Semantic models for a version of PARLOG

Frank S. de Boer[1], Joost N. Kok[2],
Catuscia Palamidessi[3] and Jan J.M.M. Rutten[1]

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

[2]Department of Computer Science, University of Utrecht,
P.O. Box 80089, 3508 TB Utrecht, The Netherlands

[3]Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, 56100 Pisa, Italy

Abstract. This paper gives four semantics for PARLOG: two operational semantics based on a transition system, a declarative semantics and a denotational semantics. One operational and the declarative semantics model the success set of a PARLOG program, that is, the set of computed answer substitutions corresponding to all successfully terminating computations. The other operational and the denotational semantics model also deadlock and infinite computations. For the declarative and the denotational semantics we extend standard notions like unification in order to cope with the synchronization mechanism of PARLOG. The basic mathematical structure for the declarative semantics is the set of finite streams of substitutions. In the denotational semantics we use tree-like structures that are labelled with streams of substitutions. We look at the relations between the different models: First we relate the two operational semantics and next we show the relation of the declarative and denotational semantics with their operational counterparts. We treat a version of PARLOG because we do not cover all aspects of the language.

Key words and phrases: operational semantics, denotational semantics, declarative semantics, parallelism, concurrent logic languages, correctness, complete metric spaces.

1 Introduction.

The language PARLOG [9,10,17], as well as most of the concurrent logic languages, is based on the Horn Clause Logic (HCL) plus some mechanisms for expressing concurrency. One of the main drawbacks of this approach is that these new mechanisms heavily affect the clean declarative understanding of HCL. Indeed, although many operational semantics have been investigated ([26,27,28,3,16,4]), a satisfactory declarative one is still to be defined. PARLOG belongs to a class of concurrent logic languages whose main features are:

- the *input-constraints*, on which the mechanism of synchronization between AND-processes is based, and

Part of this work was carried out in the context of ESPRIT 415: Parallel Architectures and Languages for Advanced Information Processing – a VLSI-directed approach.

- the presence of *commit*, that realizes the *don't know nondeterminism*, controlled by guards.

Other languages in this class are Guarded Horn Clauses [32,33], Concurrent Prolog [29,30], and their flat versions. These mechanisms affect the semantics of the pure underlying language in several ways [31]:

- the *success set* is reduced by the input-constraints,
- the *finite failure set* is enlarged by the commit, and modified (i.e., either reduced or enlarged) by the input-constraints,
- the *infinite failure set* is modified both by the commit and the input-constraints.

In this paper we address the problem of characterizing these new sets, for PARLOG, first in a declarative and then in a compositional way (i.e., by giving the meaning of a composite goal in terms of the meaning of its conjuncts). We deal with a version of PARLOG. For example, we do not consider unification related primitives, OR-parallel aspects and local deadlock in deep guards.

First, we describe these sets by a formal operational semantics based on a transition relation (in the style of [18], see also [27,16,4] for similar approaches). The operational meaning is given in terms of sets of *words* (or *streams*) of substitutions, that correspond to the answers computed during the derivation.

Next, we characterize declaratively the new success set, as the least fixpoint of an *immediate consequence* operator on interpretations. (The full model-theoretic semantics is still under investigation.) Our approach can be considered an evolution of the one developed in [21] and [22]. The basic idea there is to model the ability of a process to produce and to consume data structures. This is done by introducing annotations on data structures (terms) and by extending the Herbrand universe with variables (see also [15] and [14]). However, the declarative semantics presented in those papers is not able to fully characterize the behaviour of concurrent logic languages. The main problem has to do with the situation of *deadlock* that arises when two processes are obliged to wait for each other for bindings. Consider, for example, the goal $\leftarrow p(x,y), q(x,y)$ and the programs

$$P_1 = \{p(a,b) \leftarrow |.,\ \ q(a,b) \leftarrow |.\},$$

$$P_2 = \{p(z,b) \leftarrow |r(z).,\ \ r(a) \leftarrow |.,\ \ q(a,b) \leftarrow |.\},$$

and assume that in both cases the first argument of p and the second argument of q are input-constrained (expressed in PARLOG by the declaration modes $D_1 = \{p(?,\hat{}), q(\hat{},?)\}$, and $D_2 = \{p(?,\hat{}), q(\hat{},?), r(?)\}$, respectively). According to the operational semantics, the computation of the goal cannot succeed in P_1 (it results in a deadlock), whilst in P_2 always can. Now it is the case that the approach presented in [21] and [22] is not able to distinguish between the two situations. Indeed, $r(a^+, b^+)$ (r producing a and b) happens there to be *true* in the models (and in the least fixpoint interpretation) of both the programs. So, a full completeness result (between the declarative and the operational semantics) could not be obtained. For a detailed discussion of this problem see also [23].

Our solution to this problem consists in enriching the interpretations with streams of substitutions. Due to the presence of guards, whose evaluation has to be interpreted as an atomic action (internal action), the streams of the operational semantics offer too little structure, and we have to add some delimiters to represent *critical sections*. We call these new structures *sequences*. This allows us to characterize declaratively

(and, therefore, compositionally) the bindings obtained at different stages in the computation. In this way we obtain a full equivalence result. An other basic difference with respect to the previous approach is to annotate the variables instead of the data constructors. This allows us to extend the unification theory ([11,20]) in order to deal with input-constraints in a formal way. We also give an extended algorithm for the computation of the (extended) most general unifier. Moreover, we introduce the notion of parallel composition of substitutions, that allows us to model the combination of the substitutions computed by and-parallel processes.

Other compositional models for the success set are presented in [26] and [24]. Both these approaches are based on streams of input/output *simple* substitutions, where *simple* means that the bindings are of the form x/y or $x/f(x_1, ..., x_n)$. This restriction introduces additional complications for modeling the full unification mechanism. Thanks to our extended unification theory, we deal directly with (general) substitutions, and the correspondence with the operational semantics is therefore simpler and more intuitive.

Finally we consider the problem of characterizing the finite failures and the infinite computations in a compositional way. It turns out that the streams of the declarative semantics offer again too little structure, but also the sequences introduced in the declarative semantics are not powerfull enough. Indeed, in order to model the failure set, we need not only to distinguish between external and internal computation steps, but also between different points of nondeterministic choice.

To justify this, let's illustrate how the absence of nondeterminism informations (branching informations) causes our operational semantics to be *not* compositional. Consider the programs

$$P_1 = \{p(x) \leftarrow |q(x)., \;\; p(x) \leftarrow |r(x)., \;\; q(a) \leftarrow |. \;\; r(b) \leftarrow |. \;\; s(a) \leftarrow |.\},$$

$$P_2 = \{p(x) \leftarrow |q(x)., \;\; q(a) \leftarrow |. \;\; q(b) \leftarrow |. \;\; s(a) \leftarrow |.\},$$

with mode declarations $D_1 = \{p(?), q(?), r(?), s(\hat{})\}$, and $D_2 = \{p(?), q(?), s(\hat{})\}$, respectively. Consider the goal $\leftarrow p(y)$. Operationally, in both P_1 and P_2 it will suspend waiting for a binding on y (either a or b). However, if we extend the goal with an atom $s(x)$, thus yielding the goal $\leftarrow p(y), s(y)$, then we get *different* operational meanings. In P_1 the goal can fail (due to the choice of the wrong clause for $p(y)$), whereas in P_2 it cannot.

A possible way to provide these branching informations is to use tree-like structures. However, it still remains an open question what exactly is the minimal information needed to obtain a compositional semantics. This question is related to the problem of *full abstractness*. A fully abstract denotational semantics for Flat-Concurrent Prolog is described in [16]. Their approach is based on *suspension sets*, which are a more abstract structure than the three-like one. However, it is not clear how their result can be extended to the general case of non-flat guards. The same applies to the declarative approach taken in [13] to characterize the finite failures.

In our approach we code the branching informations by using trees labelled with streams of substitutions. We see them as elements of complete metric spaces, satisfying so-called reflexive domain equations ([8], [2]). We use a *denotational* style: for every operator in the language we define a semantic operator that can be seen as a function on these spaces. The meaning of a goal can then be given by a semantic operator that results to be the *unique* fixpoint of a *contraction* on the functional metric spaces ([8], [2]). The relation of this denotational semantics with respect to the operational one is obtained via an *abstraction operator*, that identifies some denotations. The correctness of the denotational semantics is then stated as the equality between the result of this abstraction and the operational semantics.

Due to space limitations we only give proofs in outline form. Full proofs can be found in [25] (for the declarative part) and in [6] (for the denotational part).

The definition of PARLOG has been changed with respect to the previous versions. We consider the one described in [17].

2 The language PARLOG

To describe the syntax of the language PARLOG we introduce the following sets:

- The set of atoms, with typical elements A, B, H, we denote by *Atom*.
- The set of conjunctions, with typical elements $\bar{A}, \bar{B}, \bar{G}$, we denote by *Conj*.
- The set of goals, with typical elements $\leftarrow \bar{A}, \leftarrow \bar{B}, \leftarrow \bar{G}$, we denote by *Goal*.
- The set of clauses, with typical element C, we denote by *Clause*.
- The set of programs, with typical element W, we denote by *Prog*.

Conjunctions are of the form: $\bar{A} = A_1, \ldots, A_n$. A special element in *Conj* is *true*, denoting the empty conjunct. With $\square$ we denote the goal $\leftarrow$ *true*. A clause is of the form $C = H \leftarrow \bar{G}|\bar{B}$, where H, $\bar{G}$ and $\bar{B}$ are called the head, the guard, and the body of the clause, respectively. The symbol | is called the commit operator. We do not consider operators (like ;) that impose any ordering on clauses. Every program W consists of a finite set of clauses together with a so-called *mode declaration*, which specifies for every predicate which of its arguments are *input* and *output*. They are indicated by the symbols ? and ^ respectively. So, for instance, the declaration $p(?, ?, \hat{})$ specifies that the first two arguments of p are *input* and the third one is *output*.

An atom A in a goal is seen as an (AND-) process. Its computation proceeds by looking for a *candidate clause* in W. A clause is candidate if its head H *input-unifies* with A (i.e. the input arguments unify) and the computation of the guard succeeds, both without binding the (variables in the) input arguments of A. If there are candidate clauses, then the computation of A *commits* to one of them (i.e. no backtraking will take place), the *output-unification* is performed and A is replaced by the body of the clause. If no clauses are candidate but there are *suspended* clauses (i.e. clauses in which the input unification would succeed and bind the input-arguments), then the computation of A *suspends*, and will be resumed when its (input) arguments get bound by other processes in the goal. If a guard would succeed by binding the input-arguments (of A), then an *error* is generated (*unsafe guard*). If none of this cases applies, then the process A and the whole goal *fail*. Of course, a failure occurs also when all the processes in the goal get suspended (*deadlock*).

To simplify the discussion, we do not deal with the *error* case. More precisely, we include this case into the suspension case. So, we consider a suspension mechanism similar to the one of GHC, namely: *a clause suspends if either the input-unification or the goal evaluation would instantiate the input-arguments of* A.

3 Operational semantics.

For the rest of the paper let W denote a fixed program. The set of variables occurring in a conjunction $\bar{A}$ is indicated by $\mathcal{V}(\bar{A})$. We postulate a function *invar* that gives for every atom A the set of variables occurring in those arguments of A that are specified as input by the mode declaration of W. Given a set of variables V, W_V denotes the program whose clauses are *variants* (see [20]), with respect to V, of the clauses of W.

We introduce the set of substitutions $(\vartheta, \gamma \in)$ *Subst*. ϵ is the *empty substitution*. For V a finite set of variables, we use $\vartheta_{|V}$ to denote the restriction of ϑ to V. Further we have the familiar notion of *mgu*, which is a partial function from pairs of atoms to substitutions. We introduce the notions of *input* and *output* mgu's: Consider two atoms $A = p(t_1, ..., t_n)$ and $A' = p(t'_1, ..., t'_n)$. Assume that the declaration-mode of p has the symbol ? (input-mode) on the arguments $i_1, ..., i_k$. Then, $mgu_i(A, A')$ denotes $mgu(\{\{t_{i_1}, t'_{i_1}\}, \ldots \{t_{i_k}, t'_{i_k}\}\})$. In a similar way we define $mgu_o(A, A')$ to be the *mgu* of the output arguments.

The operational semantics will be based on the following *transition relation*:

Definition 3.1 (Transition relation)
Let $\rightarrow \subseteq (Goal \times Subst) \times (Goal \times Subst)$ be the smallest relation satisfying

1. If $\exists H \leftarrow \bar{G} | \bar{B} \in W_{\mathcal{V}(A)}, \exists mgu_i(A\vartheta, H)$
   $[<\leftarrow \bar{G}, mgu_i(A\vartheta, H) > \stackrel{*}{\rightarrow} < \Box, \vartheta' >,$ *and* $\vartheta'|_{invar(A\vartheta)} = \epsilon]$,
   then $<\leftarrow A, \vartheta > \rightarrow <\leftarrow outunif(A\vartheta, H\vartheta'), \bar{B}, \vartheta\vartheta' >$
2. If $\exists mgu_o(A\vartheta, H\vartheta')$,
   then $<\leftarrow outunif(A\vartheta, H\vartheta'), \bar{B}, \vartheta' > \rightarrow <\leftarrow \bar{B}, \vartheta' mgu_o(A\vartheta, H\vartheta') > .$
3. If $<\leftarrow \bar{A}, \vartheta > \rightarrow <\leftarrow \bar{A}', \vartheta' > \mid < \Box, \vartheta' >$
   then $<\leftarrow \bar{A}, \bar{B}, \vartheta > \rightarrow <\leftarrow \bar{A}', \bar{B}, \vartheta' > \mid <\leftarrow \bar{B}, \vartheta' >$
   $<\leftarrow \bar{B}, \bar{A}, \vartheta > \rightarrow <\leftarrow \bar{B}, \bar{A}', \vartheta' > \mid <\leftarrow \bar{B}, \vartheta' >$

◇

In these transitions, ϑ represents the substitution that has been computed until that moment. In 1., it is stated that we can resolve $\leftarrow A$ if we can find a (renamed) clause in our program with a head H that can be input-unified with A; moreover, the refutation of the guard $\bar{G}$ of that clause must terminate successfully and the total substitution ϑ' must not instantiate any input variables of $A\vartheta$. The transition 2. represents the first action performed after the commitment, namely the output-unification. A conjunction, in 3., is evaluated by the parallel execution of its conjuncts, modelled here by interleaving. In the following definition we give the operational semantics.

Definition 3.2 (Operational semantics)
We define

$$\mathcal{O}_1 : Goal \rightarrow M_1, \quad \text{with } M_1 = \mathcal{P}(Subst)$$
$$\mathcal{O}_2 : Goal \rightarrow M_2, \quad \text{with } M_2 = \mathcal{P}(Subst^\infty_\delta).$$

(Here $Subst^\infty_\delta = Subst^+ \cup Subst^\omega \cup Subst^*.\{\delta\}$, with typical element $\vartheta_1 . \cdots \vartheta_n . \cdots$; the symbol δ denotes failure; $\mathcal{P}(X)$ is the set of all the subsets of X.)

We put $\mathcal{O}_i[\![\leftarrow true]\!] = \{\epsilon\}$, and

$$\mathcal{O}_1[\![\leftarrow \bar{A}]\!] = \{\vartheta_{|\mathcal{V}(\bar{A})} \mid <\leftarrow \bar{A}, \epsilon > \stackrel{*}{\rightarrow} < \Box, \vartheta >\};$$

$$\begin{aligned}\mathcal{O}_2[\![\leftarrow \bar{A}]\!] = {} & \{(\vartheta_1 . \cdots \vartheta_n)_{|\mathcal{V}(\bar{A})} \in Subst^+ \mid \\ & \quad <\leftarrow \bar{A}, \epsilon > \rightarrow <\leftarrow \bar{A}_1, \vartheta_1 > \rightarrow \cdots \rightarrow < \Box, \vartheta_n >\} \\ \cup {} & \{(\vartheta_1 . \cdots \vartheta_n)_{|\mathcal{V}(\bar{A})} . \delta \in Subst^* . \{\delta\} \mid \\ & \quad <\leftarrow \bar{A}, \epsilon > \rightarrow \cdots \rightarrow < \bar{A}_n, \vartheta_n > \not\rightarrow \wedge \leftarrow \bar{A}_n \neq \Box\} \\ \cup {} & \{(\vartheta_1 . \cdots)_{|\mathcal{V}(\bar{A})} \in Subst^\omega \mid <\leftarrow \bar{A}, \epsilon > \rightarrow <\leftarrow \bar{A}_1, \vartheta_1 > \rightarrow \cdots\}.\end{aligned}$$

◇

The *success set* for $\leftarrow \bar{A}$ is given by $\mathcal{O}_1[\![\leftarrow \bar{A}]\!]$: it contains all computed answer substitutions corresponding to all successfully terminating computations. The set $\mathcal{O}_2[\![\leftarrow \bar{A}]\!]$ takes in addition into account all failing and infinite computations, represented by elements of $Subst^* \cdot \{\delta\}$ and $Subst^\omega$, respectively. The relation between $\mathcal{O}_1$ and $\mathcal{O}_2$ is obvious: If we set

$$last(X) = \{\vartheta \mid \exists s \in Subst^*(s.\vartheta \in X)\}$$

then we have: $\mathcal{O}_1 = last \circ \mathcal{O}_2$.

In the following sections, $\mathcal{O}_1$ and $\mathcal{O}_2$ will be related to a declarative and a denotational semantics, respectively.

4 Declarative semantics.

In this section we define the declarative (fixpoint) semantics of PARLOG. We make use of an extended notion of Herbrand base and interpretations, enriched with variables (that allows to model the notion of *computed substitution*, [22],[15],[14]), and with *annotations* (that allows to model the synchronization mechanism of concurrent logic languages, see [21] and [22] for similar approaches). We extend the standard notions of the unification theory ([11], [20]) in a formal framework. Moreover, we introduce the notion of *parallel composition*, that allows to formalize the combination (plus consistency check) of the substitutions computed by subgoals run in parallel. Finally, we introduce the notion of *sequences of substitutions*, that allows to overcome the difficulties presented in [22] about the situations of deadlock. A similar construction has been made for defining the declarative semantics of GHC ([5], [25]). For the proofs we refer to [25].

4.1 Annotated variables.

In order to model the synchronization mechanism of PARLOG we introduce the notion of *annotated variable*. The annotation can occur on a variable in the goal, and it means that such a variable is in an input-argument and therefore cannot be bound, during the derivation step, before commitment. In other words, such a variable can receive bindings from the execution of other atoms in the goals, but cannot produce bindings by the execution of the atom in which it occurs (before commitment).

We will denote the set of variables, with typical elements $x, y, \ldots$, by Var, and the set of the annotated variables, with typical elements $x^-, y^-, \ldots$, by Var^-. We can consider "$^-$" as a bijective mapping $^- : Var \rightarrow Var^-$. The set of terms $Term$ is extended on the new set of variables $Var \cup Var^-$. The set of variables occurring in the term t is denoted by $\mathcal{V}(t)$. For $V \subseteq Var$, $t^{|V^-}$ is the term obtained by replacing in t every variable $x \in V$ by x^-. The term $t^{|Var^-}$ will be simply denoted by t^-.

A substitution ϑ is now a mapping $\vartheta : Var \cup Var^- \rightarrow Term$, such that only a finite number of variables are mapped into terms different from themselves. In order to model the difference between producing and receiving a binding we introduce an asymmetry in the definition of the application of a substitution ϑ to a term (or atom, or formula) t:

$$t\vartheta = \begin{cases} \vartheta(x) & \text{if } t = x \in Var \\ \vartheta(x^-) & \text{if } t = x^- \in Var^- \text{ and } \vartheta(x^-) \neq x^- \\ \vartheta(x)^- & \text{if } t = x^- \in Var^- \text{ and } \vartheta(x^-) = x^- \\ f(t_1\vartheta, \ldots, t_n\vartheta) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

The reason why the application of ϑ to x^- can result in $\vartheta(x)^-$ (instead of $\vartheta(x)$) is related to the peculiarity of the input-mode constraint of PARLOG. In fact, it does not apply to a specific variable (as in CP), but to the arguments of the atom. Therefore, when an annotated variable is bound to a term t, all the variables occurring in t get under the influence of the input-mode constraint, and therefore they have to inherit the annotation.

We factorize the set of substitution with respect to the equivalence relation $\vartheta_1 \equiv \vartheta_2$ *iff* $\forall x \in Var \cup Var^-[x\vartheta_1 = x\vartheta_2]$. From now on, a substitution ϑ will indicate its equivalence class.

Example 4.1 Consider the atom $A = p(f(x,y),x,y)$. We annotate the variables in A so to get $A^- = p(f(x^-,y^-),x^-,y^-)$. Consider now the substitution $\vartheta = \{x/g(z), y/h(w), y^-/h(a)\}$. We have: $A^-\vartheta = p(f(g(z^-),h(a)),g(z^-),h(a)))$. $\diamond$

The notion of composition $\vartheta_1\vartheta_2$, of two substitutions, ϑ_1 and ϑ_2 is extended as follows

$$\forall x \in Var \cup Var^-[x(\vartheta_1\vartheta_2) = (x\vartheta_1)\vartheta_2].$$

The composition is associative and the empty substitution ϵ is the neutral element. ϑ is called *idempotent* iff $\vartheta\vartheta = \vartheta$. Given a set of sets of terms M, we define ϑ to be an unifier for M iff

$$\forall S \in M : \forall t_1, t_2 \in S\ [t_1\vartheta = t_2\vartheta \text{ and } t_1^-\vartheta = t_2^-\vartheta].$$

The ordering on substitutions is the standard one, namely: $\vartheta_1 \leq \vartheta_2$ *iff* $\exists\vartheta_3\ [\vartheta_1\vartheta_3 = \vartheta_2]$ (ϑ_1 *is more general than* ϑ_2). The set of mgu's (most general unifiers) of a set of sets of terms M is denoted by $mgu(M)$.

We give an extended version of the unification algorithm, based on the one presented in [1], that works on finite sets of pairs. Given a finite set of finite sets of terms M, consider the (finite) set of pairs

$$M_{pairs} = \bigcup_{S \in M} \{< t, u > |\ t, u \in S\}.$$

We define the unifiers of a set $\{< t_1, u_1 >, \ldots, < t_n, u_n >\}$ as the ones of $\{\{t_1, u_1\}, \ldots, \{t_n, u_n\}\}$. Of course, M and M_{pairs} are equivalent (i.e. they have the same unifiers). A set of pairs is called *solved* if it is of the form

$$\{< x_1, t_1 >, \ldots, < x_n, t_n >\}$$

where all the x_i's are distinct elements of $Var \cup Var^-$, $x_i \notin \mathcal{V}(t_1, \ldots, t_n)$, and, if $x_i \in Var$ and $t_i \neq x_i^-$, then $x_i^- \notin \mathcal{V}(x_1, \ldots, x_n, t_1, \ldots, t_n)$. For P solved, define $\gamma_P = \{x_1/t_1, \ldots, x_n/t_n\}$, and $\delta_P = \gamma_P\gamma_P$.

The following algorithm transforms a set of pairs into an equivalent one which is solved, or halts with failure if the set has no unifiers.

Definition 4.2 (Extended unification algorithm)

- Let P, P' be sets of pairs. Define $P \Rightarrow P'$ if P' is obtained from P by choosing in P a pair of the form below and by performing the corresponding action

1. $< f(t_1, ..., t_n), f(u_1, ..., u_n) >$	replace by the pairs $< t_1, u_1 >, ..., < t_n, u_n >$

2. $< f(t_1,...,t_n), g(u_1,...,u_n) >$, where $f \neq g$		halt with failure
3. $< x, x >$	where $x \in Var \cup Var^-$	delete the pair
4. $< t, x >$	where $x \in Var \cup Var^-$, $t \notin Var \cup Var^-$	replace by the pair $< x, t >$
5. $< x, t >$	where $x \in Var$, $x \neq t$, $x^- \neq t$, x or x^- occurs in other pairs	if $x \in \mathcal{V}(t)$ or $x^- \in \mathcal{V}(t)$ then halt with failure else apply the substitution $\{x/t\}$ to all the other pairs
6. $< x, x^- >$	where $x \in Var$, and x occurs in other pairs	apply the substitution $\{x/x^-\}$ to all the other pairs
7. $< x^-, t >$	where $x^- \in Var^-$, $x^- \neq t$ and x^- occurs in other pairs	if $x^- \in \mathcal{V}(t)$ then halt with failure else apply the substitution $\{x^-/t\}$ to all the other pairs.

We will write $P \Rightarrow \mathit{fail}$ if a failure is detected (steps 2, 5 or 7).

- Let $\Rightarrow^*$ be the reflexive-transitive closure of the relation $\Rightarrow$, and let P_{sol} be the set $P_{sol} = \{P' \mid \mathit{symm}(P) \Rightarrow^* P'$, and P' is solved$\}$, where

$$\begin{aligned} \mathit{symm}(\{< t_1, u_1 >, \ldots, < t_n, u_n >\}) &= \{< t_1, u_1 >, \ldots, < t_n, u_n >\} \\ &\cup \{< t_1^-, u_1^- >, \ldots, < t_n^-, u_n^- >\}. \end{aligned}$$

The set of substitutions determined by the algorithm is

$$\Delta(P) = \{\delta_{P'} \mid P' \in P_{sol}\}. \qquad \diamond$$

The following proposition shows that the set of the idempotent most general unifiers of M is finite and can be computed in finite time by the extended unification algorithm.

Proposition 4.3 Let P be a finite set of pairs, and M be a finite set of finite sets of terms.

1. **(finiteness)** The relation $\Rightarrow$ is *finitely-branching* and *noetherian* (i.e. *terminating*).
2. **(solved form)** If P is in *normal form* (i.e. there exist no P' such that $P \Rightarrow P'$), then P is in solved form.
3. **(soundness)** $\Delta(P) \subseteq mgu(P)$
4. **(completeness)** $mgu(M) \subseteq \Delta(M_{pairs})$.
5. $P \Rightarrow^* \mathit{fail}$ iff P is not unifiable. $\diamond$

4.2 Parallel composition on substitutions.

In this section we introduce the notion of *parallel composition* on substitutions and on sets of substitutions, both denoted by $\hat{\circ}$. Intuitively, the parallel composition is meant to be the formalization of one of the basic operations performed by the *parallel execution model* of logic programs. When two atoms A_1 and A_2 (in the same goal)

are run in parallel, the associated computed answer substitutions ϑ_1 and ϑ_2 have to be combined, afterwards, in order to get the final result. This operation can be performed in the following way: Consider the set of all the pairs corresponding to the bindings of both ϑ_1 and ϑ_2. Then, compute the most general unifier of such a set. Note that the *consistency check* corresponds to a verification that such a set is unifiable.

Definition 4.4 In the following, $\mathcal{S}(\vartheta)$ is the set of sets $\{\{x,t\} \mid x/t \in \vartheta\}$. Θ_1, Θ_2 are sets of substitutions.

1. $\vartheta_1 \hat{\circ} \vartheta_2 = mgu(\mathcal{S}(\vartheta_1) \cup \mathcal{S}(\vartheta_2))$.
2. $\Theta_1 \hat{\circ} \Theta_2 = \bigcup_{\vartheta_1 \in \Theta_1, \vartheta_2 \in \Theta_2} \vartheta_1 \hat{\circ} \vartheta_2$.

We will denote the sets $\{\vartheta\} \hat{\circ} \Theta$ and $\Theta \hat{\circ} \{\vartheta\}$ by $\vartheta \hat{\circ} \Theta$ and $\Theta \hat{\circ} \vartheta$ respectively. ◇

Example 4.5

1. Consider the program $\{p(f(a)) \leftarrow |., q(f(a)) \leftarrow |.\}$, with declaration-mode $\{p(?), q(\hat{})\}$, and consider the goal $\leftarrow p(x), q(x)$. We annotate the variable x, in $p(x)$, in order to express the input-mode constraint. We have
   $mgu(p(x^-), p(f(a))) = \{\vartheta_1\}$, where $\vartheta_1 = \{x^-/f(a)\}$, and
   $mgu(q(x), q(f(a))) = \{\vartheta_2\}$, where $\vartheta_2 = \{x/f(a)\}$.
   Now observe that $\vartheta_1 \in mgu(\mathcal{S}(\vartheta_1))$, $\vartheta_2 \in mgu(\mathcal{S}(\vartheta_2))$ and $\vartheta_1 \leq \vartheta_2$, therefore $\vartheta_2 \in \vartheta_1 \hat{\circ} \vartheta_2$.
2. Consider now the same program and goal as before, but let the declaration mode be $\{p(?), q(?)\}$. We have $mgu(p(x^-), p(f(a))) = mgu(q(x^-), q(f(a))) = \{\vartheta_1\}$, and $\vartheta_1 \in \vartheta_1 \hat{\circ} \vartheta_1$, whilst $\vartheta_2 \notin \vartheta_1 \hat{\circ} \vartheta_1$.

In 1. the goal can be refuted by a suitable ordering on the execution of the atoms ($q(x)$ before $p(x)$). This corresponds to get a substitution (ϑ_2), that does not bind any annotated (i.e. input-constrained) variable. This is not the case in 2., and indeed no refutation are possible. ◇

4.3 Sequences of substitutions.

As shown in [15], and [14], the computed bindings in HCL can be declaratively modeled by using a not ground Herbrand Base, or equivalently, a set of couples atom-substitution. However, when the input-constraints are present, it is not sufficient to consider only a substitution. In fact, as shown in [21] and [22], a *flat* representation of the computed bindings is not powerful enough to model the *effects* of the possible interleavings in the executions of the atoms in a goal. Namely, the possibility for atoms to provide each other the bindings necessary for going on in the respective computations. In a sense, we have to *register* the whole history of the execution of the atom, and therefore we have to deal with *sequences of substitutions*. Since we only model declaratively the success set, we need to consider only finite sequences. Anyway, the set $Subst^+$ used for the operational semantics is still a too weak structure. Indeed, to represent the *critical sections* given by the input-unification and the guard evaluation, we need to separate a subsequence from the rest.

Definition 4.6 The finite sequences of substitutions, with typical element s, are defined by the following (abstract) syntax

$$s ::= \vartheta \mid [s] \mid s_1.s_2$$ ◇

The role of the squared brackets, is to delimitate the *critical sections*. Their meaning will be clarified by the definition of the *interleaving operator*. We introduce the following notations. If S and S' are sets of sequences, then $S.S' \stackrel{def}{=} \{s.s' \mid s \in S, s' \in S'\}$ and $[S] \stackrel{def}{=} \{[s] \mid s \in S\}$. If $s = \vartheta'.s'$, then $\vartheta \circ s \stackrel{def}{=} (\vartheta \circ \vartheta').s'$ and $\vartheta \circ [s] \stackrel{def}{=} [(\vartheta \circ \vartheta').s']$. For Θ a set of substitution we have $\Theta \circ s \stackrel{def}{=} \bigcup_{\vartheta \in \Theta} \vartheta \circ s$.

Definition 4.7 (Interleaving operator).

1. $$\begin{aligned} s_1 \parallel s_2 = \quad & \{\vartheta.s \mid \exists s' : \vartheta.s' = s_1, s \in s' \parallel s_2\} \\ \cup \quad & \{\vartheta.s \mid \exists s' : \vartheta.s' = s_2, s \in s' \parallel s_1\} \\ \cup \quad & \{[s'].s \mid \exists s'' : [s'].s'' = s_1, s \in s'' \parallel s_2\} \\ \cup \quad & \{[s'].s \mid \exists s'' : [s'].s'' = s_2, s \in s'' \parallel s_1\} \end{aligned}$$
2. $$S_1 \parallel S_2 = \bigcup_{s_1 \in S_1, s_2 \in S_2} s_1 \parallel s_2.$$ ◇

The following definition introduces the notion of *result* $\mathcal{R}$ of a sequence s (or a set of sequences S) of substitutions. Roughly, such a result is obtained by performing the parallel composition of each element of the sequence with the next one, and by checking, each time, that the partial result does not map annotated variables.

Definition 4.8

1. $\mathcal{E}(\Theta) = \{\vartheta \in \Theta \mid \vartheta_{|Var^-} = \epsilon\}$
2. $\mathcal{R}(\vartheta) = \mathcal{E}(\{\vartheta\})$
3. $\mathcal{R}([s]) = disann(\mathcal{R}(s))$
4. $\mathcal{R}(s_1.s_2) = \mathcal{R}(\mathcal{R}(s_1) \circ s_2)$
5. $\mathcal{R}(S) = \bigcup_{s \in S} \mathcal{R}(s)$.

where $disann(s)$ removes all the annotations in s. Thus, rule 3. specifies that, after a critical section, the input-constraints are released. $\mathcal{E}(\Theta)$ is a *filter* that eliminates from Θ all the substitutions mapping annotated variables. ◇

4.4 Least fixpoint semantics.

In this section we introduce the notion of interpretation, and we define a continuous mapping (associated to the program) on interpretations. The least fixpoint of this mapping will be used to define the fixpoint semantics. Such a mapping is the extension of the *immediate consequence* operator for HCL ([12],[1]).

The *Herbrand base with variables* $\mathcal{B}$ associated to the program W is the set of all the possible atoms that can be obtained by applying the predicates of W to the elements of *Term*. An interpretation I of W is a set of pairs of the form $< A, s >$, where A is an atom in $\mathcal{B}$ and s is a sequence of substitutions on *Var* and *Term*. $< A, s > \in I$ can be read declaratively as *A is true in I under the sequence s*. We remark the symilarity with the temporal logic, although we do not investigate this relation here. $\mathcal{I}$ will denote the set of all the interpretations of W.

$\mathcal{I}$ is a complete lattice with respect to the set-inclusion, where the empty set $\emptyset$ is the minimum element, and the *set union* $\cup$ and the *set intersection* $\cap$ are the *sup* and *inf* operations, respectively.

The following definition, that will be used in the least fixpoint construction, is mainly introduced for technical reasons.

Definition 4.9 Let $s_1, \ldots, s_h$ be sequences of substitutions, and let $A_1, \ldots, A_k$ ($h \leq k$) be atoms. $s_1, \ldots, s_h$ are *locally independent* on $A_1, \ldots, A_k$ iff

$$\forall s_i, \forall \vartheta \text{ in } s_i : (\mathcal{D}(\vartheta) \cup \mathcal{C}(\vartheta)) \cap \mathcal{V}(A_1, \ldots, A_k) \subseteq \mathcal{V}(A_i).$$

where $\mathcal{D}(\vartheta)$ and $\mathcal{C}(\vartheta)$ are the standard *domain* and *codomain* of ϑ. ◇

In the following, we use the notation $\bar{s}$ to denote a sequence of sequences of substitutions $s_1, \ldots, s_n$. Moreover, if $\bar{s} = s_1, \ldots, s_n$ and $\bar{A} = A_1, \ldots, A_n$, then $< \bar{A}, \bar{s} >$ stands for $< A_1, s_1 >, \ldots, < A_n, s_n >$, and $\| (\bar{s})$ stands for $s_1 \| \cdots \| s_n$.

Definition 4.10 The mapping $T : \mathcal{I} \rightarrow \mathcal{I}$, associated to W, is defined as follows:

$$\begin{aligned} T(I) = \quad \{< A, s > | \quad & \exists H \leftarrow \bar{G} | \bar{B} \in W_{\mathcal{V}(A)}, \\ & \exists \bar{s}', \bar{s}'' \text{ locally independent on } \bar{G}, \bar{B}, A, \\ & < \bar{G}, \bar{s}' >, < \bar{B}, \bar{s}'' > \in I : \\ & s \in [mgu_i(A^-, H).(\| (\bar{s}'))].mgu_o(A, H).(\| (\bar{s}''))\} \end{aligned}$$ ◇

A possible sequence for A results from the critical section containing the mgu_i with the head of a clause, and a sequence resulting from the guard. The input variables in A are annotated. The whole is followed by the mgu_o and a sequence resulting from the body.

Proposition 4.11 T is continuous. Then, its least fixpoint $lfp(T)$ exists, and $lfp(T) = \bigcup_{n \geq 0} T^n(\emptyset)$ holds. ◇

We define the least fixpoint semantics associated to W as the set $\mathcal{F}(W) = lfp(T)$.

Theorem 4.12 (Equivalence of declarative and operational semantics)

$$\begin{aligned} & \mathcal{O}_1(\leftarrow \bar{A}) = \\ & \{\vartheta \mid \exists \bar{s} \text{ locally independent on } \bar{A} : < \bar{A}, \bar{s} > \in \mathcal{F}(W) \text{ and } \vartheta \in \mathcal{R}(\| (\bar{s}))_{|\mathcal{V}(\bar{A})}\} \end{aligned}$$ ◇

Example 4.13

1. Consider the program $\{p(y) \leftarrow q(y)|., \quad q(a) \leftarrow |.\}$, with declaration-mode $\{p(?), q(\hat{})\}$, and consider the goal $\leftarrow p(x)$. The possible s's such that $< p(x), s > \in lfp(T)$, are those of the form $s = [\{y/x^-\}.\{y/a\}]$. We have: $\mathcal{R}(s) = disann(\mathcal{E}(\{y/x^-\} \circ \{y/a\})) = disann(\mathcal{E}(\{\{x^-/a, y/a\}\})) = \emptyset$, and indeed no refutations are possible.
2. Consider now the program $\{p(y) \leftarrow |q(y)., \quad q(a) \leftarrow |.\}$, with the same declaration mode. The possible s's are of the form $s = [\{y/x^-\}].\{y/a\}$. We have: $\mathcal{R}(s) = \mathcal{R}(disann(\mathcal{E}(\{y/x^-\})) \circ \{y/a\}) = \mathcal{R}(\{y/x\} \circ \{y/a\}) = \{\{x/a, y/a\}\}$, and we notice that indeed there exists a refutation for $\leftarrow p(x)$ giving the answer $\{x/a\}$. ◇

Now we consider again the example showed in the introduction (*deadlock* situation), which illustrates the necessity to use streams-like structures.

Example 4.14

1. Consider the program $\{p(a, b) \leftarrow |., \quad q(a, b) \leftarrow |.\}$, with declaration-mode $\{p(?, \hat{}), q(\hat{}, ?)\}$, and consider the goal $\leftarrow p(x, y), q(x, y)$. We have $< p(x, y), s_1 >, < q(x, y), s_2 > \in lfp(T)$, for $s_1 = [\{x^-/a\}].\{y/b\}$ and $s_2 = [\{y^-/b\}].\{x/a\}$. For all the possible interleavings $s \in s_1 \| s_2$, we get $\mathcal{R}(s) = \emptyset$. Indeed, no refutations are possible (*deadlock*).

2. Consider now the program $\{p(z,b) \leftarrow |r(z)., \ r(a) \leftarrow |., \ q(a,b) \leftarrow |.\}$, with the same declaration-mode for p and q, and with $r(?)$. We have
$< p(x,y), s_1 >, < q(x,y), s_2 > \in lfp(T)$, for $s_1 = [\{z/x^-\}].\{y/b\}.[\{z^-/a\}]$ and $s_2 = [\{y^-/b\}].\{x/a\}$. We have
$s = [\{z/x^-\}].\{y/b\}.[\{y^-/b\}].\{x/a\}.[\{z^-/a\}] \in s_1 \parallel s_2$ and $\{x/a, y/b, z/a\} \in \mathcal{R}(s)$. Indeed, there exists a refutation of the goal $\leftarrow p(x,y), q(x,y)$ giving the answer $\{x/a, y/b\}$. ◇

5 Denotational semantics.

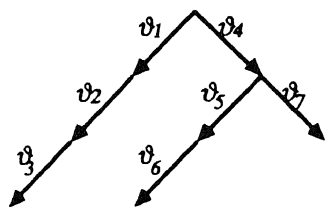
In this section, we give an overview of the definition of a denotational model for PARLOG. We refer to [6] for a more elaborate explanation and detailed formal definitions. Denotational semantics for flat versions of concurrent logic languages can be given with standard techniques. It is more difficult to assign a denotational semantics to concurrent logic languages with deep guards. Our denotational semantics does not deal with local deadlock in guards. Namely, we assume that if one of the guards has a successful computation then there will be no deadlock (cf the operational semantics).

To define our denotational model we first introduce two semantic universes ($p \in$) P and ($q \in$) Q; they are complete metric spaces satisfying the following two *reflexive domain* equations:

$$P \cong \mathcal{P}_{co}(Q) \text{ and } Q \cong \mathit{Subst} \cup (\mathit{Subst} \times Q) \cup (\mathit{Subst} \times P).$$

(Here $\mathcal{P}_{co}(Q)$ is the set of all *compact* subsets of Q; the symbol $\cong$ is interpreted "is isometric with".) In [8] and [2] it is described how to solve such equations; in [4] similar semantic universes are used. The elements p and q, called *processes*, are tree-like structures. The difference between $< \vartheta, p >$ and $< \vartheta, q >$ can be best explained by pointing out the different role played by the comma in both pairs: In $< \vartheta, p >$ it indicates an interleaving point whereas in $< \vartheta, q >$ it does not. Thus, $< \vartheta, q >$ represents an internal computation step, which will be used to model the computation of a guard. Formally, this difference appears explicitly in the definition of the semantic operator for interleaving (of processes) $\tilde{\|}$.

Example 5.1 A typical example of a process p would be $p = \{q_1, q_2\}$,
with $q_1 = \{< \vartheta_1, < \vartheta_2, \vartheta_3 >>\}$, $q_2 = \{< \vartheta_4, \bar{p} >\}$, and $\bar{p} = \{< \vartheta_5, \vartheta_6 >, \vartheta_7\}$, which can graphically be represented by



◇

In using reflexive domains with processes as elements, we follow the scheme that was used for imperative languages in [8]. Processes contain branching information, which here is used to distinguish between a failure possibility stemming from a failing guard computation and a failure resulting from the absence of a suitable candidate clause. (As it was already observed in the introduction, it is still to be investigated what kind of branching information we need in order to obtain a full abstract model.)

We introduce three semantic operators:

$\tilde{;}, \tilde{\|} : P \times P \to P$ and $str : P \to P$.

The operator $\tilde{;}$ for sequential composition is defined as usual. Namely

$$p_1 \tilde{;} p_2 = \{q \tilde{;} p_2 \mid q \in p_1\}$$

where

$$\vartheta \tilde{;} p_2 = < \vartheta, p_2 >, \quad < \vartheta, q > \tilde{;} p_2 = < \vartheta, q \tilde{;} p_2 > \quad \text{and} \quad < \vartheta, p > \tilde{;} p_2 = < \vartheta, p \tilde{;} p_2 > .$$

The operator $\tilde{\|}$ for interleaving is defined by

$$p_1 \tilde{\|} p_2 = \{q \mathbin{\underline{\|}} p_2 \mid q \in p_1\} \cup \{q \mathbin{\underline{\|}} p_1 \mid q \in p_2\}.$$

Here $\underline{\|}$ is the *left-merge* operator, which always starts with a step of the left component. It is defined by $\vartheta \mathbin{\underline{\|}} p = < \vartheta, p >$ and

$$< \vartheta, q > \mathbin{\underline{\|}} p = < \vartheta, q \mathbin{\underline{\|}} p >, \quad < \vartheta, \bar{p} > \mathbin{\underline{\|}} p = < \vartheta, \bar{p} \tilde{\|} p > .$$

Note that in the first case we stay in the "left-merge mode", whereas in the latter, where we have an interleaving point, the left-merge is changed into the normal merge again. We also observe that this definition is recursive; it can be justified by giving it as the unique fixed point of a suitable defined contraction. (The same applies to the recursive definitions of $\mathcal{D}$ and *yield*, below.) See [19] and [7] for many examples of this style of definition.

Finally, in $str(p)$ all the streams in p are collected; in other words, this operator removes all the interleaving points:

$$str(p) = \cup\{\widehat{str}(q) \mid q \in p\}$$

where

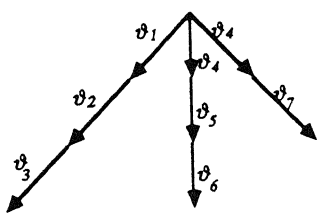
$$\widehat{str}(\vartheta) = \{\vartheta\}, \quad \widehat{str}(< \vartheta, q >) = \{< \vartheta, q' > : q' \in \widehat{str}(q)\} \quad \text{and}$$

$$\widehat{str}(< \vartheta, p >) = \{< \vartheta, q' > \mid q' \in \widehat{str}(p)\}.$$

Example 5.2 The streams of the process p given in example 5.1 are:

$$str(p) = \{< \vartheta_1, < \vartheta_2, \vartheta_3 >>, < \vartheta_4, < \vartheta_5, \vartheta_6 >>, < \vartheta_4, \vartheta_7 >\},$$

or, graphically,



◇

Definition 5.3 (Denotational semantics)
The function $\mathcal{D} : Goal \to P$ is given by the following three clauses:

1. $\mathcal{D}[\![\leftarrow true]\!] = \{\epsilon\}$,

2. $\mathcal{D}[\![\leftarrow A]\!] = \bigcup \{str(mgu_i(A^-, H)); \mathcal{D}[\![\leftarrow \bar{G}]\!]); mgu_o(A^-, H); \mathcal{D}[\![\leftarrow \bar{B}]\!] \mid H \leftarrow \bar{G} | \bar{B} \in W_{\mathcal{V}(A)}\}$,

3. $\mathcal{D}[\![\leftarrow \bar{A}, \bar{B}]\!] = \mathcal{D}[\![\leftarrow \bar{A}]\!] \| \mathcal{D}[\![\leftarrow \bar{B}]\!]$.

(these equations define a contraction, and $\mathcal{D}$ is defined as the unique fixpoint of this contraction.) ◇

In 2., the meaning of the resolution of an atom A is given as the union over all the clauses in the program W that have a head H that can be unified with A. This unification is given, similarly to the declarative semantics, in two parts: First the input variables are annotated (in A^-) and unified with H; next, after the guard evaluation, the output variables are unified. This ordering is important, since it will be used by the operator it yield, which is defined below with the help of the function $\mathcal{R}_{Var}$ (given in definition 4.8). Notice that this unification together with the computation of the guard is considered to be an atomic action, which is formally expressed by the use of the function *str*.

We conclude with the formulation of the correctness of $\mathcal{D}$ with respect $\mathcal{O}_2$. To this end, we introduce a function $yield : P \rightarrow (Subst \rightarrow M_2)$; it is given by

$$yield(p)(\vartheta) = \begin{array}{l} \bigcup_\delta \{\vartheta' \mid <\vartheta_1, <\vartheta_2, \cdots, <\vartheta_{n-1}, \vartheta_n > \cdots >> \in p\} \cup \\ \bigcup_\delta \{\vartheta'.yield(p')(\vartheta') \mid <\vartheta_1, <\vartheta_2, \cdots, <\vartheta_n, p' > \cdots >> \in p\}, \end{array}$$

where $\vartheta' \in \mathcal{R}_{Var}(\vartheta.\vartheta_1.\cdots\vartheta_n)$. (We have: $\bigcup_\delta = \bigcup X \setminus \{\delta\}$ if this is non-empty, and $\{\delta\}$, otherwise.) The correctness result now can be formulated as follows:

Theorem 5.4

$$\mathcal{O}_2[\![\leftarrow \bar{A}]\!] = \{(\vartheta_1.\cdots\vartheta_n.\cdots)_{|\mathcal{V}(\bar{A})} \mid \vartheta_1.\cdots\vartheta_n.\cdots \in yield(\mathcal{D}[\![\leftarrow \bar{A}]\!])(\epsilon)\}.$$ ◇

6 Conclusions and future work.

We have defined a declarative semantics that models the success set of PARLOG and a denotational semantics that models also the finite failures and the infinite computations. Similar approaches can be taken for GHC (see [5,25]) and for Concurrent Prolog.

If we compare the denotational semantics given here to the ones given in [4] and [5], we observe that this one is more abstract, i.e., it makes less distinctions. Moreover, this one is in some sense closer to the declarative model. In fact, the restrictions on unifications imposed by the mode declarations are formalized in the same way by the denotational and the declarative model.

Still, the denotational model is not fully abstract and the construction of such a model remains a topic for further research. Another topic to be investigated is the relation between the denotational and the declarative semantics. In this paper both models are related via their corresponding operational semantics, and no direct comparison is done.

References

[1] K.R. Apt. *Introduction to logic programming.* Technical Report CS-R8741, Centre for Mathematics and Computer Science, Amsterdam, 1987. To appear as a chapter in Handbook of Theoretical Computer Science, North-Holland.

[2] P. America and J.J.M.M. Rutten. *Solving reflexive domain equations in a category of complete metric spaces.* Proc. of the third workshop on mathematical foundations of programming language semantics, LNCS 298, 1988, pp. 254-288.

[3] L. Beckman, *Towards a Formal Semantics for Concurrent Logic Programming Languages,* Proc. of the Third International Conference on Logic Programming, LNCS 225, 1986, pp. 335-349.

[4] J.W. Bakker and J.N. Kok. *Uniform abstraction, atomicity and contractions in the comparative semantics of concurrent prolog.* Technical Report CS 88.., Centre for Mathematics and Computer Science, Amsterdam, 1988. Extended abstract in Proc. of FGCS 1988. To appear in TCS.

[5] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. *Control flow versus logic: a denotational and a declarative model for Guarded Horn Clauses.* Technical Report, Centre for Mathematics and Computer Science, 1988. To appear in Proc. MFCS 1989.

[6] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. *Semantics models for PARLOG.* Technical Report, Centre for Mathematics and Computer Science, 1988.

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

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

[9] K.L. Clark, S. Gregory, *Notes on the implementation of PARLOG,* Journal of Logic Programming 2(1), 1985, 17-42.

[10] K.L. Clark, S. Gregory, *PARLOG: Parallel programming in logic,* ACM Trans. Program. Lang. Syst. Vol. 8, 1, 1986, 1-49. Res. Report DOC 84/4, Dept. of Computing, Imperial College, London,1984.

[11] E. Eder. *Properties of substitutions and unifications.* Journal Symbolic Computation 1, 1985, pp. 31-46.

[12] M.H. van Emden and R.A. Kowalski. *The semantics of predicate logic as a programming language.* Journal of the ACM 23(4), 1976, pp. 733-742.

[13] M. Falaschi, G. Levi, *Finite Failures and Partial Computations in Concurrent Logic Languages,* Proc. of FGCS 1988, pp. 364-373.

[14] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. *Declarative modeling of the operational behaviour of logic languages.* To appear on TCS.

[15] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. *A new declarative semantics for logic languages.* In Proceedings Conf. and Symp. on Logic Programming, 1988, pp. 993-1005.

[16] R. Gerth, M. Codish, Y. Lichtenstein, and E. Shapiro. *Fully abstract denotational semantics for concurrent prolog.* In Proc. Logic In Computer Science, 1988, pp. 320-335.

[17] S. Gregory. *Parallel logic programming in PARLOG.* International Series in Logic Programming, Addison-Wesley, 1987.

[18] M. Hennessy and G.D. Plotkin. *Full abstraction for a simple parallel programming language.* In J. Becvar, editor, Proceedings 8th MFCS, Lecture Notes in Computer Science 74, Springer Verlag, 1979, pp. 108-120.

[19] J.N. Kok and J.J.M.M. Rutten. *Contractions in comparing concurrency semantics.* In Proceedings 15th ICALP, Tampere, LNCS 317, 1988, 317-332.

[20] J.-L. Lassez, M.J. Maher, and K. Marriot. *Unification revisited.* In J. Minker, editor, Foundations of deductive databases and logic programming, Morgan Kaufmann, Los Altos, 1988.

[21] G. Levi and C. Palamidessi. *The declarative semantics of logical read-only variables.* In Proc. Symp. on Logic Programming, IEEE Comp. Society Press, 1985, pp. 128- 137.

[22] G. Levi and C. Palamidessi. *An approach to the declarative semantics of synchronization in logic languages.* In Proc. 4th Int. Conf. on Logic Programming, 1987, 877-893.

[23] G. Levi. *Models, unfolding rules and fixed point semantics.* Proc. Conf. and Symp. on Logic Programming, 1988, pp. 1649-1665.

[24] M. Murakami. *A New Declarative Semantics of Parallel Logic Programs with Perpetual Processes.* Proc. of FGCS, 1988, pp. 374-381.

[25] C. Palamidessi. *A fixpoint semantics for Guarded Horn Clauses.* Technical Report CS-R8833, Centre for Mathematics and Computer Science, Amsterdam, 1988.

[26] V.A. Saraswat: *Partial Correctness Semantics for CP($\emptyset$, |, &).* Proc. of the Conf. on Foundations of Software Computing and Theoretical Computer Science, LNCS 206, 1985, 347-368.

[27] V.A. Saraswat: *The concurrent logic programming language CP: definition and operational semantics,* in: Conference Record of the Fourteenth Annual ACM Symp. on Principles of Programming Languages, Munich, 1987, pp. 49-62.

[28] V.A. Saraswat. *GHC: operational semantics, problems and relationship with* cp($\downarrow$, |). In IEEE international symposium on logic programming, 1987, pp. 347-358.

[29] E.Y. Shapiro. *A subset of concurrent prolog and its interpreter.* Tech. Report TR-003, ICOT, Tokyo, 1983.

[30] E.Y. Shapiro. *Concurrent Prolog: Collected Papers.* Vol. 1-2 MTI press, 1988.

[31] A. Takeuchi and K. Furukawa. *Parallel Logic Programming Languages.* Proc. Conf. on Logic Programming, LNCS 225, London, 1986, 242-254.

[32] K. Ueda. *Guarded Horn Clauses.* Technical Report TR-103, ICOT, Tokyo, 1985. Revised in 1986. A revised version is in Proc. Logic Programming 1985, pp. 168-179. Also in E.Y. Shapiro, editor, *Concurrent Prolog, Collected Papers.*, chapter 4.

[33] K. Ueda. *Guarded Horn Clauses, A Parallel Logic Programming Language with the Concept of a Guard.* Technical Report TR-208, ICOT, Tokyo, 1986. Revised in 1987. Also in Proc. Programming of Future Generation Computers, North Holland, 1988, pp. 441-456.