ICOT Technical Report: TR-163

TR-163

Parallel Logic Programming Languages

by
Akikazu Takeuchi and Koichi Furukawa

April, 1986

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Parallel Logic Programming Languages

Akikazu Takeuchi and Koichi Furukawa

ICOT Research Center
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108 Japan

1. Introduction

Any programming language which can be treated mathematically has its own logic in its semantic model. Logic programming languages are examples of such languages. They are based on predicate logic and characterized by the fact that logical inference corresponds to computation. Owing to this, a program can be written declaratively and can be executed procedurally by computer. Many logic programming languages can be imagined. The one based on Horn logic is the most successful and has been extensively studied.

A logic program is represented by a finite set of universally quantified Horn clauses. A program can be read procedurally and declaratively [Kowalski, 1974]. A goal statement is used to invoke computation that can be regarded as refutation of the goal statement under the given set of clauses. Prolog is the first language which realized the idea [Roussel, 1975]. Its computation rule corresponds to left-to-right and depth-first traversal of an AND-OR tree.

Given a set of Horn clauses, there are many strategies for refutation other than the one adopted in Prolog. Among these, parallel strategies are of great interest. These correspond to the parallel interpretation of logic programs. Conery *et al.* classified them into four models, OR-parallelism, AND-parallelism, Stream-parallelism and Search-parallelism [Conery and Kibler, 1981]. Stream-parallelism has received much attention recently, because of its expressive power suitable for systems programming and other applications. Several parallel logic programming languages based on stream-parallelism have been proposed. They include Relational Language [Clark and Gregory, 1981], Concurrent Prolog [Shapiro, 1983], Parlog [Clark and Gregory, 1984a], Guarded Horn Clauses [Ueda, 1985a], [Ueda, 1986] and Oc [Hirata, 1986].

The following ideas and requirements seem to be what motivated these languages. The first was to create a parallel execution model for logic programs to fully utilize new parallel computer architecture. As hardware technology evolves, highly parallel computers become realizable using VLSI technology. However, to write a program for a parallel computer is a complicated task and involves new problems quite different from those in programming on a sequential computer. The gap between hardware and software seems to increase. It is believed that the success of the parallel computer depends on the software technology. Choosing languages for parallel programming is the most important decision in parallel software technology. In order for a programmer to avoid various problems and extract parallelism easily, languages should have clear semantics and be inherently parallel themselves. Because of their semantic clarity and high level constructs useful for programming and debugging, logic programs are being regarded as a candidate to fully utilize the power of parallel architectures.

The second issue is the extension of control of logic programming languages. Control facilities of Prolog are similar to conventional procedural languages, although the model for logic programming languages includes no specific control mechanism. There have been several proposals for more flexible computation. They augment Prolog by introducing new control primitives such as coroutines [Clark, McCabe and Gregory, 1982], [Colmerauer, 1982], [Naish, 1984]. Languages based on stream-parallelism can be regarded as an alternative attempt to extend control. These languages abandoned the rules of sequential execution, and thus first introduced parallelism. A great deal of effort was devoted to finding reasonable set of control primitives managing the parallelism obtained as a result.

The third point is to exploit new programming styles in logic programming and thus to exploit new applications of logic programming. Logic programming languages such as Prolog are suitable for database applications and natural language processing, but were suspected of being inadequate for applications such as operating systems. Parallel logic programming languages with control primitives managing parallelism aims at covering such applications as systems programming, object oriented programming and simulation and thus enlarging the applications of logic programming.

The parallel logic programming languages have a relatively short history, just six years or so. In this short time researches have been intensive around the world and many fruitful results have been obtained. A general view to these languages will be presented in this paper. The purpose of this paper is to present common features of the languages, to delineate the differences between them at the abstract level and to address the problems they present.

The paper is organized as follows. The stream-parallel computation model will be informally introduced in section 2. Section 3 provides definitions of several parallel logic programming languages. Common features shared among them and their difference are discussed. In final section, unsolved problems of semantics of parallel logic programming languages will be discussed briefly.

2. Stream-parallel Computation Model

Stream-parallel computation models were studied by [Clark, McCabe and Gregory, 1982] and [van Emden and de Lucena, 1982] independently as extended interpretation models of logic programs. Without introducing specific languages, we review the stream-parallel computation models informally. Consider the following logic program (syntax similar to Edinburgh Prolog [Bowen *et al.* , 1983] is used throughout).

```
quicksort(List,Sorted) :- qsort(List,Sorted,[]).                    (1)

qsort([],H,H).                                                      (2)
qsort([A|B],H,T) :-
     partition(B,A,S,L),
     qsort(S,H,[A|T1]),
     qsort(L,T1,T).                                                 (3)

partition([],X,[],[]).                                              (4)
partition([A|B],X,[A|S],L) :- A < X, partition(B,X,S,L).            (5)
partition([A|B],X,S,[A|L]) :- A >= X, partition(B,X,S,L).           (6)
```

The predicate, `quicksort(List,Sorted)`, expresses the relation that `Sorted` is the sorted list of the list `List`. `qsort(List,H,T)` represents the fact that the difference list `H-T` is the sorted list of the list `List`. `partition(List,E,S,L)` says that `S` is a sublist of `List` each element of which is less than `E`, and `L` is a sublist each element of which is greater than or equal to `E`. Given the above program and the following goal statement,

```
?- quicksort([2,1,3],X).
```

the Prolog interpreter will return the following answer substitution,

```
X = [1,2,3].
```

The algorithm used in the above logic program is "divide and conquer". Given a list, the CDR is divided into two lists, one consisting of elements less than CAR, and the other of elements greater than or equal to CAR. Both lists are sorted independently and they are combined to construct the sorted list of the original list. The algorithm is typically embodied in the clause (3). The clause can be read procedurally in the following way: To sort a list `[A|B]`, partition `B` into `S` and `L` with respect to `A`, and sort `S` and `L`. According to the sequential computation rule of Prolog, these subgoals are executed from left to right, that is, first the list `B` is partitioned, then `S` is sorted and finally `L` is sorted.

There are two possibilities for exploiting parallelism in the above program, especially in clause (3). One is cooperative parallelism. Since the lists `S` and `L` can be sorted independently, execution of two qsorts can be done in parallel. Although they share a variable, `T1`, they can cooperate in the construction of a list `H-T` by constructing non-overlapping sublists, `H-T1` and `T1-T`, of `H-T` in parallel. The other is pipelining parallelism. Note that both lists, `S` and `L`, are constructed incrementally from the heads by partition and that these two lists are consumed from their heads by two separate qsorts. Therefore, it is possible to start execution of the two qsorts with available parts of the lists before partition completes the lists. The parallelism of the partition and the two separate qsorts resembles so-called pipelining parallelism. Both parallelisms, processed by a parallel computer, are expected to be effective in reducing computation time.

Cooperative parallelism and pipelining parallelism are typical kinds of parallelism which stream-parallel interpretation can extract from logic programs. Generally speaking, there are two kinds of parallelism in stream-parallel interpretation. One for parallel interpretation of conjunctive goals and the other for parallel search for clauses. Cooperative and pipelining parallelism are special cases of the former parallelism. The latter is not discussed in this section; it will be introduced in the next section.

In the former parallelism, goals sharing variables are not independent and can interact with each other. Stream-parallelism involves cooperation of goals executed in parallel through shared variables. This is in clear contrast with AND parallelism, where no collaboration among goals is considered. In AND-parallel interpretation, conjunctive goals are solved independently and consistent solutions are extracted from their solutions. AND-parallel interpretation is in danger of generating a lot of irrelevant computation, since unnecessary computation is only proved to be irrelevant when it terminates.

Stream-parallel interpretation avoids this problem in the following way. First, bindings created in the course of computation are transported to other computations as soon as possible. This helps parallel computations to exchange bindings of shared variables in order to

maintain consistency. Secondly, it provides new control primitives which can restrict access modes to shared variables. There can be two modes in access to a variable, although the mode is implicit and multiple in logic programming. These modes are "input (read)" and "output (write)". New primitives can be used to restrict the access mode to a shared variable to either input or output. Appropriate restriction of access modes to a shared variable enables the variable to be used as an asynchronous communication channel between parallel computations. Using such asynchronous communication channels programmers can coordinate parallel goals and suppress irrelevant computation. In sum, the parallelism explored in stream-parallelism is controlled parallelism and the languages based on stream-parallelism can extract maximum parallelism while reducing irrelevant parallel computation.

3. Languages

Several parallel logic programming languages have been proposed. They are Relational Language, Concurrent Prolog, Parlog, Guarded Horn Clauses (hereafter called GHC), Oc. We start by defining the common features of these languages. These common features were first proposed in Relational Language.

3.1 Common Features

(1) *Syntax*:

For notational convenience, we define the common syntax. A program is a finite set of guarded clauses. A guarded clause is a universally quantified Horn clause of the form:

$$H : -G1, \ldots, Gn \mid B1, \ldots, Bm. \qquad n, m \geq 0$$

"|" is called a "*commitment*" operator or "*commit*". "$G1, ..., Gn$" is called the guard part and "$B1, ..., Bm$" the body part. H is called the head of the clause. A set of clauses sharing the same predicate symbol with the same arity is defined to be the definition of that predicate. A goal statement is a conjunction of goals of the form:

$$: - P1, \ldots, Pn. \qquad n > 0.$$

(2) *Declarative semantics*:

The declarative meaning of "," is "*and*" ("$\wedge$"). The clause can be read declaratively as follows:

For all term values of the variables in the clause,
H is true if both $G1, ..., Gn$ and $B1, ..., Bm$ are true.

(3) *Sketch of operational semantics*:

Roughly speaking, "," procedurally means fork. Namely a conjunction, "p, q", indicates that goals, p and q, are to be solved in different processes. The procedural meaning of a commitment operator is to cut off alternative clauses. We give a sketch of operational semantics using two kinds of processes, an AND-process and an OR-process [Miyazaki, Takeuchi and Chikayama, 1985].

The goal statement is fed to a root-process, a special case of an OR-process. Given a conjunction of goals, a root-process creates one AND-process for each goal. When all these AND-processes succeed, the root-process succeeds. When one of these fails, it fails.

Given a goal G with the predicate symbol P, an AND-process creates one OR-process for each clause defining the predicate P and passes the goal to each process. When at least one of these OR-processes succeeds, the AND-process commits itself to the clause sent to that OR-process, and aborts all the other OR-processes. Then it creates an AND-process for each goal in the body part of the clause and replaces itself by these AND-processes. It fails, when all of these OR-processes fail.

Given a goal and a clause, an OR-process unifies the goal with the head of the clause and solves the guard part of the clause by creating an AND-process for each goal in the guard. When all these AND-processes succeed, then it succeeds. When one of these fails, it fails.

(3) *Remarks*:

Conjunctive goals are solved in parallel by AND-processes. A clause such that the head can be unified with the goal and the guard can successfully terminate is searched for in parallel by OR-processes, but only one is selected by commitment. Parallel search is similar to OR-parallelism, but not the same because it is bounded in the evaluation of guard parts. A commitment operator selects one clause, cuts off the rest and terminates OR-parallelism.

Computation is organized hierarchically as an AND- and OR-process tree. Each OR process may be associated with a local environment storing bindings that would influence other competing OR processes if they were revealed to them. This will be discussed later.

In general, if access to a variable is restricted to input mode, then no unification which instantiates the variable to a non-variable term is allowed and such unification is forced to suspend until the variable is instantiated. This kind of synchronization mechanism is useful for delaying commitment until enough information is obtained. Languages proposed so far have different syntactic primitives for specification of restriction of access mode. We review them in the next section.

3.2 Restriction of Access Mode

- *Mode Declaration*

Parlog and its predecessor, Relational Language, take this approach. Restriction of access mode is specified by mode declaration. In Parlog, each predicate definition must be associated with one mode declaration. It has the form

$mode \quad R(m_1, \ldots, m_k).$

where R is a predicate symbol with arity k. Each m_i is "?" or "^". "?" indicates that access to a variable at this position in a goal is restricted to "*input*" mode. "^" indicates "*output*" mode. Note that there is no neutral (multiple) mode. During head unification, any attempt to instantiate a variable appearing in an argument specified as input in a goal to a non-variable term is forced to suspend. Output mode indicates that a term pattern at the corresponding argument position in the head will be issued from the clause. Unification between such output patterns and corresponding variables in the goal could be performed after the clause is selected. Implementation of Parlog is presented in [Clark and Gregory, 1984b]. The approach is to translate a general Parlog program to a program (called standard form) in a simple subset of the language, called Kernel Parlog. Kernel Parlog has only AND-parallelism and has no mode declaration. Input-mode unification and output-mode unification are achieved by special one-way unification primitives. For example, if the relation "p" has a mode declaration stating that the first argument is input and the second is output, the clause

```
p(question(P), answer(A)) :- good_question(P) | solve(P,A).
```

has the standard form

```
p(X,Y) :- question(P) <= X, good_question(P) |
          Y:= answer(A), solve(P,A).
```

T <= X is one-way unification which can bind variables in T, but suspends on an attempt to bind variables in X. Y := T is assignment unification. Note that mode declaration only restricts head unification. In general, there may be a case in which a variable appearing in an input argument in a goal is instantiated to a non-variable term during computation of a guard part. In Parlog, a program indicating this possibility is regarded as a dangerous program and excluded at compile-time by mode analysis. A merge operator merging two lists into one in arbitrary order can be defined in Parlog as follows:

```
mode merge(?,?,^).

merge([A|X],Y,[A|Z]) :- true | merge(X,Y,Z).
merge(X,[A|Y],[A|Z]) :- true | merge(X,Y,Z).
merge([],Y,Y) :- true | true.
merge(X,[],X) :- true | true.
```

• *Read-only annotation*

Concurrent Prolog adopts this primitive. Read-only annotation is denoted by "?". It can be attached to any variable. A variable with read-only annotation is called a read-only variable. Read-only annotation restricts access to the variable to read mode only. Any attempt to instantiate an unbound variable with read-only annotation to a non-variable term is forced to suspend until the variable is instantiated. Read-only annotation must be handled in the general unification procedure, since read-only variables can appear anywhere in a term. Using this annotation, the merge operator can be defined as follows:

```
merge([A|X],Y,[A|Z]) :- true | merge(X?,Y,Z).
merge(X,[A|Y],[A|Z]) :- true | merge(X,Y?,Z).
merge([],Y,Y) :- true | true.
merge(X,[],X) :- true | true.
```

Invocation of the goal takes the form:

```
merge(X?,Y?,Z).
```

• *Input guard*

This is adopted in GHC and Oc. Restriction of access mode to variables in a goal is subsumed in the definition of a guard part. In GHC, given a goal G and a clause C, during head unification and computation of the guard part of C, any attempt to instantiate a variable appearing in the goal to a non-variable term is forced to suspend. Oc has no guard condition, in other words, a guard part is always "*true*". Hence, specification of synchronization in Oc is simpler than in GHC. In Oc, any attempt to instantiate a variable in the goal to a non-variable term in head unification is forced to suspend. Intuitively, a head and a guard part of GHC and Oc specify conditions to be satisfied by input data received from a goal. The definition of merge is:

```
merge([A|X],Y,Oz) :- true | Oz=[A|Z], merge(X,Y,Z).
merge(X,[A|Y],Oz) :- true | Oz=[A|Z], merge(X,Y,Z).
merge([],Y,Oz) :- true | Oz=Y.
merge(X,[],Oz) :- true | Oz=X.
```

Note that output unification must be put in the body part of each clause. Otherwise it will cause suspension, since the output pattern will be regarded as the input pattern.

• *Comparison*

Different primitives for restricting access mode are adopted by different languages. In fact, the way to represent this restriction characterizes each language. They are basically separated into two classes. One in *procedure level* representation and the other in *data level*. Relational language, Parlog, GHC and Oc belong to the first class. Concurrent Prolog belongs to the second class. The fact that procedures and data are complementary objects in a programming language indicates the clear contrast between these two approaches.

Procedure level representation of input and output: Relational language and Parlog adopt mode declaration for specification of input and output. GHC and Oc utilize a guard part for the specification of input. One mode is given for each predicate definition. On the other hand, an input guard can include input specifications for each clause. Although they put input specifications at different levels, a predicate definition and a clause, both approaches associate input specification with a procedure.

Data level representation of input and output: Concurrent Prolog adopted read-only annotation to restrict access mode. A variable with read-only annotation cannot be instantiated (written), but can be read. In general, a variable with read-only annotation can be regarded as a "*protected term*" [Hellerstein and Shapiro, 1984], [Takeuchi and Furukawa, 1985], since it is protected from instantiation. Only a process which has access to the variable without read-only annotation can instantiate it. Since input synchronization is embedded in a data object, it becomes difficult to predict where and when synchronization will occur. This may impair transparency of control flow of the program. On the other hand, embedding control in a data object will enable novel control abstraction. Authors investigated this in the implementation of bounded buffer communication using protected terms [Takeuchi and Furukawa, 1985].

3.3 OR-parallel Multiple Environments and Guard Safety

Given a goal and a clause, an OR process evaluates head unification and the guard part. Since there are competing OR processes, bindings made for variables in the goal must be hidden from processes other than descendants of the OR-process. Therefore, conceptually, each OR-process has a local environment where these bindings are stored. Local environments associated with OR-processes form a tree, since AND-processes and OR-processes are hierarchically organized. The tree can dynamically expand and contract as computation proceeds. There is no need to manage this dynamic tree if no local binding is made, but otherwise it is an unavoidable task.

A clause is defined to be safe if and only if, for any goal, evaluation of head unification and the guard part never instantiates a variable appearing in the goal to a non-variable term. The definition is due to Clark and Gregory [Clark and Gregory, 1984b]. We add a few definitions. A program is defined to be safe, if and only if each clause in the program is safe. A language is defined to be safe if and only if any program written in it is safe. If a languages is safe, then it does not need to manage local environments. The concept of safety clarifies the difference between the languages.

Parlog, GHC and Oc are safe languages. The design philosophy of Parlog excludes any program which requires multiple environments. In Parlog, a program which may be unsafe is excluded as a dangerous program at compile-time mode analysis. GHC and Oc also do not need multiple environments. In fact, the rule of suspension in GHC and Oc can be paraphrased so that any attempt to make bindings which should be stored in the local environment is forced to suspend. Thus, safety is guaranteed at run-time by the suspension mechanism.

Concurrent Prolog is not safe. Thus, the tree of local environments has to be managed. Several attempts to implement Concurrent Prolog have been reported [Levy, 1984], [Miyazaki, Takeuchi and Chikayama, 1985]. Levi proposed a lazy copying scheme for implementation of multiple environments. Miyazaki *et al.* proposed a shallow binding scheme for this purpose. Implementation of Concurrent Prolog must solve two complicated problems associated with multiple environments. One is value access control. The other is detection of inconsistency between local environments.

Local environments are organized as a tree structure. An environment in a node must be accessible from nodes under the node, but must be hidden from others until the OR process associated with the environment succeeds in being selected. Once the OR process successfully terminates and it is selected, its local environment is merged with the local environment of the parent AND process (the local environment associated with the parent OR process of the AND process). Controlling the scope of variable access in this way is called value access control. On commitment, however, it may happen that these two environments contains inconsistent bindings. When should the inconsistency be detected ? This is called the problem of detection of inconsistency of local bindings. Ueda [Ueda, 1985b] presents two possible solutions. One is called *early detection*, which seeks to detect inconsistency as soon as possible. If there exists inconsistency, the clause fails before commitment and the clause is never selected. The other solution is called *late detection* and seeks to detect inconsistency immediately after commitment. In this case, the clause succeeds in being selected, but immediately fails after commitment. Programmers may prefer early detection, but it requires a complicated locking mechanism for variables when implemented on a distributed memory

machine. Ueda examined the semantics of Concurrent Prolog from the point of view of parallel execution and highlighted several subtle issues which become crucial problems in distributed implementation of the languages [Ueda, 1985b].

Codish defines a concept of safety in Concurrent Prolog which is different from the one stated here [Codish, 1985]. He introduced output annotation into Concurrent Prolog. Output annotation is used to declare which terms will be issued to a goal in head unification. In his model, a clause is defined to be safe if, for any goal, no binding for variables in the goal is made except those declared by output annotation during head unification and guard computation. Management of local binding becomes simple in execution of a program ensured to be safe, since such bindings are syntactically predictable. Codish tries to define a subset of Concurrent Prolog with output annotation such that the safety of any program written in it can be verified syntactically.

3.4 Hierarchical Computation Structure and Flatness

As already mentioned, computation is organized as an AND- and OR-process tree. The depth of the tree corresponds to the depth of nesting of guard computations. Some parallel logic programming languages have a flat computation structure.

[Flat Concurrent Prolog] Flat Concurrent Prolog is a subset of Concurrent Prolog in which guard parts are restricted to specify system predicates [Mierowsky *et al.* , 1985]. Since no general computation is allowed in a guard, computation structure is always flat. No tree-structured multiple local environments exist. This greatly reduces the complexity of implementation of the language, but it does not eliminate the problem of detecting inconsistency. Flat Concurrent Prolog seems to adopt late detection, but it is not clear how it is realized in a distributed memory environment.

[Parlog] Owing to the safe property of a clause, OR-parallel search for a clause can be translated into AND-parallel goals. In the course of translation from a legal Parlog program to a Kernel Parlog program, clauses defining a predicate are collected into one clause. In this clause, OR-parallel evaluation of guards is expressed by AND-parallel evaluation of conjunction of meta-calls, each of which calls the guard of each clause. The commitment operator is also expressed by a goal, which receives results from meta-calls, selects one and aborts the other meta-calls. Thus, there exists simple hierarchy of AND-processes in Parlog.

[GHC] AND- and OR-process tree is essential. In GHC, unification suspends if and only if binding made by the unification has to be stored in a local environment. In order to know whether a binding of a variable has to be stored in a local environment or not, the birth place of the variable in the hierarchy has to be identified. If it is the location where the binding is about to be made, then the binding can be made. Otherwise the attempt to bind is forced to suspend. This is why the hierarchical computation structure has to be managed with appropriate information on variables.

[Oc and Flat GHC] Flat GHC is a subset of GHC. In Flat GHC, as well as Flat Concurrent Prolog, a guard part is restricted to being a set of system predicates. Both Oc and Flat GHC have no computation hierarchy, since no general computation is allowed in a guard and this makes implementation of suspension simpler than in GHC. In fact, it can be implemented by one-way unification primitives similar to those of Parlog.

3.5 Summary of Comparison

We have reviewed parallel logic programming languages from the following three viewpoints.

(1) Suspension mechanism

(2) Multiple OR-parallel environments

(3) Hierarchically organized computation

Safety and flatness contribute to reduce the complexity of implementation. Safety make the management of multiple local environments quite simple. Flatness excludes the hierarchical structure of computation.

The suspension mechanism is independent of the other mechanisms in Concurrent Prolog. However, management of hierarchy of computation and multiple environments is complicated. Safe Concurrent Prolog is an attempt to revise the language to reduce the complexity of managing multiple environments. Flat Concurrent Prolog has neither hierarchy of computation nor multiple environments.

Owing to compile-time mode analysis, at run-time a Parlog program has a simple computation model, where suspension is realized by one-way unification primitives, computation hierarchy management is simple and no multiple environments exist. What Parlog compiler does at compile-time can be regarded as detection of multiple environments over the hierarchical structure inferred from a program with mode declaration for possible data flow. One flaw of Parlog is that one cannot write a meta-interpreter for the language in itself, while in other languages this is possible. The ability to write a meta-interpreter for the language in itself is an important property of a language for the self-contained development of its programming system.

In GHC, the suspension mechanism and computation hierarchy are closely coupled, though GHC needs no multiple environments. Oc and Flat GHC are similar to Kernel Parlog. In fact, any program written in Oc and Flat GHC can be translated into a Kernel Parlog program. If we can imagine Flat Kernel Parlog which prohibits general goals and meta-calls in a guard, then Oc, Flat GHC and Flat Kernel Parlog are equivalent to each other and constitute the simplest parallel logic programming language.

4. Semantics of Parallel Logic Programming Languages

In this final section, we present an open problem on semantics of parallel logic programs.

The semantics of logic programs has been extensively investigated [van Emden and Kowalski, 1976], [Apt and van Emden, 1982], [Lloyd, 1984]. These provide a rigid basis for various mathematical manipulations of logic programs such as program verification, equivalent program transformation and declarative debugging. Logical foundations for parallel logic programming languages are also indispensable for the development of the theory of parallel logic programming including verification, transformation and debugging. However, the results for pure logic programs are not directly applicable to parallel logic programming languages because of the new control primitives.

Given a program P (a set of Horn clause), the success set of the program is defined to be the set of all A in the Herbrand base of P such that $P \cup \{\leftarrow A\}$ has an SLD-refutation. The finite-failure set is defined to be the set of all A in the Herbrand base of P such that there exists a finitely-failed SLD-tree with $\leftarrow A$ as root. It is well known that the success set, the minimum model and the least fixpoint of the function associated with the program are equivalent. The finite-failure set is characterized by the greatest fixpoint under a certain condition. If a goal succeeds under sound computation rules, the result is assumed of being included in the success set. If a goal finitely fails, then the result is ensured to be included in the finite-failure set.

The declarative semantics of parallel logic programming languages recommends reading a guarded clause as just a Horn clause. This is sufficient as long as a goal succeeds, but this does not happen sufficient in many cases. Suppose that a goal failed. This implies neither that the result is not in the success set, nor that the result is in the finite-failure set, since the goal may fail even if there is a possibility of success because of commitment to an incorrect clause. The declarative semantics becomes insufficient also if two programs with different input/output behavior need to be distinguished.

Parallel logic programming languages have two control primitives not appearing in pure logic programs. These are a commitment operator and a synchronization primitive. Parallel logic programming relies heavily on these control primitives. However, a commitment operator changes the semantics of failure and a synchronization primitive introduces procedural flavor. It is now obvious that declarative semantics for pure logic programs cannot characterize such aspects of parallel logic programs as failure and input/output behavior.

Let us consider the algorithmic debugging for parallel logic programming languages, where the intended interpretation of a program plays an important role in guiding debugging. Declarative semantics such as success set is no longer sufficient. Intended interpretations should be abstract semantics characterizing all aspects which programmers intend to express. One of the authors developed an algorithmic debugger for GHC, where the intended interpretation with procedural flavor of a GHC program was defined [Takeuchi, 1986]. Lloyd *et al.* refined the framework for the above algorithmic debugging and discussed some difficult cases to handle [Lloyd and Takeuchi, 1986]. These are just starting points.

Semantics of parallel logic programming languages discussed in this paper have been defined only operationally. None of them provides a method to modelling abstract meaning of a program. What is required is semantics of parallel logic programs that can characterize what a programmer intends to express in a program. Meanings of programs should be abstract and independent from concrete implementation since the detail of implementation is not of interest. Furthermore semantics should be mathematically manipulatable so that important properties of programs can be derived from their meanings. Such semantics is strongly desired for the theory of parallel logic programming.

References

Apt, K. R. and van Emden, M. H. [1982] Contributions to the Theory of Logic Programming. *J. ACM*, Vol. 29, No. 3 (1982), pp. 841-862.

Bowen, D. L. (ed.), Byrd, L., Pereira, F. C. N., Pereira, L. M. and Warren, D. H. D. [1983] *DECsystem-10 Prolog User's Manual.* Dept. of Artificial Intelligence, Univ. of

Edinburgh.

Clark, K. L. and Gregory, S. [1981] A Relational Language for Parallel Programming. In *Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture*, ACM, pp. 171-178.

Clark, K. L. McCabe, F. and Gregory, S. [1982] IC-Prolog language features. In *Logic Programming*, Clark, K. L. and Tarnlund, S. A. (ed.), Academic Press, pp. 253-266.

Clark, K. L. and Gregory, S. [1984a] *PARLOG: Parallel Programming in Logic*. Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and Technology, London.

Clark, K. L. and Gregory, S. [1984b] *Notes on the Implementation of PARLOG*. Research Report DOC 84/16, Dept. of Computing, Imperial College of Science and Technology, London, 1984. Also in *J. of Logic Programming*, Vol. 2, No. 1 (1985), pp. 17-42.

Codish, M. [1985] *Compiling OR-parallelism into AND-parallelism*. Master Thesis, Computer Science, Feinberg Graduate School of the Weizmann Institute of Science, Rehovot.

Colmerauer, A. et al. [1982] *PROLOG II Reference Manual and Theoretical Model*. Groupe Intelligence Artificielle, Faculte des Sciences de Luminy, Marseille.

Conery, J. S. and Kibler, D. F. [1981] Parallel Interpretation of Logic Programs. In *Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture*, ACM, pp. 163-170.

Hellerstein, L. and Shapiro, E. [1984] Implementing Parallel Algorithms in Concurrent Prolog: The MAXFLOW Experience. In *Proc. 1984 Symp. on Logic Programming*, IEEE Computer Society, pp. 99-117.

Hirata, M. [1985] Self-Description of Oc and Its Applications. In *Proc. Second National Conf. of Japan Society of Software Science and Technology*, pp. 153-156. (in Japanese)

Kowalski, R. [1974] Predicate Logic as Programming Language. In *Proc. IFIP-74 Congress*, North-Holland, pp. 569-574.

Levy, J. [1984] A Unification Algorithm for Concurrent Prolog. In *Proc. Second Int. Logic Programming Conf.*, Uppsala Univ., Sweden, pp. 331-342.

Lloyd, J. W. [1984] *Foundations of Logic Programming*. Springer-Verlag, Berlin Heidelberg New York Tokyo.

Lloyd, J. W. and Takeuchi, A. [1986] *A Framework of Debugging GHC*. to appear Tech. Report, Institute for New Generation Computer Technology, Tokyo.

Mierowsky, C. , Taylor, S. , Shapiro, E. , Levy, J. and Safra, M. [1985] *The Design and Implementation of Flat Concurrent Prolog*. Tech. Report CS85-09, The Weizmann Institute of Science, Rehovot.

Miyazaki, T. , Takeuchi, A. and Chikayama, T. [1985] A Sequential Implementation of Concurrent Prolog Based on the Shallow Binding Scheme. In *Proc. 1985 Symp. on Logic Programming*, IEEE Computer Society, pp. 110-118.

Naish, L. [1984] *MU-Prolog 3.1db Reference Manual*. Internal Memorandum, Department of Computer Science, Univ. Melbourne.

Roussel, P. [1975] *Prolog: Manuel reference et d'utilisation.* Tech. Report, Groupe d'Intelligence Artificielle, Marseille-Luminy.

Shapiro, E. Y. [1983] *A Subset of Concurrent Prolog and Its Interpreter.* Tech. Report TR-003, Institute for New Generation Computer Technology, Tokyo.

Takeuchi, A. and Furukawa, K. [1985] Bounded Buffer Communication in Concurrent Prolog. *New Generation Computing*, Vol. 3, No. 2 (1985), pp. 145-155.

Takeuchi, A. [1986] *Algorithmic Debugging of GHC programs.* to appear Tech. Report, Institute for New Generation Computer Technology, Tokyo.

Ueda, K. [1985a] *Guarded Horn Clauses.* ICOT Tech. Report TR-103, Institute for New Generation Computer Technology. Also to appear in *Lecture Notes in Computer Science*, Springer-Verlag, Berlin Heidelberg (1986).

Ueda, K. [1985b] *Concurrent Prolog Re-examined.* ICOT Tech. Report TR-102, Institute for New Generation Computer Technology, Tokyo.

Ueda, K. [1986] *Guarded Horn Clauses.* Doctoral Thesis, Information Engineering Course, Faculty of Engineering, Univ. of Tokyo.

van Emden, M. H. and Kowalski, R. [1976] The Semantics of Predicate Logic as a Programming Language, *J. ACM*, Vol. 23, No. 4 (1976), pp. 733-742.

van Emden, M. H. and de Lucena Filho, G. J. [1982] Predicate logic as a programming language for parallel programming. In *Logic Programming*, Clark, K. L. and Tarnlund, S. A. (ed.), Academic Press, pp. 189-198.