Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.N. Kok

A compositional semantics for concurrent prolog

Computer Science/Department of Software Technology Report CS-R8809 February

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

A Compositional Semantics for Concurrent Prolog

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

Concurrent logic languages like Concurrent Prolog have mechanisms in common with imperative concurrent languages: concurrency, communication, synchronization and indeterminacy, finite and infinite behaviour. A goal statement can be considered as a network of processes which run in parallel and can communicate. For imperative concurrent languages a lot of research on the semantics has been done. In this paper we show that a model, which was originally designed for imperative languages, can be used to give a formal description of Concurrent Prolog. This model is compositional: for any two conjunctions C_1, C_2 the meaning of $C_1 \wedge C_2$ can be obtained by applying a function to the meanings of C_1 and C_2. To be more specific: we shall employ a domain where some choices made during the computation are recorded in certain tree-like structures, and which allows for interleaving. We use tree like structures because it is not possible to use flat structures (like traces or sequences of substitutions). They do not contain enough information to handle deadlock or infinite computations. Such phenomena require that we know at a certain moment in time all the alternatives. The domain is obtained as a solution of a domain equation. We apply metric topological tools to find a solution of the domain equation, and to define operations on elements of the domain. The semantic function maps goals, given a program P, to elements of the domain. This function is defined recursively. The model is able to handle both finite and infinite computations.

1985 Mathematics Subject Classification: 68Q55, 68Q10.
1987 Computing Reviews Categories: D.1.3, D.3.1, F.1.2, F.3.2.

Key words and phrases: denotational semantics, parallelism, concurrent logic languages, compositionality, metric topology.

Note: This work is partially supported by the Netherlands Organization for Scientific Research (N.W.O.), grant 125-20-04.

Note: this report is also published in the proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS), Bordeaux, february 1988, in the Lecture Notes in Computer Science series of Springer Verlag.

1. Introduction

Programming languages are sometimes divided in two groups: descriptive languages and constructive languages. Functional and logic languages are often classified as descriptive languages. Constructive languages are sometimes called imperative languages. Following [Gelernter 1984] algorithms tend to be described in descriptive languages and constructed in constructive ones. Both categories of languages have different styles of giving semantics to them. Often different domains and techniques are used. Descriptive languages have more natural mathematical domains. In our opinion, this division has not a straight border line. Logic languages, for example, can be given different interpretations. We have a declarative, a procedural and a process interpretation: a clause $a \leftarrow b_1 \wedge \cdots \wedge b_n$ is read in a declarative interpretation as "a is true if all b_i are true", and the procedural reading would be "to solve problem a, solve subproblems b_i", and in the process interpretation "a process a can replace itself by the system of processes that contains all b_i". In a process interpretation we can consider a unit goal to be a process, a conjunctive goal a network of processes, a variable shared by different processes in a goal a communication channel and a conditional clause defines a network reconfiguration. If we take the first interpretation we can say that a logic language is a declarative

Report CS-R8809
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

language, with the second interpretation this is not so clear any more, and with the third interpretation one can argue that a logic language is a constructive language. If we look at the semantics given to logic languages (see for example [Lloyd 1984]) we see three kinds of semantics: declarative semantics, procedural semantics, and perpetual processes. The declarative semantics is formulated with logical models, and it tries to characterize the desired model for a program with the help of fixed points. It corresponds to the declarative reading of logic programs. The procedural semantics speaks in terms of resolution, SLD trees and success sets. One can say that this corresponds to the procedural reading of logic programs. Perpetual processes are completions of models: they are used because infinite behaviour can give rise to infinite data structures. The semantics is still formulated with models, which now can contain infinite ground terms, and with fixed point characterizations of them. This semantics does not totally correspond to the process interpretation. It is more an extension (taking possible infinite behaviour into account) to the declarative semantics. In this paper we try to give a process based semantics to a logic language. We use Concurrent Prolog because this languages has many features (maybe too many). We think this kind of semantics can also be used for other (concurrent) logic languages.

Concurrent Prolog, as defined in [Shapiro 1983], has two major extensions to "normal" logic programming languages. It is possible for two processes to synchronize. This is done by a special mechanism: *read only variables.* Variables in programs can be marked read only with a question mark. There are some constraints on the unification of read only variables, and this has as effect that it is possible that at some moment in the computation the unification between two atoms (processes) can not succeed, and at some later stage they can unify because (at least) one of them has communicated with another process. As second extension Concurrent Prolog has a mechanism to control the nondeterminism: *the commit operator.* If execution comes to a point where it has to execute a commit operator, a choice can be made. Some alternative processes can be killed and it can be decided that the body of the clause will be executed. Such a choice can not be undone. A comparison can be made with the cut operator of Prolog.

There are some problems with the definition of Concurrent Prolog given in [Shapiro 1983]. They were pointed out in [Saraswat 1986]. In order to overcome these problems, Saraswat defines a family of languages which he calls Concurrent Prolog. Each member of this family has a specific set of features. For example, he proposes a new kind of annotation (the wait annotation) to be used instead of the read only annotation. Also a new commit operator is introduced: besides the don't care commit he uses the don't know commit. In this paper we take the CP language which has both commits and the wait only annotation as a starting point. In [Saraswat 1987] an operational semantics, based on transition systems, is given. This semantics, however, is not compositional.

For a number constructive languages, like CSP or ADA, process based models have been developed. We try here to adapt these semantic techniques. We will use a process based semantics based on techniques which were first described in [de Bakker & Zucker 1982]. The processes described in [de Bakker & Zucker 1982] (for a language containing assignment and a merge operator) are state transforming processes. A state is used to record the values of the variables. Our processes are a variant of these processes: instead of using state transforming processes, we use substitution transforming processes.

We will try to describe the main ideas behind our model. The semantic model is based on substitution transforming functions. The execution of an atom a can be considered as a state transforming function: given a substitution θ, we try to resolve $\theta(a)$, and if this succeeds, an answer substitution θ' results. We require that our semantic function is compositional. The semantic function assigns to each conjunct, given a Concurrent Prolog program P, an element of the semantic domain. Compositionality implies that the meaning of the conjunct $a_1 \wedge a_2$ should be a function of the semantic

meanings of a_1 and a_2. This requirement complicates our domain. The execution of a conjunct $a_1 \wedge a_2$ is done by executing a_1 and a_2 in parallel. The execution of a_1 can influence the execution of a_2 and vice versa. We have to use a more complex semantic domain than just state transforming functions: elements of the domain should allow for interleaving of other elements. Concurrent Prolog allows processes to synchronize: as indicated above a process can be suspended till another process reaches a certain point in its execution. This has implications if we want to model deadlock (a process fails and it has no alternatives left nor it is suspended) or infinite behaviour. If we have two alternative processes, from which the first one fails and the second delivers a substitution, the result will be the substitution. If there is a choice between failure and infinite behaviour, we are forced to choose infinite behaviour. To model this correctly, we have to record information about at the timing of certain choices during a computation. Our processes are called resumptions: mathematical structures which allow for interleaving and in which timing of choices can be recorded. They can be infinite structures, and are defined with a recursive equation. We use metric topological tools to show that they exist and also to define operations on them. For example, the operator merge (shuffle) on resumptions, which is the semantic counterpart of $\wedge$, is defined as a fixed point of a higher order function. This higher order function is contracting on a complete metric space, and hence, by Banach's fixed point theorem, has a unique fixed point.

The semantic function (which assigns processes to a goal, given a Concurrent Prolog program) is defined as the unique fixed point of a higher order function. With an abstraction operator we can derive from our compositional semantics a non compositional semantics.

The outline of the rest of the paper is as follows: in section 2 we introduce the syntax of Concurrent Prolog, section 3 describes the informal semantics, in section 4 the processes (resumptions) and some operations on them are given, in section 5 we define the semantic functions, section 6 gives the conclusion and in section 7 we have the references. We also provide an appendix with the (metric topological) mathematical preliminaries.

2. Introduction to Concurrent Prolog

2.1 Syntax of Concurrent Prolog

In this subsection we introduce the syntax of Concurrent Prolog. Let the following sets be given:

$$Cons = \{c,d, \ldots\} \textit{ of constants,}$$

$$Var = \{u,v,w, \ldots\} \textit{ of variables,}$$

$$Func = \{f,g, \ldots\} \textit{ of functions.}$$

Each function has an arity $n \geq 1$. There is a special function $\downarrow$ of arity 1. This functor is called the wait functor. We construct the set of terms:

$$Term = \{s,t, \ldots\} \textit{ of terms,}$$

with the help of a BNF grammar:

$$t ::= c \mid v \mid f(t_1, \ldots, t_n) : n \geqslant 1 \wedge f\ n-adic.$$

We also define the set $Term_{\downarrow} \subset Term$ to be the least set such that

- $\{ \downarrow(t) : \text{no} \downarrow \text{inside } t \} \subset Term_{\downarrow}$
- $\{ t : \text{no} \downarrow \text{inside } t \} \subset Term_{\downarrow}$
- if $t_1, \ldots, t_n \in Term_{\downarrow}$ and if at least one of them has as principal functor $\downarrow$ we have that, for any n-adic function f that $\downarrow(f(t_1, \ldots, t_n)) \in Term_{\downarrow}$.

Motivation for this definition can be found in [Saraswat 1986].

Let also be given a set

$$Pred = \{Q, \ldots\} \textit{ of predicates.}$$

The set

$$Atom = \{a,b, \ldots\} \textit{ of atoms}$$

is given by

$$a ::= Q(t_1, \ldots, t_n) : n \geqslant 1 \wedge Q\ n-adic \wedge t_1, \ldots, t_n \in Term_{\downarrow}$$

The set

$$Conj = \{C, \ldots\} \textit{ of conjunctions}$$

is the set of conjunctions of one or more atoms in which no $\downarrow$ appear. The set

$$\begin{aligned} Clause = {} & \{a \leftarrow C_1 \not\in C_2 : a \in Atom \wedge C_1, C_2 \in Conj \wedge \not\in \in \{|, \&\}\} \cup \\ & \{a \leftarrow C_1 \not\in : a \in Atom \wedge C_1 \in Conj \wedge \not\in \in \{|, \&\}\} \cup \\ & \{a \leftarrow \not\in C_2 : a \in Atom \wedge C_2 \in Conj \wedge \not\in \in \{|, \&\}\} \cup \\ & \{a \leftarrow \not\in : a \in Atom \wedge \not\in \in \{|, \&\}\} \end{aligned}$$

is the set of clauses. The bar "|" in a clause is called the *don't care commit* operator and & is called the *don't know commit* operator. The conjunct before a commit operator is called the guard of a clause and the conjunct after the commit operator is called the body of the clause. A Concurrent Prolog program is a finite set of clauses: the set

$$Prog = \{P, \ldots\} \textit{ of programs}$$

is defined by $Prog = \mathcal{P}_{finite}(Clause)$, where $\mathcal{P}_{finite}(\cdot)$ denotes finite subsets of $(\cdot)$.

2.2 Substitutions

In this subsection we introduce substitutions. Let $\sigma, \theta \in Subst = Var \rightarrow Term$ be substitutions. The substitution *id* is the identity substitution. We can extend a substitution $\theta : Var \rightarrow Term$ to a function $\hat{\theta} : Term \rightarrow Term$ by

$$\hat{\theta}(c) = c,\ \hat{\theta}(v) = \theta(v),\ \hat{\theta}(f(t_1, \ldots, t_n)) = f(\hat{\theta}(t_1), \ldots, \hat{\theta}(t_n))$$

and to a function $\dot{\theta} : Atom \rightarrow Atom$ by

$$\dot{\theta}(P(t_1, \ldots, t_n)) = P(\hat{\theta}(t_1), \ldots, \hat{\theta}(t_n))$$

and to a function $\ddot{\theta} : Conj \rightarrow Conj$ by

$$\ddot{\theta}(a_1 \wedge \cdots \wedge a_n) = \dot{\theta}(a_1) \wedge \cdots \wedge \dot{\theta}(a_n)$$

and to a function $\tilde{\theta} : Prog \rightarrow Prog$ by

$$\tilde{\theta}(P) = \bigcup \{ (\dot{\theta}(a) \leftarrow \ddot{\theta}(C_1) \mid \ddot{\theta}(C_2)) : (a \leftarrow C_1 \mid C_2) \in P \}$$

Define the composition of substitutions by

$$\theta_1 \hat{\circ} \theta_2 = \lambda v\ .\ \hat{\theta}_1(\theta_2(v)).$$

3. Informal introduction to the semantics of Concurrent Prolog

First we look at the unification of terms. By the syntactic restriction $\downarrow$ functions only appear in heads of clauses. Therefore we have only to consider terms t_1, t_2 with $t_1 \in Term_{\downarrow}$ and t_2 which does not contain $\downarrow$ functions. The unification is an extension to normal unification. A term $\downarrow(t)$ will suspend when unifying against a variable. It waits till the variable becomes instantiated to a term which is either a constant or a term with a functor. Then it tries to unify the term with the instantiation of the variable. The arguments of a compound term are unified in parallel. A formal definition of this unification is given in [Saraswat 1986]. For the understanding of the rest of the paper, it suffices to remember that there are three possibilities for unification: success, failure and suspension. In other words, the $\downarrow$ annotation is used to restrict the goals for which a clause is applicable by specifying which terms in the goal need to be instantiated.

Next we take look at the execution of a Concurrent Prolog program. We start with a goal $a_1 \wedge \cdots \wedge a_n$. With each atom a_i, $i = 1, \ldots, n$, a process is associated. Each process tries to reduce itself to other processes. A process a can reduce itself finding a clause whose head $\bar{a}$ unifies with a and whose guard system C_1, if present, terminates following that unification. The computation gives rise to a hierarchy of systems of processes. Each process may invoke several guard systems, in an attempt to find a reducing clause, and the computation of these guard systems in turn may evoke other systems. The communication between these systems is governed by the commitment mechanism. If a guard system terminates, it can commit. This means that the bindings are communicated to all other systems. If such a binding is inconsistent with a binding in another system, this system fails. Moreover, the body C_2, if present, of the chosen clause replaces a. If no body is present, we have termination. The difference between the two commits is that for the don't care commit alternatives are killed, and for the don't know commit they are also searched in parallel. The system terminates when all processes are terminated. Note that a process can terminate only when there are clauses in the program with an empty body.

4. Resumptions

First we present an intuitive introduction of the notion of resumption. We use the terminology of *processes* $p,q, \ldots$ and process domains *Process*, ... 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.* We let $A, B, C, \ldots$ stand for arbitrary (given) sets (which do not interact with the process domains to be constructed.) A very simple equation is

$$Process = A \cup (A \times Process) \tag{4.1}$$

We can read this equation as follows: a process is either an atomic action, or it is a pair $<a,q>$ where a is the first action taken, and q is the *resumption,* describing the rest of p's actions. Clearly, (4.1) has as solution either the set of all finite sequences $<a_1, a_2, \ldots, a_n>$, $n \geq 0$, or these and, in addition, all infinite sequences $<a_1, a_2, \ldots, a_n, \ldots>$. We next consider

$$Process = A \rightarrow (B \cup (C \times Process)) \tag{4.2}$$

This is already a much more interesting equation: each process p is a function which, when supplied with an argument a, yields as outcome either a pair $p(a) = <c,p'>$ or an action b. We see that p maps a to c or b, and in the first case at the same time turning itself into the resumption p'. We can say that p determines its first step b or c and resumption p' on the basis of a. The following equation we consider is

$$Process = A \rightarrow \mathcal{P}_{comp}(B \cup (C \times Process)) \tag{4.3}$$

where $\mathcal{P}_{comp}(.)$ stands for the set of all compact nonempty subsets of (.). Now, if we feed a process p with an $a \in A$, a whole set X of possible actions b and pairs $<c,q>$ results, among which the process can choose freely. For reasons of cardinality, (4.3) has no solution when we take *all* (rather than all compact) subsets of (.) It will turn out that (4.3) is the right type of domain equation for our purposes. We shall specialize A to *Subst* and B to

$$Subst_\delta^{st} =^{def} Subst^\omega \cup Subst^+ \cup Subst^* \cdot \{\delta\},$$

the set of sequences of substitutions which can end in δ (deadlock). The set $Subst^*$ is the set of finite sequences, $Subst^\omega$ is the set of infinite sequences, $Subst^\infty$ is the set of finite and infinite sequences, $Subst^+$ is the set of finite sequences of length greater than 0 of substitutions. We shall specialize C to $Subst^+$. In this way we obtain

$$Process = Subst \to \mathcal{P}_{comp}(Subst_\delta^{st} \cup (Subst^+ \times Process)) \tag{4.4}$$

Thus, each process p when supplied with a substitution σ as an argument yields a set. This set can contain pairs of the form $<x,p'>$ or sequences of substitutions y which can end in $\{\delta\}$. An important advantage of processes as in (4.4) is that they allow a natural definition of their *merge* which is a quite familiar operator in concurrency semantics.

We show how to construct a process domain that satisfies equation (4.4). Define metric spaces M_i, $i \in N$, as follows:

$$M_1 = Subst \to \mathcal{P}_{comp}(Subst_\delta^{st})$$
$$M_{n+1} = Subst \to \mathcal{P}_{comp}(Subst_\delta^{st} \cup (Subst^+ \times M_n))$$

where metrics d_n, $n \geq 1$ are defined as follows. Let $y_1, y_2 \in Subst_\delta^{st}$. Define

$$d_{st}(y_1,y_2) = \begin{cases} 2^{-\max\{\, n : y_1[n] = y_2[n] \,\}} & \text{if } y_1 \neq y_2 \\ 0 & \text{if } y_1 = y_2 \end{cases}$$

where $y[n]$ denotes the prefix of length n of y, and let d_H be the Hausdorff distance on $\mathcal{P}_{comp}(Subst_\delta^{st})$ generated by d_{st}. Define

$$d_1(p_1,p_2) = sup_{\sigma \in Subst} d_H(p_1(\sigma),p_2(\sigma)).$$

Define a metric on $Subst_\delta^{st} \cup Subst^+ \times M_n$ as follows:

$$d(y_1,y_2) = d_{st}(y_1,y_2)$$

$$d(<y_1,p_1>,y_2) = \begin{cases} d_{st}(y_1,y_2) & \text{if } y_1 \neq y_2 \\ 2^{-length(y_1)} & \text{if } y_1 = y_2 \end{cases}$$

$$d(y_1,<y_2,p_2>) = \begin{cases} d_{st}(y_1,y_2) & \text{if } y_1 \neq y_2 \\ 2^{-length(y_1)} & \text{if } y_1 = y_2 \end{cases}$$

$$d(<y_1,p_1>,<y_2,p_2>) = \begin{cases} d_{st}(y_1,y_2) & \text{if } y_1 \neq y_2 \\ 2^{-length(y_1)} \cdot d_n(p_1,p_2) & \text{if } y_1 = y_2 \end{cases}$$

and let

$$d_{n+1}(p_1,p_2) = sup_{\sigma \in Subst} d_H(p_1(\sigma),p_2(\sigma))$$

where d_H is the Hausdorff distance generated by d. Remark that for each M_n, M_m, $m \geqslant n$, there exist an isometric embedding $i_{nm} : M_n \to M_m$. We define a metric on $\cup\{M_n : n \geqslant 1\}$. Take any p_1 and p_2. Suppose $p_1 \in M_n$ and $p_2 \in M_m$ with $m \geqslant n$. Define $d(p_1,p_2) = d_m(i_{nm}(p_1),p_2)$. If $m<n$ change the roles of p_1 and p_2. Now if we define

$$Process = \overline{\cup\{M_n : n \geqslant 1\}}$$

where the bar denotes the completion, then it is possible to prove that the domain equation (4.4) holds for this domain. The proof is an adaptation of the proofs in [de Bakker & Zucker 1982] or in [America & Rutten 1987]. For the role of the compactness consult [de Bakker & Zucker 1983].

The next step is the definition of some operations. Some operations are defined as fixed points of a higher order function. The main advantage of such kind of definitions is that we do not have to consider first the finite case and then define the infinite case with a limit construction. First we define the merge operator $\|$ to be the fixed point of

$$F_{\|} : (Process \times Process \to Process) \to (Process \times Process \to Process)$$

$$\begin{aligned} F_{\|}(\phi)(p_1,p_2)(\sigma) = {} & \{<y,p_2> : y \in p_1(\sigma) \wedge y \in Subst^+ \} \cup \\ & \{y : y \in p_1(\sigma) \wedge y \in Subst^* \cdot \{\delta\} \cup Subst^\omega\} \cup \\ & \{<y,\phi(p,p_2)> : <y,p> \in p_1(\sigma)\} \cup \\ & \{<y,p_1> : y \in p_2(\sigma) \wedge y \in Subst^+ \} \cup \\ & \{y : y \in p_2(\sigma) \wedge y \in Subst^* \cdot \{\delta\} \cup Subst^\omega\} \cup \\ & \{<y,\phi(p_1,p)> : <y,p> \in p_2(\sigma)\} \end{aligned}$$

LEMMA (4.6)
The function $F_{\|}$ is contractive

PROOF
Easily seen by the following argument: each "recursive" appearance of ϕ is guarded by a sequence of substitutions and by the definition of the metric get a factor smaller than or equal to ½.

□

Informally, when we take the merge of two processes p_1 and p_2 we do first a step from p_1 or from p_2 and then we continue with the merge of what is left. Note that the merge is not a fair merge. If we want to have AND fairness, we have to extend our model along the lines of [de Bakker & Zucker 1983].

The function *trace* flattens the tree structure to a set of words: it is the fixed point of

$$F_{trace} : (Process \to (Subst \to \mathcal{P}_{comp}(Subst_\delta^{st}))) \to (Process \to (Subst \to \mathcal{P}_{comp}(Subst_\delta^{st})))$$

$$\begin{aligned} F_{trace}(\phi)(p)(\sigma) = {} & \{ y : y \in p(\sigma) \wedge y \in Subst_\delta^{st} \} \cup \\ & \bigcup \{ y \cdot ((\phi(\overline{p}))(last(y))) : <y,\overline{p}> \in p(\sigma) \} \end{aligned}$$

where the function *last* is only defined on sequences in $Subst^+$, and delivers the last element of such a sequence.

LEMMA (4.7)

The function F_{trace} is contractive

REMARK : In [de Bakker et al 1984] it is shown that if we take closed subsets instead of compact subsets in our processes, the resulting trace sets are not necessarily closed. However, if we take compact sets, the resulting trace sets are compact.

The function *restr* models that if there is a choice between deadlock and other behaviour, that other behaviour is chosen. It is the fixed point of

$$F_{restr} : (Process \to Process) \to (Process \to Process)$$

is given by

$$F_{restr}(\phi)(p)(\sigma) = \begin{cases} p(\sigma) & \text{if } p(\sigma) \subseteq Subst^* \cdot \{\delta\} \\ (p(\sigma) \cap Subst^\infty) \cup \{(y, \phi(q)) : (y,q) \in p(\sigma)\} & \text{otherwise} \end{cases}$$

LEMMA (4.8)
The function F_{restr} is contractive

The function

$$\otimes : Subst_\delta^{sf} \times Process \to \mathcal{P}_{comp}(Subst_\delta \cup Subst^+ \times Process)$$

is given by

$$\otimes(y,p) = \{y\} \cap (Subst^\omega \cup Subst^* \cdot \{\delta\}) \cup (\{y\} \cap Subst^+) \times \{p_2\}.$$

The function

$$\oplus : Subst_\delta^{sf} \times Process \to \mathcal{P}_{comp}(Subst_\delta^{sf} \cup (Subst^+ \times Process))$$

is given by

$$\begin{aligned} \oplus(y,p) = {} & \{y\} \cap (Subst^\omega \cup Subst^* \cdot \{\delta\}) \cup \\ & \{(y \cdot y_1, p_1) : y \in Subst^+ \wedge (y_1, p_1) \in p(last(y))\} \cup \\ & \{y \cdot y_1 : \{y\} \in Subst^+ \wedge y_1 \in p(last(y))\}. \end{aligned}$$

5. COMPOSITIONAL SEMANTICS

Before we come to the definition of the semantic function we need some more definitions. In Concurrent Prolog two or more atoms can be developed in parallel. We have to do some renaming of variables, because otherwise it would be possible to get undesirable bindings.

DEFINITION (5.1)
Let $Seq = \mathbb{N}^*$ be the set of finite words of integers and let r be a typical element of this set. Let $\cdot$ denote concatenation of these words, and let ϵ be the empty word.

We suppose from now on that we can divide the set of variables in the following way

$$Var = \bigcup \{Var_r : r \in Seq\}$$

such that they are pairwise disjoint:

$$\forall r,r' \in Seq : r \neq r' \; [Var_r \cap Var_{r'} = \varnothing],$$

and that there exist injections, for each $r \in Seq$,

$$\Psi_r : Var_\epsilon \to Var_r.$$

Let $Term_r$, $Clause_r$, $Conj_r$ and $Prog_r$ be those subsets of *Term*, *Clause*, *Conj* and *Prog* in which all variables are taken from Var_r. In the same way as we have done for substitutions (see section 2.2) we define

$$\tilde{\Psi}_r : Prog_\epsilon \to Prog_r.$$

Now we are prepared to give the definition of the semantic function. First we try to give some intuition. The size of the "steps" in our process will be the computation till a commit operator. The commitment is the place where choices are made definite. Given a substitution σ and an atom it tries to find a clause $a \leftarrow \cdots$ in the program such that $unify(a, \dot{\sigma}(C))$ is defined. Recall that the dot on σ denotes the generalization to atoms (see section 2.2). If no such clause exist, we have failure and the set $\{\delta\}$ is delivered. We have to be careful: we do not look for a candidate clause $a \leftarrow \cdots$ in P, but in a syntactic variant $\tilde{\Psi}_r(P)$, where all variables are taken from Var_r.

Suppose we can find a clause in $\tilde{\Psi}_r(P)$ of the form $a \leftarrow C_1 \mid C_2$, such that a unifies with $\dot{\sigma}(C)$. We try to resolve the guard C_1. This done recursively in our definition: we assume that we "know" which process is associated with C_1, given P and $r{\cdot}1$ (the concatenation with 1 is to avoid clashes with variables). There is no interaction with other processes during the resolving of the guard, so we can abstract from the tree like structure by applying the operators *trace* and *restr*, and only consider what the possible substitutions are that resolve the guard. We also signify an infinite behaviour. If the set of sequences substitutions is non empty, we can take an element from it ("commit") and continue with the execution of the body C_2, given P and $r{\cdot}2$.

When we have as goal a conjunct, we put them in parallel in a safe way by our mechanism of renaming. The parallelism is modeled by interleaving. It is not difficult to see that this yields the same results as "true" concurrency.

The semantic function *CS* will be the (unique) fixed point of

$$F_{CS} : (Conj \times Prog_\epsilon \times Seq \to Process) \to (Conj \times Prog_\epsilon \times Seq \to Process).$$

First we consider the case that the conjunct is an atom:

$$F_{CS}(\phi)(a,P,r)(\sigma) = \{\delta\}$$

if there is no clause in $\tilde{\Psi}_r(P)$ whose head unifies with $\dot{\sigma}(a)$, and if there exist such a clause we define

$$F_{CS}(\phi)(a,P,r)(\sigma) =$$

$$\{(\overline{\sigma} \mathbin{\hat{\circ}} \sigma \cdot y) \bigcirc \phi(C_2,P,r{\cdot}2) :$$
$$\exists (a' \leftarrow C_1 \not\mid C_2) \in \tilde{\Psi}_r(P)\, [\overline{\sigma} = unify(a', \dot{\sigma}(a)) \wedge$$
$$y \in trace(restr(\phi(C_1,P,r{\cdot}1))(\overline{\sigma} \mathbin{\hat{\circ}} \sigma) \wedge$$
$$\not\mid = | \Rightarrow \bigcirc = \otimes \wedge \not\mid = \& \Rightarrow \bigcirc = \oplus\,]\} \cup$$

$$\{(\overline{\sigma} \mathbin{\hat{\circ}} \sigma) \bigcirc \phi(C_2,P,r{\cdot}2) :$$
$$\exists (a' \leftarrow \not\mid C_2) \in \tilde{\Psi}_r(P)\, [\overline{\sigma} = unify(a', \dot{\sigma}'(a)) \wedge$$
$$\not\mid = | \Rightarrow \bigcirc = \otimes \wedge \not\mid = \& \Rightarrow \bigcirc = \oplus\,]\} \cup$$

GHC, and Parlog. In [Levi & Palamidessi 1987] a language is defined which contains all the languages mentioned above as subset. It would be interesting to see whether our techniques can be used to assign a process based semantics to this language.

An other issue for further research is the fully abstractness of the semantics. We have added extra information to make our semantics compositional. The question arises: did we add not too much information? We expect that we have to remove some of the information that is contained in the tree like structures to make our semantics fully abstract. Interesting work on the comparison between operational and denotational semantics can be found in [Debray & Mishra 1987] and in [van Veen & de Vink 1985]. Both consider semantics for PROLOG. Due to the deterministic nature of this language the compositional denotational semantics equals the operational semantics.

Acknowledgements: we would like to thank the members of the concurrency groups at the Centre for Mathematics and Computer Science and at the Free University, both in Amsterdam. Jaco de Bakker for his insistence on a pure topological model (not mixed with order theory) and for providing the mathematical basis (processes) for this work. The non closedness of some operators on processes were also pointed out by him. Stylistic remarks on this paper are also acknowledged. The work of Erik de Vink on the foundations of logic programming was a starting point from the logic programming side. Some errors in previous versions of this paper were found by him and by Frank de Boer. Jan Rutten and Pierre America gave a fresh look at domain equations. Some of the notation was borrowed from papers from members of the concurrency groups. Also we would like to thank Catuscia Palamidessi for the discussions about read only variables and Krzysztof Apt for providing some material on CP.

7. References

[America & Rutten 1987] P. America, J. Rutten, Solving Reflexive Domain Equations in a Category of Complete Metric Spaces, Proceedings 3rd workshop on the Mathematical Foundations of Programming Language Semantics, to appear.

[Apt & van Emden 1982] K.R. Apt, M.H. van Emden, Contributions to the theory of logic programming, Journal of the ACM 29, 1982, pp. 841-862.

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

[de Bakker & Zucker 1983] J.W. de Bakker, J.I. Zucker, Compactness in Semantics for Merge and Fair Merge, Proceedings Workshop Logic of Programs, Springer, 1983, pp. 18-33.

[de Bakker et al 1984] 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.

[Debray & Mishra 1987] S.K. Debray, P. Mishra, Denotational and Operational Semantics for Prolog, Proceedings of 3rd Working Conference on the Formal Description of Programming Concepts, North-Holland, 1987, pp. 245-270.

[Dugundji 1966] J. Dugundji, Topology, Allyn and Bacon Inc, 1966.

[van Emden 1986] M. van Emden, Quantitative Deduction and its fixpoint theory, Journal on Logic Programming 1, 1986, pp. 37-53.

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

[FITTING 1985A] Fitting, A Deterministic Prolog Fixpoint Semantics, Journal on Logic Programming 2, 1985, pp. 111-118.

[FITTING 1985B] Fitting, A Kripke-Kleene semantics for Logic Programs, Journal on Logic Programming 4, 1985, pp. 295-312.

[GELERNTER 1984] D. Gelernter, A Note on Systems Programming in Concurrent Prolog, Proceedings 1984 Symposium on Logic Programming, IEEE Comp. Society Press, 1984, pp. 76-82.

[LLOYD 1984] J.W. Lloyd, Foundations of Logic Programming, Springer, 1984.

[LEVI & PALAMIDESSI 1985] G. Levi, C. Palamidessi, The declarative semantics of logical read-only variables, Proceedings 1985 Symposium on Logic Programming, IEEE Comp. Society Press, 1985, pp. 128-137.

[LEVI & PALAMIDESSI 1987] G. Levi, C. Palamidessi, An approach to the declarative semantics of synchronization in logic languages, to appear.

[NAIT ABDALLAH 1984] M.A. Nait Abdallah, On the interpretation of infinite computations in logic programming, Proceedings 11th ICALP, (J. Paredaens ed.), Springer, 1984, pp. 358-370.

[SARASWAT 1986] V.A. Saraswat, Problems with Concurrent Prolog, CMU-CS-86-100, Departement of Computer Science, Carnegie-Mellon University, 1986

[SARASWAT 1987] V.A. Saraswat, The concurrent logic programming language CP: Definition and Operational Semantics, Proceedings 1987 Principles Of Programming Languages, 1987, pp. 49-62.

[SHAPIRO 1983] E.Y. Shapiro, A subset of Concurrent Prolog and its interpreter, Techn. Rep. TR-003, ICOT, 1983.

[VAN VEEN & DE VINK 1985] S. van Veen, E. de Vink, Semantics of Logic Programming, Note CS-N8508, Centre for Mathematics and Computer Science, 1985.

APPENDIX: MATHEMATICAL PRELIMINARIES

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

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*), 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 consider only metric spaces with bounded diameter: the distance between two points never exceeds 1.

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:
$\forall\epsilon>0\ \exists N\in N\ \forall n,m>N\ [d(x_n,x_m)<\epsilon]$.
(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:
$\forall\epsilon>0\ \exists N\in N\ \forall n>N\ [d(x,x_n)<\epsilon]$.
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:
$\forall x,y\in M_1\ [d_2(f(x),f(y))=d_1(x,y)]$. 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.*

THEOREM 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) A subset X of a complete metric space (M,d) is called *closed* whenever each Cauchy sequence in X converges to an element of X.
(b) A subset X of a complete metric space (M,d) is called *compact* whenever each sequence in X has a converging subsequence which converges to an element of X.

REMARK :
(a) The definition of compactness given here is in fact the definition of sequential compactness. In a metric space this is equivalent to compactness.
(b) In a metric space every compact set is closed.

DEFINITION A.6
Let $(M,d),(M_1,d_1),(M_2,d_2)$ 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 $f_1,f_2 \in M_1 \to M_2$

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

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

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

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

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

PROPOSITION A.7
Let (M,d), $(M_1,d_1),(M_2,d_2)$, d_F, d_H be as in definition A.6 and suppose that (M,d), $(M_1,d_1),(M_2,d_2)$ are complete. We have that
(a) $(M_1 \to M_2, d_F)$,
(b) $(\mathcal{P}_{closed}(M), d_H)$
(c) $(\mathcal{P}_{comp}(M), d_H)$
are complete metric spaces. (Strictly speaking, for the completeness of $M_1 \to M_2$ we do not need the completeness of M_1.)

The proof of proposition A.7 (a) is straightforward because the distance between two points never exceeds 1. Part (b) and (c) are more involved. They can be proved with the help of the following characterization of the completeness of $(\mathcal{P}_{closed}(M), d_H)$.

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:

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

Proofs of proposition A.7(b) and A.8 can be found in (for instance) [Dugundji 1966] and [Engelking 1977]. The proofs are also repeated in [de Bakker & Zucker 1982].

THEOREM A.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 $\overline{j} : \overline{M} \to M'$ such that $\overline{j} \circ i = j$.*

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.

□