straightforward:

COROLLARY 7.3: $\Theta_U = abstr \circ \Theta_U'$

Part 2. Comparing Θ_U' and $\mathcal{D}_U$.
In order to compare $\Theta_U' : \mathcal{P}_{fin}(LStat) \to \bar{P}$ and $\mathcal{D}_U \in Q_E \times Q_S$ we define an extension of $\mathcal{D}_U$ $(= <\mathcal{D}_E, \mathcal{D}_S>)$ in two steps. First, we define $\mathcal{D}_U'$ $(= <\mathcal{D}_E', \mathcal{D}_S'>) \in Q_E' \times Q_S'$, with

$$Q_E' = L_E' \to AObj \to Cont_E \to^1 \bar{P},$$
$$Q_S' = L_S' \to AObj \to Cont_S \to^1 \bar{P},$$

which is as $\mathcal{D}_U$ but with the extended sets of expressions and statements, L_E' and L_S', for its domain. (Recall that L_S' is used in the definition of $LStat = AObj \times L_S'$.) Next, we extend $\mathcal{D}_U'$ to $\mathcal{D}_U^* : \mathcal{P}_{fin}(LStat) \to \bar{P}$, which takes sets of labelled statements for its arguments.

DEFINITION 7.4 ($\mathcal{D}_U'$)
Let $\Psi'_U : (Q_E' \times Q_S') \to (Q_E' \times Q_S')$ be defined as follows. For $F = <F_E, F_S>$, we denote $\Psi'_U(F)$ by $\bar{F} = <\bar{F}_E, F_S>$. Let $\alpha \in AObj$, $p \in Cont_S = \bar{P}$ and $f \in Cont_E = Obj \to \bar{P}$. Now $\bar{F}$ is defined similarly to $\Psi_U(F)$ (definition 5.3) but with the following clauses added:

$\bar{F}_E(\beta)(\alpha)(f) = f(\beta)$, for $\beta \in Obj \supseteq AObj$,
$\bar{F}_E((e,\varphi))(\alpha)(f) = \bar{F}_E(e)(\alpha)(\lambda\beta \cdot \bar{F}_E(\varphi(\beta))(\alpha)(f))$
$\bar{F}(E)(\alpha)(p) = p$

$$\bar{F}_S\ (\textbf{release}(\beta,s))(\alpha)(p) = p \| \bar{F}_S(s)(\beta)(p_0)$$

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

Finally, we set

$$\mathcal{D}_U' = <\mathcal{D}_E', \mathcal{D}_S'>$$
$$= \text{Fixed Point } (\Psi'_U).$$

The meaning of (e,φ) is obtained by first evaluating the expression e, then substituting the result β into the parameterized expression φ and finally evaluating the expression $\varphi(\beta)$. The interpretation of $\mathcal{D}_S'[\![(e,\psi)]\!]$ is similar. In $\mathcal{D}_S'[\![\textbf{release}(\beta,s)]\!](\alpha)(p)$, the meaning of the statement s (when executed by the object β and with the empty continuation p_0) is computed and composed in parallel with the process p, the continuation of the release statement.

DEFINITION 7.5 ($\mathcal{D}_U^*$)
Let $\mathcal{D}_U^*: \mathcal{P}_{fin}(LStat) \to \bar{P}$ be given by

$$\mathcal{D}_U^* = (\tilde{\mathcal{D}_U'}),$$

where $\sim: (Q_E \times Q_S) \to (\mathcal{P}_{fin}(LStat) \to \bar{P})$ is defined as follows: If $F = <F_E, F_S>$, then $\sim(F)$, here being denoted by F is given by

$$\tilde{F}(\{(\alpha_1,s_1), \ldots, (\alpha_k,s_k)\}) = F_S(s_1)(\alpha_1)(p_0) \| \cdots \| F_S(s_k)(\alpha_k)(p_0).$$

(We put $\tilde{F}(\varnothing) = p_0$.)

Note that we have: $\tilde{F}(X \cup Y) = \tilde{F}(X) \| \tilde{F}(Y)$.

The omission of parentheses in the parallel composition above is justified by the fact that $\|$ is associative.

Given a finite set X of labelled statements (α_i,s_i), the value of $\mathcal{D}_U^*(X)$ is obtained by first computing the semantics of every labelled statement $(\alpha_i,s_i) \in X$. This is given by $\mathcal{D}_S[\![s_i]\!](\alpha_i)(p_0)$, where the label α_i indicates the name of the object executing the statement and where p_0 indicates that after s_i nothing remains to be done. Next, all the resulting processes are composed in parallel.

Now that we have extended the domain of $\mathcal{D}_U$ to $\mathcal{P}_{fin}(LStat)$ we are ready to prove the fact that $\mathcal{D}_U^* = \Theta_U'$. It is a straightforward corollary of theorem 7.7 below. The proof of this theorem makes use of the following

LEMMA 7.6
For all $\alpha \in AObj$ and $\psi \in L_{PS}$ we have:

$$\forall\beta\ [\Phi_U'(\mathcal{D}_U^*)(\{(\alpha,\psi(\beta))\}) = \mathcal{D}_U^*(\{(\alpha,\psi(\beta))\})] \Rightarrow$$
$$\forall e \in L_E'\ [\Phi_U'(\mathcal{D}_U^*)(\{(\alpha,(e,\psi))\}) = \mathcal{D}_U^*(\{(\alpha,(e,\psi))\})]$$

PROOF
The proof uses induction on the complexity of expressions. We treat two simple basic cases, being (lazy and) confident that these will show the reader how to proceed in the other cases. So let $\alpha \in AObj$ and $\psi \in L_{PS}$ and suppose

$$\forall\beta\ [\Phi_U'(\mathcal{D}_U^*)(\{(\alpha,\psi(\beta))\}) = \mathcal{D}_U^*(\{(\alpha,\psi(\beta))\})].$$

For $e = \beta$ we have

$$\Phi_U'(\mathcal{D}_U^*)(\{(\alpha,(\beta,\psi))\}) = \Phi_U'(\mathcal{D}_U^*)(\{(\alpha,\psi(\beta))\})$$

$$= \text{[hypothesis]}$$
$$\mathcal{D}^*_U(\{(\alpha,\psi(\beta))\})$$
$$= \mathcal{D}_S'[\![\psi(\beta)]\!](\alpha)(p_0)$$
$$= \mathcal{D}_S'[\![(\beta,\psi)]\!](\alpha)(p_0)$$
$$= \mathcal{D}^*_U(\{(\alpha,(\beta,\psi))\});$$

if $e = \beta_1!m(\beta_2)$ then

$$\Phi_U'(\mathcal{D}^*_U)(\{(\alpha,(\beta_1!m(\beta_2),\psi))\}) = \lambda\sigma\cdot\{<\beta_1, m, \beta_2, \lambda\beta\cdot\mathcal{D}^*_U(\{(\alpha,\psi(\beta))\}),\ p_0>\}$$
$$= \lambda\sigma\cdot\{<\beta_1, m, \beta_2, \lambda\beta\cdot\mathcal{D}_S'[\![\psi(\beta)]\!](\alpha)(p_0),\ p_0>\}$$
$$= \mathcal{D}_E'[\![\beta_1]\!](\alpha)(\lambda\beta'_1\cdot\mathcal{D}_E'[\![\beta_2]\!](\alpha)(\lambda\beta'_1\cdot$$
$$\lambda\sigma\cdot\{<\beta'_1, m, \beta'_2,\ \lambda\beta\cdot\mathcal{D}_S'[\![\psi(\beta)]\!](\alpha)(p_0),\ p_0>\}))$$
$$= \mathcal{D}_E'[\![\beta_1!m(\beta_2)]\!](\alpha)(\lambda\beta\cdot\mathcal{D}_S'[\![\psi(\beta)]\!](\alpha)(p_0))$$
$$= \mathcal{D}_S'[\![(\beta_1!m(\beta_2),\ \psi)]\!](\alpha)(p_0)$$
$$= \mathcal{D}^*_U(\{(\alpha,(\beta_1!m(\beta_2),\ \psi))\}).$$

THEOREM 7.7: $\Phi_U'(\mathcal{D}^*_U) = \mathcal{D}^*_U$

PROOF
We show: $\forall X \in \mathcal{P}_{fin}(LStat)\ [\Phi_U'(\mathcal{D}^*_U)(X) = \mathcal{D}^*_U(X)]$, using induction on the number of elements in X.

Case 1: $X = \{(\alpha,s)\}$, with $\alpha \in AObj$, $s \in L_S'$.
The proof uses induction on the complexity of the statement s. We treat some typical cases.

(i) **answer** m:

$$\Phi_U'(\mathcal{D}^*_U)(\{(\alpha,\ \textbf{answer}\ m)\}) = \lambda\sigma\cdot\{<\alpha, m, g_m>\}$$

with

$$g_m = \lambda\beta\cdot\lambda f\cdot\lambda\bar{\sigma}\cdot\{<\bar{\sigma}', \mathcal{D}_E[\![e_m]\!](\alpha)(\lambda\beta'\cdot\lambda\hat{\sigma}\cdot\{<\hat{\sigma}', f(\beta')\| \mathcal{D}^*_U(\{(\alpha,E)\})>\})>\}$$
$$= \lambda\beta\cdot\lambda f\cdot\lambda\bar{\sigma}\cdot\{<\bar{\sigma}', \mathcal{D}_E[\![e_m]\!](\alpha)(\lambda\beta'\cdot\lambda\hat{\sigma}\cdot\{<\hat{\sigma}', f(\beta')>\})>\}$$

(and $\bar{\sigma}'$ and $\hat{\sigma}'$ as in definition 6.1). If we compare this to the definition of $\mathcal{D}_S[\![\textbf{answer}\ m]\!]$ (definition 5.3(S3)) we see

$$\lambda\sigma\cdot\{<\alpha, m, g_m>\} = \mathcal{D}_S[\![\textbf{answer}\ m]\!](\alpha)(p_0)$$
$$= \mathcal{D}^*_U(\{(\alpha,\ \textbf{answer}\ m)\}).$$

(ii) $x \leftarrow e$: we distinguish two subcases. First , if $e = \beta$, then

$$\Phi_U'(\mathcal{D}^*_U)(\{(\alpha, x\leftarrow\beta)\}) = \lambda\sigma\cdot\{<\sigma\{\beta/\alpha, x\}, p_0>\}$$
$$= \mathcal{D}_E'[\![\beta]\!](\alpha)(\lambda\beta\cdot\lambda\sigma\cdot\{<\sigma\{\beta/\alpha, x\},\ p_0>\})$$
$$= \mathcal{D}_S'[\![x\leftarrow\beta]\!](\alpha)(p_0)$$
$$= \mathcal{D}^*_U(\{(\alpha, x\leftarrow\beta)\}).$$

If $e \notin Obj$, then:

$$\Phi_U'(\mathcal{D}^*_U)(\{(\alpha, x\leftarrow e)\}) = \text{[definition } -U\rightarrow \text{]}$$

$$\Phi_U'(\mathcal{D}_U^*)(\{(\alpha,(e,\lambda u\cdot x\leftarrow u))\})$$
$$= [\text{ see (v) below }]$$
$$\mathcal{D}_U^*(\{(\alpha,(e,\lambda u\cdot x\leftarrow u))\})$$
$$= \mathcal{D}_E'[\![e]\!](\alpha)(\lambda\beta\cdot\mathcal{D}_S'[\![x\leftarrow\beta]\!](\alpha)(p_0))$$
$$= \mathcal{D}_S'[\![x\leftarrow e]\!](\alpha)(p_0)$$
$$= \mathcal{D}_U^*(\{(\alpha,x\leftarrow e)\}).$$

(iii) $s_1;s_2$: case analysis for s_1.

(iv) **do** e **then** s **od**:

$$\Phi_U'(\mathcal{D}_U^*)(\{(\alpha,\ \textbf{do}\ e\ \textbf{then}\ s\ \textbf{od}\ )\})$$
$$= \lambda\sigma\cdot\{<\sigma,\mathcal{D}_U^*(\{(\alpha,\ \textbf{if}\ e\ \textbf{then}\ s;(\textbf{do}\ e\ \textbf{then}\ s\ \textbf{od})\ \textbf{else}\ E\ \textbf{fi}\ )\})>\}$$
$$= \lambda\sigma\cdot\{<\sigma,\mathcal{D}_E'[\![e]\!](\alpha)(\lambda\beta\cdot\ \text{if}\ \beta=tt\ \text{then}$$
$$\mathcal{D}_S'[\![s]\!](\alpha)(\mathcal{D}_S'[\![\textbf{do}\ e\ \textbf{then}\ s\ \textbf{od}]\!](\alpha)(p_0))\ \text{else}\ p_0\ \text{fi}\ )>\}$$
$$= \mathcal{D}_S'[\![\textbf{do}\ e\ \textbf{then}\ s\ \textbf{od}]\!](\alpha)(p_0)$$
$$= \mathcal{D}_U^*(\{(\alpha,\ \textbf{do}\ e\ \textbf{then}\ s\ \textbf{od}\ )\}).$$

(v) (e,ψ): by induction we have that the theorem holds for $(\alpha,\psi(\beta))$, for every $\beta\in Obj$. Now we can apply lemma 7.6.

Case 2: $X\in\mathcal{P}_{fin}(LStat)$ and X has at least two elements.
Suppose we have two disjoint sets X_1 and X_2 in $\mathcal{P}_{fin}(LStat)$ with $X=X_1\cup X_2$ such that

$$\Phi_U'(\mathcal{D}_U^*)(X_i)=\mathcal{D}_U^*(X_i)$$

for $i=1,2$. Assume $X_1,\ X_2\neq\{<\alpha_1,E>,\ldots,<\alpha_n,E>\}$. We shall show that from this induction hypothesis it follows that

$$\Phi_U'(\mathcal{D}_U^*)(X_1\cup X_2)=\ \mathcal{D}_U^*(X_1\cup X_2).$$

(This is proved in very much the same way as the fact that $\Phi'(\tilde{\mathcal{D}})(\rho)=\tilde{\mathcal{D}}(\rho)$ and $\Phi'(\tilde{\mathcal{D}})(\pi)=\tilde{\mathcal{D}}(\pi)$ implies $\Phi'(\tilde{\mathcal{D}})(\rho^\frown\pi)=\tilde{\mathcal{D}}(\rho^\frown\pi)$, which occurs in theorem 2.14 of section 2.)

From the definition of $-U\rightarrow$ (definition 4.8, rules 10 and 11) it follows that

$$\Phi'_U(\mathcal{D}_U^*)(X_1\cup X_2)=\ \lambda\sigma\cdot(X_1^\forall\cup X_2^\forall\cup Z).$$

Here

$$X_1^\forall\ =\ \{<\sigma',\ \mathcal{D}_U^*(X'_1\cup X_2)>:\ <X_1,\ \sigma>\ -U,\tau\rightarrow\ <X'_1,\ \sigma'>\}$$
$$\cup\{<\beta_1,\ m,\ \beta_2,\ \lambda\beta\cdot\mathcal{D}_U^*(\{(\alpha,\ \psi(\beta))\}),\ \mathcal{D}_U^*(X_1'\cup X_2)>:$$
$$<X_1,\sigma>\ -U,\ (\alpha,\beta_1!m(\beta_2))\rightarrow\ <X_1'\cup\{(\alpha,\psi)\},\sigma>\}$$
$$\cup\{<\alpha,m,g_m>:\ <X_1,\sigma>\ -U,\ (\alpha?m)\rightarrow\ <X_1'\cup\{(\alpha,s)\},\ \sigma>\}$$

with

$$g_m\ =\ \lambda\beta\cdot\lambda f\cdot(\lambda\sigma\cdot\{<\overline{\sigma}',\ \mathcal{D}_E[\![e_m]\!](\alpha)(\lambda\beta'\cdot\lambda\hat{\sigma}\cdot\{<\hat{\sigma}',\ f(\beta')\|\mathcal{D}_U^*(\{(\alpha,s)\})>\})>\}$$
$$\|\ \mathcal{D}_U^*(X_1'\cup X_2))$$

and e_m, $\overline{\sigma}'$ and $\hat{\sigma}'$ as in definition 6.1. The set $X_2^\forall$ is like $X_1^\forall$ but with the roles of X_1 and X_2 interchanged. Finally,

$$Z = \{<\sigma', \mathcal{D}_U^*(\{(\beta_1, (e_m, \lambda u\cdot(u_m \leftarrow \sigma_2(\beta_1)(u_m);\ \textbf{release}\ (\alpha, \psi(u));\ s)))\} \cup X'_1 \cup X_2') > :$$
$$<X_i, \sigma> -U, (\alpha, \beta_1 ! m(\beta_2)) \rightarrow <\{(\alpha, \psi)\} \cup X_i', \sigma> \text{ and}$$
$$<X_j, \sigma> -U, (\beta_1 ? m) \rightarrow <\{(\beta_1, s)\} \cup X_j', \sigma>, \text{ for } i=1,\ j=2 \text{ or } i=2,\ j=1\}$$

(and $\sigma' = \sigma\{\beta_2/\beta_1,\ u_m\}$, $m \Leftarrow <u_m,\ e_m> \in U$). The steps in X_i^{∇} correspond to the transition steps that can be made from $X_1 \cup X_2$ as a result of a transition step from X_i (by an application of rule 10 in the definition of $-U\rightarrow$), for $i=1,2$.

The set Z contains those steps that correspond with a communication transition from $X_1 \cup X_2$ which results from a send transition from X_i and an answer transition from X_j (for $i=1,\ j=2$ or $i=2,\ j=1$) by an application of rule 11.

Now we have

$$X_1^{\nabla} = \Phi_U'(\mathcal{D}_U^*)(X_1)(\sigma) \between \mathcal{D}_U^*(X_2),$$
$$X_2^{\nabla} = \Phi_U'(\mathcal{D}_U^*)(X_2)(\sigma) \between \mathcal{D}_U^*(X_1),$$
$$Z = \Phi_U'(\mathcal{D}_U^*)(X_1)(\sigma) |_\sigma \Phi_U'(\mathcal{D}_U^*)(X_2)(\sigma).$$

The proofs of these facts are not difficult (but tiresome and therefore omitted). It follows that

$$\begin{aligned}
\Phi_U'(\mathcal{D}_U^*)(X_1 \cup X_2) &= \lambda\sigma\cdot(X_1^{\nabla} \cup X_2^{\nabla} \cup Z) \\
&= \lambda\sigma\cdot(\Phi_U'(\mathcal{D}_U^*)(X_1)(\sigma) \between \mathcal{D}_U^*(X_2) \cup \\
&\qquad \Phi_U'(\mathcal{D}_U^*)(X_2)(\sigma) \between \mathcal{D}_U^*(X_1) \cup \\
&\qquad \Phi_U'(\mathcal{D}_U^*)(X_2)(\sigma) |_\sigma \Phi_U'(\mathcal{D}_U^*)(X_2)(\sigma)) \\
&= [\text{ induction hypothesis }] \\
&\quad \lambda\sigma\cdot(\mathcal{D}_U^*(X_1)(\sigma) \between \mathcal{D}_U^*(X_2) \cup \\
&\qquad \mathcal{D}_U^*(X_2)(\sigma) \between \mathcal{D}_U^*(X_1) \cup \\
&\qquad \mathcal{D}_U^*(X_1)(\sigma) |_\sigma \mathcal{D}_U^*(X_2)(\sigma)) \\
&= [\text{ definition } \| \,] \\
&\quad \mathcal{D}_U^*(X_1) \| \mathcal{D}_U^*(X_2) \\
&= \mathcal{D}_U^*(X_1 \cup X_2).
\end{aligned}$$

This concludes the proof of theorem 7.7. □

Since Θ_U' and $\mathcal{D}_U^*$ are both fixed points of the same contraction Φ_U', they must be equal:

COROLLARY 7.8: $\Theta_U' = \mathcal{D}_U^*$

Part 3. Collecting the results

We have proved that $\Theta_U = \mathit{abstr} \circ \Theta_U'$ and that $\Theta_U' = \mathcal{D}_U^*$. Thus

THEOREM 7.9: $\Theta_U = \mathit{abstr} \circ \mathcal{D}_U^*$

From this theorem we deduce the main theorem of this paper:

THEOREM 7.10: $[\![U]\!]_\Theta = \mathit{abstr}([\![U]\!]_{\mathcal{D}})$

PROOF

Let $U = <(\ldots, C_n \Leftarrow s_n), \ldots >$. Then

$$
\begin{aligned}
[\![U]\!]_{\mathcal{O}} &= \Theta_U[\![\{(\nu(\varnothing), s_n)\}]\!] \\
&= abstr(\mathcal{D}_U^*(\{(\nu(\varnothing), s_n)\})) \\
&= abstr(\mathcal{D}_S'[\![s_n]\!](\nu(\varnothing))(p_0)) \\
&= abstr(\mathcal{D}_S[\![s_n]\!](\nu(\varnothing))(p_0)) \\
&= abstr([\![U]\!]_{\mathcal{D}}).
\end{aligned}
$$

8. References

[Am85] P. America, *Definition of the programming language* POOL-T, ESPRIT project 415, Doc. No. 0091, Philips Research Laboratories, Eindhoven, 1985.

[Am86] P. America, *Rationale for the design of* POOL, ESPRIT project 415, Doc. No. 0053, Philips Research Laboratories, Eindhoven, 1986.

[Am87] P. America, POOL-T — *A parallel object-oriented language,* in: "Object-Oriented Concurrent Systems" (A. Yonezawa and M. Tokoro, Eds.), MIT Press, 1987.

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

[ABKR86(a)] P. America, J.W. de Bakker, J.N. Kok, J.J.M.M. Rutten, *Operational semantics of a parallel object-oriented language,* in: "Conference Record of the 13th Symposium on Principles of Programming Languages, St. Petersburg, Florida," 1986, pp. 194-208.

[ABKR86(b)] P. America, J.W. de Bakker, J.N. Kok, J.J.M.M. Rutten, *A denotational semantics of a parallel object-oriented language,* Technical Report (CS-R8626), Centre for Mathematics and Computer Science, Amsterdam, 1986. (To appear in: Information and Computation.)

[ANSI83] ANSI, *Reference manual for the Ada programming language,* ANSI / MIL-STD 1815 A, United States Department of Defense, Washington D. C., 1983.

[AR88] P. America, J.J.M.M. Rutten, *Solving reflexive domain equations in a category of complete metric spaces,* in: Proceedings of the Third Workshop on Mathematical Foundations of Programming Language Semantics (M. Main, A. Melton, M. Mislove, D. Schmidt, Eds.), Lecture Notes in Computer Science 298, Springer-Verlag, 1988, pp. 254-288. (To appear in the Journal of Computer and System Sciences.)

[Br86] A. de Bruin, *Experiments with continuation semantics: Jumps, backtracking, dynamic networks,* Ph. D. thesis, Free University of Amsterdam, 1986.

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

[BKMOZ86] 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 (J.W. de Bakker, W.P. de Roever, G. Rozenberg, Eds.), Lecture Notes in Computer Science 224, Springer-Verlag, 1986, pp. 51-121.

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

[Cl81] W.D. Clinger, *Foundations of actor semantics,* Ph. D. thesis, Massachusetts Institute of Technology (AI-TR-633), 1981.

[Du66] J. Dugundji, *Topology,* Allyn and Bacon, inc., Boston, 1966.

[En77] E. Engelking, *General topology,* Polish Scientific Publishers, 1977.

[Go79] M.J.C. Gordon, *The denotational description of programming languages,* Springer-Verlag, 1979.

[HP79] M. Hennessy, G.D. Plotkin, *Full abstraction for a simple parallel programming*

language, in: Proceedings 8th MFCS (J. Bečvář ed.), Lecture Notes in Computer Science 74, Springer-Verlag, 1979, pp. 108-120.

[KR88] J.N. KOK, J.J.M.M. RUTTEN, *Contractions in comparing concurrency semantics*, in: Proceedings 15th ICALP, Tampere, Lecture Notes in Computer Science 317, Springer-Verlag, 1988, pp. 317-332.

[Mi51] E. MICHAEL, *Topologies on spaces of subsets*, Trans. AMS 71, 1951, pp.152-182.

[Mil80] R. MILNER, *A calculus of communicating systems*, Lecture Notes in Computer Science 92, Springer-Verlag, 1980.

[Od87] E.A.M. ODIJK, *The* DOOM *system and its applications: a survey of ESPRIT 415 subproject A*, in: "Parallel Architectures and Languages Europe, Volume I" (J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, Eds.), Lecture Notes in Computer Science 258, Springer-Verlag, 1987, pp. 461-479.

[Pl76] G.D. PLOTKIN, *A powerdomain construction*, SIAM Journal of Computing, Vol. 5, no. 3, 1976, pp. 452-487.

[Pl81] G.D. PLOTKIN, *A structural approach to operational semantics*, Report DAIMI FN-19, Comp. Sci. Dept., Aarhus Univ. 1981.

[Pl83] G.D. PLOTKIN, *An operational semantics for CSP*, in: Formal Description of Programming Concepts II (D. Bjørner ed.), North-Holland, Amsterdam, 1983, pp. 199-223.

APPENDIX I: MATHEMATICAL DEFINITIONS

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

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

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

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

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

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

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

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

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

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

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

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

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

PROPOSITION A.4 (Banach's fixed-point theorem)
Let (M,d) be a complete metric space and $f: M \to M$ a contracting function. Then there exists an $x \in M$ such that the following holds:

(1) $f(x) = x$ *(x is a fixed point of f),*
(2) $\forall y \in M\ [f(y) = y \Rightarrow y = x]$ *(x is unique),*
(3) $\forall x_0 \in M\ [\lim_{n\to\infty} f^{(n)}(x_0) = x]$, *where* $f^{(n+1)}(x_0) = f(f^{(n)}(x_0))$ and $f^{(0)}(x_0) = x_0$.

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

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

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

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

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

(b) With $M_1 \dot\cup \cdots \dot\cup M_n$ 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 $M_1 \dot\cup \cdots \dot\cup M_n$ as follows. For every $x,y \in M_1 \dot\cup \cdots \dot\cup M_n$

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

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

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

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

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

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

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

The following spaces

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

$$\mathcal{P}_{ncompact}(M) = \{X: X\subseteq M \wedge X \text{ is nonempty and compact}\}$$

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

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

PROPOSITION A.7

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

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

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

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

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

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

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

PROPOSITION A.8

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

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

The proof of proposition A.8 can be found in [Du66] and [En77]. The completeness of the Hausdorff space containing compact sets is proved in [Mi51].

APPENDIX II: THE FUNCTION *abstr*

The definition of $abstr:\overline{P}\rightarrow P$ can be viewed as a fixed point characterization of a somewhat differently and more intuitively defined operation

$$abstr^*:\overline{P}\rightarrow P,$$

which we introduce below. Next, we show that $abstr=abstr^*$.

DEFINITION II.1 ($abstr^*$)

Let $p\in\overline{P}$ and $\sigma\in\Sigma$, and let $w\in\Sigma^{\infty}_{\partial}$.

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

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

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

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

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

$$w = \sigma_1 \cdots \sigma_n \cdot \partial \wedge \forall 1 \leqslant i < n\ [<\sigma_{i+1},p_{i+1}> \in p_i(\sigma_i)] \wedge$$
$$<\sigma_1,p_1> \in p(\sigma) \wedge p_n \neq p_0 \wedge p_n(\sigma_n) \cap (\Sigma \times \overline{P}) = \varnothing.$$

Now we define a function $abstr^*: \overline{P} \to P$ by

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

We have to verify that for every $p \in \overline{P}$ and $\sigma \in \Sigma$ the set $abstr^*(p)(\sigma)$ is compact. This is not trivial and is proved in theorem II.3 below (which is a slightly generalised form of lemma AII.4 in [BBKM84]). The fact that we use in the definition of $\overline{P}$ *compact* subsets rather than closed ones is essential for the proof. (For a process domain defined with *closed* subsets, [BBKM84] provides a counterexample of the theorem.)

In the proof of theorem II.3 below, we need the following lemma:

LEMMA II.2
Let $q = \lim_{n\to\infty} q_n$, *for* $q, q_n \in \overline{P}$: *assume (without loss of generality) that for all* $n \geqslant 0$

$$d(q,q_n) \leqslant 2^{-(n+1)}.$$

Let $\sigma \in \Sigma$ *and let* $(w_i)_i$ *be a sequence in* Σ_δ^∞ *with* $w_i \in abstr^*(q_i)(\sigma)$, *for every* $i \geqslant 0$. *Then*

$$\forall n\ \exists u\ [w_n[n] \cdot u \in abstr^*(q)(\sigma)].$$

PROOF
Let $w_n[n] = \sigma_1 \cdots \sigma_n$. (In the case of termination or deadlock the rest of the proof is analogous to this case.) Now there must be $q^1, \ldots, q^n$ with

$$<\sigma_1,q^1> \in q_n(\sigma) \text{ and } <\sigma_{i+1},q^{i+1}> \in q^i(\sigma_i)$$

for $1 \leqslant i \leqslant n$. We shall show that there are $\overline{q}^1, \ldots, \overline{q}^n$ with $<\sigma_1,\overline{q}^1> \in q(\sigma)$ and $<\sigma_{i+1},\overline{q}^{i+1}> \in \overline{q}^i(\sigma_i)$ for $1 \leqslant i \leqslant n$. We do this inductively: For $i = 1$ we observe that $d(q,q_n) \leqslant 2^{-(n+1)}$, so $d(q(\sigma),q_n(\sigma)) \leqslant 2^{-n} \leqslant \frac{1}{2}$. Because $<\sigma_1,q^1> \in q_n(\sigma)$, there must be a $\overline{q}^1$ with

$$<\sigma_1,\overline{q}^1> \in q(\sigma) \text{ and } d(q^1,\overline{q}^1) \leqslant 2^{-n}.$$

For the inductive step, let $1 \leqslant i \leqslant n$ and let $\overline{q}^i$ be such that $d(q^i,\overline{q}^i) \leqslant 2^{-(n+1)+i}$. Then

$$d(q^i(\sigma_i),\overline{q}^i(\sigma_i)) \leqslant 2^{-n+i} \leqslant \tfrac{1}{2}.$$

Because $<\sigma_{i+1},q^{i+1}> \in q^i(\sigma_i)$ there must be a $\overline{q}^{i+1}$ with

$$<\sigma_{i+1},\overline{q}^{i+1}> \in \overline{q}^i(\sigma_i) \text{ and } d(q^{i+1},\overline{q}^{i+1}) \leqslant 2^{-n+i}.$$

With $\overline{q}^1, \ldots, \overline{q}^n$ suitably chosen, we can take $u \in abstr^*(\overline{q}^n)(\sigma_n)$ arbitrary, and then $w_n[n] \cdot u$ will be in $abstr^*(q)(\sigma)$.

THEOREM II.3: *For every* $p \in \overline{P}$ *and* $\sigma \in \Sigma$ *the set* $abstr^*(p)(\sigma)$ *is compact.*

PROOF
Let $(w_i)_i$ be a sequence in $abstr^*(p)(\sigma)$. We shall show that there exists a subsequence of $(w_i)_i$ that has its limit in $abstr^*(p)(\sigma)$. First we introduce some notation: For an arbitrary word $w \in \Sigma_\delta^\infty$, $w<k>$ indicates the word that is obtained from w by omitting the first k elements. We call $p_0 = p$, $\sigma_0 = \sigma$ and $f_0 = id_{\mathbb{N}}$, the identity function on the set of natural numbers. We shall inductively construct for every $n \geqslant 0$ a function $f_n: \mathbb{N} \to \mathbb{N}$, a process $p_n \in \overline{P}$, and a state σ_n such that:

1. $\forall i \geqslant 0\ [w_{f_n(i)}[n] = \sigma_1 \cdots \sigma_n]$
2. $\forall i,\ 0 \leqslant i < n\ [<\sigma_{i+1}, p_{i+1}> \in p_i(\sigma_i)]$
3. $\exists (v_i)_i$ in $abstr^*(p_n)(\sigma_n)\ \forall i \geqslant 1\ [v_i[i] = w_{f_n(i)} <n>[i]]$
4. f_n is monotonic and there exists a monotonic h with $f_n = f_{n-1} \circ h$.

Once we have constructed such sequences $(f_n)_n$, $(p_n)_n$ and $(\sigma_n)_n$, we are done: We can define

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

This function is monotonic and we have

$$\lim_{i \to \infty} w_{g(i)} = \sigma_1 \cdot \sigma_2 \cdots$$

Since $\sigma_1 \cdot \sigma_2 \cdots \in abstr^*(p)(\sigma)$ we thus have found a subsequence $(w_{g(i)})_i$ of $(w_i)_i$, which has its limit in $abstr^*(p)(\sigma)$.

The construction is as follows: Suppose we are at stage $n \geqslant 0$. Let $(v_i)_i$ be a sequence in $abstr^*(p_n)(\sigma_n)$ satisfying property 3. above. Let for every $i \geqslant 1$

$$v_i = \tau_1^i \cdot \tau_2^i \cdots.$$

Then there are $q_1^i, q_2^i, \ldots \in \overline{P}$ with

$$<\tau_1^i, q_1^i> \in p_n(\sigma_n), \text{ and}$$

$$\forall j \geqslant 1\ [<\tau_{j+1}^i, q_{j+1}^i> \in q_j^i(\tau_j^i)].$$

Since the set $p_n(\sigma_n)$ is compact, the sequence $(<\tau_1^i, q_1^i>)_i$ has a converging subsequence, which is given by, say, the monotonic function h and which has a limit, say $<\tau, q>$ in $p_n(\sigma_n)$. We may assume

$$\forall j \geqslant 1\ [\tau_1^{h(j)} = \tau \wedge d(q_1^{h(j)}, q) \leqslant 2^{-(j+1)}].$$

Now we take

$$p_{n+1} = q,\ \ \sigma_{n+1} = \tau,\ \ f_{n+1} = f_n \circ h.$$

In order to show that this construction works, we have to verify that p_{n+1}, σ_{n+1}, and f_{n+1} again satisfy properties 1. through 4. above.

1. We have for every $i \geqslant 1$:

$$\begin{aligned} w_{f_{n+1}(i)}[n+1] &= w_{f_{n+1}(i)}[n] \cdot w_{f_{n+1}(i)}(n+1) \\ &= \sigma_1 \cdots \sigma_n \cdot w_{f_{n+1}(i)} <n> (1) \\ &= \sigma_1 \cdots \sigma_n \cdot v_{h(i)}(1) \\ &= \sigma_1 \cdots \sigma_n \cdot \sigma_{n+1}. \end{aligned}$$

2. We have $<\sigma_{n+1}, p_{n+1}> = <\tau, q> \in p_n(\sigma_n)$.
3. In order to prove this property, we are going to apply the following version of lemma II.2: For all $q, q_1, q_2, \ldots \in \overline{P}$, and for all $x_1, x_2, \ldots \in \Sigma_\delta^\infty$,

$$\forall i \geqslant 1\ [d(q, q_i) \leqslant 2^{-(i+1)} \wedge x_i \in abstr^*(q_i)(\sigma)] \Rightarrow$$

$$\exists (y_i)_i \text{ in } abstr^*(q)(\sigma)\ \forall i \geqslant 1\ [y_i[i] = x_i[i]].$$

This we now use: Since

$$\forall i \geqslant 1\ [d(p_{n+1}, q_1^{h(i)}) \leqslant 2^{-(i+1)} \wedge v_{h(i)} <1> \in abstr^*(q_1^{h(i)})(\sigma_{n+1})]$$

there must exist a sequence $(v_i')_i$ in $abstr^*(p_{n+1})(\sigma_{n+1})$ with

$$\forall i \geqslant 1\ [v_i'[i] = v_{h(i)} <1> [i]].$$

Now

$$\begin{aligned} v_{h(i)}<1>[i] &= v_{h(i)}[h(i)]<1>[i] \\ &= w_{f_{n+1}(i)}<n>[h(i)]<1>[i] \\ &= w_{f_{n+1}(i)}<n><1>[i] \\ &= w_{f_{n+1}(i)}<n+1>[i]. \end{aligned}$$

(Here we have used twice the fact that $h(i)>i$, for all $i \geqslant 1$.)

4. By definition.

This concludes the proof of theorem II.3.

Next we show that the function $abstr:\bar{P}\to P$, given in definition 7.1, can be defined as the fixed point of a contraction.

DEFINITION II.4 (Formal definition *abstr*)

We define $\Xi:(\bar{P}\to^1 P)\to(\bar{P}\to^1 P)$; let $F\in\bar{P}\to^1 P$, $P\in\bar{P}$ and $\sigma\in\Sigma$. We put

$$\Xi(F)(p_0)(\sigma) = \{\epsilon\},$$
$$\Xi(F)(p)(\sigma) = \{\partial\}, \quad \text{if } p(\sigma)\cap Comp_{\bar{P}} = \varnothing.$$

Otherwise, we set

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

Finally, we define

$$abstr = \text{Fixed Point}(\Xi)$$

It is straightforward to show Ξ is contracting. The fact that for every $p\in\bar{P}$ and $\sigma\in\Sigma$ the set $\Xi(F)(p)(\sigma)$ is compact needs some explanation. In order to prove this, it is convenient to adapt the definition of Ξ a little. Recalling that $P=\Sigma\to\mathcal{P}_{ncomp}(\Sigma^\infty_\partial)$ we define

$$\Xi': ((\bar{P}\times\Sigma)\to^1\mathcal{P}_{ncomp}(\Sigma^\infty_\partial)) \to ((\bar{P}\times\Sigma)\to^1\mathcal{P}_{ncomp}(\Sigma^\infty_\partial)),$$

where the superscript 1 above the arrow indicates that we consider only non-expansive (and hence continuous) functions, by

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

Now

$$\begin{aligned} \Xi'(F)(<p,\sigma>) &= \bigcup_{<\sigma',p'>\in p(\sigma)}\{\sigma'\cdot F(<p',\sigma'>)\} \\ &= \bigcup_{\sigma'}(\sigma'\cdot\{F(<p',\sigma'>): <\sigma',p'>\in p(\sigma)\}) \\ &= \bigcup_{\sigma'}(\sigma'\cdot F(\{<p',\sigma'>: <\sigma',p'>\in p(\sigma)\})) \end{aligned}$$

This union can be seen to be compact by first observing that from the compactness of p it follows that the union is finite: the set

$$\{\sigma': \exists p'\in\bar{P}[<\sigma',p'>\in p(\sigma)]\}$$

is finite. The compactness of $p(\sigma)$ further implies the compactness of the isomorphic set

$$\{<p',\sigma'>: <\sigma',p'>\in p(\sigma)\},$$

for every $\sigma' \in \Sigma$, which is preserved under the continuous mapping F and the concatenation with σ'. So we have a finite union of compact sets, which is again compact. Now the compactness of $\Xi(F)(p)(\sigma)$ follows straightforwardly from the compactness of $\Xi'(F')(<p',\sigma'>)$, for arbitrary F', p' and σ'. The fact that $\Xi(F)$ is again non-expansive is also easily verified.

We conclude this appendix by showing that *abstr* and *abstr** are equal:

THEOREM II.5: $abstr = abstr^*$

PROOF: Consider $p \in \overline{P} - \{p_0\}$ and $\sigma \in \Sigma$ such that $p(\sigma) \cap (\Sigma \times \overline{P}) \neq \emptyset$. Then:

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

The other cases are easy. We see: $abstr^* = \Xi(abstr^*)$. Because Ξ is a contraction the theorem follows. (Note the similarity of this proof and the one of theorem 4.14.)

APPENDIX III: STANDARD OBJECTS

We want to extend the language under consideration with a few standard classes of so-called *standard* objects, namely 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 23+11 is indicated by the send expression 23! add (11), sending a message with method name add and parameter 11 to the standard object 23. The set of expressions L_E, given in definition 3.1, is extended with these standard objects:

$$e ::= x \mid u \mid e_1!m(e_2) \mid \mathbf{new}(C) \mid s;e \mid \mathbf{self} \mid \alpha,$$

where $\alpha \in SObj$, with

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

Recall that we already defined (in definition 4.1):

$$Obj = AObj \cup SObj$$

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

Intuitively, the evaluation of the expression α, with $\alpha \in SObj$, results in that object itself. For instance, the value of the expression 29 will be the integer 29.

Below, we shall first extend the definition of the operational semantics, next we adapt the definition of the denotational semantics (following [ABKR86(b)]), and finally we shall prove that the equivalence result of section 7 still holds.

III.1 *Standard objects in the operational semantics*

We extend the set $L_{E'}$, given in definition 4.2, with the standard objects:

$$e ::= x \mid u \mid e_1!m(e_2) \mid \mathbf{new}(C) \mid s;e \mid \mathbf{self} \mid \alpha \mid (e, \phi),$$

where now $\alpha \in Obj = AObj \cup SObj$.

Next we add to the set of labeled statements (definition 4.5) an abstract element S_t that represents all standard objects and for which transitions will be specified in a moment:

$$LStat^* = LStat \cup \{S_t\}.$$

The following transitions are possible from S_t:

$$<\{S_t\}, \sigma> -n?\mathsf{add}\rightarrow <\{S_t\}, \sigma>$$
$$<\{S_t\}, \sigma> -n?\mathsf{sub}\rightarrow <\{S_t\}, \sigma>$$
$$<\{S_t\}, \sigma> -b?\mathsf{and}\rightarrow <\{S_t\}, \sigma>$$
$$<\{S_t\}, \sigma> -b?\mathsf{or}\rightarrow <\{S_t\}, \sigma>$$
$$<\{S_t\}, \sigma> -b?\mathsf{not}\rightarrow <\{S_t\}, \sigma>$$

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

If $<\{(\alpha,s)\}, \sigma> -(\alpha,n!\mathsf{add}\ (m))\rightarrow <\{(\alpha,\psi)\}, \sigma>$

then $<\{(\alpha,s), S_t\}, \sigma> -\gamma\rightarrow <\{(\alpha, \psi(n+m)), S_t\}$.

If $<\{(\alpha,s)\}, \sigma> -(\alpha, b_1!\mathsf{and}\ (b_2))\rightarrow <\{(\alpha,\psi)\}, \sigma>$

then $<\{(\alpha,s), S_t\}, \sigma> -\gamma\rightarrow <\{(\alpha,\psi(b_1\wedge b_2)), S_t\}, \sigma>$,

and by similar transitions for the other operations. The result of, for example, an addition of the integers n and m is computed and passed through to the parameterized statement of the object requesting the execution of the method add.

Finally, the operational semantics of a unit (definition 4.11) is changed by taking into account the standard objects; we put

$$[\![U]\!]_\Theta = \Theta_U[\![\{(\nu(\varnothing),s_n), S_t\}]\!].$$

(In the operational semantics defined in [ABKR86(a)], the standard objects are treated somewhat differently. There no special rules are given for the communication with a standard object; instead, some axioms are added that replace in one step a send expression that addresses a standard object by the corresponding value of the result.)

III.2 *Standard objects in the denotational semantics*

The denotational meaning of a standard object $\alpha\in L_E$ is given by

$$\mathcal{D}_E[\![\alpha]\!](\beta)(f) = f(\alpha),$$

where $\beta\in AObj$, and $f\in Cont_E$.

We follow [ABKR86(b)] in introducing a process $p_{St}\in\overline{P}$ that represents the denotational meaning of the standard objects. For this we have to adapt our semantic process domain $\overline{P}$. In definition 5.1 the domain $\overline{P}$ is given by

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

In order to let the standard process p_{St}, to be defined below, fit into our semantic domain nicely, we are forced to use closed subsets of steps rather than compact ones. Let us indicate the process domain given in definition 5.1 by $\overline{P}_{co}$. We introduce here $\overline{P}_{cl}$, which satisfies:

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

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

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

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

where

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

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

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

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

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

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

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

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

The standard objects are assumed to be present at the execution of every unit U. Therefore we adapt the denotational semantics of a unit (definition 5.4) as follows:

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

III.3 *Semantic equivalence*

Finally, we extend the arguments presented in section 7 in order to show that for the modified versions of $[\![U]\!]_{\mathfrak{O}}$ and $[\![U]\!]_{\mathfrak{D}}$, as presented above, we still have:

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

We begin by adapting the intermediate semantics Θ_U' (definition 6.1), which will now be of type

$$\Theta_U': \mathcal{P}_{fin}(LStat^*)\to\bar{P}_{cl}.$$

We put:

$$\Theta_U'(\{S_t\}) = p_{St}$$

and for $X\subseteq LStat^*-\{S_t\}$ $(=LStat)$:

$$\Theta_U'(X\cup\{S_t\}) = \Theta_U'(X)\|\Theta_U'(\{S_t\}),$$

with $\Theta_U'(X)$ as defined according to definition 6.1.

Next we extend the definition of *abstr* to an operation:

$$abstr^*:\bar{P}_{cl}\to(\Sigma\to\mathcal{P}(\Sigma_\delta^\infty)),$$

where $abstr^*$ is defined as in definition II.1. Please note, however, that for processes $p\in\bar{P}_{cl}$ it is in general *not* the case that $abstr^*(p)(\sigma)$ is a *closed* subset of Σ_δ^∞. Fortunately we can prove the following, which turns out to be all we need:

THEOREM III.1: *For every* $p\in\bar{P}_{co}$ *and* $\sigma\in\Sigma$: $abstr^*(p\|p_{St})(\sigma)$ *is compact.*

PROOF

The proof is analogous the one for theorem II.3, given the additional observation that for every $p \in \bar{P}_{co}$ the set

$$(p \| p_{St})(\sigma) \cap (\Sigma \times \bar{P}_{cl})$$

is compact, which we prove now.

According to the definition of $\|$ we have

$$(p \| p_{St})(\sigma) = p(\sigma) \mathbb{L} p_{St} \cup p_{St}(\sigma) \mathbb{L} p \cup p(\sigma)|_{\sigma} p_{St}(\sigma).$$

From the continuity of $\|$ and the compactness of $p(\sigma)$ it follows that

$$(p(\sigma) \mathbb{L} p_{St}) \cap (\Sigma \times \bar{P}_{cl}) = \{<\sigma', p' \| p_{St}> : <\sigma', p'> \in p(\sigma)\}$$

is compact. Secondly, the set

$$(p_{St}(\sigma) \mathbb{L} p) \cap (\Sigma \times \bar{P}_{cl})$$

is empty. Finally, we show that

$$(p(\sigma)|_{\sigma} p_{St}(\sigma)) \cap (\Sigma \times \bar{P}_{cl})$$

is compact. Consider a sequence $(<\sigma, q_i>)_i$ in this intersection. We show that it has a converging subsequence $(<\sigma, q_{k(i)}>)_i$. According to the definition of $|_{\sigma}$ there exist sequences $(<\alpha_i, m_i, \beta_i, f_i, p_i>)_i$ in $p(\sigma)$ and $(<\alpha_i, m_i, g_i>)_i$ in $p_{St}(\sigma)$ such that

$$q_i = g_i(\beta_i)(f_i) \| p_i.$$

Because $p(\sigma)$ is compact there exists a monotonic function $k: \mathbb{N} \to \mathbb{N}$ such that

$$(<\alpha_{k(i)}, m_{k(i)}, \beta_{k(i)}, f_{k(i)}, p_{k(i)}>)_i$$

is convergent. From the definition of the metric on $\bar{P}_{cl}$ it follows that we may assume that there exist α, m and β such that for all i

$$\alpha_{k(i)} = \alpha, \; m_{k(i)} = m, \text{ and } \beta_{k(i)} = \beta.$$

The definition of p_{St} implies that for every $<\alpha, m, g>$ in $p_{St}(\sigma)$ the function g is entirely determined by α and m. Thus

$$(<\alpha_{k(i)}, m_{k(i)}, g_{k(i)}>)_i = (<\alpha, m, g_{k(i)}>)_i = (<\alpha, m, g>)_i,$$

for some g. Suppose we have

$$f = \lim_{i \to \infty} f_{k(i)} \wedge p = \lim_{i \to \infty} p_{k(i)};$$

then $<\alpha, m, \beta, f, p> \in p(\sigma)$ and

$$\lim_{i \to \infty} <\sigma, q_i> = <\sigma, g(\beta)(f) \| p> \in (p(\sigma)|_{\sigma} p_{St}(\sigma)) \cap (\Sigma \times \bar{P}_{cl}). \qquad \square$$

COROLLARY III.2: $abstr^* \circ \Theta_U' \in \mathcal{P}_{fin}(LStat^*) \to P$

(Recall that $P = \Sigma \to \mathcal{P}_{ncompact}(\Sigma_{\delta}^{\infty})$.)

THEOREM III.3: $\Theta_U = abstr^* \circ \Theta_U'$

This theorem can be proved by showing that in addition to Θ_U also $abstr^* \circ \Theta_U'$ is a fixed point of Φ_U. This can be done analogously to the proof of theorem 7.2. From this observation and the fact that Φ_U is a contraction the theorem follows.

The definition of $\mathcal{D}_U^*$, which is given in definition 7.5, is also changed. It will be a function of type

$\mathfrak{D}_U^*$: $\mathcal{P}_{fin}(LStat^*) \rightarrow P_{cl}$

that is like the original $\mathfrak{D}_U^*$ but for the clause that

$$\mathfrak{D}_U^*(\{S_t\}) = p_{St}.$$

A last step towards the goal of this third appendix, which is to prove the semantic equivalence of the denotational and operational semantics with standard objects present, consists of the observation that theorem 7.6, stating that

$$\Phi_U'(\mathfrak{D}_U^*) = \mathfrak{D}_U^*,$$

can be proved for the new version of $\mathfrak{D}_U^*$ as well. The extended proof involves some new case analysis (within Case 2), concerning the communications with standard objects. This being the last appendix, this step being the last step towards our goal, and the author being only human, we omit the details and state without proof:

THEOREM III.4: *(Extended version of 7.6):* $\Phi_U'(\mathfrak{D}_U^*) = \mathfrak{D}_U^*$

COROLLARY III.5: *(Extended version of 7.7):* $\mathcal{O}' = \mathfrak{D}_U^*$

Finally we are ready to prove the extended version of the main theorem (7.9) of our paper:

THEOREM III.6: $[\![U]\!]_{\mathcal{O}} = abstr^*([\![U]\!]_{\mathfrak{D}})$

PROOF

$$\begin{aligned}
[\![U]\!]_{\mathcal{O}} &= \Theta_U[\![\{(\nu(\varnothing), s_n), S_t\}]\!] \\
&= \text{[theorem III.3]} \\
&\quad abstr^*(\Theta_U'(\{(\nu(\varnothing), s_n), S_t\})) \\
&= \text{[corollary III.5]} \\
&\quad abstr^*(\mathfrak{D}_U^*(\{(\nu(\varnothing), s_n), S_t\})) \\
&= abstr^*(\mathfrak{D}_S[\![s_n]\!](\nu(\varnothing))(p_0)\|p_{St}) \\
&= abstr^*([\![U]\!]_{\mathfrak{D}}).
\end{aligned}$$

□

A Semantic Approach to Fairness

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

J.I. Zucker **
Department of Computer Science
State University of New York at Buffalo, USA

In the *semantic* framework of metric process theory, we undertake a general investigation of *fairness* of processes from two points of view: (1) *intrinsic fairness* of processes, and (2) *fair operations* on processes. Regarding (1), we shall define a "fairification" operation on processes called ***Fair*** such that for every (generally unfair) process p the process ***Fair***(p) is fair, and contains precisely those paths of p that are fair. Its definition uses systematic alternation of random choices. The second part of this paper treats the notion of fair operations on processes: suppose given an operator on processes (like merge, or infinite iteration), we want to define a fair version of it. For the operation of infinite iteration we define a fair version, again by a "fair scheduling" technique.
1980 Mathematical Subject classification: 68B10, 68C01, 68C05.
1986 Computing Reviews Categories: D.1.3, F.1.2.
Key words and phrases: Fairness, semantic domains of metric processes, fair infinite iteration, alternation of random choices.

1. Introduction

The most basic context in which the notion of *fairness* can be defined is that of a repetitive choice among alternatives. In [F] the reader can find an elaborate introduction to the notion(s) of fairness, with an extensive overview of the research in this area. Here "fairness" means that, in having to choose repeatedly among alternatives, no alternative will be postponed forever. Usually a nondeterministic programming language is taken as the context for such a study, especially the language of *guarded commands* ([D]).

In this paper we propose a different approach, which could be called a *semantic* one, as opposed to the *language* (or *syntax*) *directed* approach mentioned above. Our point of departure is a *semantic domain* for nondeterministic languages in general, without limiting ourselves to the choice of a particular language. Such a semantic domain will in general be a solution of some reflexive *domain equation*

$$FP \cong P,$$

where F is a functor on some category of mathematical domains, and "$\cong$" means "is isomorphic to". Various techniques have been developed for solving this type of equation. We follow a *metric*

* The research of Jan Rutten was partially supported by ESPRIT project 415: Parallel Architectures and Languages for Advanced Information Processing — a VLSI-directed approach.
** The research of Jeffery Zucker was supported by the National Science Foundation under grant no. DCR-8504296.

approach, introduced by De Bakker and Zucker in [BZ1], and reformulated and extended in a category-theoretic setting in [AR]. The category $\mathcal{C}$ under consideration consists of complete metric spaces, and the functors on $\mathcal{C}$ are so-called *contracting functors*. These spaces are composed from basic metric spaces (sets provided with the trivial 0-1 metric) by the operations of union, Cartesian product, forming function spaces, and forming the set of all (closed) subsets of a given space. Examples would be complete metric spaces satisfying one of the following equations:

$$P \cong A \cup (B \times P), \text{ or}$$

$$P \cong A \cup (B \to (C \times P)),$$

where A, B and C are arbitrary sets and $\cong$ stands for "is isometric to". (Since elements of $\mathcal{C}$ are pairs $<P,d_P>$, consisting of a set P and a metric d_P on P, domain equations over $\mathcal{C}$ should also specify a condition on these metrics. In this introduction, however, we omit such details.)

Another example of a domain is a metric space P satisfying the domain equation:

$$P \cong \{p_0\} \cup \mathcal{P}_{cl}(B \times P).$$

(Here $\mathcal{P}_{cl}(\cdots)$ denotes the set of all closed subsets of $(\cdots)$.) Since this is the domain we shall use in this paper as a starting point for our study of fairness, we discuss it in some detail. The (possibly infinite) set $B = \{b_1, b_2, \ldots\}$ is called the *alphabet* of P. The elements of P are called *processes*. A process $p \in P$ is either p_0, the so-called *nil* process, or a (closed) set of the form

$$p = \{<b_i, p_i> \mid <b_i, p_i> \in B \times P, i \in I\}$$

for some set I of indices. (Here the set I represents the *choice* among alternatives.) Then p can be regarded as a process that for each $i \in I$ can take a step b_i, and then continues with the process p_i (called the *resumption* of b_i). This is itself either p_0, indicating that the process p has terminated after performing step b_i, or again a (closed) set of possible next steps and corresponding resumptions.

Roughly, one can think of these processes as *tree-like* entities. However, there are some differences. Trees with a left branch labeled a and a right branch labeled b, and with a left branch labeled b and a right branch labeled a, are identified, and both are represented by $\{<a,p_0>, <b,p_0>\}$. A tree with only one branch labeled a is identified with a tree with two branches both labeled a. Furthermore, we do not consider arbitrary subsets of $B \times P$, but only closed ones. For an extensive comparison of trees and processes we refer to [BK].

In our approach the elements of B, which are called *basic steps*, are atomic actions, whose possible interpretations have been abstracted from. One such interpretation would be to associate a basic step b_i with each component of a *guarded command*, indicating that the i-th component of that command is selected. Another interpretation would be to regard b_i as an arbitrary action of the i-th component of a system of (possibly infinitely many) active components, indicating that "progress" is being made by that component. A context in which this interpretation makes sense is that of *object-oriented programming* (see e.g. [ABKR]). The basic steps could also be thought of as being different possible actions (e.g. read, write, assignment, etc.) which a single component can perform.

In this framework of metric process theory, we undertake a general investigation of *fairness* of processes from two points of view: (1) *intrinsic fairness* of processes, and (2) *fair operations* on processes.

Regarding (1), a process p is called (*intrinsically*) *fair* if all its paths are fair. A *path* for p is a sequence of pairs: $<a_1,p_1>, <a_2,p_2>, \ldots$, such that $<a_1,p_1> \in p$ and $<a_{i+1},p_{i+1}> \in p_i$ for all $i \geq 1$. The difference between fair and unfair paths can easily be illustrated with a simple example: consider a process $p \in P$ satisfying

$$p = \{<0,p>, <1,p>\}.$$

This process must choose infinitely often (in fact at every step) whether to perform the basic step "0" or the basic step "1". The following path in p

$$<0,p>, <0,p>, <0,p>, \ldots$$

is unfair (with respect to basic step "1"), because step "1" can be taken infinitely often, but never is. An example of a fair path is

$$<0,p>, <1,p>, <0,p>, <1,p>, \ldots$$

There are actually two notions (at least) of fairness current in the literature. The notion we are considering in this paper is often called "strong" fairness (e.g. in [OA]), as opposed to "weak" fairness. In our context a path π would be called weakly fair if every basic step that is from some moment on *continually* enabled in π occurs infinitely often in π. (For the definition of *enabled* see 2.3.) This notion is also called *justice* ([LPS]). A path is *strongly fair* if every basic step that is enabled *infinitely often* (but not necessarily continually) in π occurs infinitely often in π.) The difference between these two notions can again be illustrated with a simple example: consider a process $p \in P$ satisfying

$$p = \{<0,\{<0,p>\}>, <1,\{<1,p>\}>\}.$$

This process can choose infinitely often whether to perform twice the basic step "0", or twice the basic step "1". Then the path in p

$$<0,\{<0,p>\}>, <0,p>, <0,\{<0,p>\}>, <0,p>, \ldots$$

is weakly fair but *not* strongly fair. We do not consider weak fairness further in this paper.

We shall define in section 3 (for a finite alphabet B) a "fairification" operation

$$\textbf{\textit{Fair}}: P \rightarrow P^{Ind}$$

(where P^{Ind} is a suitably extended version of P), such that the process $\textbf{\textit{Fair}}(p)$ is fair, and contains precisely those paths of p that are fair, or, more precisely, representatives of such paths. The relation between $\textbf{\textit{Fair}}(p)$ and p will be clarified by the definition of a mapping from the paths of $\textbf{\textit{Fair}}(p)$ to those paths of p which they represent. Roughly, $\textbf{\textit{Fair}}(p)$ is defined by associating *indices* with the subprocesses (or "*nodes*") of p so as to provide a "bookkeeping" of the way in which alternative subprocesses are chosen in forming paths. These indices indicate priorities for each of the basic steps b_j. During the construction of $\textbf{\textit{Fair}}(p)$, new sets of indices will from time to time be chosen by certain random choices. (This idea of implementing fair scheduling by means of systematic alternation of random choices is well known (see e.g. [AO], [BZ2,3], [P]).) In section 4 this theory is extended to an infinite alphabet, with an "expanding" system of indices (i.e. increasing in length), so that an index at a node records all the (finitely many) basic steps already encountered on the path to that node.

We turn now to (2), the notion of *fair operations* on processes. Suppose given an operation Θ on processes, which is, say, binary: $\Theta: P \times P \rightarrow P$. We want to define a fair version $\Theta_f: P \times P \rightarrow P$ of Θ, such that for all $p_1, p_2 \in P$: first, if p_1 and p_2 are (intrinsically) fair, then so is $\Theta_f(p_1,p_2)$; and second, $\Theta_f(p_1,p_2)$ is *fair with respect to the operation* Θ. This second condition must be explicated for each operation Θ. A good example is the *merge* operation $\| : P \times P \rightarrow P$. In [BZ2,3] a fair version $\|_f$ is defined. In this case the second condition is the requirement that all paths in the resulting process $p_1 \|_f p_2$ must be fair with regard to *alternate scheduling* from p_1 and p_2. A trivial and wrong solution to the problem would be to define

$$p_1 \|_f p_2 = \textbf{\textit{Fair}}(p_1 \| p_2).$$

Obviously, the first condition would be satisfied, but not so the second. The reason for this is, roughly, that in the resulting process $p_1 \|_f p_2$, (intrinsically) unfair paths of $p_1 \| p_2$ that *are* fair with respect to the alternate scheduling from p_1 and p_2 should still be present. The operation ***Fair***, however, would remove them from $p_1 \| p_2$. So this solution would be too coarse. A satisfactory solution was given in [BZ2,3], where the fair merge was defined on the basis of alternate sequences of random choices.

In this paper (section 5) we shall consider another example of an operation on processes, namely *infinite iteration* $(\cdots)^\omega: P \rightarrow P$, defined by

$$p^{\omega} = \lim_{n\to\infty} p^n,$$

where $p^0 = p_0$ and $p^{n+1} = p^n \circ p$. (Here "$\circ$" stands for sequential composition of processes.) We define the *fair* infinite iteration p^{ω_f} of a process $p \in P$ and, after explicating the notion of *fairness with respect to infinite iteration*, prove that the conditions above are indeed satisfied.

An area that remains to be investigated is that of fairness for *non-uniform* processes [BZ1], where our uninterpreted basic actions are replaced by basic state transformations, since here even the *definition* of fairness of paths in such processes is problematic.

RELATED WORK: We already mentioned [F] above, where the reader can find an introduction to the notion(s) of fairness. Next, we mention a few related papers without the intention of giving a complete overview of this area of research.

In [DM], fairness properties are imposed through metrics that allow convergence to fair processes only. The starting point is a simple concurrent language for which a semantics is given with the help of so-called concurrent histories, which are partial orderings describing 'true' concurrency. In [AO] and [CS], proof rules are given for fair transformations in concurrent systems: in the first paper for a fixed number of concurrent components, and in the latter for a (possibly) growing number.

The main difference between the above approaches and ours, is that they consider fairness with respect to parallel (or concurrent) behavior of subprocesses, and we relate fairness to nondeterministic choice (represented by the nodes in our processes). Furthermore, the fact that we study fairness of processes purely at a semantic level, enables us to consider the notion of *arbitrary fair operation* on processes, of which the merge (of concurrent, possibly infinitely many, processes) is just one example.

ACKNOWLEDGEMENTS: It was Jaco de Bakker who first noticed that fair scheduling, implemented by systematic alternation of random choices (as in [P]), could be used to model fair merge in the semantic framework of process domains, as in [BZ2,3]. The second author had useful discussions with Shenquan Xie on fairification and fair infinite iteration.

2. MATHEMATICAL PRELIMINARIES

DEFINITION 2.1 (Domains)
We shall use mathematical domains P of processes p, which are such that:
(1) P is a complete metric space,
(2) P satisfies the following reflexive equation:

$$P \cong \{p_0\} \cup \mathcal{P}_{cl}(A \times P),$$

where $\cong$ stands for "is isometric to", p_0 is a null process, $\mathcal{P}_{cl}(\cdots)$ denotes the set of all closed subsets of $(\cdots)$ and A, with typical elements a, is such that it contains as a subset a (possibly infinite) alphabet

$$B = \{b_1, b_2, \ldots\}$$

of *basic steps*.

We shall not dwell too long upon the mathematical details of the construction of a domain P which satisfies the above definition. Let us just briefly mention two different approaches. First, one can take the metric *completion* of the union of metric spaces $P_0 \subset P_1 \subset \cdots$ defined inductively by

$$P_0 = \{p_0\},$$

$$P_{n+1} \cong \{p_0\} \cup \mathcal{P}_{cl}(A \times P_n).$$

(The metric on P_0 is trivial, the metric on P_{n+1} can be defined using the metric on P_n.) For this

method, full mathematical details and extensive motivation are supplied in [BZ1]. Secondly, one can interpret the reflexive equation for P as defining a functor F on a category of complete metric spaces, thus:

$$FP = \{p_0\} \cup \mathcal{P}_{cl}(A \times P).$$

(The definition of F should also specify a metric for FP.) In [AR] it is shown how to define F as a so-called *contraction*, which has a (unique) fixed point; so

$$FP \cong P.$$

Thus this method also presents us with a solution.

REMARK: We should be more precise about the metrics involved. We should have written the equation above like

$$FP = \{p_0\} \cup \mathcal{P}_{cl}(A \times id_{1/2}(P)),$$

where, for any positive real number c, id_c maps a metric space (M,d) onto (M,d') with $d'(x,y) = c \cdot d(x,y)$. For the details see [AR].

We now introduce a number of concepts related to processes.

DEFINITION 2.2 (Paths)
A *path* for a process $p \in P$ is a (finite or infinite) sequence

$$\pi = (<a_1,p_1>,<a_2,p_2>, \ldots)$$

such that

$$<a_1,p_1> \in p \wedge \forall i \geq 1\, [<a_{i+1},p_{i+1}> \in p_i].$$

We say that π *passes through* p_i, and p_i will be called a *node* of p or a *subprocess* of p (for $i \geq 1$). The set of all paths for p will be called $\boldsymbol{Paths}(p)$.

The following definition explains which processes we want to consider *fair*.

DEFINITION 2.3 (Fairness)
(a) Let $b_i \in B$. Consider a path

$$\pi \equiv (<a_1,p_1>,<a_2,p_2>, \ldots).$$

We say that b_i is *enabled* in π (or i is enabled in π) whenever

$$\exists k \in \mathbb{N}\, \exists q \in P\, [<b_i,q> \in p_k].$$

If $<b_i,q> \in p_k$ we also say that b_i is enabled at step k. We say that b_i *occurs* in π, whenever

$$\exists k \in \mathbb{N}\, [a_k = b_i].$$

(b) We call a path π *fair* whenever for all $b_i \in B$, if b_i is enabled infinitely often in π, then it occurs infinitely often in π.
(c) A process $p \in P$ is called *fair* if all its paths are fair.

EXAMPLE: Let $p \in P$ be such that $p = \{<a,p>,<b,p>\}$. Then b is continually enabled in

$$\pi = (<a,p>,\ <a,p>, \ldots),$$

but never occurs in it. Thus, the path π is unfair.

Please note that only *basic steps* $b_i \in B$ are taken into account in the definition of fairness.

Let P be defined by

$$P \cong \{p_0\} \cup \mathcal{P}_{cl}(B \times P),$$

with B a *finite* alphabet:

$$B = \{b_1, \ldots, b_m\}.$$

Given a process $p \in P$, we want to form a new process ***Fair***(p), which is, in some sense, a fair version of p. For this purpose we want to define a function

$$\textbf{\textit{Fair}}: P \rightarrow P^{Ind}$$

such that there is an obvious correspondence between the paths of ***Fair***(p) and the *fair* paths of p. Here P^{Ind} is given by:

$$P^{Ind} = \{p_0\} \cup \mathcal{P}_{cl}(A \times P^{Ind}),$$

where $A = B \cup$ ***Index***, and ***Index*** is a set of indices (to be defined below). A node p' of a process $p \in P^{Ind}$ with

$$p' = \{<\nu, p_\nu> \mid \nu \in I\},$$

for some subset I of ***Index***, is called a *sum node* and is denoted by

$$p' = \sum_{\nu \in I'} p_\nu.$$

After having defined the function ***Fair***, we shall clarify the relation between p and ***Fair***(p) by defining a mapping

$$\Phi: \textbf{\textit{Paths}}(\textbf{\textit{Fair}}(p)) \rightarrow \textbf{\textit{Paths}}(p),$$

that will satisfy the following two properties. First, for every path $\pi \in$ ***Paths***$($***Fair***$(p))$ we have that $\Phi(\pi)$ is fair. Secondly, any fair path in p will be in the range of Φ. The function ***Fair*** will be defined in such a way that it transforms a process p into a fair process ***Fair***(p) by labeling each node of p with an *index* and, moreover, interspersing some new nodes consisting of *sums* of indices (to be defined below). Indices are the main building blocks in the definition of the function ***Fair***. They are defined as follows.

DEFINITION 3.1 (Indices)
The set ***Index*** of indices, with typical elements ν, is given by

$$\textbf{\textit{Index}} = \{<n_1^{s_1}, \ldots, n_m^{s_m}> \mid$$
$$\forall i \in \{1, \ldots, m\}\ [n_i \geqslant 0 \wedge 0 < s_i \leqslant \infty \wedge (n_i = 0 \Leftrightarrow s_i = \infty)]\},$$

where m is the number of elements in B, and $n_i^{s_i}$ denotes the Cartesian pair $<n_i, s_i>$.

Let p be a process and ν an index. The process p^ν, which is defined below, can be viewed, informally speaking, as a process that behaves like p as far as is allowed by the index ν. Consider the i-th element of ν, say $n_i^{s_i}$. It is related to b_i, the i-th element of our alphabet B. The interpretation of $n_i^{s_i}$ (relative to p) is that in paths starting in p, a step b_i is permitted n_i times with *priority* s_i.

For the priorities s_i we have the convention that a *low* number indicates a *high* priority. It is possible that two or more s_i's have the same value, the corresponding b_i's having the same priority. The symbol ∞ indicates the lowest priority possible. Because it is always associated with an n that is 0, it can also be interpreted as indicating no priority at all.

REMARK
The interpretation of the i-th component $n_i^{s_i}$ is in a sense orthogonal to the approach taken in e.g. [AO]. There a single number z_i is used to indicate the priority of the i-th component of some system of active components. This number z_i indicates, roughly, the number of times a computation can "allow itself" *not to choose* this component as the next one to make progress. In our approach the number n_i indicates the number of times we are allowed *to choose* b_i (the i-th component) as the next step, before another component gets the highest priority.

Now suppose we have a process p containing a step $<b_i,q>$:

$$p = \{\ldots,<b_i,q>,\ldots\};$$

and assume furthermore that we have $\nu \in \textbf{\textit{Index}}$ with

$$\nu = <\ldots,n_i^{s_i},\ldots>$$

where $n_i>0$ and $s_i=\min\{s_1,\ldots,s_m\}$. Then, according to our interpretation of p^ν, it is permitted to choose $<b_i,q>$ as the first step of a path starting from p. With the resumption q of this step will be associated a new index $\nu^-[i]$, in which n_i is decreased by one. If $n_i>1$ nothing happens to the priority s_i of b_i. If $n_i=1$ (and so decreased to 0) it is, for the time being, the last time that b_i is allowed, and s_i is changed to ∞ (the lowest priority possible). As we will see, at some later stage it will be taken care of that n_i and s_i are reset again, so that $n_i>0$ and $s_i<\infty$. All this is formalized in the following definition.

DEFINITION 3.2
Let $\nu \in \textbf{\textit{Index}}$ be such that

$$\nu = <n_1^{s_1},\ldots,n_i^{s_i},\ldots,n_m^{s_m}>,$$

and let $i\in\{1,\ldots,m\}$. We define

$$\nu^-[i] = \begin{cases} <n_1^{s_1},\ldots,(n_i-1)^{s_i},\ldots,n_m^{s_m}> & \text{if } n_i>1 \\ <n_1^{s_1},\ldots,0^{\infty},\ldots,n_m^{s_m}> & \text{if } n_i=1 \\ \text{undefined} & \text{if } n_i=0. \end{cases}$$

There is one other operation on indices we shall need.

DEFINITION 3.3
Let $\nu \in \textbf{\textit{Index}}$ be such that

$$\nu = <n_1^{s_1},\ldots,n_m^{s_m}>,$$

then

$$N(\nu) = \{<\tilde{n}_1^{\tilde{s}_1},\ldots,\tilde{n}_m^{\tilde{s}_m}> \mid$$
$$\forall j\in\{1,\ldots,m\}\ [(n_j=0\wedge s_j=\infty)\Rightarrow(\tilde{n}_j>0\wedge\tilde{s}_j=s+1)\wedge$$
$$(n_j>0\wedge s_j<\infty)\Rightarrow(\tilde{n}_j=n_j\wedge\tilde{s}_j=s_j)]\}$$

where $s = \max(\{s_1, \ldots, s_m\} \setminus \{\infty\})$.

The elements $\tilde{\nu}$ in $N(\nu)$ are obtained from ν by changing, for all i with $n_i = 0$ and $s_i = \infty$, the value of n_i to an arbitrary positive number and the value of s_i to $s+1$. In words, this means that b_i is again allowed to be chosen ($\tilde{n}_i$ times) but with a priority lower than all other priorities present in ν that are not ∞. This definition will also be used in the definition of ***Fair***, where it will be further elucidated. We now give this definition, upon which an explanation will follow.

DEFINITION 3.4 (Fairification)
We define a function

$$\textit{Fair}: P \to P^{Ind}.$$

Let $p \in P$. Then

$$\textit{Fair}(p) = \sum_{\nu \in I_0} \textit{fair}(p, \nu),$$

where

$$I_0 = \{<n_1^1, \ldots, n_m^1> \mid n_i > 0,\ i = 1, \ldots, m\}$$

and

$$\textit{fair}: P \times \textit{Index} \to P^{Ind}$$

is defined as follows. (We often write p^ν for $\textit{fair}(p, \nu)$.) For all $\nu \in \textit{Index}$ we define

$$\textit{fair}(p_0, \nu) = p_0.$$

For $p \neq p_0$ we distinguish two cases.
Case 1:

$$\text{If } \exists i \in \{1, \ldots, m\}\, [n_i > 0 \wedge s_i < \infty \wedge \textit{enabled}(i)],$$
$$\text{then } p^\nu = \{<b_j, q^{\nu[i]}> \mid <b_j, q> \in p \wedge s_j = \min\{s_1, \ldots, s_m\}\}.$$

Case 2:

$$\text{If } \forall i \in \{1, \ldots, m\}\, [\textit{enabled}(i) \Rightarrow (n_i = 0 \wedge s_i = \infty)],$$
$$\text{then } p^\nu = \sum_{\tilde{\nu} \in N(\nu)} p^{\tilde{\nu}}.$$

REMARKS
(1) The definition of $\textit{fair}: P \times \textit{Index} \to P^{Ind}$ is self-referential and therefore needs some justification. We observe that ***fair*** could be defined as the fixed point of a mapping

$$\Phi: (P \times \textit{Index} \to P^{Ind}) \to (P \times \textit{Index} \to P^{Ind}),$$

which can be defined according to the definition scheme of ***fair*** above. It is straightforward to see that such a definition yields a contracting function, which thus has a unique fixed point (cf. Banach's fixed point theorem ([BZ1], [AR])).
(2) Because case 2 never occurs twice in succession, $\textit{fair}(p, \nu)$ never contains two sum nodes successively.
(3) Every node in $\textit{Fair}(p)$ is either a sum node, or of the form $\{<b_{i_j}, p_j> \mid j \in I\}$, for some set of indices I.
(4) We give some informal intuition for this definition. The indices $\nu \in \textit{Index}$ in the definition above can be interpreted as strategies for the construction of a process $\textit{Fair}(p)$ such that every path in

this process will be fair with respect to every b_i in B. An element ν in I_0 can be regarded as permission, for each i, to choose b_i n_i times. All i are supplied at the beginning with the same priority, that is 1.

We will treat p^ν for the case that $p \neq p_0$. As long as case 1 applies there is no need to change our strategy or, in other words, to choose a new ν. Each b_i that is enabled at p, and for which $n_i > 0$ and $s_i = \min\{s_1, \ldots, s_m\}$, may be chosen as the next step in the new process we are constructing. The index ν is changed according to the definition of $\nu^-[i]$, so n_i is decreased by 1 and the priority s_i remains constant, unless n_i was 1. Then it is set to ∞, indicating no priority at all. Because every application of case 1 causes the decrease of an n_i, it is obvious that after a finite number of such applications case 2 must hold. For didactic purposes we shall now make a conceptual distinction between two possible situations that may arise in this case. Formally however, as may be inferred from the definition of case 2, this is not necessary.

First, it may be the case that all n_i's have been decreased to 0 (and all s_i's have been set to ∞). Then we can consider the strategy suggested by the ν we started with to be a great success: every b_i has been chosen the number of times we had in mind for it (n_i). The fact that originally all n_i's were strictly positive implies that so far we have made sure that all b_i's have been treated fairly. It is clear what to do next: we can just restart by choosing a new index ν, with all n_i strictly positive and all s_i set to 1. According to the definition of $N(\nu)$, this is exactly what happens in this case.

The second situation is more typical. It concerns the case that for all i that are *enabled* at p, $n_i = 0$ and $s_i = \infty$. But we have not finished the strategy suggested by the original ν, because there exists at least one j *not* enabled at p, with $n_j > 0$ and $s_j < \infty$. Although we have not finished our first strategy, we are forced to change it because it does not tell us what to do about the i's that are enabled at p. A new strategy $\tilde{\nu}$ is defined such that for all j with $n_j > 0$ and $s_j < \infty$ these values remain unchanged, thus preserving that part of the first strategy (ν) that has not yet been dealt with. For all other i (enabled or not enabled) the value of n_i is set to an arbitrary strictly positive number, and the value of s_i to $\max\{s_1, \ldots, s_m\} + 1$. So the new priority introduced here is lower than all the already existing priorities. When at a later stage one of the j's, for which n_j and s_j remain unchanged here, is enabled, it will take precedence over those i's for which a new priority is introduced. Thus a fair treatment of such j's is ensured for the future.

Now for the rest of this section let $p \in P$ be fixed. We define a mapping

$$\Phi: \textbf{\textit{Paths}}(\textbf{\textit{Fair}}(p)) \rightarrow \textbf{\textit{Paths}}(p),$$

relating to each path π in $\textbf{\textit{Fair}}(p)$ a fair path in p. For its formal definition we shall make use of the following lemma.

LEMMA 3.5
For all $\bar{p} \in P$ with $\bar{p} \neq p_0$, $\nu \in \textbf{\textit{Index}}$ and $<a,q> \in \textbf{\textit{fair}}(\bar{p},\nu)$, there exist $p' \in P$ and $\nu' \in \textbf{\textit{Index}}$ such that

$$q = \textbf{\textit{fair}}(p',\nu') \wedge$$

$$a \in \textbf{\textit{Index}} \Rightarrow p' = \bar{p} \wedge$$

$$a \in B \Rightarrow <a,p'> \in \bar{p}.$$

The proof is straightforward from the definition of $\bar{p}^\nu$ $(= \textbf{\textit{fair}}(\bar{p},\nu))$.

DEFINITION 3.6 (The mapping Φ)
Let

$$\pi = <a_0,q_0>, <a_1,q_1>, \ldots$$

be a path in $\textbf{\textit{Fair}}(p)$. By the above lemma and the definition of $\textbf{\textit{Fair}}(p)$ we can rewrite it as

$$\pi = <a_0,p^{\nu}>, <a_1,p_1^{\nu_1}>, \ldots,$$

for certain $\nu,\nu_1, \ldots \in$ ***Index*** and $p_1,p_2, \ldots \in P$. Now if we delete all pairs $<a_i,p_i^{\nu_i}>$ with $a_i \in$ ***Index***, and all superscripts ν_i, we get a sequence

$$\Phi(\pi) = <a_{i_1},p_{i_1}>, <a_{i_2},p_{i_2}>, \ldots,$$

which is a path in p. We call $\Phi(\pi)$ the path in p *corresponding* to the path π in ***Fair***(p). This defines a mapping

$$\Phi: \textbf{\textit{Paths}}(\textbf{\textit{Fair}}(p)) \rightarrow \textbf{\textit{Paths}}(p).$$

Next, we have an important theorem.

THEOREM 3.7
Fair(p) *is fair. That is, for all* $\pi \in$ ***Paths***$(\textbf{\textit{Fair}}(p))$, π *is fair.*

PROOF
Let $\pi \in$ ***Paths***$(\textbf{\textit{Fair}}(p))$ be such, that

$$\begin{aligned}\pi &= <a_1,q_1>,<a_2,q_2>, \ldots \\ &= <a_1,p_1^{\nu_1}>,<a_2,p_2^{\nu_2}>, \ldots.\end{aligned}$$

Suppose b_i is enabled infinitely often in π. We must show that b_i occurs infinitely often within π. It is sufficient to show that for any j, if b_i is enabled at the node $p_j^{\nu_j}$ of π, then b_i occurs further on in the path π, that is, for some $j' \geqslant j$: $b_i = a_{j'}$.
We consider the sequence $\nu_j, \nu_{j+1}, \ldots$ and observe that for every $k \in \mathbb{N}$, ν_{k+1} is obtained from ν_k by an application of case 1 or 2 in the definition of ***fair***(p,ν) (definition 3.4). Now let

$$\nu_j = <n_1^{s_1}, \ldots, n_m^{s_m}>.$$

We consider all possible cases.

(1) $n_i = 0$:
Then $s_i = \infty$. For every application of case 1 (above) one of the n_k's must decrease. Therefore eventually case 2 must apply, which makes all n_k's positive and brings us to the next case.

(2) $n_i > 0$: This implies $s_i < \infty$. As long as s_i is not the highest priority, the following may happen. Any application of case 1 results in *either* the decrease of an n_k, not to 0, *or* the decrease of an n_k to 0 and the removal of a higher priority than s_i. After a finite number of applications of case 1, the latter must happen. Any application of case 2 introduces only priorities that are lower than s_i, and must be followed by an application of case 1. Furthermore, during any of these applications, n_i and s_i remain constant. It follows then that eventually s_i will be the highest priority. Because b_i is enabled infinitely often in π, it must be enabled at some step beyond this, at which point case 1 will be applied to it and b_i will occur at the next step.

Now that we have proved that we did not promise too much, that is to say that ***Fair***(p) indeed contains only fair paths, let us also make sure that for all fair paths in p there is a corresponding path in ***Fair***(p).

THEOREM 3.8
Any fair path in p *is in the range of the mapping* Φ.

PROOF
Given a fair path $\pi' \in$ ***Paths***(p), we must construct a path $\pi \in$ ***Paths***$(\textbf{\textit{Fair}}(p))$ such that

$\Phi(\pi) = \pi'$.

First, we partition the set $\{1, \ldots, m\}$ into two parts F and I, where F is the set of all i such that b_i is enabled finitely often (perhaps never) in π', and I is the set of all i such that b_i is enabled infinitely often in π'. Thus:

$$\{1, \ldots, m\} = I \cup F.$$

Note that for all $i \in F$, b_i occurs only finitely often in π', and for all $i \in I$, b_i occurs infinitely often in π', since π' is fair. Let $l_1 \in \mathbb{N}$ be so big that

(1) no b_i with $i \in F$ is enabled in the part of π' at or after step l_1;

(2) every b_i with $i \in I$ occurs at least once by then.

Now for $i = 1, \ldots, m$, let n_i' be the number of times that b_i occurs before (or at) step l_1 and then define

$$n_i = \begin{cases} n_i' + 1 & \text{if } i \in F \\ n_i' & \text{if } i \in I. \end{cases}$$

We define our first index ν_1 by

$$\nu_1 = \langle n_1^1, \ldots, n_m^1 \rangle.$$

Now we can construct the first part of the path π corresponding with the part of π' before step l_1, by starting with p^ν, and repeatedly applying case 1 for the appropriate b_i, thus decreasing the n_i's until (at step l_1) our index is such that for all $i \in \{1, \ldots, m\}$:

$$i \in F \Rightarrow n_i = 1 \wedge s_i = 1,$$
$$i \in I \Rightarrow n_i = 0 \wedge s_i = \infty.$$

Now case 2 must be applied to get a sum node, since no $i \in F$ is enabled at step l_1. To determine the following index ν_2 we again choose a number $l_2 \in \mathbb{N}$, with $l_2 > l_1$, such that every b_i with $i \in I$ occurs at least once between steps l_1 and l_2 (including l_1, excluding l_2). Then choose an index ν_2 such that, for $i \in I$, n_i denotes the number of occurrences of b_i between l_1 and l_2. We proceed as before, constructing the part of π' between l_1 and l_2. Continuing in this way, we construct a path π in $\boldsymbol{Fair}(p)$ such that $\Phi(\pi) = \pi'$.

REMARK: This function Φ is *not* bijective. In general there are more than one (in fact, infinitely many) paths in $\boldsymbol{Fair}(p)$ that are mapped by Φ to the same path in p.

4. FAIRIFICATION OF PROCESSES WITH INFINITE ALPHABET

We now want to extend our technique of fairification to a set of processes, which we shall (again) call P, defined by

$$P \cong \{p_0\} \cup \mathcal{P}_{cl}(B \times P),$$

with B an *infinite* alphabet:

$$B = \{b_1, b_2, \ldots\}.$$

We shall again define a function

$$\boldsymbol{Fair}: P \to P^{Ind},$$

where P^{Ind} is given by

$$P^{Ind} = \{p_0\} \cup \mathcal{P}_{cl}(A \times P^{Ind}),$$
$$A = B \cup \textbf{\textit{Index}},$$

with ***Index*** to be defined below. We shall repeat the approach of the previous section with some small but essential changes. The definitions, lemmas and theorems that need not be changed will be mentioned, but not repeated in full.
An important change is the new definition of indices. They no longer have a fixed length.

DEFINITION 4.1 (Indices)
The set ***Index*** of indices, with typical elements ν, is given by

$$\textbf{\textit{Index}} = \bigcup_{m \in \mathbb{N}} \textbf{\textit{Index}}^{[m]},$$

with

$$\textbf{\textit{Index}}^{[m]} = \{<n_1^{s_1}, \ldots, n_m^{s_m}> \mid \forall i \in \{1, \ldots, m\}\ [n_i \geqslant 0 \wedge 0 < s_i \leqslant \infty \wedge (n_i = 0 \Leftrightarrow s_i = \infty)]\}.$$

An index of length k is related to the first k elements of our alphabet B. The interpretation of n_i and priority s_i is as before. When we define, for a given process p, a fair version ***Fair***(p), we shall, during the construction, increase the length of the indices used, thus considering fairness with respect to a growing number of basic steps b_i. Once the length of an index is bigger than or equal to some $i \in \mathbb{N}$, it is ensured that b_i is treated fairly thereafter. The definition of the first operation on indices, $\nu^-[\cdots]$, remains unchanged, but for the fact that the original definition (3.2) should hold for indices of arbitrary length. The most important adaptation of this section lies in the following new definition of $N(\nu)$.

DEFINITION 4.2
Let $\nu \in$ ***Index*** be such that $\nu = <n_1^{s_1}, \ldots, n_m^{s_m}>$ and let $p \in P$. We define

$$\begin{aligned} N(\nu,p) = \{<\tilde{n}_1^{\tilde{s}_1}, \ldots, \tilde{n}_{m'}^{\tilde{s}_{m'}}> \mid\ & \\ & m' > m \wedge \\ & \{k \mid 1 \leqslant k \leqslant m' \wedge \tilde{n}_k > 0\} \cap \{k \mid k \text{ enabled at } p\} \neq \varnothing \wedge \\ & \forall j\ [((1 \leqslant j \leqslant m) \wedge n_j = 0 \wedge s_j = \infty) \Rightarrow (\tilde{n}_j > 0 \wedge \tilde{s}_j = s+1)) \wedge \\ & ((1 \leqslant j \leqslant m) \wedge n_j > 0 \wedge s_j < \infty) \Rightarrow (\tilde{n}_j = n_j \wedge \tilde{s}_j = s_j)) \wedge \\ & m < j \leqslant m' \Rightarrow (\tilde{n}_j > 0 \wedge \tilde{s}_j = s+1) \vee (\tilde{n}_j = 0 \wedge \tilde{s}_j = \infty))]\} \end{aligned}$$

where $s = \max\{s_1, \ldots, s_m\}$.

Let us see how this definition is used in the definition of the function ***Fair*** below, and then try to comment on its intuitive interpretation. Although we do not change the definition of ***Fair*** (definition 3.4), we repeat its most interesting part and discuss it in the context of the altered definition of $N(\nu)$.

If $p \in P$ with $p \neq p_0$, then p^ν $(= \textbf{\textit{fair}}(p,\nu))$ is given by:
Case 1:
If $\exists i \in \{1, \ldots, \textit{length}(\nu)\}\ [n_i > 0 \wedge s_i < \infty \wedge \textit{enabled}(i)]$,
then $p^\nu = \{<b_j, q^{\nu^-[i]}> \mid <b_j, q> \in p \wedge s_j = \min\{s_1, \ldots, s_{\textit{length}(\nu)}\}\}$.
Case 2:

If $\forall i \in \{1, \ldots, length(\nu)\}\ [enabled(i) \Rightarrow (n_i = 0 \wedge s_i = \infty)]$,

then $p^{\nu} = \sum_{\tilde{\nu} \in N(\nu,p)} p^{\tilde{\nu}}$.

The interpretation of case 1 is the same as before. When the condition of case 2 holds, we are obliged to change our strategy, that is to choose a new index, because our current strategy does not say anything about the i's that are enabled at p. This can have two reasons. For such an i we either have $n_i = 0$ and $s_i = \infty$ or $i > length(\nu)$. In order to be able to continue our construction, we therefore allow several new strategies $\tilde{\nu} \in N(\nu,p)$, which all must satisfy the following constraints. First, the part of the old strategy ν that has not been dealt with yet has to be preserved: for $1 \leqslant i \leqslant length(\nu)$ with $n_i > 0$ and $s_i < \infty$ we have $\tilde{n}_i = n_i$ and $\tilde{s}_i = s_i$. Then, for $1 \leqslant i \leqslant length(\nu)$ with $n_i = 0$ and $s_i = \infty$, the values of n_i and s_i are reset: $\tilde{n}_i$ arbitrary positive, $\tilde{s}_i = 1 + s$. As in the finite case, the new priority is lower than the existing ones. Because we want each $b_k \in B$ eventually to be treated fairly, for each k there should be a moment in our construction where an index ν is introduced with $length(\nu) > k$. Therefore we require the length of the new index $\tilde{\nu}$ to be strictly greater than the length of ν. For the newly introduced j's ($length(\nu) < j \leqslant m$) we require

$$(\tilde{n}_j > 0 \wedge \tilde{s}_j = s + 1) \vee (\tilde{n}_j = 0 \wedge \tilde{s}_j = \infty).$$

Although here $\tilde{n}_j = 0$ is allowed, we know that the next time that case 2 is applied $\tilde{n}_j$ will be set to a strictly positive value. The newcomers, so to speak, are granted one (and only one) moment of respite. The motivation for this generosity lies in the rather selfish wish to prove theorem 4.4. It appears that it would be too restrictive to demand for all such j that $\tilde{n}_j > 0$. Finally, the condition that

$$\{k \mid 1 \leqslant k \leqslant m \wedge \tilde{n}_k > 0\} \cap \{k \mid k \text{ enabled at } p\} \neq \varnothing$$

entails that case 2 can never occur twice in succession.

Now for the rest of this subsection let $p \in P$ be fixed. We define a mapping

$$\Phi: \textbf{\textit{Paths}}(\textbf{\textit{Fair}}(p)) \rightarrow \textbf{\textit{Paths}}(p),$$

relating to each path π in $\textbf{\textit{Fair}}(p)$ a fair path in p, in exactly the same way as in definition 3.6. We finally repeat theorems 3.7 and 3.8 of the previous section, which together show that the definition of $\textbf{\textit{Fair}}(p)$ (using the new definition of $N(\nu,p)$) is satisfactory. The former proofs of these theorems have to be altered, as can be seen below.

THEOREM 4.3
$\textbf{\textit{Fair}}(p)$ is fair. That is, for all $\pi \in \textbf{\textit{Paths}}(\textbf{\textit{Fair}}(p))$, π is fair.

PROOF
Let $p \in P$ and let $\pi \in \textbf{\textit{Paths}}(\textbf{\textit{Fair}}(p))$ be such that

$$\pi \equiv <a_1, q_1>, <a_2, q_2>, \ldots$$
$$\equiv <a_1, p_1^{\nu_1}>, <a_2, p_2^{\nu_2}>, \ldots.$$

Suppose b_i is enabled infinitely often in π. We must show that b_i occurs infinitely often within π. From the construction of $\textbf{\textit{Fair}}(p)$ it follows that in the sequence $(\nu_j)_j$ each index ν_{j+1} is obtained from ν_j by an application of case 1 or 2. Since case 1 can be applied only finitely many times in succession, it follows that case 2 must have been applied infinitely many times, each application increasing the length of the index. Therefore there is an $N \in \mathbb{N}$ such that for all $j > N$:

$$length(\nu_j) > i.$$

Now we are back in the old situation of the previous section! The proof can be completed as before, but for the new observation that with the increase of the length of an index, only priorities lower than the existing ones are introduced.

THEOREM 4.4
Any fair path in p is in the range of the mapping Φ.

PROOF
Given a fair path $\pi' \in \textbf{\textit{Paths}}(p)$,

$$\pi' = <b_{i_1}, p_1>, <b_{i_2}, p_2>, \ldots,$$

we must construct a path $\pi \in \textbf{\textit{Paths}}(\textbf{\textit{Fair}}(p))$ such that

$$\Phi(\pi) = \pi'.$$

First, we partition $\mathbb{N}$ into two parts F and I, where F is the set of all i such that b_i is enabled finitely often (perhaps never) in π', and I is the set of all i such that b_i is enabled infinitely often in π'. Thus:

$$\mathbb{N} = I \cup F.$$

Note that (as in 3.4) for all $i \in F$ b_i occurs only finitely often in π' and for all $i \in I$ b_i occurs infinitely often in π', since π' is fair. Secondly, we introduce the following functions that will be very useful in our proof.

(a) For all $L \in \mathbb{N}$ we define a (*position*) function $Pos_L : B \to \mathbb{N}$ by

$$Pos_L(b_k) = \begin{cases} \text{smallest } L' \geqslant L \text{ such that} \\ \forall j \geqslant L' \, [b_k \notin p_j] & \text{if } k \in F \\ \text{smallest } L' \geqslant L \text{ such that} \\ \exists j \, [L < j < L' \wedge b_{i_j} = b_k] & \text{if } k \in I. \end{cases}$$

For $k \in F$ this function gives the smallest position greater than L after which b_k is never enabled again. For $k \in I$ the smallest position greater than L is chosen such that b_k has occurred (at least) once since L.

(b) For all $L, L' \in \mathbb{N}$, with $L \leqslant L'$, we define a (*number*) function $Num_{L,L'} : B \to \mathbb{N}$ by

$$Num_{L,L'}(b_k) = \begin{cases} 1 + \text{ (number of occurrences in } \pi' \\ \quad \text{of } b_k \text{ between } L \text{ and } L') & \text{if } k \in F \\ \text{(number of occurrences in } \pi' \\ \quad \text{of } b_k \text{ between } L \text{ and } L') & \text{if } k \in I. \end{cases}$$

(In this definition *between* L and L' means *including* L and *excluding* L'.)
We shall define, at each of an infinite sequence of stages k, an index ν_k and, corresponding to that index, the k-th part of the path π corresponding to π'. After we have constructed, at stage $k-1$, the $(k-1)$-th approximation of path π corresponding to the initial segment

$$<b_{i_1}, p_1>, \ldots, <b_{i_l}, p_l>$$

of path π', then at stage k we shall take into account the basic steps $b_{i_{l+1}}$ and all the b_j's we have encountered in the preceding stages. We shall make sure that the length of the index ν_k will be, as prescribed by definition 4.2, strictly bigger than the length of ν_{k-1}. Note that in the previous section, where our alphabet was finite, from the beginning we could focus on all b_j's at the same time.

Stage 1
For the definition of our first index ν_1 we focus on basic step b_{i_1}. We define

$$L_1 = Pos_1(b_{i_1}),$$

$$R_1 = \{i_1, \ldots, i_{L_1-1}\},$$

$$M_1 = \max R_1.$$

Our first index ν_1, with $\nu_1 = <n_1^{s_1}, \ldots, n_{M_1}^{s_{M_1}}>$, is defined so that

$$\forall 1 \leqslant j \leqslant M_1\ [j \in R_1 \Rightarrow (n_j = Num_{1,L_1}(b_j) \wedge s_j = 1) \wedge$$
$$j \notin R_1 \Rightarrow (n_j = 0 \wedge s_j = \infty)].$$

The length of ν_1 is M_1, because according to the definition of indices no holes are allowed in ν_1, that is: every index is related to an initial part of the enumeration of our infinite alphabet $\{b_1, b_2, \ldots\}$. For those basic steps b_j that do not occur in the path π' before place L_1, default values $n_j = 0$ and $s_j = \infty$ are chosen in ν_1. (Here we use the fact that for newly introduced j's, n_j can get the value 0 once. See the corresponding remark in the explanation following definition 4.2.) With ν_1 we can construct the first part of π corresponding to the part of π' before L_1, starting with p^{ν_1}, and repeatedly applying case 1 for the appropriate b_i, thus decreasing the n_i's until (at step L_1) our index is such that for all $1 \leqslant i \leqslant M_1$:

$$(i \in F \cap R_1) \Rightarrow (n_i = 1 \wedge s_i = 1),$$

$$(i \in I \cap R_1 \vee i \notin R_1) \Rightarrow (n_i = 0 \wedge s_i = \infty).$$

Now case 2 must be applied, since no $j \in F \cap R_1$ is enabled at step L_1. This brings us to stage 2.

Stage 2

We define our next index ν_2, taking into account all steps encountered at stage 1, that is all b_i's with $1 \leqslant i \leqslant M_1$, and the next step in the path π', that is $b_{i_{L_1}}$. We define

$$L_2 = \max(\{Pos_{L_1}(b_{i_{L_1}})\} \cup \{Pos_{L_1}(b_k) \mid 1 \leqslant k \leqslant M_1\}),$$

$$R_2 = \{1, \ldots, M_1\} \cup \{i_{L_1}, \ldots, i_{L_2-1}\},$$

$$M_2 = 1 + \max R_2.$$

We define our second index ν_2, with $\nu_2 = <\tilde{n}_1^{\tilde{s}_1}, \ldots, \tilde{n}_{M_2}^{\tilde{s}_{M_2}}>$, such that

$$\forall 1 \leqslant j \leqslant M_2\ [((1 \leqslant j \leqslant M_1 \wedge n_j = 0) \vee (j > M_1 \wedge j \in R_2) \Rightarrow$$
$$\tilde{n}_j = Num_{L_1,L_2}(b_j) \wedge \tilde{s}_j = 1 + \max\{s_k \mid 1 \leqslant k \leqslant M_1\}) \wedge$$
$$(1 \leqslant j \leqslant M_1 \wedge n_j = 1) \Rightarrow (\tilde{n}_j = n_j \wedge \tilde{s}_j = s_j) \wedge$$
$$(j \notin R_2) \Rightarrow (\tilde{n}_j = 0 \wedge \tilde{s}_j = \infty)].$$

Note that M_2, the length of ν_2, is strictly bigger than M_1, the length of ν_1. We proceed as before, constructing the part of π corresponding to the part of π' between L_1 and L_2. Continuing in this way, we construct a path π in ***fair***(p) such that $\Phi(\pi) = \pi'$.

5. Infinite iteration

Let P be the mathematical domain of section 3, that is, a complete metric space satisfying

$$P \cong \{p_0\} \cup \mathcal{P}_{cl}(B \times P)$$

where B is a finite alphabet

$$B = \{b_1, \ldots, b_m\}.$$

The operation of sequential composition on P is defined in

Definition 5.1 (Sequential composition)
Let $\circ$: $P \times P \to P$ be given by

$$p \circ q = \begin{cases} q & \text{if } p = p_0 \\ \{<b,p' \circ q> \mid <b,p'> \in p\} & \text{if } p \neq p_0 \end{cases}$$

for all p and q in P.

Remarks
(1) Because this definition is self-referential, it needs some justification. We observe that $\circ$ can be defined as the unique fixed point of a contraction Φ of type Φ: $(P \times P \to P) \to (P \times P \to P)$. (Cf. definition 3.4.)
(2) It is not very difficult to show that:

$$\forall p,q,q' \in P\ [p \neq p_0 \Rightarrow d_P(p \circ q, p \circ q') \leqslant \tfrac{1}{2} d_P(q,q')].$$

We shall use this property below.

In this section we want to study the operation of *infinite iteration* of a process $p \in P$. It is defined as follows:

Definition 5.2 (Infinite iteration)
Let $(\ \cdots\ )^\omega$: $P \to P$ be given by

$$p^\omega = \lim_{n \to \infty} p^n$$

for $p \in P$, where $p^0 = p_0$ and $p^{n+1} = p^n \circ p$.
(This limit exists, as can be easily proved using the property of remark (2) above).

Let us now explain how fairness issues come into play by taking the infinite iteration of $p \in P$. Generally, taking the infinite iteration of a process $p \in P$ introduces new infinite paths in p^ω that were not yet present in p. When we take, for example, $p = \{<a,p_0>, <b,p_0>\}$, then p does not contain any infinite paths, whereas p^ω, which satisfies

$$p^\omega = \{<a,p^\omega>, <b,p^\omega>\},$$

contains many. Some of these are unfair, such as

$$\pi = <a,p^\omega>, <a,p^\omega>, <a,p^\omega>, \ldots,$$

which is unfair with respect to b_1. Such unfair paths π we call *globally* unfair. We do *not* call every unfair path in p^ω globally unfair, only those that are introduced, so to speak, by taking the infinite iteration of p. Another example may illustrate this point. (Formal definitions follow below.) Consider a process $p \in P$ satisfying

$$p = \{<a,p>,<b,p_0>\}.$$

Then p^ω will contain the unfair paths

$$<a,p>,<a,p>,\ldots,$$

$$<b,p>,<a,p>,<a,p>,\ldots,$$

$$<b,p>,<b,p>,<a,p>,<a,p>,\ldots, \text{ etc.}$$

The unfairness of these paths is, as it were, reducible to the unfairness of the path

$$<a,p>,<a,p>,\ldots,$$

which was already present in p. Therefore they will not be called globally unfair paths.

There is a second notion of unfairness, which plays a role here. It is called *node* (or *local*) unfairness. Again we explain it here by giving an example, the formal definition following below. Let $p \in P$ contain the node $p' = \{<a,p_1>,<b,p_2>\}$. Let $\pi \in \textbf{\textit{Paths}}(p^\omega)$ and suppose π passes through p' infinitely many times. If it is the case that in π the next step that is taken after passing through p' is always a, and never b, we call π *node* unfair (with respect to the node p'). The reason for this terminology is obvious: although b is infinitely often enabled in π *at node* p', it is never chosen in π as the next step *after* p'.

The notions of global and node unfairness are in a sense independent. Let $p \in P$ be given by

$$p = \{<b,p'>\}, \quad \text{where}$$

$$p' = \{<a,p>,<b,p_0>\}.$$

Consider $\pi \in \textbf{\textit{Paths}}(p^\omega)$, given by

$$\pi = <b,p'>,<a,p>,<b,p'>,<a,p>,\ldots.$$

This path is not globally unfair, but is node unfair with respect to the node p'. Thus node unfairness does not imply global unfairness. The same holds in the opposite direction. Let $p \in P$ be defined by

$$p = \{<a^n,\{<a,p_0>,<b,p_0>\}> \mid n \in \mathbb{N}\} \cup \{a^\omega\},$$

using a^n and a^ω as shorthand with an obvious interpretation. (The fact that $a^\omega \in p$ is not important for the point we want to make with this example, but is implied by the (topological) closedness of p.) Now it is not difficult to find a path

$$\pi = <a,p_1>,<a,p_2>,<a,p_3>,\ldots$$

in $\textbf{\textit{Paths}}(p^\omega)$ (with $p_1,p_2,p_3,\ldots$ nodes of p) that is globally unfair (with respect to b), but fair with respect to every node of p, although it passes through p infinitely many times.

Let us now proceed with formally defining these notions of global and node unfairness. Actually, we shall define what we consider to be globally fair and node fair. For this we need the following notion.

DEFINITION 5.3 (Iteration paths)
Let $p \in P$, $\pi \in \textbf{\textit{Paths}}(p^\omega)$. We call π an *(infinite) iteration path*, whenever π is the concatenation of an infinite sequence of finite paths $\pi_1,\pi_2,\ldots \in \textbf{\textit{Paths}}(p)$:

$$\pi = \pi_1 \circ \pi_2 \circ \pi_3 \circ \ldots.$$

For a basic step b occurring in π_k we say that b occurs in the *k-th instantiation* of p.

REMARK
We have not defined the concatenation of finite paths. It is just what one would expect: if $\pi_1 = <a_1,p_1>,\ldots,<a_m,p_0>$, and $\pi_2 = <b_1,q_1>,\ldots,<b_m,p_0>$ are finite paths in $\textbf{\textit{Paths}}(p)$,

then:

$$\pi_1 \circ \pi_2 = <a_1,p_1>, \ldots, <a_n,p>, <b_1,q_1>, \ldots, <b_m,p_0>.$$

(Note that finite paths always end in $<a,p_0>$, for some $a \in B$.)

DEFINITION 5.4 (Global fairness)
Let $p \in P$, $\pi \in \mathbf{Paths}(p^\omega)$. We call π *globally fair* whenever
(1) π is fair (in the sense of definition 2.3); or
(2) π is *not* an iteration path.
We call p^ω globally fair whenever all paths in p^ω are globally fair.

REMARK: It follows that a path in p^ω is globally unfair if and only if it is an iteration path and unfair.

DEFINITION 5.5 (Node fairness)
Let $p \in P$, $\pi \in \mathbf{Paths}(p^\omega)$. We call π *node fair with respect to* p', for a subnode p' of p, whenever it is the case that: if π passes through p' infinitely often, then for all $b \in B$ that are enabled in p': b occurs infinitely often in π, *immediately after* p'. We call π node fair if it is node fair with respect to every subnode p' of p. Finally we call p^ω node fair if all paths in $\mathbf{Paths}(p^\omega)$ are node fair.

REMARK
In this definition the phrase "π passes through p' infinitely often" is not altogether clear: it may be the case that a subnode p' occurs in p on more than one place; p might even contain infinitely many instances of p'. Below we shall overcome this ambiguity by being more precise in identifying subnodes of p.

The aim of this section is to define two fair versions of the infinite iteration operator:

$$(\ \cdots\ )^{\omega_{fair}} : p \to p$$

such that the result $p^{\omega_{fair}}$ will be globally fair and node fair respectively. For this purpose we first give an alternative definition of infinite iteration, which will be used as a starting point for defining $(\ \cdots\ )^{\omega_{fair}}$.

PROPOSITION 5.6 (Alternative definition of infinite iteration)
Let $p \in P$. We define $\mathbf{App}_p : P \to P$ by

$$\mathbf{App}_p(p_0) = p \circ \mathbf{App}_p(p)$$

$$\mathbf{App}_p(q) = \{<a,\mathbf{App}_p(q')> \mid <a,q'> \in q\}, \text{ if } q \neq p_0.$$

(Read "append" for $\mathbf{App}$.) Then we have:

$$p^\omega = \mathbf{App}_p(p)$$

REMARKS
(1) Formally, $\mathbf{App}_p$ can be defined as the unique fixed point of the function $\Phi_p : (P \to P) \to (P \to P)$, given by

$$\Phi_p(F)(p_0) = p \circ F(p),$$

$$\Phi_p(F)(q) = \{<a,F(q')> \mid <a,q'> \in q\}, \text{ if } q \neq p_0.$$

(It is straightforward to show that Φ_p is contracting.)
(2) The function $\mathbf{App}_p$ applied to an argument $q \in P$ replaces all occurrences of p_0 in q by p, in

which, recursively, all occurrences of p_0 are again replaced by p.

(3) From proposition 5.6 it follows that $\mathit{App}_p(p_0) = \mathit{App}_p(p)$.

PROOF OF THE PROPOSITION

We define, for fixed $p \in P$, a function $\phi_p : P \to P$ by

$$\phi_p(q) = q \circ p^\omega.$$

We have

$$\begin{aligned}\phi_p(p_0) &= p_0 \circ p^\omega = p^\omega = p \circ p^\omega \\ &= p \circ (p \circ p^\omega) = p \circ \phi_p(p)\end{aligned}$$

and, for $q \in P$, $q \neq p_0$:

$$\begin{aligned}\phi_p(q) &= q \circ p^\omega \quad \text{(definition of } \circ) \\ &= \{<a, q' \circ p^\omega> \mid <a, q'> \in q\} \\ &= \{<a, \phi_p(q')> \mid <a, q'> \in q\}.\end{aligned}$$

From this it follows that ϕ_p is also a fixed point of Φ_p. Because Φ_p is contracting, it has a unique fixed point, thus $\phi_p = \mathit{App}_p$. Thus

$$p^\omega = p \circ p^\omega = \phi_p(p) = \mathit{App}_p(p).$$

(1) Global fairness

In this subsection we set out to define a fair version

$$(\ \cdots\ )^{\omega_{fair}} : P \to P^{FInd}$$

of the operation of infinite iteration such, that for p in P the result $p^{\omega_{fair}}$ will be globally fair. The range P^{FInd} of this mapping $(\ \cdots\ )^{\omega_{fair}}$ is given by

$$P^{FInd} = \{p_0\} \cup \mathcal{P}_{cl}(A \times P^{FInd}),$$

with

$$A = B \cup \mathit{FIndex},$$

where ***FIndex*** is a set of indices to be defined below. A naive first attempt would be to define

$$p^{\omega_{fair}} = \mathit{Fair}(p^\omega),$$

with the function ***Fair*** as in definition 3.4. This would be wrong, according to our definition of global fairness. The function ***Fair*** transforms its argument into a process, in which *all* unfair paths have disappeared. However, not every unfair path in p^ω is globally unfair, only those that are iteration paths. Thus the function ***Fair*** removes too many paths from p^ω. (For an illustration see the informal explanation above.) Therefore we have to come up with another solution. We shall use the definition of p^ω as $\mathit{App}_p(p)$ as a starting point for the definition of $p^{\omega_{fair}}$, but changing it by again using indices (as we did in the definition of ***Fair***) to label the nodes of p. After having defined $p^{\omega_{fair}}$, we shall clarify the relation between $p^{\omega_{fair}}$ and p^ω by defining a mapping

$$\Phi: \mathit{Paths}(p^{\omega_{fair}}) \to \mathit{Paths}(p^\omega).$$

Although the idea of defining $p^{\omega_{fair}}$ as $\mathit{Fair}(p^\omega)$ does not work (as was mentioned above), the definition of $(\ \cdots\ )^{\omega_{fair}}$ will be surprisingly similar to that of the function ***Fair***. The reason is the following: in constructing $p^{\omega_{fair}}$ for a given $p \in P$, we do two things at the same time. On the one hand we construct (a special version of) the infinite iteration of p, and on the other hand we select certain paths, namely

those that are globally fair. The first task is performed along the lines of the definition of App_p, the second task is realised following the definition of ***Fair***. So in some sense the definition of $p^{\omega_{fair}}$ will be a combination of the definitions of App_p and ***Fair*** (see proposition 5.6 and definition 3.4).

DEFINITION 5.7 (Flag indices). The set of *flag indices,* with typical element μ, is defined by:

$$\mathit{FIndex} = \{<<n_1,s_1,f_1>,\ldots,<n_m,s_m,f_m>> \mid n_i \geqslant 0,\ 0 \leqslant s_i \leqslant \infty,\ f_i \in \{U,D\}\}$$

where m is the number of basic steps in our finite alphabet B, and $\{U,D\}$ is the set of flags, containing two elements: U (for "up") and D (for "down").

The interpretation of n_i and s_i is as in definition 3.1 (see the informal explanation that follows there), but for the difference that only the *first* occurrence of b_i in each instantiation of p in $p^{\omega_{fair}}$ will cause n_i to be decreased by 1. Whether or not b_i has been chosen in a given instantiation of p, is indicated by the flag f_i. If it is up, b_i has not yet been chosen, and if it is down, b_i has been chosen at least once in the current instantiation of p.

We need the following operations on indices.

DEFINITION 5.8

Let $\mu \in \mathit{FIndex}$, with $\mu = <<n_1,s_1f_1>,\ldots,<n_m,s_m,f_m>>$, and let $i \in \{1,\ldots,m\}$. We define

$$\mu^-[i] = \begin{cases} \mu & \text{if } f_i = D \\ <<n_1,s_1,f_1>,\ldots,<n_i-1,s_i,D>,\ldots,<n_m,s_m,f_m>> & \text{if } f_i = U \wedge n_i > 1 \\ <<n_1,s_1,f_1>,\ldots,<0,\infty,D>,\ldots,<n_m,s_m,f_m>> & \text{if } f_i = U \wedge n_i = 1 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

For $\mu \in \mathit{FIndex}$ with $f_i = U$ the interpretation of $\mu^-[i]$ is as in definition 3.2, with the difference that U is changed to D. This indicates that in the current instantiation of p the basic step b_i has been chosen (at least once). If $f_i = D$, then $\mu^-[i] = \mu$, as indicated above. This will be explained below, after the definition of $p^{\omega_{fair}}$.

DEFINITION 5.9

Let $\mu \in \mathit{FIndex}$, with $<<\mu = n_1,s_1,f_1>,\ldots,<n_m,s_m,f_m>>$. We define

$$\begin{aligned} N(\mu) = \{&<<\tilde{n}_1,\tilde{s}_1,\tilde{f}_1>,\ldots,<\tilde{n}_m,\tilde{s}_m,\tilde{f}_m>> \mid \\ &\forall i \in \{1,\ldots,m\}\ [(n_i = 0 \wedge s_i = \infty \Rightarrow \tilde{n}_i > 0 \wedge \tilde{s}_i = 1 + \max\{s_j \mid 1 \leqslant j \leqslant m\}) \\ &\wedge (n_i > 0 \wedge s_i < \infty \Rightarrow \tilde{n}_i = n_i \wedge \tilde{s}_i = s_i) \\ &\wedge \tilde{f}_i = f_i]\}. \end{aligned}$$

The interpretation of $N(\mu)$ is as in definition 3.3, because the flags do not matter here.

DEFINITION 5.10

Let $\mu \in \mathit{FIndex}$ with, $\mu = <<n_1,s_1,f_1>,\ldots,<n_m,s_m,f_m>>$. Then

$$\mu^U = <<n_1,s_1,U>,\ldots,<n_m,s_m,U>>.$$

This operation sets all flags to "up" and is used upon entrance to a new instantiation of p. Now we are ready to define $(\cdots)^{\omega_{fair}}$.

DEFINITION 5.11 (Fair infinite iteration)

We define $(\cdots)^{\omega_{fair}} : P \to P^{FInd}$. Let $p \in P$. Then

$$p^{\omega_{fair}} = \sum_{\mu \in I_0} \mathit{App}_p(p,\mu),$$

where

$$I_0 = \{\ll n_1, 1, U\gg, \ldots, \ll n_m, 1, U\gg \mid n_i > 0\}$$

and for given $p \in P$

$$\textbf{\textit{App}}_p : P \times \textbf{\textit{FIndex}} \to P^{FInd}$$

is defined as follows. (We write q^μ for $\textbf{\textit{App}}_p(q,\mu)$.) Let $\mu \in \textbf{\textit{FIndex}}$. We define

$$\textbf{\textit{App}}_p(p_0,\mu) = \textbf{\textit{App}}_p(p,\mu^U).$$

For $q \in P$, $q \neq p_0$, we distinguish two cases.
Case 1:

$$\text{If } \exists i \in \{1, \ldots, m\}\ [\text{enabled}(i) \wedge (f_i = D \vee (s_i < \infty \wedge n_i > 0))],$$
$$\text{then } q^\mu = \{\langle b_i, \bar{q}^{\mu^-[i]}\rangle \mid \langle b_i, \bar{q}\rangle \in q \wedge$$
$$(f_i = D \vee (s_i < \infty \wedge n_i > 0 \wedge s_i = \min\{s_1, \ldots, s_m\}))\}.$$

Case 2:

$$\text{If } \forall i \in \{1,\ldots,m\}\ [\text{enabled}(i) \Rightarrow (f_i = U \wedge s_i = \infty \wedge n_i = 0)]$$
$$\text{then } q^\mu = \sum_{\mu' \in N(\mu)} q^{\mu'}.$$

REMARKS

(1) The remarks (1), (2), and (3) following definition 3.4 apply also to the above definition.

(2) We give some informal explanation of this definition by referring to remark (4) after definition 3.4 and making explicit what is different here. First, when we reach p_0 in the definition (3.4) of ***fair***, we are done: $\textbf{\textit{fair}}(p_0,\nu) = p_0$. Here we continue by appending p to p_0, together with the index μ changed into μ^U: $\textbf{\textit{App}}_p(p_0,\mu) = \textbf{\textit{App}}_p(p,\mu^U)$. The reason why we append p to p_0 is obvious: we are building the infinite iteration of p. (See proposition 5.6.) The index μ is changed to μ^U, that is all flags f_i of μ are set to U to indicate the entrance of a new instantiation of p. The second important difference between this definition and definition 3.4 is the role played by the flags. Let $q \in P$ with $\langle b_i, \bar{q}\rangle \in q$ for some $\bar{q} \in P$, $b_i \in B$. If $f_i = D$ (down), then b_i has already been chosen (at least once) in the current instantiation of p. Therefore it may be chosen unrestrictedly, even infinitely many times, within *this instantiation* of p (no matter what the values of n_i and s_i are). In this case we have: $\mu^-[i] = \mu$, formally expressing that b_i may pass "for free" without changing the values of n_i and s_i. The reason for letting b_i pass for free is that it provides us with the presence within $p^{\omega/\sim}$ of those infinite paths (possibly unfair) that are *not* iteration paths (and, hence, not globally unfair). If on the other hand $f_i = U$ and $n_i > 0$ and $s_i = \min\{s_1, \ldots, s_m\} < \infty$, then b_i may be chosen (as in case 2 of definition 3.4), but now μ is changed into $\mu^-[i]$ by changing the values of n_i and s_i (as in definition 3.4) and by changing the flag f_i to D.

Now for the rest of this subsection let $p \in P$ be fixed. We define a mapping

$$\Phi: \textbf{\textit{Paths}}(p^{\omega/\sim}) \to \textbf{\textit{Paths}}(p^\omega),$$

relating to each iteration path in $p^{\omega/\sim}$ a corresponding fair iteration path in p^ω. We start by re-stating lemma 3.5.

LEMMA 5.12
Let $\bar{p} \in P$, with $\bar{p} = p_0$, $\mu \in \textbf{\textit{FIndex}}$, and $\langle a, q\rangle \in \textbf{\textit{App}}_p(\bar{p},\mu)$ for $a \in B$ and $q \in P$. Then there exist $p' \in P$ and $\mu' \in \textbf{\textit{FIndex}}$ such that

$$q = \mathit{App}_p(p',\mu') \wedge$$
$$a \in \mathit{FIndex} \Rightarrow p' = \bar{p} \wedge$$
$$a \in B \Rightarrow \langle a,p' \rangle \in \bar{p}.$$

The proof is straightforward from the definition of $\bar{p}^{\mu}$ $(= \mathit{App}_p(\bar{p},\mu))$.

DEFINITION 5.13 (The mapping Φ)
Let

$$\pi = \langle a_0,q_0 \rangle, \langle a_1,q_1 \rangle, \ldots$$

be a path in $p^{\omega_{fair}}$. We can rewrite it as:

$$\pi = \langle a_0,p^{\mu} \rangle, \langle a_1,p_1^{\mu_1} \rangle, \langle a_2,p_2^{\mu_2} \rangle, \ldots$$

for certain $\mu,\mu_1,\mu_2,\ldots \in \mathit{FIndex}$ and $p_1,p_2,\ldots \in P$. If we omit in π all pairs $\langle a_i,p_i^{\mu_i} \rangle$ with $a_i \in \mathit{FIndex}$, and further all superscripts μ_i, we get a sequence

$$\Phi(\pi) = \langle a_{i_1},p_{i_1} \rangle, \langle a_{i_2},p_{i_2} \rangle, \ldots$$

which is a path in p^{ω}. We call $\Phi(\pi)$ the path in p *corresponding to* the path π in $p^{\omega_{fair}}$. This defines a mapping

$$\Phi\colon \mathit{Paths}(p^{\omega_{fair}}) \rightarrow \mathit{Paths}(p^{\omega}).$$

THEOREM 5.14
$p^{\omega_{fair}}$ is globally fair. That is, for all $\pi \in \mathit{Paths}(p^{\omega_{fair}})$, if π is an iteration path, then π is fair.

PROOF
Let $\pi \in \mathit{Paths}(p^{\omega_{fair}})$ and suppose π is an iteration path. We reduce the proof of this theorem to that of theorem 3.7 by making the following observation. Since π is an infinite iteration paths it enters infinitely often into a new instantiation of p. Upon each entrance, all flags are raised (set to "up"). As was observed above, if $f_i = U$ (for $i \in \{1,\ldots,m\}$), then b_i is treated in case 1 of definition 5.11 above in exactly the same way as in case 1 of definition 3.4. Because this situation arises infinitely often, the argument given in the proof of theorem 3.7 also applies here. (Note that case 2 in both definitions 3.4 and 5.11 is the same.)

REMARK: Formally we have to extend definition 5.4 of global fairness to processes in P^{FInd}. This can be done straightforwardly.

THEOREM 5.15: *Any globally fair path in p^{ω} is in the range of the mapping Φ.*

PROOF
Let $\pi' \in \mathit{Paths}(p^{\omega})$ such that π' is globally fair. We must construct a path $\pi \in \mathit{Paths}(p^{\omega_{fair}})$ such that

$$\Phi(\pi) = \pi'.$$

We distinguish between two cases: first, that π' is *not* an iteration path (and possibly unfair); second, that π' *is* an iteration path and fair.

(1) Suppose π' is not an iteration path. Without loss of generality we may assume that π' lies entirely within p (that is, the first instantiation of p in p^{ω}). We define a flag index μ by

$$\mu = \langle\langle 1,1,U \rangle, \ldots, \langle 1,1,U \rangle\rangle$$

and take $<\mu,p^{\mu}>$ as the first element of the path π that we are constructing. Now we can continue the construction of π by repeatedly applying case 1 (of definition 5.11) for the appropriate b_i's (namely, those that occur in π'). Each time we encounter a b_i for the first time, the corresponding triple $<1,1,U>$ in the index is changed into $<0,\infty,D>$. >From this moment on b_i may be chosen unrestrictedly within this instantiation of p (in which the path π' lies), without changing the index. The path π thus constructed is an element of $\boldsymbol{Paths}(p^{\omega_{fair}})$. Furthermore: $\Phi(\pi)=\pi'$. (Note that it is of no importance whether π' is fair or not.)

(2) Suppose π' is a fair infinite iteration path. As in the proof of theorem 5.14, we reduce this proof to that of the corresponding theorem in section 3 (theorem 3.8) by observing that the latter only needs a slight modification. When we count the number of times that a certain b_i occurs before a given step l_j in the path π', we have to count only the first occurrences of b_i in different instantiations of p. With this change in mind the proof of 3.8 can easily be transformed into a proof of this theorem.

(2) Node fairness

Let us now forget about global fairness and focus on the second notion: node fairness. We again set out to define a fair version

$$(\ \cdots\ )^{\omega_{fair}} : P \rightarrow P^{NInd}$$

of the operation of infinite iteration but now such, that for all $p \in P$ the result $p^{\omega_{fair}}$ will be *node* fair. The domain P^{NInd} is like P^{Ind} and P^{FInd}, but with

$$A \ = \ B \cup \boldsymbol{NIndex},$$

with $\boldsymbol{NIndex}$ a set of indices to be defined below.

In constructing this second version of infinite iteration we proceed globally as in the previous subsection, now using *node* indices in order to ensure the node fairness of $p^{\omega_{fair}}$, instead of flag indices, which were used above. We shall characterize (and even identify) a subnode of a given process $p \in P$ by the subpath in p that leads to it.

DEFINITION 5.16 (Nodes)
Let $p \in P$. We define the set of nodes of p by

$$\boldsymbol{Nodes}(p) \ = \ \{\pi \mid \exists \pi' \in \boldsymbol{Paths}(p)\ [\pi \text{ is a finite initial part of } \pi']\}.$$

For $\pi \in \boldsymbol{Nodes}(p)$, with $\pi = <a_1,p_1>, \ldots, <a_n,p_n>$, we define

$$\boldsymbol{end}(\pi) \ = \ p_n.$$

(When no confusion is possible we sometimes identify π and $\boldsymbol{end}(\pi)$.) We set $\boldsymbol{end}(\epsilon)=p$, where ϵ is the empty path.

The set of *node indices* for a given $p \in P$ is defined as follows. Each node index for p has two components: the first is a finite mapping, associating with each of a (finite) set of nodes of p an index as defined in 3.1; and the second is a node of p. Such a node index schedules the fairness of paths with respect to this second component. At each moment in the construction of $p^{\omega_{fair}}$, we consider only a finite number of nodes (the domain of the first component), namely those that we have encountered thus far.

DEFINITION 5.17 (Node indices)
Let $p \in P$. We define the set of node indices for p as follows:

$$\boldsymbol{NIndex}_p \ = \ (\boldsymbol{Nodes}(p) \rightarrow^{fin} \boldsymbol{Index}) \times \boldsymbol{Nodes}(p),$$

where $\rightarrow^{fin}$ denotes the set of partial functions on ***Nodes***(p) with a finite domain, and ***Index*** is defined as in definition 3.1. A typical element of ***NIndex*** is denoted $\rho=(\rho_1,\rho_2)$. For $\rho_1 \in$ ***Nodes***$(p) \rightarrow^{fin}$ ***Index*** we use the variant notation for functions: for $\pi,\pi' \in$ ***Nodes***(p) and $\nu \in$ ***Index***,

$$\rho_1\{\nu/\pi\}(\overline{\pi}) = \begin{cases} \nu & \text{if } \pi=\overline{\pi} \\ \rho_1(\overline{\pi}) & \text{if } \pi \neq \overline{\pi}. \end{cases}$$

(We shall use this notation whether or not $\pi \in$ ***domain***(ρ).)

We again need the operations $\nu^-[i]$ and $N(\nu)$ on indices $\nu \in$ ***Index*** (see definitions 3.2 and 3.3). They are used in the following

DEFINITION 5.18 (Fair infinite iteration)
We define $(\ \cdots\ )^{\omega_{fair}}: P \rightarrow P^{NInd}$. Let $p \in P$. Then

$$p^{\omega_{fair}} = \sum_{\rho_1 \in \{\epsilon\} \rightarrow I_0} \textbf{\textit{App}}_p(p,(\rho_1,\epsilon))$$

where ϵ is the empty subpath of p (identifying p as a subnode of itself),

$$I_0 = \{<n|_1, \ldots, n|_m^1> \mid n_i > 0\}$$

and for given $p \in P$

$$\textbf{\textit{App}}_p\colon P \times \textbf{\textit{NIndex}} \rightarrow P^{NInd}$$

is defined as follows. (We write q^ρ for $\textbf{\textit{App}}_p(q,\rho)$.) Let $\rho \in$ ***NIndex***, $\rho=(\rho_1,\rho_2)$ and let $q \in P$. If $q \neq$ ***end***(ρ_2), then $\textbf{\textit{App}}_p(q,\rho)$ is undefined. Now suppose that $q =$ ***end***(ρ_2). Then we define

$$\textbf{\textit{App}}_p(p_0,\rho) = \textbf{\textit{App}}_p(p_0,(\rho_1,\epsilon)).$$

For $q \neq p_0$ we distinguish two cases.
Case (a): $\rho_2 \in$ ***domain***(ρ_1). Let $\rho_1(\rho_2)=\nu=<n_1^{s_1}, \ldots, n_m^{s_m}>$. Then

$(a\,1)$ If $\exists i \in \{1, \ldots, m\}$ [enabled$(i) \wedge n_i > 0 \wedge s_i < \infty$], then

$$q^\rho = \{<b_i,\overline{q}^{(\rho_1\{\nu^-[i]/\rho_2\},\ \rho_2\circ<b_i,\overline{q}>)}> \mid <b_i,\overline{q}> \in q \wedge s_i = \min\{s_j \mid 1 \leqslant j \leqslant m\}\}.$$

$(a\,2)$ If $\forall i \in \{1, \ldots, m\}$ [enabled$(i) \Rightarrow n_i = 0 \wedge s_i = \infty$], then

$$q^\rho = \sum_{\nu' \in N(\nu)} q^{(\rho_1\{\nu'/\rho_2\},\rho_2)}.$$

Case (b): $\rho_2 \notin$ ***domain***(ρ_1). Then

$$q^\rho = \sum_{\nu' \in I_0} q^{(\rho_1\{\nu'/\rho_2\},\rho_2)},$$

where I_0 is as above.

REMARKS

(1) The remarks (1), (2), and (3) following definition 3.4 apply also to the above definition.
(2) We have that $\textbf{\textit{App}}_p(q,\rho)$ is undefined whenever $q \neq$ ***end***(ρ_2). This implies (since $\rho_2 \in$ ***Nodes***(p)) that $\textbf{\textit{App}}_p$ is defined on nodes of p only, which seems quite natural.
(3) We give some informal explanation of the definition above. If we arrive at p_0, with index p, we continue with $\textbf{\textit{App}}_p(p,\rho')$. Here $\rho'=(\rho_1,\epsilon)$, that is, the second component of ρ' now indicates that the node we are treating next is (***end***$(\epsilon)=$) p itself. The interpretation of cases $(a1)$ and $(a2)$ above is entirely similar to that of the cases 1 and 2 in definition 3.4: if $\rho_2 \in$ ***domain***(ρ_1), then $\nu=\rho_1(\rho_2)$ is treated exactly as before. A small difference is that, in $(a1)$, the second component

ρ_2 is extended with $<b_i,\overline{q}>$ to denote that the next node of p that is treated is $\overline{q}$ $(=\textit{end}(\rho_2 \circ <b_i,\overline{q}>))$. If $\rho_2 \notin \textit{domain}(\rho_1)$, an extension of the domain of ρ_1 takes place. Here I_0 is the set of initial indices (as in definition 3.4).

Now for the rest of this subsection let $p \in P$ be fixed. As in definitions 3.6 and 5.13 we can define a mapping

$$\Phi\colon \textit{Paths}(p^{\omega_{fair}}) \rightarrow \textit{Paths}(p^{\omega}).$$

The following two theorems can be proved by easy generalizations of the corresponding proofs (3.7 and 3.8) in section 3.

THEOREM 5.19 *$p^{\omega_{fair}}$ is node fair.*

THEOREM 5.20 *Any node fair path in p^{ω} is in the range of the mapping Φ.*

Combining global and node fairness
We could now combine the two definitions (5.11 and 5.18) of fair infinite iteration and construct a function

$$(\ \cdots\ )^{\omega_{fair}}\colon P \rightarrow P$$

such that $p^{\omega_{fair}}$ would be both globally and node fair. We do not do this and confine ourselves to the observation that it would be a straightforward and dull exercise. Similarly for the generalization to an infinite alphabet.

6. REFERENCES

[ABKR] P. AMERICA, J.W. DE BAKKER, J.N. KOK, J.J.M.M. RUTTEN, *A denotational semantics for a parallel object-oriented language,* report CS-R8626, Centre for Mathematics and Computer Science, Amsterdam, August 1986. (To appear in: Information and Computation.)

[AO] K.R. APT, E.-R. OLDEROG, *Proof rules dealing with fairness,* Science of Computer Programming 3, 1983, pp. 65-100.

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

[BK] J.A. BERGSTRA, J.W. KLOP, *A convergence theorem in process algebra,* Report CS-R8733, Centre for Mathematics and Computer Science, Amsterdam, Netherlands, 1987.

[BZ1] J.W. DE BAKKER, J.I. ZUCKER, *Processes and the denotational semantics of concurrency,* Information and Control 54, 1982, pp. 70-120.

[BZ2] J.W. DE BAKKER, J.I. ZUCKER, *Processes and a fair semantics for the ADA rendez-vous,* in: Proceedings 10th ICALP (J. Diaz, Ed.) Lecture Notes in Computer Science 154, Springer-Verlag, 1983, pp. 52-66.

[BZ3] J.W. DE BAKKER, J.I. ZUCKER, *Compactness in semantics for merge and fair merge,* Proceedings Workshop Logics of Programs, (E. Clarke & D. Kozen, Eds.) Pittsburgh, Lecture Notes in Computer Science 164 Springer-Verlag, 1983, pp. 18-33.

[CS] G. COSTA, C. STIRLING, *Weak and strong fairness in CCS,* Information and

Computation 73, 1987, pp. 207-244.

[D] E.W. DIJKSTRA, *A discipline of programming,* Prentice-Hall, 1976.

[DM] P. DEGANO, U. MONTANARI, *Liveness properties as convergence in metric spaces,* Proceedings of the sixteenth annual ACM Symposium on Theory of Computing, Washington, D.C., 1984, pp. 31-38.

[F] N. FRANCEZ, *Fairness,* Springer-Verlag, 1986.

[LPS] D. LEHMANN, A. PNUELI, J. STAVI, *Impartiality, justness and fairness: the ethics of concurrent termination,* Proceedings 8th ICALP, Acre, July 1981 (O. Kariv, S. Even, Eds.), Lecture Notes in Computer Science 115, Springer-Verlag, 1981.

[OA] E.-R. OLDEROG, K.R. APT, *Transformations realizing fairness assumptions for parallel programs,* Proceedings STACS 1984, Lecture Notes in Computer Science 166, Springer-Verlag, 1984.

[P] G.D. PLOTKIN, *A powerdomain for countable nondeterminism,* in: Automata, Languages and Programming, Proceedings 9th ICALP, Aarhus, July 1982 (M. Nielsen, E.M. Schmidt, Eds.), Lecture Notes in Computer Science 140, Springer-Verlag, 1982, pp. 418-428.

Samenvatting

Een parallelle object-georiënteerde taal: Ontwerp en semantische grondslagen

Dit proefschrift bestaat uit een zevental artikelen (de hoofdstukken 2 tot en met 8) vooraf gegaan door een inleiding (hoofdstuk 1).

In hoofdstuk 2 wordt POOL2, een parallelle object-georiënteerde programmeertaal, geïntroduceerd. Daarbij wordt uitgebreid verslag gedaan van de overwegingen die aan het ontwerp van POOL2 ten grondslag liggen. Deze taal integreert de structureringstechnieken van object-georiënteerd programmeren met mechanismen voor het uitdrukken van parallellisme. De grondslagen van het object-georiënteerd programmeren en het belang van deze programmeerstijl voor de methodologie van programmaontwerp worden uiteengezet. Verschillende manieren om objecten en parallellisme te integreren worden met elkaar vergeleken. Verder wordt een beknopt overzicht gegeven van het onderzoek naar de formele aspecten van POOL.

In hoofdstuk 3 wordt een stukje wiskundig gereedschap ontwikkeld dat in hoofdstuk 4 gebruikt zal worden: Er wordt een algemene methode beschreven om oplossingen te vinden van zogenaamde reflexieve (of recursieve) *domeinvergelijkingen* binnen de klasse van volledige metrische ruimten. Eerst wordt een categorie van volledige metrische ruimten gedefinieerd. Vervolgens wordt voor een vergelijking een oplossing gevonden door het dekpunt te nemen van een met de vergelijking geassocieerde functor gedefinieerd op deze categorie. Dit dekpunt is de directe limiet van een convergerende toren. Deze constructie werkt voor functoren die contraherend zijn. Ook worden twee additionele voorwaarden gegeven die elk afzonderlijk voldoende zijn om te garanderen dat het dekpunt uniek is. Tot slot wordt voor een grote klasse van functoren aangetoond dat aan deze voorwaarden is voldaan, zodat ze een uniek dekpunt bezitten.

In het volgende hoofdstuk wordt een denotationele semantiek voor POOL gedefinieerd. De belangrijkste eigenschap van dit model is compositionaliteit: De betekenis van een samengesteld programma wordt gegeven in termen van de betekenis van de samenstellende delen. Het semantisch universum van het model is een volledige metrische ruimte, verkregen met behulp van de technieken uit hoofdstuk 3; de elementen van dit domein worden processen genoemd. De semantische vergelijkingen geven een betekenis aan iedere syntactische constructie afhankelijk van het POOL object dat de constructie uitvoert, de omgeving die wordt gedefinieerd op grond van de klasse- en methode-declaraties, en een continuatie, die alle berekeningsstappen representeert die zullen volgen op het uitvoeren van deze constructie. Nadat we het proces hebben geconstrueerd dat het uitvoeren van een geheel POOL programma voorstelt, kunnen we met behulp van een "yield"(resultaat)-functie de verzameling van alle mogelijke executie-rijen eruit afleiden.

Vervolgens wordt in hoofdstuk 5 een eerste aanzet gegeven tot het bewijzen van de equivalentie van dit denotationele model en een al eerder gedefinieerde operationele semantiek voor POOL. Hiertoe wordt een aantal eenvoudige taaltjes bestudeerd, die opgevat kunnen worden als benaderingen van POOL. Voor ieder van hen wordt een

operationele en een denotationele semantiek gedefinieerd en wordt de equivalentie van beide modellen bewezen. Achtereenvolgens passeren een uniform/statisch, een uniform/dynamisch, een nonuniform/statisch en een nonuniform/dynamisch taaltje de revue. Hierbij duidt het onderscheid tussen uniform en nonuniform op het verschil tussen ongeïnterpreteerde en geïnterpreteerde elementaire acties. De tegenstelling tussen statisch en dynamisch wijst op het verschil tussen talen met een vast dan wel een veranderlijk aantal van parallelle processen. Voor de uniforme talen worden zogenaamde "linear time" modellen gebruikt, terwijl de nonuniforme talen met "branching time" modellen beschreven worden; de laatste maken gebruik van Plotkins resumpties. De operationele modellen berusten op transitiesystemen in de stijl van Hennessy en Plotkin.

In hoofdstuk 6 wordt een andere weg ingeslagen voor het bewijzen van de equivalentie van operationele en denotationele modellen in het algemeen. Door het eerste model als het dekpunt van een contractie te definiëren en vervolgens aan te tonen dat ook het tweede model een dekpunt van dezelfde contractie is volgt hun gelijkheid uit de stelling van Banach. Op deze wijze wordt een drietal eenvoudige (CCS-achtige) statische taaltjes onder de loep genomen: eerst twee uniforme taaltjes en tenslotte een nonuniform taaltje.

Het volgende hoofdstuk laat zien hoe deze methode kan worden toegepast op dynamische talen in het algemeen en POOL in het bijzonder. Eerst wordt weer een eenvoudig dynamisch taaltje (met procescreatie) behandeld en vervolgens komt de volledige taal POOL zelf aan bod. Een complicerende factor bij dynamische talen is het gebruik van continuaties in de definitie van de denotationele semantiek. De oplossing hiervoor bestaat uit de definitie van een intermediair model. Dit wordt met het operationele model vergeleken door middel van een abstractieoperator. Deze functie beeldt processen, die een "boom-achtige" structuur hebben, af op verzamelingen van stromen. Vervolgens wordt aangetoond dat zowel het intermediaire model als het denotationele model dekpunten zijn van dezelfde contractie. Hieruit volgt dan hun gelijkheid. Beide feiten impliceren tot slot het belangrijkste resultaat van dit hoofdstuk: De operationele betekenis van een POOL programma is gelijk aan de denotationele betekenis waarop de abstractie operator is toegepast. Dit wordt ook wel de correctheid van de denotationele semantiek met betrekking tot de operationele semantiek genoemd.

Tot slot geeft hoofdstuk 8 een eerste benadering voor de oplossing van het in hoofdstuk 4 gesignaleerde probleem van *fairness*. In een algemeen semantisch kader van metrische processen worden twee soorten van fairness-problematiek behandeld: (1) zogenaamde intrinsieke fairness en (2) faire operaties op processen. Met betrekking tot (1) zullen we een "fairificatie"-operatie op processen definiëren, geheten *Fair*, zodanig dat voor ieder proces p het proces $Fair(p)$ fair is, en precies die paden van p bevat die fair zijn. Deze functie wordt gedefinieerd door gebruik te maken van een systematische afwisseling van willekeurige keuzen (random choices). De tweede helft van het laatste hoofdstuk behandelt de notie van faire operaties op processen: Er wordt, gegeven een operator op processen, zoals parallelle compositie of oneindige iteratie, een faire versie van deze operator gedefinieerd.

Curricula vitae

Pierre America werd geboren op 23 juli 1959 in Kerkrade. Zijn middelbare schoolopleiding genoot hij van 1971 tot 1977 aan het gymnasium "Rolduc" in diezelfde plaats. Daarna studeerde hij aan de Rijksuniversiteit Utrecht. In 1980 behaalde hij kandidaatsexamens in wiskunde en natuurkunde met bijvak sterrenkunde, en in 1982 het doctoraalexamen wiskunde met bijvakken informatica en natuurkunde. Zijn afstudeerscriptie was getiteld "Complete and continuous lattices, with applications in the theory of models of typed and typefree lambda calculus" en kwam tot stand onder begeleiding van Dr. H. P. Barendregt en Prof. Dr. D. van Dalen. Sinds 1982 is hij in dienst van het Philips Natuurkundig Laboratorium in Eindhoven.

Jan Rutten werd op 24 maart 1959 te Tilburg geboren. Hij volgde het VWO aan het Hertog Jan college te Valkenswaard. Van 1977 tot 1985 studeerde hij wiskunde te Nijmegen. Zijn afstudeerscriptie schreef hij bij Prof. J. J. de Iongh onder begeleiding van Dr. W. H. M. Veldman. Sinds 1985 is hij medewerker aan het Centrum voor Wiskunde en Informatica te Amsterdam.

Stellingen

behorend bij het proefschrift

A Parallel Object-oriented Language:
Design and Semantic Foundations

Pierre America

17 mei 1989

1. De eigenschap die het meest bijdraagt tot de geschiktheid van object-georiënteerd programmeren voor het maken van grote systemen en van meermaals bruikbare software-componenten is het dicht bij elkaar plaatsen van data en de operaties daarop. Het veelgeroemde inheritance-mechanisme blijft daarbij in belangrijkheid ver achter.

2. Veel te laat begint in de object-georiënteerde wereld het besef door te dringen dat het nuttig is om een onderscheid te maken tussen *inheritance* en *subtyping*, waarbij het eerste begrip een overeenkomst aanduidt in de implementatie van objecten en het tweede een overeenkomst in hun observeerbare gedrag. Subtyping in deze vorm ontstaat op natuurlijke wijze als men een type opvat als een specificatie van observeerbare eigenschappen van objecten. Helaas wordt nog te weinig onderkend dat een dergelijke specificatie meer dient te omvatten dan alleen de signatuur van het object (de namen van de methoden en de types van hun parameters en resultaten).

   Zie: Pierre America. A behavioural approach to subtyping in object-oriented programming languages. Te verschijnen in het *Philips Journal of Research*.

3. Sinds meer dan tien jaar worden declaratieve programmeerstijlen (zoals logisch en functioneel programmeren) aangeprezen als bijzonder geschikt voor het programmeren van parallelle machines. Hun grootste voordeel zou zijn dat ze de berekening beschrijven in termen van een wiskundig, statisch universum in plaats van in dynamische, operationele termen, waardoor een automatische analyse van het programma veel gemakkelijker wordt. Tegelijk is dit echter het grootste nadeel van deze programmeerstijlen, want er zijn nu twee vertaalslagen nodig: eerst van het applicatiedomein, dat meestal dynamisch en niet-wiskundig van aard is, naar een statisch programma, en vervolgens weer van dit statische programma naar een dynamische machine. Deze laatste vertaling begint men na jaren van intensieve studie redelijk onder de knie te krijgen (althans voor sequentiële machines), maar de eerste zou wel eens een onoplosbaar probleem kunnen blijven.

4. Het kernprobleem van parallel programmeren is het omgaan met nondeterminisme. Een zekere mate van nondeterminisme is nodig om implementatiedetails voor de programmeur af te schermen, maar het redeneren over niet-deterministische systemen leidt al te gemakkelijk tot onhandelbare gevals-onderscheidingen. Daarom zijn formalismen die parallellisme beschrijven in termen van nondeterminisme (zoals CCS en procesalgebra) wellicht niet het meest geschikt om het ontwerp van parallelle systemen te ondersteunen.

5. Bij het ontwikkelen van parallelle programma's kan een gezond en volledig bewijssysteem voor de gebruikte programmeertaal een goed hulpmiddel zijn. Veel nuttiger zou echter een uitgewerkte ontwerp*methode* zijn die voor een voldoende grote klasse van problemen tot formeel gefundeerde oplossingen leidt.

6. De techniek, gepresenteerd in hoofdstuk 5 van dit proefschrift, om continuaties

te vervangen door semantische operatoren, kan ook worden gebruikt om de equivalentie van operationele en denotationele semantiek van POOL aan te tonen.

7. Het belang van fundamenteel onderzoek in de industriële research, zeker voor wat betreft de informatica, ligt daarin dat het leidt tot een beter begrip van de basisprincipes en van daaruit tot betere ontwerpen van concrete systemen. Dit belang is niet goed te quantificeren, maar wel degelijk aanwezig. Dit verklaart de relatief zwakke positie van de mensen die dit onderzoek doen en de sterke positie van de organisaties die het belang ervan inzien.

8. De ontwerper van een programmeertaal zou er goed aan doen ook enkele natuurlijke talen te bestuderen van zo veel mogelijk uiteenlopende aard. Op zijn minst leert hij daarvan bescheidenheid.

9. Voor een nog realistischer muziekweergave dan mogelijk is met de tegenwoordig gangbare stereo-apparatuur, is het niet zozeer van belang het frekwentiebereik of de vervormingscijfers te verbeteren als wel het toevoegen van een extra dimensie. Veel belangrijker dan de dimensie voor-achter is daarbij de dimensie boven-beneden.

10. Sinds het begin van deze eeuw hebben zich in de klassieke, serieuze, gecomponeerde muziek (onbevredigende termen die alle hetzelfde willen aanduiden) ingrijpende veranderingen voorgedaan. Helaas hebben die tot gevolg gehad dat de in deze lijn nieuw gecomponeerde muziek geen maatschappelijk relevant verschijnsel meer is.

11. Het doel van het onderzoek naar kunstmatige intelligentie zou niet moeten zijn om computers te leren precies dezelfde fouten te maken als mensen.

Stellingen

behorend bij het proefschrift

A Parallel Object-oriented Language:
Design and Semantic Foundations

Jan Rutten

17 mei 1989

1. Zij $\mathcal{T} = \langle S, A, \rightarrow \rangle$ met $\rightarrow \subseteq S \times A \times S$ een eindig vertakkend gelabeld transitiesysteem en zij P de volledige metrische ruimte gegeven door de reflexieve vergelijking $P \cong \mathcal{P}_{compact}(A \times P)$. We kunnen een afbeelding $M_{\mathcal{T}} : S \rightarrow P$ definieren door: $M_{\mathcal{T}}[\![s]\!] = \{ \langle a, M_{\mathcal{T}}[\![s']\!] \rangle : (s, a, s') \in \rightarrow \}$, voor $s \in S$. We schrijven $s \underline{\leftrightarrow} t$ voor s en t in S als er een bisimulatie-relatie $R \subseteq S \times S$ bestaat met sRt. Nu geldt voor alle s en t in S: $s \underline{\leftrightarrow} t \Leftrightarrow M_{\mathcal{T}}[\![s]\!] = M_{\mathcal{T}}[\![t]\!]$.

   Vgl.: Rob van Glabbeek, *The linear time–branching time spectrum*, CWI rapport CS-R89.., Amsterdam, 1989, nog te verschijnen, en: Jan Rutten, *Deriving metric models for bisimulation from transition system specifications*, CWI rapport CS-R89.., Amsterdam, 1989, nog te verschijnen.

2. De begrippen *operationele*, *denotationele*, *declaratieve* en *procedurele* semantiek zijn niet eenduidig en geven daarom vaak aanleiding tot verwarde discussie.

3. Het is mogelijk een parallel logisch programma (geschreven in bijvoorbeeld Concurrent Prolog, Guarded Horn Clauses of Parlog) van een declaratieve semantiek te voorzien door het kleinste vaste punt te beschouwen van een continue operator (de zogenaamde eenstapsafleidings operator) op de complete tralie van interpretaties voor dat programma. (Een interpretatie is hier een verzameling van paren bestaande uit een atoom (mogelijk met variabelen) en een eindig rijtje substituties.) Het is eenvoudig in te zien hoe deze complete tralie is uit te breiden tot een volledige metrische ruimte. De genoemde continue operator blijkt dan een contractie te zijn. Deze observatie vormt een eerste aanzet tot het geven van een equivalentiebewijs voor dit declaratieve model en een denotationele semantiek voor parallelle logische programma's in de imperatieve traditie.

   Vgl.: Catuscia Palamidessi, *A fixed point semantics for Guarded Horn Clauses*, CWI rapport CS-R8833, Amsterdam, 1988, en: Frank de Boer, Joost Kok, Catuscia Palamidessi en Jan Rutten, *Semantic models for PARLOG*, te verschijnen in het congresverslag van ICLP 1989, Lissabon.

4. Het gebruik van volledige metrische ruimten bij het beschrijven van semantiek van programmeertalen is zeer vruchtbaar gebleken. Met name vormt de eenvoudig te bewijzen stelling van Banach *(iedere contraherende afbeelding van een volledige metrische ruimte naar zichzelf heeft een uniek vast punt)* een verrassend nuttig hulpmiddel, zowel bij het geven van definities (waarbij het *bestaan* van het vaste punt wordt gebruikt) als voor het leveren van bewijzen van semantische equivalenties (berustend op de *uniciteit* van het vaste punt), die verschillende modellen met elkaar relateren.

   Vgl.: Jaco de Bakker en Jeffery Zucker, *Processes and the denotational semantics of concurrency*, Information and Control **54**, 1982; Jan Rutten, *Correctness and full abstraction of metric semantics for concurrency*, te verschijnen in: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency (J. W. de Bakker, W. P. de Roever, G. Rozenberg, eds.), Proc. REX Workshop 1988, LNCS, Springer-Verlag, 1989; en: dit proefschrift.

5. De methode om bij de semantische studie van een geavanceerde programmeertaal eerst een aantal eenvoudige benaderingen van deze taal te onderzoeken heeft haar nut bewezen bij het bestuderen van de taal POOL. Ook voor de studie van parallelle logische programmeertalen lijkt dit de aangewezen weg.

   Vgl.: Jaco de Bakker, *Comparative semantics for flow of control in logic programming without logic*, CWI rapport CS-R8840, Amsterdam, 1988, en: dit proefschrift.

6. Het is niet eenvoudig zelfs zeer kleine programma's geschreven in een van de momenteel in zwang zijnde parallelle logische programmeertalen te begrijpen.

7. Het is voor de algemene problematiek van de equivalentie van operationele en denotationele semantiek essentieel om te begrijpen hoe de specificatie van het transitiesysteem, waarop het operationele model gebaseerd is, de semantische operatoren van het denotationele model induceert.

   Vgl.: Ph. Darondeau en B. Gamatié, *Modelling infinite behaviors of communicating systems*, rapport IRISA, Rennes, 1987, en: Jan Rutten, *Deriving metric models for bisimulation from transition system specifications*, CWI rapport CS-R89.., Amsterdam, 1989, nog te verschijnen.

8. In bijvoorbeeld de Verenigde Staten en Italië worden promovendi PhD-studenten genoemd en volgen zij, al dan niet verplicht, vaak nog colleges. Het verdient aanbeveling in Nederland de tweede-fase opleiding voor promovendi in de informatica in te richten naar het model van dergelijke PhD-opleidingen. Duidelijk is nu al dat in deze postdoctorale scholing aan de wiskunde ruime aandacht zou behoren te worden geschonken, zeker door hen die een universitaire studie informatica hebben gevolgd.

9. Goede wetenschappers zijn dapper: Ze zijn in staat de angst voor het onbekende te overwinnen.