A formal specification of PARLOG *†

Egon Börger
Dip. di Informatica
C.so Italia 40
I-56100 PISA
boerger@di.unipi.it

Elvinia Riccobene
Dip. di Matematica
V.le Andrea Doria 6
I-95125 CATANIA
riccobene@mathct.cineca.it

Abstract

We provide a complete mathematical semantics for the parallel logic programming language PARLOG. This semantics is abstract but nevertheless simple and supports the intuitive operational understanding of programs. It is based on Gurevich's notion of *Evolving Algebras* ([20]) and is obtained adapting ideas from the description of full (Sequential) Standard PROLOG in [5] and the specification of imperative parallel computation phenomena of OCCAM developed in [24]. We develope a complete specification of the core of PARLOG which governs the computation of goals by user defined predicates. The built-in predicates can be described as for Standard PROLOG (see [4]-[6]) and are therefore omitted here. We give an explicit formalization of the two kinds of parallelism occurring in PARLOG: the AND-Parallelism and the (orthogonal) OR-Parallelism. Our description uses an abstract notion of PARLOG terms and PARLOG substitutions which is unburdened by representation details and implementation constraints.

*Part of this work was done when the first author was guest scientist at the Scientific Center of IBM Germany GmbH in Heidelberg, on sabbatical from University of Pisa, and when the second author from July 1990 till November 1990 worked at the Institut für Logik, Komplexität und Deduktionssysteme of University of Karlsruhe (Germany). The second author has been partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR, under Grant n.90.00671.69.

†In: M.Droste and Y.Gurevich (Eds.): *Semantics of Programming Languages and Model Theory*, Gordon and Breach, 1993, 1-42. Also: TR - 1/93, Dipartimento di Informatica, Universita di Pisa, pp.42.

1 Introduction

Gurevich's notion of *Evolving Algebras*, initially developed to provide operational semantics for programs and programming languages by improving on Church's thesis (see [21], [22]), has since been shown to apply as formal specification method for real programming languages like Modula2 [23], Smalltalk [3], Occam [24], Standard (Sequential) Prolog ([4], [5], [6], [12]), Prolog III [14], an object-oriented data base language [18]. In [13] and [7] it has allowed a systematic analysis of Prolog database views and their implementation. In [10] and [11] a series of *Evolving Algebras* extensions, starting from the Prolog algebras of [5] and proved to be correct w.r.t. the latter, has been developed which yields an entirely mathematical but transparent specification of Warren's Abstract Machine for executing Prolog. This specification together with the correctness proof has been extended to type-constraint logic programming for the case of Protos-L by Beierle & Börger in [2].

In this paper we provide a complete mathematical semantics, based on *Evolving Algebras*, for the parallel logic programming language Parlog. This seems to be the first attempt of a complete, formal (machine independent) specification of Parlog in the literature. In the conclusion we relate our result to other approaches.

We start from the *Evolving Algebras* description of (Standard) Prolog given by Börger (see [5]) and combine it with basic ideas developed by Gurevich and Moss for an *Evolving Algebras* specification of functional parallel computation phenomena of Occam (see [24]). We develope a complete specification of the core of Parlog which governs the computation of goals by user defined predicates. The built-in predicates occurring in Parlog (see [16]), can be treated in a similar way as already shown for (Standard) Prolog (see in particular [4], [5], [6]).

We give an explicit formalization of the two kinds of parallelism occurring in Parlog: the AND-Parallelism and the (orthogonal) OR-Parallelism. It turned out that the AND-Parallelism and the OR-Parallelism of Parlog can be specified almost independently one from the other. Both phenomena are described using an abstract notion of Parlog terms and Parlog substitutions which is unburdened by representation details and implementation constraints. This is similar to what has been done for the conjunctive and disjunctive components of the WAM in [10] and we believe that a natural extension of the term algebras developed in [11] for the WAM can be defined to obtain a Parlog - JAM from our present Parlog specification.

The paper is organized as follows:

- in Section 2 we adapt Gurevich's [20] definition of Evolving Algebras to our purposes;

- in Section 3 we define universes and functions of Parlog Algebras (extending Börger's [5] definitions for Sequential Prolog) and give the Transition Rules for AND-Parallelism in Parlog;

- in Section 4 we extend the previous Parlog Algebras (introducing new universes and functions and modifying the existing ones) to formalize OR-Parallelism in Parlog and give the Transition Rules for it;

- in the Appendix the complete Transition Rule System describing the semantics of Parlog is listed.

A preliminary version of this paper has appeared in [8], [9]. Indeed in [8] we have given a formalization of the AND-Parallelism of Parlog which uses an abstract notion of OR-Parallelism and therefore can be applied mutatis mutandis to other parallel logic programming languages like Concurrent Prolog or GHC. In [9] we have made this abstraction explicit by a system of rules for the OR-Parallelism, thus showing that it is orthogonal to the AND-Parallelism. We correct here the candidate clause search of [9] which turned out to be unnecessary complicated. (For details see sections 4.2.3 - 4.2.6). In [25] the correctness of this "implementation" is proved.

2 Evolving Algebras

To describe the semantics of a programming language from an operational point of view means to describe the way in which an ideal abstract machine executes the commands of the given language.

In algebraic operational semantics, an instantaneous configuration of a computing (abstract) machine is interpreted as a finite, many-sorted structure having finite sets, called *universes*, and partial *functions* defined on cartesian products of the *universes*. (Supposing that the set **Bool** of boolean values is always present as universe, we can represent predicates by their characteristic functions.)

Since a computation is a sequence of machine configurations, an abstract machine can be mathematically seen as an algebra whose *universes* and *functions* may change in time by application of finite number of *transition rules* which guide the evolution of the algebra from one state to another. The *transition rule* system describes the way in which the abstract machine executes the commands of the language.

The framework proposed by Y.Gurevich in [21] (see also [20]) and based

on the notion of **Evolving Algebras**, allows us to capture the intuition of the *dynamic* and *resource-bounded* aspects of computations. For motivation and complete definitions we refer the reader to [20] - [22].

Adapting [20] to our purposes, we define an **Evolving Algebra** as follows:
An **Evolving Algebra** *is a pair* (A, T) *consisting of a (finite), many-sorted, partial, first-order algebra* A *of some finite signature and a finite set* T *of transition rules of the same signature.*
A **computation** *of* A *is a (finite or infinite) sequence* $s_0, s_1, \cdots, s_k, s_{k+1}, \cdots$, *where* $s_0 = A$ *and each* s_{k+1} *is obtained from* s_k *by applying one or more* transition rules.

2.1 Transition Rules

A *transition rule* has the form:

If b(p) then F

where b(p) is a first order expression of the given signature of the algebra (sometimes called *guard*); only very simple expression for b(p) will be used, mostly boolean;
F is a set of **updates** $U_1(p), U_2(p), \ldots, U_m(p)$;
p is a set of variables ranging over some universes of the given algebra.

To execute a *transition rule* on a given algebra A means to perform simultaneously the $U_1(p)$, $U_2(p)$,..., $U_m(p)$ updates in A for each value of p such that b(p) is true in A. So doing the given algebra A changes to another algebra of the same signature.

Note that in writing down our rules, we often make use of the notational device "Let abbreviation long-expression" which allows to write a short "abbreviation" instead of the full "long-expression".

Remark: Following the given definition, a system of *transition rules* is non-deterministic because in a given moment more than one guard may be true (and the corresponding rule executed) in a given algebra. If a language is deterministic, the system of *transition rules* must be formulated so that the guards are mutually exclusive and the whole system turns out to be deterministic, too.

A Parlog computation is non-deterministic and so will be the Parlog system of transition rules. Moreover a Parlog computation may involve a concurrent computation of more than one processes. This implies that the transition from a current algebra A_{s_k} to the subsequent algebra $A_{s_{k+1}}$ will be realized

applying simultaneously more than one rule (one for each process which works in parallel with others). We will only use non-conflicting rules.

2.2 Updates

We have said that an algebra represents an instantaneous configuration (a *state*) of the (abstract) machine. Passing from one state to another, the algebra is updated according to the rules which encode the execution of the language commands by the machine.

We will use two kinds of **updates**:

a) function updates;

b) universe extensions.

Note. We will not use universe contraction rules which were considered in [21]. For Parlog their application would in most cases correspond to garbage collection, which need not be part of the formal description of the semantics of programming languages. This practice was already adopted in [4].

A **function update** has the form:

$$f(t_1, t_2, \cdots, t_n) := t$$

where f is a function symbol and $t_1, t_2, \cdots, t_n, t$ are terms in the signature of the language.

To execute a **function update** means to compute the terms $t_1, t_2, \cdots, t_n, t$ in the given algebra, say with result $a_1, a_2, \cdots, a_n, a$; then the result of the update's execution is to assign the new value a to the function f on the argument $(a_1, a_2, \cdots, a_n)$.

A **universe extension** update has the form:

Extend U by temp_1, temp_2,..., temp_t with
S
end Extend

where U is a universe of the algebra, t a term and S a set of function updates where temp_1, temp_2,..., temp_i,..., temp_t may occur. To apply this **universe extension** update on an algebra A means to compute t in A, say with resulting number n, to extend the universe U of A with new elements temp_1, temp_2,..., temp_n and to perform the sequence S of function updates for all values i between 1 and n simultaneously.

2.3 Auxiliary functions for Evolving Algebras

We suppose that each **Parlog Algebra** is equipped with some standard functions which we summarize here.

- Given a set $\mathbf{D}$, let $\mathbf{D}\Phi*$ be (a subset of) the set of sequences of elements of $\mathbf{D}$.

  For $f : D \to T$, we define $f\Phi* : D\Phi* \to T\Phi*$ such that

  $$f\Phi*(d_1, \cdots, d_n) = (f(d_1), \cdots, f(d_n)).$$

- The Parlog Algebras are equipped with the standard **list operations**. In particular we see a list as:

  $$List = [head|tail]$$

  where *head* is the first sequence element and *tail* the rest of the sequence. The function

  $$proj \ : \mathbf{Index} \times \mathbf{D}\Phi* \to \mathbf{D},$$

  is the *projection* function. **Index** is a set of indices equipped with a *successor* function

  $$succ \ : \mathbf{Index} \to \mathbf{Index}.$$

  On lists we assume the "integrity constraint"

  $$List = [proj(1, List), \cdots, proj(length(List), List)],$$

  where *length* is the function which yields the number of list components.

- We will use an [*op*]-*decomp* function defined as follows:
  if *op* is an associative term building operator (like ",", "&", ".", ";") and t a *term*,

  $$[op]\text{-}decomp(t)$$

  denotes the sequence $[t_1, \cdots, t_n]$ of immediate components of t w.r.t. *op*, i.e. such that

  $$t \equiv t_1 \ op \cdots op \ t_n \text{ and } \forall i : t_i \neq op(a, b).$$

  [*op*]-*decomp* comes with its inverse function [*op*]-*comp* which yields the term t built by the elements of the sequence $[t_1, \cdots, t_n]$ using the operator *op*.

- We introduce two other auxiliary functions for lists to simplify the notation of *transition rules*.

  If t is a term built by an operator op (like "," ,"&", ".", ";") having form $t_1\ op \cdots op\ t_n$, we define the following two functions:

$$[op]\text{-}head(t) = head([op]\text{-}decomp(t))$$

  which gives the first element t_1 of term t;

$$[op]\text{-}tail(t) = [op]\text{-}comp(tail([op]\text{-}decomp(t)))$$

  which yields the term t without its first element t_1.

3 AND-Parallelism

A Parlog program is a set of guarded clauses.

To run a program P means to give a query Q to the system and try to satisfy the goal(s) represented by Q, using facts and rules defined by P.

A query is a term built up from literals by "," and "&":

- "," is the parallel conjunction operator;
- "&" is the sequential conjunction operator.

To compute $g_1, g_2, \cdots, g_n$, means to compute g_1 and g_2 and ... and g_n in parallel.
To compute $g_1 \& g_2 \& \cdots \& g_n$, means to compute first g_1 and then g_2 and ... and finally g_n. In Parlog it is allowed to mix parallel with sequential conjunction operators. The parallel operator "," has higher binding priority than the sequential operator "&". For ex., the query $g_1, g_2 \& g_3$ has the effect of running g_1 concurrently with g_2 and, if and when both have terminated successfully, then running g_3. The higher precedence of "&" operator can be altered using parentheses, for ex. in $g_1, (g_2 \& g_3)$.

3.1 Universes and functions for AND-Parallelism

We imagine the computation of a query Q w.r.t. a Parlog program P as evolution (dynamic construction and traversal) of a "computation tree".

This "tree" is not a static structure defined by P; its form (the number of vertices and edges) and its labeling depend on the given query and its evaluation w.r.t. P.

We formalize this "tree structure", each incarnation of which represents the essentials of an instantaneous description of a Parlog computation, by introducing a set **Node** as basic universe of the Parlog Algebra.

Node is an evolving universe, because during the computation the "tree" grows by extending **Node** and the functions defined on it. (Note that in absence of discard rules **Node** will never shrink.)

3.1.1 Universes and functions adapted from Standard (Sequential) Prolog

Since a Parlog computation can be seen, in a certain way, as parallel computation of different Prolog computations, some of the universes and functions introduced in [5] to describe Standard (Sequential) Prolog can be used to define Parlog Algebras.

In this section we list those universes and functions which we have adapted to Parlog Algebras from Börger's Prolog Algebras.

Each subtree, having a *node*($\in$ **Node**) as root, performs a subcomputation which is associated to its root.

In order to describe this (sub)computation, we have, in analogy to the case of Sequential Standard Prolog, the following decorating functions:

a)

$$goal \ : \mathbf{Node} \rightarrow \mathbf{Goal}$$

associates with a *node* the *goal* to be evaluated during the subcomputation starting from *node*.
If **Lit** is the universe of all Parlog literals, then **Goal** is the set of all terms constructed applying a finite number of times the operators "," and "&" to elements of **Lit**.

In Parlog Algebras the *substitution* is treated by the following three functions:

b)

$$subres \ : \mathbf{Term}\Phi* \times \mathbf{Sub} \rightarrow \mathbf{Term}\Phi*,$$

where **Sub** is the universe of substitutions. *subres* yields the result of applying the given substitution to the terms in the given sequence.

On universe **Sub** of substitutions we introduce a 0-ary function *sub*, representing the current variable bindings.

This *global* substitution allows for Parlog's *stream-And-Parallelism* where calls can be evaluated concurrently, communicating incrementally through bindings to shared variables.

c) We also introduce a function *res* (for *result*) - taken from [24] -

$$res \ : \mathbf{Node} \rightarrow \{success, failure\}.$$

to report *success* or *failure* of the computation performed by $node \in \mathbf{Node}$.

d)

$$rename : \mathbf{Term} \times \mathbf{Node} \rightarrow \mathbf{Term}$$

is an external function (see [20]) which to a term associates a new copy where all variables are renamed at the level determined by the given node.

¿From the Standard (Sequential) Prolog description we also adopt the 0-ary function (a "constant" which functions as global variable)

$$database \in \mathbf{Clause}\Phi*$$

representing the current database. **Clause** is the universe of (user-defined) program *clauses*.

3.1.2 New Parlog universes and functions

On the universe **Node** we introduce some functions to realize the dynamic tree structure we are interested in.

$$children : \mathbf{Node} \rightarrow \mathbf{Node}\Phi*$$

yields the sequence of children associated with a node. A specific child of a node can be obtained using the projection function

$$child(i, p) = proj(i, children(p)).$$

By the integrity constraint assumed for lists, we have

$$children(\mathrm{p}) = [child(1, \mathrm{p}), \cdots, child(length(children(\mathrm{p})), \mathrm{p})].$$

When a node p has only one child, we will write $child(p)$ instead of $child(1, p)$. Sometimes we will also use the function *parent* to refer to a parent of a given node p having p as root. We use the expression $p = parent(q)$ as a short way to write $\exists i \in \mathbf{Index} : q = child(i, p)$. To keep control of the "tree" extension and traversal we take an idea from the description of OCCAM in [24] to introduce a set of *modes* each of which tells the current phase of the

computation associated with the given node.

Formally speaking we have a universe **Mode** and a function

$$mode : \mathbf{Node} \rightarrow \mathbf{Mode}.$$

For **Mode** we assume

$$\mathbf{Mode} = \{dormant, ready, starting, waiting, reporting.\}$$

A node has *mode*:

- *ready* when it is ready to receive the control;
- *starting* when it has received the control;
- *waiting* when it waits for an answer of (one of) its subcomputations;
- *reporting* when it reports to its parent node;
- *dormant* when it becomes inactive.

Each node of **Node** has also a label which can be :*and-par*, *and-seq* or *or-par*. The node's label is given in order to encode the type of the computation managed by the node. If a node is labelled by

- *and-par*, it coordinates the computation of concurrent (conjunctive) processes;
- *and-seq*, it performs the computation of sequential (conjunctive) processes;
- *or-par*, it controls the reduction process associated with the execution of a *call* in the given program.

Formally we have a universe

$$\mathbf{Tag} = \{and - par, and - seq, or - par\}$$

with the function

$$tag : \mathbf{Node} \rightarrow \mathbf{Tag}.$$

The following function

$$\tilde{tag} : \mathbf{Term} \rightarrow \mathbf{Tag},$$

associates a label of **Tag** with a term t according to the principal functor of t. It is defined as follows:

$$\tilde{tag}(t) := \begin{cases} \text{and-par} & \text{if} \quad t \equiv t_1, t_2, \cdots, t_n \\ \text{and-seq} & \text{if} \quad t \equiv t_1 \& t_2 \& \cdots \& t_n \\ \text{or-par} & \text{if} \quad t \in \mathbf{Lit} \end{cases}$$

Remark on notation When we write, for example, "*and-par* node p" we mean the expression "$tag(\mathrm{p}) = \textit{and-par}$ & $\mathrm{p} \in \mathbf{Node}$".

Instead of writing "$tag(\mathrm{p}) = \textit{and-par}$ & $mode(\mathrm{p}) = \textit{starting}$", we write "the *and-par* node p is *starting*".

In the same way we use the expression "*and-par* node p reports *success*" instead of "$tag(\mathrm{p}) = \textit{and-par}$ & $mode(\mathrm{p}) = \textit{reporting}$ & $res(\mathrm{p}) = \textit{success}$".

3.2 Transition Rules for AND-Parallelism

The general form of the *Transition Rule System* for a Parlog Algebra is as described in Section 2.1. We now give the definition of the single *Transition Rules*.

The variable p occurring in Parlog *Transition Rules* ranges over the universe **Node**.

In writing down these rules we will make use of simplifying notation, like abbreviating:

for all children s of p : $\mathsf{b}(\mathsf{s},\mathsf{p})$ for $\forall \mathsf{s}(\mathsf{p} = \mathsf{parent}(\mathsf{s}) \rightarrow \mathsf{b}(\mathsf{s},\mathsf{p}))$;

for some child s of p : $\mathsf{b}(\mathsf{s},\mathsf{p})$ for $\exists \mathsf{i} \in \mathbf{Index}(\mathsf{s} = \mathsf{child}(\mathsf{i},\mathsf{p}) \& \mathsf{b}(\mathsf{s},\mathsf{p}))$.

In **function updates** we write "for all children s of p : $\mathsf{f_o}(\mathsf{s}) = \mathsf{e_o} \& \ldots \& \mathsf{f_n}(\mathsf{s}) = \mathsf{e_n}$", instead of all **function updates** "$\mathsf{f_o}(\mathsf{s}) = \mathsf{e_o}, \ldots, \mathsf{f_n}(\mathsf{s}) = \mathsf{e_n}$" where s ranges over all the nodes s such that $\mathsf{p} = \mathsf{parent}(\mathsf{s})$.

3.2.1 Beginning and end of computations

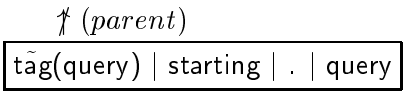
In our description of a Parlog computation started from a query w.r.t. a given program, we give no rule for the initialization of the "computational tree", but assume the following initialization of Parlog Algebras.

The universe **Node** contains a unique element p (the *root*) and the functions are initialized as follows:

database:= "program",
goal(p):= "query",
sub:= empty,
tag(p):= $\tilde{\mathsf{tag}}$("query"),

mode(p):= starting.

On the *root* of the "tree" the *parent* function is undefined. (We may also consider $res(p)$ as undefined in the initial algebra.) We may represent this pictorially as follows (leaving out database because it is never changed during a Parlog computation):



When the whole computation has terminated the control comes back to the *root*.

Based on our assumption that the *root* is uniquely determined by having no parent, we can formalize the **stop rule** as follows:

```
If mode(p) = reporting & parent(p) = undef
then
      mode(p):= dormant.
```

This **stop rule** only stops the system without giving output. The latter could be provided by some **output rule** (which we do not formalize here) using the substitution information coded in *sub*.

3.2.2 The *and-par* node operation

When an *and-par* node receives the control (i.e. when its mode is *starting*), it creates as many children as there are computations that should be executed in parallel. The $[,]$-*decomp* function on the goal g associated with the node, provides those new parallel processes $g_i = proj(i, [,]\text{-}decomp(g)) \quad for\ i = 1, 2, \cdots, n$. The tag of children nodes is determined by $\tilde{tag}$ on term g_i. The *and-par* node becomes *waiting* and the control passes to each of its children which become *ready*. We therefore have the following:

and-par starting rule

```
If tag(p) = and-par & mode(p) = starting
then
      Let parlist [,]-decomp(goal(p))
      Create p-subtree of children temp(1),..., temp(length(parlist))
      with mode ready and
            tag(temp(i)):= tag̃(proj(i,parlist))
            goal(temp(i)):= proj(i,parlist)
      end Create
```

where

```
   Create p-subtree of child{ren} temp(1),..., temp(l)
   { with tag t, mode m, } { passing g ... and }
         updates
   end Create
```

is an abbreviation for the following update:

```
      Extend Node by temp(1),..., temp(l) with
            child(i,p):= temp(i),
            {tag(temp(i)):= t,
            mode(temp(i)):= m,}
            {g(temp(i)):= g(p),
             ⋮}
            updates
      end Extend,
      mode(p):= waiting.
```

Typically we will use goal for g; updates stands for a set of function updates; updates written between { } are optional. Note that - in order to avoid the necessity to distinguish the case of a root labelled with *and-par* but with empty query - we understand the preceding rule as doing nothing in case l=0.

Remark. In our formalization of *and-par* nodes we have to depart from the fully parallel point of view taken in Gurevich and Moss (see [24]) for Occam PAR nodes in one major respect: we do not describe how the children of an *and-par* node become *starting*. How such a node becomes *starting* depends on the implementation. In a fully parallel system all children would have *starting* mode because all of them must work simultaneously. This would correspond to identify *ready* with *starting*, or respectively to add the rule

```
if mode(p) = ready then mode(p):= starting
```

If our machine has fewer processors than (parallel) processes, then one or more processors must work on more than one process. In this case we have not a parallel run, but concurrent runs which are realized by time sharing or (memory) interleaving.
We have adopted this point of view in order to point out to the reader that different implementations of the system may differ in this respect, and that a full description has to take a definite decision with respect to the crucial phenomenon we have isolated here.

Each child which has been created by the above rule computes one element of the conjunction. A conjunction succeeds if each of its calls succeeds, and fails if one of its calls fails. Therefore an *and-par* node gives back the control either when all of its children have finished their subcomputation with success - the *and-par* node reports *success* - or when one of them has failed its subcomputation - the *and-par* node reports *failure* to the parent and all siblings' computations of the reporting child are aborted (i.e. set to *dormant*) -. These two cases are formally described by the following two rules:

and-par success rule

If tag(p) = and-par & mode(p) = waiting
& for all children q of p : mode(q) = reporting & res(q) = success
then
report from p-*subtree with* success.

and-par failure rule

If tag(p) = and-par & mode(p) = waiting
& for some child q of p: mode(q) = reporting & res(q) = failure
then
report from p-*subtree with* failure.

The abbreviation *"report from* p-*subtree (or -leaf) with* success *(or* failure*)"* stands for the following three function updates:
mode(p):= reporting,
res(p):= success *(or* failure*)*,
for each child s of p: mode(s):= dormant.

The latter update is not there if p is a leaf of the "tree". We include the update for later use for reasons of uniformity.

Once a node has become *dormant*, it will never be used any more in our algebra. This means that the whole "subtree" of a *dormant* node is marked

for **garbage collection**.

To abort all children's subcomputations of a *reporting* node p, it is not (always) sufficient to change the mode of p's children to *dormant* because a descendant node s of p may remain active when its parent subcomputation has been killed. It can be proved however (see [25]) that these possibly still active sub-sub-computations could not affect the substitution *sub*. How the computation of a child node of a *dormant* node is aborted depends on the implementation.

Note also that the *and-par success rule* does not update the associated goals. This is because *and-par* nodes can appear only at the root (if the initial query has form $t_1, \cdots, t_n$) or under an *and-seq* node (which will update its restgoalsequence discarding the goal of its *and-par* child).

3.2.3 The *and-seq* node operation

When a node p labeled *and-seq* has mode *starting*, *goal(*p*)* contains a term of the form $a_1 \& a_2 \cdots \& a_n$, where

$$a_i = \begin{cases} b_{i_1}, b_{i_2}, \cdots, b_{i_s} \\ \text{lit} \end{cases}$$

are goals to be executed in the indicated order, starting with a_1.

In *starting* mode the *and-seq* node creates a child having label according to $\tilde{tag}$ function on [&]-*head*(*goal*(p)) :
- *or-par* if $a_1 \in$ **Lit**;
- *and-par* if $a_1 = b_{1_1}, b_{1_2}, \cdots, b_{1_s}$.

To its child the *and-par* node passes the first element of the sequence [&]-*decomp*(*goal*(p)) as value of the function *goal*. Thus we have the following

and-seq starting rule

```
If tag(p) = and-seq & mode(p) = starting & goal ≠ nil
then
      Let goal [&]-head(goal(p))
      Create p-subtree of child temp
      with tag t̃ag(goal), mode starting and
            goal(temp):= goal
      end Create
```

When its child (*or-par* or *and-par*) is reporting, the *and-seq* node gets

the control again. If the child reports *failure*, then p reports *failure*, since the computation of $a_1 \& a_2 \cdots \& a_n$ fails.

and-seq failure rule

If tag(p) = and-seq & mode(p) = waiting
& mode(child(p)) = reporting & res(child(p)) = failure
then
report from p-*subtree with* failure.

If the child reports *success*, the computation of a_1 has been evaluated with success.

The following step will be the computation of $a_2 \& \cdots \& a_n$ under a certain substitution θ satisfying a_1.

The *goal* function is updated to the value $a_2 \& \cdots \& a_n$, whereas the substitution θ has been included in *sub* (see below).

and-seq continuation rule

If tag(p) = and-seq & mode(p) = waiting
& mode(child(p)) = reporting & res(child(p))=success
then
mode(p):= starting,
goal(p):= [&]-tail(goal(p)),
mode(child(p)):= dormant.

If the whole sequential conjunction $a_1 \& a_2 \cdots \& a_n$ has been run successfully eventually we will have *goal*(p) = *nil* (*tag*(p) = *and-seq*). Then p becomes *reporting* and it reports *success*.

and-seq success rule

If tag(p) = and-seq & mode(p) = starting & goal(p) = nil
then
report from p-*subtree with* success.

4 OR-Parallelism

A Parlog program consists of a set of procedures defining Parlog relations. If $R(t_1, t_2, \ldots, t_n)$ is a relation, its definition (procedure) consists of the following two elements:

1) a "mode declaration" of the form

$$mode\, R(m_1, m_2, \ldots, m_n)$$

where m_i is either "?" (to indicate that this argument is *input*) or "^" (to indicate that this argument is *output*);

2) a "sequence of clauses". The clauses are defined using "." (parallel search) and ";" (sequential search) operators in a particular way explained below.

When the Parlog computation system begins the reduction process of a given procedure call *lit* (of form $R(t_1, t_2, \ldots, t_n)$), it comes into the so-called **test-commit-output-spawn** phase.

The system first tries to find a clause which satisfies the **candidate clause** condition among those that define the procedure of *lit*.

For any call *lit*, a clause "$head \leftarrow guard : body$" is a *candidate clause* if:

- it is an unguarded clause for which **input matching** [1] of *lit* and *head* succeeds,
  or
- it is a guarded clause for which
  a) **input matching** of *lit* and *head* succeeds with a substitution s;
  b) **guard evaluation** (i.e. the computation of the clause guard by the program) succeeds with a substitution s';
  c) the substitutions s and s' are consistent.

When a *candidate clause* is found, the calling literal *lit* commits to it (**commit** phase) interrupting the search for (other) candidate clauses - for reasons which concern the substitution handling, the real implementation is slightly different, see section 4.2.4 -, output unification is performed between the output mode argument terms of *lit* and those of the selected clause (**output** phase); the *lit* computation is reduced to the evaluation of the *candidate clause*'s body under the computed output substitution (**spawn** phase).

[1] **Input matching** (of a literal *lit* and a term *t*) is defined as unification of *lit* and *t* in which no variable is bound which occurs in an *input argument* of *lit* (in the given program).
In contrast to input matching one speaks of **output unification** to refer to a unification of two terms which appear in an *output mode argument* (of a literal w.r.t. a given program). This façon de parler stresses that for the unification of terms occurring in *output mode arguments* there is no restriction on the direction of the bindings (from goal to clause head (*output*) or viceversa (*input*)).

4.1 New universes and functions of Parlog Algebras for OR-Parallelism

To formalize in terms of *evolving algebras* the structure of a Parlog program we define a universe **Program** whose elements are Parlog *programs* which are sets of *procedures* taken from a universe **Procedure** = **Decl** ×**Seqbloc** with

- **Decl** the universe of relation mode declarations;
- **Seqbloc** defined as set of sequences, called *seqblocs*, of the form:

  $$S_1; S_2; \cdots; S_n$$

  where the S_i are going to be executed in sequential order and are themselves sequences of form

  $$C_1.C_2.\ldots.C_m$$

  with clauses $C_i \in$ **Clause**. The latter sequences are also called *par-clauses* (clauses to be executed in parallel) and are elements of the **Parclause** universe.

  Therefore an element of **Seqbloc** has the form

  $$\underbrace{C_{1_1}.C_{1_2}.\ldots.C_{1_q}}_{S_1}; \cdots\cdots; \underbrace{C_{n_1}.C_{n_2}.\ldots.C_{n_p}}_{S_n}$$

  with $C_{i_j} \in$ **Clause**; the operator "." has higher binding power than ";".

On the new universes we define the functions:

a)

$$decl : \mathbf{Lit} \times \mathbf{Program} \rightarrow \mathbf{Decl},$$

associates with a literal of a Parlog program its mode declaration in the given program.

b)

$$in\_var : \mathbf{Term} \times \mathbf{Program} \rightarrow \wp(\mathbf{Var}),$$

assigns to a term t the set of all variables occurring in *input argument positions* of its mode declaration (in the given program). **Var** is the universe of variables and $\wp(\mathbf{Var})$ is its powerset.

From Sequential (Standard) Prolog Algebras (see [5]), we take the following functions adapting their definition to Parlog Algebras:

c)

$$unify \ : \mathbf{Term} \times \mathbf{Term} \rightarrow \mathbf{Sub} \cup nil,$$

where **Term** is the set of (Parlog) *terms*. We suppose $\mathbf{Goal} \subseteq \mathbf{Term}$. This function assigns to two terms either a substitution (the *unifier* and possibly the *most general unifier*) if they are unifiable or *nil* if they are not. *nil* is different from the empty substitution.

d)

$$procdef : \mathbf{Program} \times \mathbf{Lit} \rightarrow \mathbf{Seqbloc},$$

yields a copy of the procedure in the given program which defines the predicate having the same functor as $lit \in \mathbf{Lit}$.

e) In connection with the universe **Clause** we need three standard auxiliary functions whose names suggest their meaning:

$$clhead \ : \mathbf{Clause} \rightarrow \mathbf{Lit},$$

$$clguard \ : \mathbf{Clause} \rightarrow \mathbf{Goal},$$

$$clbody \ : \mathbf{Clause} \rightarrow \mathbf{Goal}.$$

For an explicit description of the *candidate clause* search and the subsequent commitment to it, we imagine each *or-par* node as root of a computation subtree performing the **test-commit-output** phase and of another (later created) subtree to perform the **spawn** phase.

The procedure defining the calling literal *lit* comes in the form of

$$procdef(database, lit) = \underbrace{C_{1_1}.C_{1_2}.\ldots.C_{1_{m_1}}}_{S_1}; \cdots\cdots; \underbrace{C_{n_1}.C_{n_2}.\ldots.C_{n_{m_n}}}_{S_n.}$$

First we search through the clauses of the first block S_1; among the S_1-clauses the search is in parallel.

If no *candidate clause* is found in the block S_1, the clauses of the second block S_2 are tried, in parallel. If the search fails again the subsequent blocks are considered in the same way. For each clause we have to perform in parallel **input matching** and **guard evaluation**.

If there is no clause which satisfies the *candidate clause* condition, it is impossible to reduce the calling literal *lit* and its computation fails. For a particular clause the search process may also be suspended (see below).

This is illustrated by fig.1.

To formalize this search subtree we extend the previous **Mode** and **Tag** Parlog universes as follows:

a) We extend the **Tag** universe with *seq-search, or- search, try-clause, input-match, guard-eval* elements.
A node has *tag*:

- *seq-search* if it coordinates the *candidate clause* search among blocks (of clauses) which have to be tried sequentially (the first level S_1, S_2, ... in fig.1);
- *or-search* if the node performs the disjunctive search of a *candidate clause* through a block of clauses that are tried in parallel (the second level of C_{i_j} for fixed i in fig.1);
- *try-clause* when its subtree tests if the selected clause is or not a *candidate clause* for the given literal *lit* (the third level C_1, C_2, ... in fig.1);
- *input-match* when its subcomputation controls the **input matching** condition (leaf level in fig.1);
- *guard-eval* if the node performs the **guard evaluation** condition (leaf level in fig.1).

b) We extend the **Mode** universe with an element *suspend*.
A node has mode *suspend* when there is a unifier for the calling literal and the clause head which however tries to bind a variable occurring (in a term) in an *input* argument position of the call.

We need the following new functions on the extended universe **Node**:

1.

$$parcl : \mathbf{Node} \rightarrow \mathbf{Parclause}$$

associates with a *node* a sequence of clauses (for parallel search of a *candidate clause*);

2.

$$seqbloc : \mathbf{Node} \rightarrow \mathbf{Parclause}\Phi*$$

associates with a *node* a sequence of "*parclauses*" (for sequential search of a *candidate clause* through subsequent blocks of clauses);

3.

$$clause : \mathbf{Node} \rightarrow \mathbf{Clause}$$

associates a *clause* with a *node* (the one which will be considered for the *candidate clause* test).

In the *or-par* node subtree description, the function *goal* will be used to pass from parent to child the calling literal *lit* which is responsible for the *candidate clause* search. The same function will be used to report from child to parent the *body* of the identified *candidate clause* for the calling literal.

In connection with the input matching during the *candidate clause* search, Parlog also uses a clause transformation which shifts the unification of certain arguments - so called *output arguments* - from the input matching phase to the clause body computation, introducing fresh variables. Formally we represent this by a function

$$out - unif - shift : \mathbf{Clause} \times \mathbf{Node} \rightarrow \mathbf{Clause}$$

which associates with a given clause $Head \leftarrow Guard : Body$ and a node p the clause

$$Head[\overline{t}/\overline{y}] \leftarrow Guard : \overline{t} = \overline{y}, Body$$

where $\overline{t}$ is the sequence of output arguments of $Head$, $\overline{y}$ is a corresponding sequence of pairwise distinct variables renamed at level p, $Head[\overline{t}/\overline{y}]$ denotes the result of substituting the output arguments from $\overline{t}$ by the corresponding variables from $\overline{y}$, and $\overline{t} = \overline{y}$ denotes the sequence of unifications of $\overline{y}$-variables with the corresponding $\overline{t}$-arguments.

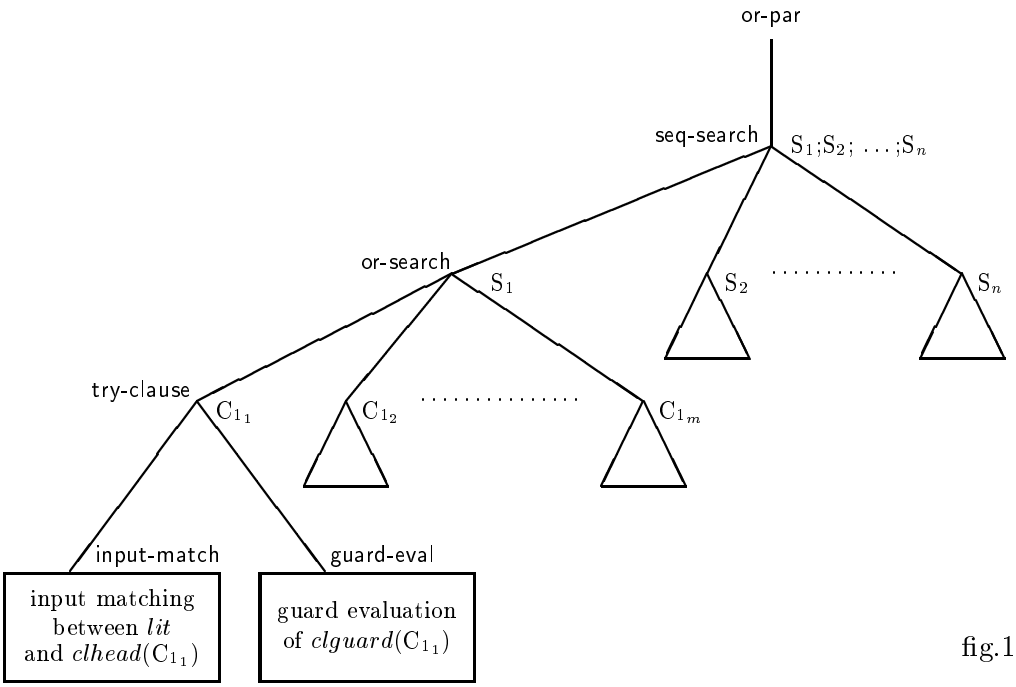


fig.1

4.2 Transition Rules for OR-Parallelism

4.2.1 The *or-par* node operation

When the *or-par* node has mode *starting*, its goal *lit* has to be reduced by starting the *candidate clause* search through its $procdef(database, lit)$, followed later by the computation of the selected body (if any).

For this purpose the control is first passed to a newly created child labeled *seq-search* whose subtree manages the search of a *candidate clause* through the blocks $S_1, S_2, \cdots, S_n$ of $procdef(database, varindex, lit)$ that are tried in sequence.

Remember that we are specifying the computation of user-defined predicates, which occur as our goals.

or-par starting rule

If tag(p) = or-par & mode(p) = starting
then
Create p-*subtree of child* temp
with tag seq-search, *mode* starting, *passing* goal
end Create

The *or-par* node is *waiting* until the child *seq-search* becomes *reporting*.

If the child *seq-search* is *reporting* with *failure*, no *candidate clause* has been found and the *or-par* node becomes *reporting* with *failure*:

or-par failure rule

If tag(p) = or-par & mode(p) = waiting
& mode(child(p)) = reporting & res(child(p)) = failure
then
report from p-*subtree with* failure.

If the child *seq-search* is *reporting* with *success*, a *candidate clause* has been found and the system has committed to it (by effecting the corresponding output substitution). Therefore the body of this *candidate clause* is reported by the *goal* function of the *seq-search* node.

The subsequent step, triggered by the following **or-par spawn rule**, is the computation of the selected (really extended, see section 4.2.4) clause body: a new node of the *or-par* child is created to perform the body computation. The *seq-search* node becomes *dormant*.

or-par spawn rule

If tag(p) = or-par & mode(p) = waiting & tag(child(p)) = seq-search
& mode(child(p)) = reporting & res(child(p)) = success
then
Create p-*subtree of child* temp
with tag $\tilde{\text{tag}}$(goal(child(p))), *mode* starting *and*
goal(temp):= goal(child(p)),
mode(child(p)):= dormant
end Create

If the body computation ends with *failure* the previous *or-par failure rule* (for the new child) is performed and the *or-par* node becomes *reporting* with *failure*.

Instead, if the body computation ends with *success*, the *or-par* node becomes *reporting* with *success*.

or-par success rule

If tag(p) = or-par & mode(p) = waiting & tag(child(p)) $\neq$ seq-search
& mode(child(p)) = reporting & res(child(p)) = success
then
report from p-*subtree with* success

4.2.2 The *seq-search* node operation

The *seq-search* node rules are similar to those of our *and-seq* nodes and to the description of SEQ node for OCCAM [24].

A node labeled *seq-search* with goal *lit* manages the search of a *candidate clause* through the ";"-sequence of blocks of clauses that form the procedure defining *lit*, i.e. of *procdef(database,lit)*.

When the node *seq-search* is *starting* and $procdef(database, lit) = nil$, then *failure* is reported:

seq-search failure rule 1

If tag(p) = seq-search & mode(p) = starting
& procdef(database,goal(p)) = nil
then
report from p-*subtree with* failure.

When the node *seq-search* is *starting* and $procdef(database, lit) \neq nil$, we create a child labelled *or-search*. By the function *parcl* the first *parclause* of those defining the procedure of *lit* (which have to be tried in sequence searching a *candidate clause*) is passed to the child *or-search* together with goal *lit* and *seqbloc* at the node *seq-search* is updated to the remaining *parclauses*. (Note that the necessary renaming of clauses will be done only at *or-search* nodes where individual clauses are set for being tried).

seq-search starting rule

If tag(p) = seq-search & mode(p) = starting
& procdef(database,goal(p)) ≠ nil
then
Create p-*subtree of children* temp
with tag or-search, *mode* starting, *passing* goal *and*,
parcl(temp):= [;]-head(procdef(database,goal(p)))
seqbloc(p):= [;]-tail(procdef(database,goal(p)))
end Create

If the search performed by the child labeled *or-search* fails, a new child *or-search* is created with the same function for the next *parclause* in *seqbloc*; the previous node *or-search* becomes *dormant*:

seq-search continuation rule

If tag(p) = seq-search & mode(p) = waiting
& p = parent(q)
& mode(q) = reporting & res(q) = failure
& seqbloc(p) ≠ nil
then
Create p-*subtree of children* temp
with tag or-search, *mode* starting, *passing* goal *and*,
parcl(temp):= [;]-head(seqbloc(p))
seqbloc(p):= [;]-tail(seqbloc(p))
end Create,
mode(q):= dormant

If all *parclauses* have been tried and no *candidate clause* has been found, the *seq-search* node becomes *reporting* and reports *failure*:

seq-search failure rule 2

If tag(p) = seq-search & mode(p) = waiting
& p = parent(q)
& mode(q) = reporting & res(q) = failure
& seqbloc(p):= nil
then
    mode(p):= reporting,
    res(p):= failure,
    mode(q):= dormant.

If the subcomputation of one *or-search* child node ends with *success* (namely a *candidate clause* has been found and - through committing to it - the *sub* substitution has been extended by the **output substitution**), the node *seq-search* becomes *reporting* and reports to its parent the body of the *candidate clause* by the *goal* function, thus preparing the start of the spawn phase. All children *or-search* become *dormant*:

seq-search success rule

If tag(p) = seq-search & mode(p) = waiting
& p = parent(q)
& mode(q) = reporting & res(q) = success
then
    report from p-*subtree with* success
        saving goal *from child* q.

The abbreviation
"report from p-*subtree with* success *(or* failure*)*
{*saving f,g . . . from child q ,*
updates }"
stands for the following sequence of function updates:

    mode(p):= reporting,
    res(p):= success *(or* failure*)*,
    for each child s of p: mode(s):= dormant,
    {f(p):= f(q),
     g(p):= g(q),
     ⋮
     updates }

It is an extension of the previously introduced *"reporting"* abbreviation.

4.2.3 The *or-search* node operation

A node *or-search* performs the parallel search of a *candidate clause* through a *parclause*, say $C_1.C_2.\ldots.C_m$.

When a node labelled *or-search* becomes *starting*, as many *or-search*'s children labelled *try-clause* are created as there are clauses in the given *parclause*. (The number of these clauses is computed by the function *length* applied to [.]-*decomp* evaluated on *parcl(p)*.)

Each child *try-clause* receives the calling literal and one clause (renamed corresponding to the node to ensure that all variables are fresh with respect to the whole computation) of the parent's *parclause*. All children get mode *ready*:

or-search starting rule

If tag(p) = or-search & mode(p) = starting
then
    Let parlist [.]-decomp(parcl(p)),
    Create p-*subtree of children* temp(1),..., temp(length(parlist))
    with tag try-clause, *mode* ready, *passing* goal *and*
        clause(temp(i)):= rename(proj(i,parlist),temp(i))
    end Create

When at least one (but maybe more than one) child *try-clause* ends its subcomputation with *success*, one computed *candidate body* (really its transformed version, see section 4.2.4) is selected by the waiting *or-search* parent all of whose children's computations are aborted (by changing their mode into *dormant*); the *or-search* node becomes *reporting* thus realizing the commit phase of the current call. (As we will see in section 4.2.4, during the spawn phase, still part of the substitution consistency from input matching and guard evaluation is realized. Substitution inconsistency will lead to failure).

or-search success rule

Let succ-resp(q) (parent(q) = p & mode(q) = reporting & res(q) = success)
If tag(p) = or-search & mode(p) = waiting & for some q succ-resp(q)
then
    report from p-*subtree with* success
        saving goal *from child* ε q succ-resp(q)

where Hilbert's ε-operator is used as (external) choice function. This reflects the decision taken by the Parlog design to consider the *or-search* as

non deterministic (implementation defined) language feature.

If all *try-clause* children end their subcomputation with *failure*, no *candidate clause* is in the initially given *parclause*. The node *or-search* changes its mode to *reporting* and reports *failure*:

or-search failure rule

If tag(p) = or-search & mode(p) = waiting
& for all children q of p : mode(q) = reporting & res(q) = failure
then
report from p-*subtree with* failure.

4.2.4 The *try-clause* node operation

The subcomputation of a node p labelled *try-clause* consists of verifying whether the associated clause *clause(p)* is a *candidate clause* for the calling literal *goal(p)* or not.

To be *candidate* the clause must satisfy the **input matching** and the **guard evaluation** conditions. These are tried in parallel, controlled by two newly created p-children labelled *input-match* resp. *guard-eval*. The child *input-match* receives from the parent, via *goal*, the literal to be considered, whereas the child *guard-eval* receives the guard of the clause. This explains the **try-clause starting rule**, except for its last update which transforms the clause associated to p. This clause transformation presents a subtlety of Parlog implementation by which the need for multiple environments for "local" substitutions is avoided.

Since it has a semantical consequence we have to take it into consideration here. In [9] we had given a formalization which was inspired by some features in the JAM [17], creating "local" substitutions which became transparent to the main system only after commitment. Our input matching operation there unifies the entire goal with the clause head, and the consistency of the guard evaluation is checked with the output arguments bindings before commitment. In reality in Parlog the input matching operation should unify only the input argument bindings postponing the unification of the output arguments, and the consistency of the guard evaluation bindings with the output argument bindings should be checked by output unification only after commitment.

Let us look at the problem in more detail. Since in Parlog input matching and guard evaluation are performed in parallel, a variable occurring both in the head and in the guard of the same clause might be bound to not unifyable terms by the two computations (in this case the clause should be

consedered to be rejected. As a matter of fact it will produce failure, should the system commit to it. See below.). Since we assume that guards are *safe* (i.e. the guard evaluation never attempts to bind a variable that occurs in an input mode argument position of the clause head), the problem of possible incompatible bindings can come up only for those variables of the guard which occur in output mode argument positions of the head. The problem is solved by postponing unification of the output arguments, shifting it from the input matching phase to the moment of the computation of the body (whereby the compatibility with the bindings of the guard evaluation is also checked.) Formally this shift of unification of output arguments is realized by the clause transforming function *out-unif-shift* (applied in the following rule to the newly created node *temp*(1) as second argument to ensure that the dummy variables $\overline{y}$, introduced for the output arguments $\overline{t}$, are new for the whole computation).

try-clause starting rule

If tag(p) = try-clause & mode(p) = starting
then
    Create p-*subtree of children* temp(1),temp(2)
    with mode ready *and*
        tag(temp(1)):= input-match,
        tag(temp(2)):= guard-eval,
        goal(temp(1)):= goal(p),
        goal(temp(2)):= clguard(clause(p)),
        clause(p):= out-unif-shift(clause(p),temp(1))
    end Create

If one of the two children fails its subcomputation, the *try-clause* node changes its mode into *reporting* and reports *failure* (because one of the two conditions, **input matching** or **guard evaluation**, has failed):

try-clause failure rule

If tag(p) = try-clause & mode(p) = waiting
& for some child q of p: mode(q) = reporting & res(q) = failure
then
    report from p-*subtree with* failure.

If both children report *success*, the *try-clause* node ends its subcomputation with *success* and the computed clause is *candidate* for the calling literal.

Thus the node *try-clause* becomes *reporting* and it reports, via the *goal*

function, the body of the *candidate clause* by which the calling literal will be replaced (if this clause will be selected in the *or-search success rule*). Remember that at this moment, the clause attached to the node *try-clause* is the result of the *out-unif-shift* application to the original clause (when entering *try-clause*).

try-clause success rule

If tag(p) = try-clause & mode(p) = waiting
& for all children s of p :
mode(s) = reporting & res(s) = success
then
report from p-*subtree with* success,
goal(p):= clbody(clause(p)).

4.2.5 The *input-matching* node operation

A node labelled *input-match* is a leaf node of the Parlog computation "tree".

When the control arrives to an *input-match* node, we try to compute **input matching** between the head of the given clause (*clhead(clause(parent(p)))*) and the calling literal under the current substitution (*subres(goal(p),sub)*).

If there is no unification at all - formally if

$$unify(subres(goal(p)sub), clhead(clause(parent(p)))) = nil\ -,$$

then surely the node *input-match* becomes *reporting* with *failure*:

input-match failure rule

If tag(p) = input-match & mode(p) = starting
& unify(subres(goal(p),sub),clhead(clause(parent(p)))) = nil
then
report from p-*leaf with* failure.

If there is some unifier, then we need to check that it satisfies the *input matching* condition, i.e. that it does not bind variables occurring in *input argument* positions of the calling literal *lit* (w.r.t. the underlying program *db*).

In order to check this condition we use a function:

$$domain : \mathbf{Sub} \rightarrow \wp(\mathbf{Var})$$

which associates with a substitution $\sigma \in \mathbf{Sub}$ the set of variables bound by σ.

A unifier *unif* satisfies the *input matching* condition for *lit*, if the set of variables bound by *unif* has empty intersection with the set of variables occurring in input positions of *lit* w.r.t. the underlying program *db*; formally if

$$domain(unif) \cap in\_var(lit, db) = \emptyset.$$

We have to apply this for

$$lit = subres(goal(p), sub),$$

$$head = clhead(clause(parent(p)),$$

$$unif = unify(lit, head).$$

If the intersection is empty, the node *input-match* becomes *reporting* with *success* and the current substitution *sub* is updated by the unifier substitution. We note that the following transition rule is performed if the guard condition $allowed(p) = 1$ is true. *Allowed* is an external function (see [20]) having value 1 on at most one node p at any moment. The role of this function is to avoid that the current value of *sub* could be simultaneously updated by more than one node p performing parallel processes. Using this function we simulate Parlog's approach to the commit/output access to shared information by parallel processes.

input-match success rule

Let lit subres(goal(p),sub)
    head clhead(clause(parent(p)))
If tag(p) = input-matching & mode(p) = starting
& unify(lit,head) $\neq$ nil
& domain(unify(lit,head)) $\cap$ in_var(lit,db)) $= \emptyset$
& allowed(p) = 1
then
    report from p-*leaf with* success,
    sub:= unify(lit,head)

If the intersection is not empty, the unifier exists but it tries to bind an input variable.

In this case the node *input-match* changes its mode into *suspend* and the *input* variables of *lit* that the unifier tries to bind are put into a set *susp_var(p,db)* of suspended variables.

input-match suspension rule

Let lit subres(goal(p),sub)
    head clhead(clause(parent(p)))
If tag(p) = input-matching & mode(p) = starting
& unify(lit,head) $\neq$ nil
& domain(unify(lit,head)) $\cap$ in_var(lit,db) $\neq \emptyset$
then
    mode(p):= suspend,
    susp_var(p,db):= in_var(lit,db) $\cap$ domain(unify(lit,head)).

If and when one variable of *susp_var(p,db)* is bound by one of its producer processes - which means that it enters into the substitution *sub* -, the node *input-match* starts to work again (its mode changes from *suspend* to *starting*) in its attempt to find an **input matching**.

input-match re-starting rule

If tag(p) = input-match & mode(p) = suspend
& susp_var(p,db) $\cap$ domain(sub) $\neq \emptyset$
then
    mode(p):= starting

4.2.6 The *guard-eval* node operation

In our description we assume that guards are $safe$ (i.e. a guard evaluation never attempts to bind a variable that occurs in an input mode argument position of the calling literal) as is required for a Parlog program. This *guard* $safety$ property has to be assumed by the programmer or may be considered at compile time.

If the current clause has no guard, i.e. the function *goal* has empty value on the node, the *guard-eval* node becomes *reporting* with *success*. Therefore we have:

guard-eval empty rule

If tag(p) = guard-eval & mode(p) = starting & goal(p) = empty
then
    report from p-*subtree with* success

If the clause guard is not empty, the *guard-eval* node has to start a (sub)computation which performs the **guard evaluation**. It consists of computing a substi-

tution satisfying the guard of the given clause. (Note that guard evaluation may be nested).

The *guard-eval* node works in the same way as the Parlog "tree" *root* having the renamed clause guard as its query. Therefore its subcomputation is performed by the rules seen so far. The operation of a node labeled *guard-eval* is therefore as follows.

When the guard evaluation starts, the node labeled *guard-eval* (in *starting* mode) creates a child to which the new queries are passed by the function *goal*.

The label of that child is computed by the function $\tilde{tag}$ on *goal(p)*.

guard-eval starting rule

If tag(p) = guard-eval & mode(p) = starting & goal(p) $\neq$ empty
then
Create p-*subtree of child* temp
with tag $\tilde{\text{tag}}$(goal(p)), *mode* starting, *passing* goal
end Create

The *guard-eval* node receives the control again when its subcomputation ends. If the latter ends with *success*, the node changes its mode into *reporting*, while its child becomes *dormant*.

guard-eval success rule

If tag(p) = guard-eval & state(p) = waiting
& mode(child(p)) = reporting & res(child(p)) = success
then
report from p-*subtree with* success.

If the child's computation ends with *failure*, the node *guard-eval* becomes *reporting* with *failure*:

guard-eval failure rule

If tag(p) = guard-eval & state(p) = waiting
& mode(child(p)) = reporting & res(child(p)) = failure
then
report from p-*subtree with* failure.

In the appendix we list all thus obtained transition rules for the semantics of Parlog.

5 Conclusion

Let us conclude by a comparison of our approach to related work in the literature. The interesting approach in [1] which is based on Milner's CCS has difficulties with a precise description of the commit-operator. Our approach allows to describe commitment in a natural and simple way due to the fact that a notion of time (temporal order) is built-in into *evolving algebras*. (By the way this makes also the connection of *evolving algebras* to dynamic logic, observed by Gurevich in [20].)

The program of ca. 150 "Horn clauses with negation" in [26], proposed as formal specification of Parlog, presupposes a semantics for standard logic programs with negation; it is unstructured and technically rather involved.

Our formal model for Parlog starts from scratch - we presuppose only the notion of *evolving algebras*, but our rules can be read also simply as pseudocode over abstract data, avoiding even the notion of *evolving algebra* -; it is developed by stepwise refinement (which allowed us to isolate the independence of a semantical definition of AND-parallelism and OR-parallelism) and is a direct formalization of the basic intuition of Parlog, thus helping the programmer to understand and to control the computational effect of his programs. Furthermore it is to be expected that the present high-level formal specification of Parlog can be transformed naturally into an implementation of the WAM-like machine model underlying the Parlog implementation, along the lines of the corresponding way for sequential Prolog in [10, 11].

Our description of Parlog semantics is a complete specification of the core of Parlog which governs the computation of goals by user defined predicates. The model can be easily extended to the usual built-in predicates using the *evolving algebras* based methods developed for Sequential (Standard) Prolog in [4]-[6]. To perform the execution of built-in predicates we need other transitions rules (one or more for each predicate) for the operation of the node *or-par* which, in this case, will be a leaf of the "Parlog tree". For those built-in predicates having input mode arguments, the *or-par* node may have mode *suspend* and can be removed from such a state using the same technique as for the *input-match* node.

In our model we have no rule which tests for a possible *deadlock* state reached by the system during the computation when all processes suspend. The test could be performed introducing an external function which represents the state of the system and which is updated to the value "*deadlock*" when no transition rule can be applied.

6 Acknowledgements

We would like to thank Steve Gregory for its careful reading of the draft version of this paper and his helpful criticism and suggestions.

Appendix: List of final Transition Rules

stop rule

If mode(p) = reporting & parent(p) = undef
then
    mode(p):= dormant

and-par starting rule

If tag(p) = and-par & mode(p) = starting
then
    Let parlist [,]-decomp(goal(p))
    Create p-*subtree of children* temp(1), ..., temp(length(parlist))
    with mode ready *and*
        tag(temp(i)):= $\tilde{\text{tag}}$(proj(i,parlist))
        goal(temp(i)):= proj(i,parlist)
    end Create

and-par success rule

If tag(p) = and-par & mode(p) = waiting
& for all children q of p : mode(q) = reporting & res(q) = success
then
    report from p-*subtree with* success

and-par failure rule

If tag(p) = and-par & mode(p) = waiting
& for some child q of p: mode(q) = reporting & res(q) = failure
then
    report from p-*subtree with* failure

and-seq starting rule

If tag(p) = and-seq & mode(p) = starting & goal ≠ nil
then
    Let goal [&]-head(goal(p))
    Create p-*subtree of child* temp
    with tag tag(goal), *mode* starting *and*
        goal(temp):= goal
    end Create

and-seq continuation rule

If tag(p) = and-seq & mode(p) = waiting
& mode(child(p)) = reporting & res(child(p)) = success
then
    mode(p):= starting,
    goal(p):= [&]-tail(goal(p)),
    mode(child(p)):= dormant

and-seq success rule

If tag(p) = and-seq & mode(p) = starting & goal(p) = nil
then
    report from p-*subtree with* success

and-seq failure rule

If tag(p) = and-seq & mode(p) = waiting
& mode(child(p)) = reporting & res(child(p)) = failure
then
    report from p-*subtree with* failure

or-par starting rule

If tag(p) = or-par & mode(p) = starting
then
    Create p-*subtree of child* temp
        with tag seq-search, *mode* starting, *passing* goal
    end Create

or-par spawn rule

If tag(p) = or-par & mode(p) = waiting & tag(child(p)) = seq-search
& mode(child(p)) = reporting & res(child(p)) = success
then
Create p-*subtree of child* temp
with tag tag(goal(child(p))), *mode* starting *and*
goal(temp):= goal(child(p)),
mode(child(p)):= dormant
end Create

or-par success rule

If tag(p) = or-par & mode(p) = waiting & tag(child(p)) ≠ seq-search
& mode(child(p)) = reporting & res(child(p)) = success
then
report from p-*subtree with* success

or-par failure rule

If tag(p) = or-par & mode(p) = waiting
& mode(child(p)) = reporting & res(child(p)) = failure
then
report from p-*subtree with* failure.

seq-search failure rule 1

If tag(p) = seq-search & mode(p) = starting
& procdef(database,goal(p)) = nil
then
report from p-*subtree with* failure

seq-search starting rule

If tag(p) = seq-search & mode(p) = starting
& procdef(database,goal(p)) ≠ nil
then
Create p-*subtree of children* temp
with tag or-search, *mode* starting, *passing* goal *and*
parcl(temp):= [;]-head(procdef(database,goal(p)))
seqbloc(p):= [;]-tail(procdef(database,goal(p)))
end Create

seq-search continuation rule

If tag(p) = seq-search & mode(p) = waiting
& p = parent(q)
& mode(q) = reporting & res(q) = failure
& seqbloc(p) $\neq$ nil
then
Create p-*subtree of children* temp
with tag or-search, *mode* starting, *passing* goal *and*
parcl(temp):= [;]-head(seqbloc(p))
seqbloc(p):= [;]-tail(seqbloc(p))
end Create,
mode(q):= dormant

seq-search success rule

If tag(p) = seq-search & mode(p) = waiting
& p = parent(q)
& mode(q) = reporting & res(q) = success
then
report from p-*subtree with* success
saving goal *from child* q

seq-search failure rule 2

If tag(p) = seq-search & mode(p) = waiting
& p = parent(q)
& mode(q) = reporting & res(q) = failure
& seqbloc(p):= nil
then
mode(p):= reporting,
res(p):= failure,
mode(q):= dormant

or-search starting rule

If tag(p) = or-search & mode(p) = starting
then
Let parlist [.]-decomp(parcl(p)),
Create p-*subtree of children* temp(1), ..., temp(length(parlist))
with tag try-clause, *mode* ready, *passing* goal *and*
clause(temp(i)):= rename(proj(i,parlist),temp(i))
end Create

or-search success rule

Let succ-resp(q) (parent(q) = p & mode(q) = reporting & res(q) = success)
If tag(p) = or-search & mode(p) = waiting
& for some q succ-resp(q)
then
report from p-*subtree with* success
saving goal *from child* ε q succ-resp(q)

or-search failure rule

If tag(p) = or-search & mode(p) = waiting
& for all children q of p : mode(q) = reporting & res(q) = failure
then
report from p-*subtree with* failure

try-clause starting rule

If tag(p) = try-clause & mode(p) = starting
then
Create p-*subtree of children* temp(1),temp(2)
with mode ready *and*
tag(temp(1)):= input-match,
tag(temp(2)):= guard-eval,
goal(temp(1)):= goal(p),
goal(temp(2)):= clguard(clause(p)),
clause(p):= out-unif-shift(clause(p),temp(1))
end Create

try-clause failure rule

If tag(p) = try-clause & mode(p) = waiting
& for some child q of p: mode(q) = reporting & res(q) = failure
then
report from p-*subtree with* failure

try-clause success rule

If tag(p) = try-clause & mode(p) = waiting
& for all children s of p :
mode(s) = reporting & res(s) = success
then
report from p-*subtree with* success,
goal(p):= clbody(clause(p))

input-match failure rule

If tag(p) = input-match & mode(p) = starting
& unify(subres(goal(p),sub),clhead(clause(parent(p)))) = nil
then
report from p-*leaf with* failure

input-match success rule

Let lit subres(goal(p),sub)
head clhead(clause(parent(p)))
If tag(p) = input-matching & mode(p) = starting
& unify(lit,head) $\neq$ nil
& domain(unify(lit,head)) $\cap$ in_var(lit,db)) = $\emptyset$
& allowed(p) = 1
then
report from p-*leaf with* success,
sub:= join(sub, unify(lit,head))

input-match suspension rule

Let lit subres(goal(p),sub)
head clhead(clause(parent(p)))
If tag(p) = input-matching & mode(p) = starting
& unify(lit,head) $\neq$ nil
& domain(unify(lit,head)) $\cap$ in_var(lit,db) $\neq \emptyset$
then
mode(p):= suspend,
susp_var(p,db):= in_var(lit,db) $\cap$ domain(unify(lit,head))

input-match re-starting rule

If tag(p) = input-match & mode(p) = suspend
& susp_var(p,db) $\cap$ domain(sub) $\neq \emptyset$
then
mode(p):= starting

guard-eval empty rule

If tag(p) = guard-eval & mode(p) = starting & goal(p) = empty
then
report from p-*subtree with* success

guard-eval starting rule

If tag(p) = guard-eval & mode(p) = starting & goal(p) ≠ empty
then
Create p-*subtree of child* temp
with tag $\tilde{\text{tag}}$(goal(p)), *mode* starting, *passing* goal
end Create

guard-eval success rule

If tag(p) = guard-eval & state(p) = waiting
& mode(child(p)) = reporting & res(child(p)) = success
then
report from p-*subtree with* success.

guard-eval failure rule

If tag(p) = guard-eval & state(p) = waiting
& mode(child(p)) = reporting & res(child(p)) = failure
then
report from p-*subtree with* failure

References

[1] L.Beckmann, 1986 *Towards a Formal Semantics for Concurrent Logic Programming Languages*, 3rd International Conference on Logic Programming, Springer LNCS 225, pp. 335-349.

[2] C.Beierle & E.Börger, 1992, *Correctness Proof for the WAM with types*, CSL'91 5th Workshop on Computer Science Logic (Eds. E.Börger, H.Kleine Büning, G.Jaeger, M.M.Richter), Springer LNCS (to appear).

[3] R.Blakley, 1991, *Ph.D. Thesis*, University of Michigan.

[4] E.Börger, 1990 *A Logic Operational Semantics of full Prolog. Part I. Selection Core and Control*, CSL'89 3rd Workshop on Computer Science Logic, Springer LNCS 440, pp. 36-64.

[5] E.Börger, 1990 *A Logic Operational Semantics of full Prolog. Part II. Built-in Predicates for Database Manipulations*, MFCS'90 Mathematical Foundation of Computer Science (Ed. B.Rovan), Springer LNCS 452, pp. 1-14.

[6] E.Börger, 1991 *A Logic Operational Semantics of full Prolog. Part III. Built-in Predicates for Files, Terms, In-Output and Arithmetic*, Proc. Workshop Logic for Computer Science (Ed. Y.Moschovakis), Berkeley November 1989, MSRI Proceedings, Springer Verlag (to appear).

[7] E.Börger & B.Demoen, 1991, *A Framework to specify Database Update Views for Prolog*, PLILP'91 3rd International Symposium on Programming Languages Implementation and Logic Programming (Eds. J.Maluszynski & M.Wirsing), Springer LNCS 528, pp. 147-158.

[8] E.Börger & E.Riccobene, 1991, *Logical Operational Semantics of Parlog. Part I: And-Parallelism*, in: H.Boley & M.M.Richter (Eds.), *Processing Declarative Knowledge*, Springer LNAI 567, 1991, pp. 191-198.

[9] E.Börger & E.Riccobene, 1992, *Logical Operational Semantics of Parlog. Part II: Or-Parallelism*, in: A.Voronkov (Ed.),*Logic Programming*, Springer LNAI 592, 1992, 27-34

[10] E.Börger & D.Rosenzweig, 1991, *From Prolog Algebras Towards WAM - A Mathematical Study of Implementation*, Computer Science Logic (Eds. E.Börger, H.Kleine Büning, M.M.Richter, W.Schönfeld), Springer LNCS 533, pp. 31-66.

[11] E.Börger & D.Rosenzweig, 1992, *WAM Algebras - A Mathematical Study of Implementation. Part II*, pp.1-21 Technical Report, CSE-TR-88-91, Dept. of EECS, University of Michigan, Ann Arbor; 2nd Russian Conference on Logic Programming, Springer LNCS (to appear).

[12] E.Börger & D.Rosenzweig, 1991, *Prolog Tree Algebras. A formal specification of Prolog*, Proceedings ITI'91 13th International Conference on "Information Technology Interface", pp. 513-518.

[13] E.Börger & D.Rosenzweig, 1991, *An Analysis of Prolog Database Views and Their Uniform Implementation*,Technical Report, CSE-TR-89-91, Dept. of EECS, University of Michigan, Ann Arbor, pp. 44, April 1991; ISO/IEC JTC1 SC22 WG17 Standardization Report no.80, "Prolog Paris papers-2", July 1991, pp. 87-130.

[14] E.Börger & P.Schmitt, 1991 *A Formal Semantics for languages of type Prolog III.*, Computer Science Logic (Eds. E.Börger, H.Kleine Büning, M.M.Richter, W.Schönfeld), Springer LNCS 533, pp. 67-79.

[15] T.Conlon, *Programming in Parlog*, Addison Wesley 1989.

[16] T.Conlon & S.Gregory, *Hands on MacPARLOG 2.0 A User's Guide*, PLP Ltd 1990.

[17] J.A.Crammond, 1988, *Implementation of committed choice logic languages on shared memory multiprocessors*, Ph.D. Thesis, Heriot-Watt University, Edinburgh.

[18] G.Gottlob, G.Kappell, M. Schrefl, 1991, *Semantics of Object-Oriented Data Models - The Evolving Algebra Approach*, International Workshop on Information System for the 90's, Springer LNCS.

[19] S.Gregory, *Parallel Logic Programming in PARLOG*, Addison Wesley 1989.

[20] Y.Gurevich, 1991, *Evolving Algebras. A Tutorial Introduction*, EATCS Bulletin 43, February 1991.

[21] Y.Gurevich, 1988, *Logic and Challenge of Computer Science*, Trends in Theoretical Computer Science (Ed. E.Börger), Computer Science Press, pp. 1-57

[22] Y.Gurevich, 1988, *Algorithms in the World of Bounded Resources.* In: *The Universal Turing Machine - a Half-Century Story* (Ed. R.Herken), Oxford University Press, pp. 407-416.

[23] Y.Gurevich & J.M.Morris, 1988, *Algebraic Operational Semantics and Modula-2*, CSL'87 1st Workshop on Computer Science Logic (Eds. E.Börger, H.Kleine Büning, M.M.Richter), Springer LNCS 329, pp. 81-101.

[24] Y.Gurevich & L.S.Moss, 1990, *Algebraic Operational Semantics and Occam*, CSL'89 3rd Workshop on Computer Science Logic (Eds. E.Börger, H.Kleine Büning, M.M.Richter), Springer LNCS 440, pp. 176-192.

[25] E.Riccobene, *Tesi di Dottorato* (in preparation).

[26] G.Richard & A.Rizk, March 1988, *Semantique de PARLOG un language logique parallele*, INRIA Rapports de Recherche n.815.