# DAP Prolog: A Set-oriented Approach to Prolog

P. KACSUK* AND A. BALE
*Queen Mary College DAP Support Unit, Queen Mary College, Mile End Road, London E1 4NS*

***Prolog is based on first-order predicate logic and works by generating sets of values for variables expressed as arguments to rules. However, current Prolog systems, implemented on sequential machines, work by using the Left-to-Right, Depth First (LRDF) search strategy and thus generate successive members of the solution set one at a time. This approach is unsuitable for implementation on many parallel machines.***

***At Queen Mary College a project is under way to implement Prolog on an SIMD machine – the ICL Distributed Array Processor (DAP) – involving a set-oriented view of Prolog which is suitable for implementation on such a machine, and leads to an efficient execution of many symbolic problems.***

***SIMD (Single Instruction, Multiple Data set) computers are a class of machines that are ideally suited to exploit the rapidly growing field of VLSI research to produce high-power computation at a low cost. They do so by replicating a simple processing unit many times. This makes them very efficient for a large class of homogeneous, regular problems as diverse as database applications and fluid-flow calculations. The set-oriented view of Prolog interpretation exploits the associative parallelism of an SIMD machine by distributing the database over the processors and implementing unification of constants on a within-processing element basis.***

***In the first section we describe the research background to the current project involving defining and implementing DAP Prolog on the DAP. We then outline a general view of DAP Prolog and explain the set-oriented semantics in some detail. Section 3 is a brief overview of the basis of a sequential implementation of Prolog, while section 4 describes the main data structures and algorithms involved in implementing DAP Prolog on the DAP.***

*Received June 1987*

## 1. RESEARCH BACKGROUNDS

### 1.1 Parallel implementations of Prolog

Since Prolog was selected to be the basic language for the Japanese fifth-generation project the language has become more and more popular, particularly in the fields of artificial intelligence and expert systems. However, two things have become clear:

(i) the Left-to-Right Depth First (LRDF) search strategy of Prolog on sequential machines is not suited to large AI problems; and

(ii) by modifying Prolog's execution mechanism a large amount of inherent parallelism can be expressed, making implementations on parallel machines feasible.

Three basic types of parallelism can be distinguished in the execution of a Prolog program as follows.

(1) *Parallelism within unification.* When the number of arguments in a clause head is large, the unification of different arguments may be performed in parallel. This type of parallelism has only been touched in a few papers because of the large problems of consistency arising from shared variables in a clause head.[1,2]

(2) *Parallelism among unifications.* When a definition contains several alternative clauses the unification of the current goal can be done in parallel with all the alternative clause heads. This form of parallelism was utilized in Ref. 3 and is the parallelism expressed in DAP Prolog.

(3) *Parallelism in the control strategy.* Instead of using the sequential LRDF strategy for traversing the search tree there are many possible parallel strategies. These can be divided into three major classes as follows.

(*a*) *OR-Parallelism.* When the search tree has several alternative routes to solving a goal, these routes can be traversed in parallel by separate processes. This type of parallelism is particularly suited to MIMD architectures.[4,5]

(*b*) *AND-Parallelism.* When a goal consists of several sub-goals, these goals can be solved in parallel. This is the parallel breadth-first strategy. However, the common variables in the sub-goals present a similar problem to that found in within-unification parallelism.[6,7]

(*c*) *Stream parallelism.* Stream parallelism is a combination of OR- and AND-parallelism where the sub-goals are organised into a pipeline. The partial results produced by sub-goals are immediately consumed by the next sub-goal, resulting in a parallel operation on the sub-goals.[8] In order to utilise the control-strategy parallelisms, two basic computational paradigms have been proposed, control flow and data flow. The control-flow idea was first proposed to investigate the suitability of shared-memory multiprocessors for Prolog,[4,9] though some projects have explored implementations on distributed multiprocessors.[10,11] The data-flow paradigm has generally been targeted at distributed multiprocessors.[12–14] All of these projects have concentrated on MIMD computers, and there have so far been no proposals for implementing Prolog on SIMD machines.

### 1.2 Parallel logic languages

When implementing high-level languages on parallel computers a major question is how to express the parallelism. Two basic possibilities can be identified.

(*a*) *Implicit parallelism.* No explicit parallel construct appears in the language, while the compiler performs all the work of expressing any inherent parallelism. Typical examples are the vectorising Fortran compilers.[15]

(*b*) *Explicit parallelism.* The high-level language is extended with certain parallel constructs, and it is the programmer's responsibility to use these constructs to express the parallelism in his algorithm. For example

* Academic Visitor from Computer Research and Innovation Center, H-1015 Budapest I. Donati u. 35-45, Hungary.

Fortran Plus (the main language for the DAP) provides parallel matrix operations to express a regular parallel operation on a large data structure.[16]

Though implicit parallelism is frequently assumed to be more attractive to the user, there are many reasons for considering explicit parallelism. This is particularly true in Prolog, where implicit parallelism has several serious drawbacks.

(*a*) *Implicit OR-parallelism* can (i) cause unexpected changes in the semantics of the program due to the side-effects of certain built-in predicates (for example the cut or the database operations); (ii) lead to unregulated parallelism where the number of parallel processes becomes much larger than the number of processors. This results in most of the work that the system performs being wasted in administration, and any gains through parallelism being totally negated.

(*b*) *Implicit AND-parallelism.* Automatic detection of AND-parallelism requires the administration of all the independent subsets of goals which are to be executed in parallel, and the dependencies of goals to be maintained in the right order. This run-time analysis of programs can also cause a significant performance degradation.

All these reasons have led many researchers to extend Prolog with language constructs regulating the parallelism, [17,18] or to modify the semantics and syntax of Prolog to create a parallel programming language like PARLOG[19] or Concurrent Prolog.[20]

Another reason for introducing new language constructs into Prolog is to express the constraints introduced by the particular architecture under consideration as the target machine for our implementation. This reason has influenced our choice of DAP Prolog as a parallel extension of Prolog in order to effectively implement it on the DAP architecture.

### 1.3 DAP architecture

The DAP is designed to handle the wide class of problems which are homogeneous over the problem space. In other words the same algorithm can be applied to each item in a large data set. This class of problem obviously includes tasks such as text searching and large database handling as well as more numerically intensive tasks like fluid-flow calculations or many-body problems. The DAP has traditionally been used almost exclusively for numerical applications, largely due to the lack of a suitable language to handle symbolic tasks.

The hardware of the DAP in some sense models the nature of the tasks it is designed to handle:[21]

(1) Large problem spaces leads to a large number of processing elements.

(2) The localised nature of data interchange common in this class of problem demands a highly efficient short-range connection network between processing elements at the expense of long-range connections.

(3) Since algorithms frequently can be performed using simple data – short word-length data – the DAP is a 'bit-serial' machine, giving high efficiency for short data with very little loss in efficiency for long data.

The DAP is therefore designed as a regular (square) array of simple, fast, one-bit processing elements (PEs) which can communicate with their four nearest neighbours. They are all under the control of one central processor called the Master Control Unit (MCU) which specifies the actions of all the PEs concurrently, and hence they are frequently referred to as one unit: the PE 'plane'. This fits with a concept of a large, regular data structure on which certain well-defined operations can be performed (see Fig. 1).

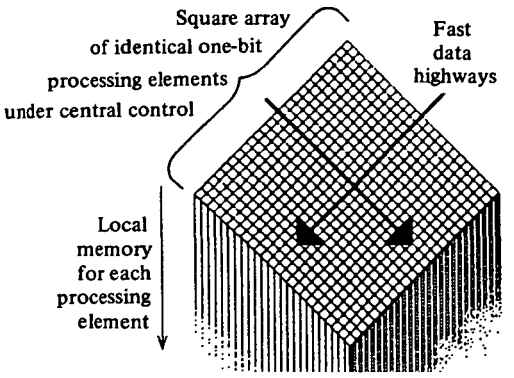



**Figure 1. The DAP–A distributed array of processors.**

The PEs each have several one-bit registers which enable them to perform arithmetic on their local memory. These registers can again be referred to as composite objects, of which the most important is the 'Activity register plane', or Activity plane. This register can be loaded from local memory or from a register in the MCU, and can control whether a particular PE participates in any given operation. The Activity plane can therefore serve to partition an algorithm across a data set, or permit operations to be dependent on some local condition.

The interprocessor connections of the DAP, while forming a two-dimensional network, can model any other interconnection pattern. In particular it is very efficient to consider the DAP to have a one-dimensional mode of operation, and it is in this mode that much of the current implementation of Prolog operates.

The DAP can be considered as an associative memory: each PE can simultaneously set a local register to depend on the contents of a memory plane; this register plane can then be considered as a 'parallel pointer' to all the selected objects, making the DAP a very efficient search engine.

The particular version of the DAP on which the work was undertaken is the pre-production model called the Mini-DAP which has an array of 32 by 32 PEs.

## 2. DAP PROLOG

### 2.1 General aspects of DAP Prolog

Most of the work done on parallel implementations of Prolog has been based on MIMD architectures. They have frequently attempted to modify the control mechanisms of Prolog in order to use the multi-tasking parallelism of such machines, and have often, therefore, neglected features of Prolog like its databasing capabilities.

The approach taken in the current project is based on the strengths of SIMD machines such as the DAP, i.e.: they are efficient at data-parallel rather than task-parallel problems, enabling them to work efficiently with large, homogeneous data structures; they are good associative engines, well engineered for database applications.

It was also decided that any new implementation of

Prolog should be a pure extension to current Prolog. That is, all existing Prolog programs (in some common implementation such as C-Prolog) should run without modification in the new language. Of course, no efficiency guarantees could be made about such unmodified code!

DAP Prolog is therefore standard Prolog extended with two new data structures and relevant support code. The structures are as follows.

(i) *Sets.* These utilise the associative nature of an SIMD machine, and provide for efficient implementation of such built-in predicates as 'member', 'delete', etc. They enable Prolog to be used efficiently in relational database applications.

(ii) *Arrays.* Arrays are the most obvious software realisation of the SIMD aggregate of processing elements. The extension of Fortran to Fortran Plus involved the introduction of arrays for the same reason. Arrays enable Prolog to be used efficiently in applications with a large numerical part. We shall not describe the array part of DAP Prolog any further in this paper.

DAP Prolog can therefore be described by:

$$\text{DAP Prolog} = \text{Prolog} + \text{Sets} + \text{Arrays}$$

In line with this, DAP Prolog has three main 'modes of operation'. They are a normal mode semantically identical to Prolog; a Set mode; and an Array mode. Communication between the various modes is through built-in procedures. Thus a program which wishes to use the Set mode will use the procedure

$$set\_mode(goal(V_1,V_2 \ldots V_n),sdef)$$

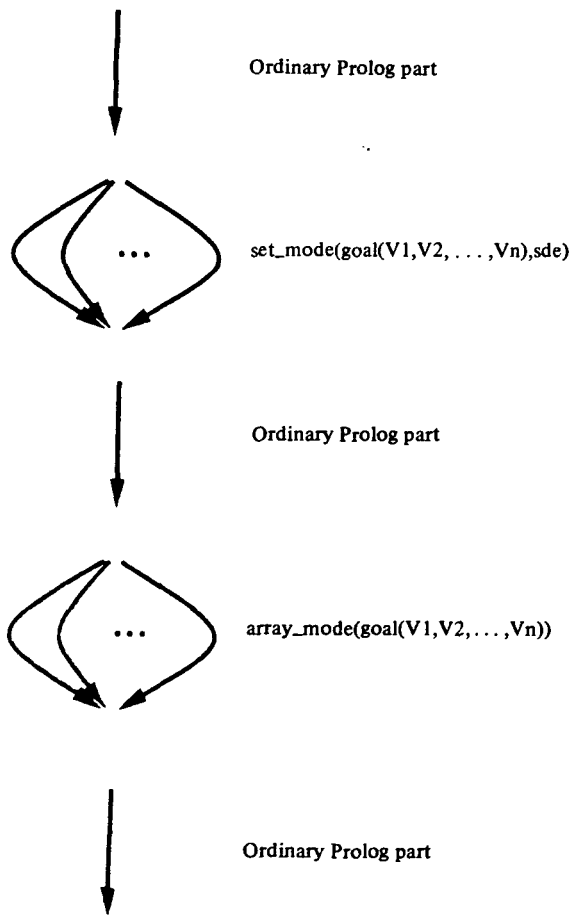



**Figure 2. DAP Prolog Execution trail.**

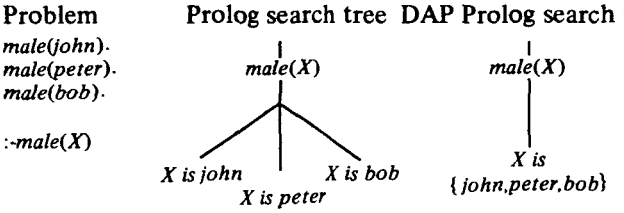
in order to call the *goal* procedure, within whose scope all interpretation is carried out in set mode. Based on this operational behaviour, Fig. 2 gives a paradigm of execution for a DAP Prolog program.

In this paper we concentrate on the set mode, which is most relevant to symbolic computation. The Array mode of DAP Prolog is explained in Ref. 22.

### 2.2 Set mode

An SIMD machine is able to operate on large data sets in a naturally associative fashion. DAP Prolog is therefore designed to use this associative power by considering sets of acts (unit clauses) as unitary objects, enabling database-oriented Prolog programs to execute very effectively with little rewriting.

In the DAP Prolog Set mode, a set-oriented interpreter adopts a mixed depth-first/breadth-first search strategy in which the multiple-fact branches of a conventional Prolog search tree are considered as generating binding sets rather than search non-determinism.




The semantics of this fit well with predicate logic: given the set

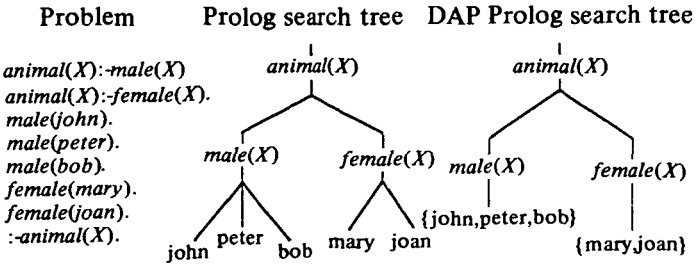
$$male = \{john,peter,bob\}$$

the solution of

$$X : X \in male$$

is exactly the set {*john,peter,bob*} returned by DAP Prolog (and returned in a sequential fashion by Prolog).

Of course, multiple *rule* branches are still preserved as true choice points by the interpreter:




so that the 'true' set solution is the union of the solutions returned by DAP Prolog.

There are several advantages to this set-oriented approach.

(i) *Speed-up.* The interpreter can handle the breadth-first stage in a single step (on an SIMD machine), eliminating a significant amount of backtracking when there are a large number of clauses in a set.

(ii) *Semantics.* In the above query :-*male*(*X*) the semantics are that *X* should be bound to the full set of

solutions. There is no semantic ordering in the above program. A conventional Prolog interpreter will return with the answer $X = john$, however, and must be forced to backtrack to generate any further solutions.

### 2.2.1 Set restriction

The sequence of unifications:

*...,number(X),succ(X,Z),...,greater_than_two(X),...*

with $X$ and $Y$ initially unbound does several things:

$X$ becomes bound to the values in the set *numbers* – which we shall suppose to contain {0,1,2,3,4,5}

the tuple $(X,Z)$ is unified with the set *succ* – let us say the value {(0,1),(1,2),(2,3),(3,4),(4,5),(5,6)}

thus $Z$ is bound to {1,2,3,4,5,6}

$X$ is unified with the value of *greater_than_two* – containing {3,4,5,...}. This 'restricts' $X$ to the value {3,4,5}.

All values which are bound to $X$ must also be restricted including that of $Z$, which must be restricted to {4,5,6}.

This set restriction is the key to set-oriented Prolog, and is the underlying mechanism of the interpreter. However, note that its effects are non-local in the sense that variables can be bound in any part of the set mode.

In the above example we see one kind of association between sets. However, we should identify two kinds of set, associated and derived. Two sets are associated if they represent two arguments in the same clause. For example, in the *succ* definition

*succ*(0,1).
.....
*succ*(4,5).

the two sets of arguments {0,1,2,3,4} and {1,2,3,4,5} are associated sets. This is because they are selections from one set of two-tuples

$$\binom{X}{Y} = \left\{\binom{0}{1}\binom{1}{2}\binom{2}{3}\binom{3}{4}\binom{4}{5}\right\}$$

In other words, individual elements of the two sets are associated. We denote this association by $\mathbf{X} \leftrightarrow \mathbf{Y}$.

A set $\mathbf{X}$ is derived from $\mathbf{Y}$ if each element in $\mathbf{X}$ is derived from an element of $\mathbf{Y}$:

$$\forall x_t \in \mathbf{X} \quad \text{and} \quad \forall y_t \in \mathbf{Y}: x_t = f(y_t)$$

The same relation can be defined in Prolog:

*derived_set*(X,Y):-
X *is f*(Y).

The difference between Prolog and DAP Prolog is that while Prolog defines the elements of $\mathbf{X}$ by the backtrack mechanism, DAP Prolog defines $\mathbf{X}$ in one step. '$\mathbf{X}$ is derived from $\mathbf{Y}$' will be denoted $\mathbf{X} \leftarrow \mathbf{Y}$.

In terms of association and derivation, restriction is defined as follows. Whenever a set $\mathbf{X}$ is restricted, all sets which are derived from or associated with $\mathbf{X}$ should also be restricted, i.e. for all elements $X_i$ deleted from $\mathbf{X}$, corresponding elements $Y_i$ of $\mathbf{Y}$ $(\mathbf{Y} \leftarrow \mathbf{X})$ and $Z_i$ of $\mathbf{Z}$ $(\mathbf{Z} \leftrightarrow \mathbf{X})$ are also deleted.

Sets can be restricted in three ways: by implicit restriction; by a restriction condition; or by unification.

*Implicit restriction* is the process of restriction that occurs when $\mathbf{X}$ is associated with or derived from some other set that is restricted. A *restriction condition* can be viewed as a 'filter' to be applied to the elements of a set. For example in:

..., *number*(X), $X > 2$, ...

the original set of $X$ ({1,2,3,4,5}) has selected from it all elements greater than 2. $X$ is restricted to {3.4.5}. Notice that again the same set is generated by Prolog through backtracking.

### 2.2.2 Set unification

Unification must deal with both associated (and derived) sets, and independent sets.

(1) *Associated set unification.* This involves pair-wise unification on set elements:

*if* $\mathbf{X} \leftrightarrow \mathbf{Y} \vee \mathbf{X} \leftarrow \mathbf{Y} \vee \mathbf{Y} \leftarrow \mathbf{X}$ *then*
*unify*$(\mathbf{X},\mathbf{Y}) \Leftrightarrow \forall (x,y) \in <(\mathbf{X},\mathbf{Y})$: *unify*$(x,y)$

Associated sets can be handled as parallel vectors, and the unification step can be performed in parallel. Consider the following program:

*triangle*(3,4,5)
*triangle*(3,3,5)
*triangle*(1,3,3)
*triangle*(5,5,9)
*isosceles*:-(X,X,Y).
?*isosceles*.

The unification process proceeds in three steps:

(1) $\mathbf{X} = \{3,3,1,5\}$

(2) *unify*($\mathbf{X}$,{4,3,3,5}) – the two sets are associated and so the unification is pair-wise:

| 3 | 3 | 1 | 5 |
|---|---|---|---|
| ↓ | ↓ | ↓ | ↓ |
| 4 | 3 | 3 | 5 |
| ⇓ | ⇓ | ⇓ | ⇓ |
| F | T | F | T |

$\mathbf{X}$ is now the set { ,3,.,5}

(3) $\mathbf{Y}$ is now restricted, since $\mathbf{Y} \leftrightarrow \mathbf{X}$. Thus $\mathbf{Y}$ is the set {.,5,.,9}. This is equivalent to:

$\{(X,Y): (X,A,Y) \in triangle \wedge X = A\}$

The reader can again see that Prolog generates the same solutions through backtrack.

(2) *Independent set unification.* If two sets are neither associated nor derived, they are 'independent', denoted ' '

$\mathbf{X}$ – $\mathbf{Y}$ *if* $\neg \mathbf{X} \leftrightarrow \mathbf{Y}$
*and* $\neg \mathbf{X} \leftarrow \mathbf{Y}$
*and* $\neg \mathbf{Y} \leftarrow \mathbf{X}$

Unification of two independent sets is defined as the intersection of the sets:

*if* $\mathbf{X}$—$\mathbf{Y}$ *then*
*unify*$(\mathbf{X},\mathbf{Y}) \Leftrightarrow \forall y \in \mathbf{X} \wedge \forall y \in \mathbf{Y}$: *unify*$(x,y)$

Independent set unification appears in those cases where we are unifying the argument sets from different definitions. For example, consider the following program:

```
capital(italy,rome).        river(london,thames).
capital(austria,vienna).    river(rome,tiber).
capital(hungary,budapest).  river(budapest,danube).
capital(england,london).    river(moscow,volga).
capital(france,paris).
?capital(X,Y), river(Y,Z).
```

where the query corresponds to:

$$\{(X,Y,Z): (X,Y) \in capital \wedge (A,Z) \in river \wedge Y = A\}$$

The first goal generates

$$\mathbf{Y} = \{rome,vienna,budapest,london,paris\}$$

The second goal unifies:

$$\mathbf{Y} \text{ and } \{london,rome,budapest,moscow\}$$

The intersection gives

$$\mathbf{Y} = \{rome,budapest,london\}$$

being the capitals for which information is available. Performing the restriction of associated sets **Z** and **X** gives:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \left\{ \begin{pmatrix} italy \\ rome \\ tiber \end{pmatrix} \begin{pmatrix} hungary \\ budapest \\ danube \end{pmatrix} \begin{pmatrix} england \\ london \\ thames \end{pmatrix} \right\}$$

Again, note that Prolog generates the same set through backtrack.

### 2.2.3 *Defining sets and using set mode*

Definitions to be used as sets should be declared as set definitions:

$$set\_definition(sdef/n)$$

*sdef* is declared to be an n-tuple set which can be used to return the value from a piece of *Set mode* code. The set-oriented execution of DAP Prolog can be used through the built-in procedure:

$$set\_mode(goal(V_1,V_2,\ldots,V_n),sdef)$$

which has the semantics:

switch into set working mode
solve the *goal* using set-oriented semantics
put the result tuples of $\{V_i\}$ into *sdef*
return to Normal mode

The work in Set mode is non-deterministic if rule branches are in the search tree. From the point of view of the Normal mode code it behaves as a non-deterministic built-in procedure. After the call of *set mode* sdef behaves just as any other set of definitions. In particular, on backtrack it will generate successive members of the solution.

## 2.3 Programming style

Two basic programming styles can be distinguished in Prolog: list oriented and database oriented. In the list-oriented style of programming compound data structures are represented by lists, and operated on by recursive list-processing predicates. This is generally a clean, side-effect-free programming style, and is well suited to functional programming one of Prolog's major application fields.

The database programming style represents data structures by clause sets. Frequently this type of program manipulates the database directly (using *asserts* and *retracts* and so on), which are generally extra-logical side-effecting operations. Backtracking is generally necessary to explore all possible solutions (leading to the *set_of* class of built-in predicates). However, this style is well suited to database applications such as expert systems.

Prolog programmers have frequently preferred the list-oriented style of programming even in the case of database operations. In DAP Prolog, however, the set-oriented features of the language make database style operations much more attractive. Let us consider a database problem to illustrate the difference between the styles, and the difference between Prolog and DAP Prolog. (This problem was originally presented in Ref. 23.)

A travel agency offers one- and two-week holidays in various cities in Europe and Africa. For each destination the brochures contain the transport cost and the price of a week's trip, which depends on the destination, the season and the level of accommodation (hotel, bed and breadfast, camping).

The list-oriented program is as follows.

```
town(europe,
    [rome,...,budapest]).
town(africa,[tunis,...]).

travel(rome,120).          %The journey to Rome
travel(tunis,200).             %costs £120
    ...

stay(rome,hotel,210)       %A week in a Rome hotel
stay(budapest,camping,70).     %is £210
    ...

duration(1).
duration(2).

high_season(europe,[june,
    july,august]).
high_season(africa,[decem-
    ber,january,february]).

low_season(europe,[jan-
    uary:60,february:
55,...,december:75])
low_season                 %a week in June in Africa
(africa,[june:65,...])         %is 65% of the high-
                               %season cost

trip(Town,Accommodation,Price,Travel):-
    travel(Town,Travel),stay(Town,Accommodation,
    Price).

trip_cost(Continent,Month,Weeks,Price,Travel,Cost):-
    high_season(Continent,L),member(Month,L),
    duration(Weeks),
    Cost is Price*Weeks+Travel.
trip_cost(Continent,Month,Weeks,Price,Travel,Cost):-
    low_season(Continent,L),member(Month:Percent,L),
    duration(Weeks),
    Cost is (Price*Weeks*Percent)/100+Travel.

economy_trip(Continent,Month,Town,Weeks,Cost,Max
_cost):-
    town(Continent,L),member(Town,L),
    trip(Town,_,Price,Travel),
    trip_cost(Continent,Month,Price,Travel,Weeks,Cost),
    Cost < Max_cost.
member(H,[H|T]).
member(X,[H|T]):-member(X,T).
```

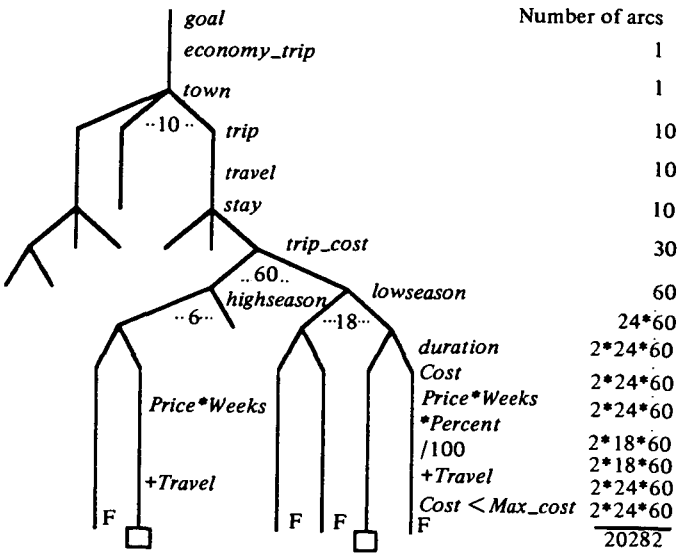



**Figure 3. Prolog search tree for the travel agency program.**

The same program written in a database form is as follows:

```
town(europe,rome).
town(africa,tunis).
   ...
town(europe,budapest).

travel(rome,120).
travel(budapest,150).
   ...

stay(rome,hotel,210)
stay(budapest,camping,70).
   ...

duration(1).
duration(2).

high_season(europe,june).
high_season(africa,jan).
...

low_season(europe,january,60).
   ...
low_season(africa,june,65).
   ...

trip(Town,Accom,Price,Travel):-
   travel(Town,Travel),stay(Town,Accom,Price).

trip_cost(Continent,Month,Price,Travel,Weeks,Cost):-
   high_season(Continent,Month),
   duration(Weeks),
   Cost is Price*Weeks+Travel.
trip_cost(Continent,Month,Price,Travel,Weeks,Cost):-
   low_season(Continent,Month,Percent),
   duration(Weeks),
   Cost is (Price*Weeks*Percent)/100+Travel.

economy_trip(Continent,Month,Town,Weeks,Cost,Max_
cost):-
   town(Continent,Town),
   trip(Town,_,Price,Travel),
   trip_cost(Continent,Month,Price,Travel,Weeks,Cost),
   Cost < Max_cost.
```

The main difference between the programs is in the representation of the database. In the first case the unit clauses contain repeating groups (lists), while in the second case new clauses are introduced into the database representation. This is the same transformation that is used in the relational database model to obtain the first normal form (1NF) of the relations.

Let us compare the database access in the two approaches:

*...,town(Continent,L,),member(Town,L),...* I
*...,town(Continent,Town),...* II

In case I *member* involves 2***i** unifications to return the **i**th member of the list, so 2***i**+1 unifications in total are required.

Case II requires only **i** unifications. Moreover, an array processor can perform the access in one associative step in case II, but not in case I since the list access is inherently sequential. With the set-oriented interpreter the complete set of alternatives can be picked up at once.

Fig. 3 depicts a search tree for the sequential interpreter of the 1NF form of the following problem:

*?economy_trip(Continent,Month,Town,Week,Cost,350).*

*What trips are available for under* £350?

given a certain (small) size of database.

Fig. 4 shows the set-oriented search tree, which contains only one branch. In Fig. 3 each leaf represents only one possible member of the solution set, while in Fig. 4 a leaf represents a whole partition of the set.

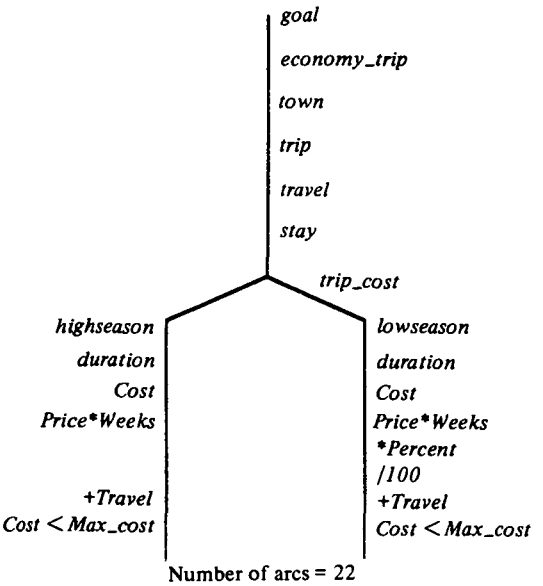



**Figure 4. DAP Prolog search tree for the travel agency program.**

The theoretical speed-up (TS) obtainable by the set-oriented interpreter is:

$$TS = \frac{\text{Number of arcs in sequential search tree}}{\text{Number of arcs in set-oriented search tree}}$$

For the travel agency program, with the size of database indicated in Fig. 3, the theoretical speed-up is:

$$TS = \frac{20282}{22} = 921$$

TS is, of course, an upper limit on actual speed-up since the execution time for an arc in the set-oriented search tree is likely to be higher than in the normal search tree

because of the set restriction administration. However, note that the speed-up would increase with the size of the problem until all PEs in the array processor are in use.

## 3. PROLOG INTERPRETERS ON SEQUENTIAL MACHINES

Any interpreter must have certain functional parts, namely: some way of representing the user's input script – the static data structures; some representation of partially completed evaluations – the dynamic data structures; the mechanism to evaluate the static data structures and produce a 'result' – the interpretation mechanism; input/output routines including a parser and output conversion routines.

### 3.1 Static data structures

*3.1.1 Clause representation*

A Prolog statement has the form:
*atom(term,term,...) :-functor,functor,....*

We therefore need some way of representing three kinds of item.

(*a*) The rule head – with its name (*atom*); its arity (number of arguments); its actual arguments; and a pointer to its body. Also useful is to store the number of variables in the rule.

(*b*) The rule body – each of whose functors will have a similar representation to that of a rule head.

(*c*) The items of information, such as atoms, numbers and rules. Numbers are represented by their values; atoms are represented as pointers to their string values, and terms are represented as pointers to another storage area with similar structure to the two structures outlined above.

### 3.2 Dynamic data structures

*Stacks.* Prolog's non-deterministic behaviour is one of its most unconventional features, and the element that gives it much of its expressive power. A Prolog interpreter works by searching a database of facts according to a list of rules. Whenever it fails to find a solution via one route it will 'backtrack' and find another route to a solution until all routes have been tried.

The dynamic data structures must therefore contain more than the environments of most other languages. Not only must they contain results of partial evaluations but also the information necessary to allow the backtracking mechanism to operate. These dynamic structures are in the form of stacks which build up as a program progresses and then decreases as it backtracks. They therefore need to contain the values of variables (Variable Stack), and information as to when a variable received its value (Trail Stack); information about which procedure to backtrack to (Environment Stack); and which procedure will be the next to be called in the event of success.

### 3.3 Interpretation process

*Clause selection.* Given the simple program
*male(john).*
*female(mary).*
*male(bob).*
*married(fred,mary).*
*male(X):-married(X,Y),female(Y).*
*:-...,male(X),...*
it is clearly necessary for the interpreter to find all the clauses corresponding to the name *male*, and no other name. Thus we should have some means of finding the correct item in the head data structures. Generally this is done using some kind of hash-based search strategy, but having an associative processor is clearly an advantage in this context!

*Unification.* In the program above, once a clause has been found that satisfies the criteria of name and arity, any arguments in the calling procedure must be 'unified' with its arguments. Unification is a matching process whereby variables receive values. The rules are that a variable matches anything (there is no typing in Prolog); a constant only matches itself; and a functor matches a functor with the same name and arity, if and only if all the arguments match.

*Shallow backtrack.* If this matching process fails, then the interpreter must proceed to the next rule in the database with the same name and arity. In the above program, if *X* has been bound to something that does not match with *john*, the interpreter must look for the rule *male(bob)*. This process of attempting a match and proceeding in the case of a fail is known as 'shallow backtrack'.

*Backtrack.* Once unification has been performed on the clause head, the interpreter proceeds to the clause body (if any). Now each of the parts of the body are tried in turn to try to find a match. Again, in the above program, suppose the interpreter has reached the rule:

*male(X):-married(X,Y),female(Y).*

The interpreter must be capable of recovering its state should any call in the rule body fail. If the *married* call should fail, the interpreter must undo any bindings that occurred in the rule head unification, and find another *male(X)* procedure to call. If the fail occurs in the *female(Y)* call, then the interpreter must undo any bindings that occurred due to the *married(X,Y)* call, and try for another successful *married* rule. Thus we must save several pointers: the next call in the current rule; the rule being searched for unification with the current call; the place where a search for re-satisfying each previous call will have to begin; all previous values of variables, and where they received their values, which are the contents of the dynamic data structures. This process of going back to the previous call to find a re-satisfaction is called 'backtracking'.

## 4. IMPLEMENTING DAP PROLOG ON THE DAP

DAP Prolog is an extension to Prolog. Therefore the first step in the implementation of DAP Prolog is to implement ordinary Prolog. The further features of DAP Prolog such as the Array and Set modes can be built on to a regular foundation.

### 4.1 Ordinary Prolog implementation

The operation of a Prolog interpreter is an exhaustive search of a tree representing the input program. Thus the

existence of a powerful search engine (using the associative power of an SIMD machine) is clearly useful in implementing even ordinary Prolog.

### 4.1.1 Static data structures

The definitions of the static data structures can be expressed in Prolog:

(1) *Clause head representation*

```
clause_head_description(N,ANO,VNO,AL,BP):-
    clause_name(N),
    arg_number(ANO),
    var_number(VNO),
    arg_list(AL),
    body_pointer(BP).
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| data plane | *N* | *ANO* | *VNO* | *AL*1 | ••• | *ALn* | *BP* | Clause head records are stored in the head plane (Fig. 5) |
| type plane | *TN* | | | *TA*1 | ••• | *TAn* | | |

Note that clause records are not stored in any linked-list structure since they are not found by a sequential search. Instead they are identified by their name (*N*) and the type name flag (*TN*) and found in an associative lookup.

(2) *Clause body representation*

```
clause_body([]).
clause_body([goal_descriptor(_,_,_,_)|GL]):-
    clause_body(GL).
goal_descriptor(N,ANO,AL,GP:-
    goal_name(N),
    arg_number(ANO),
    arg_list(AL),
    next_goal_pointerGP).
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| data plane | *N* | *ANO* | *AL*1 | ••• | *ALn* | *GP* | Goal descriptors are stored in the body plane (Fig. 5) |
| type plane | *TN* | | *TA*1 | ••• | *TAn* | | |

(3) *Term representation*

```
term(N,ANO,AL):-
    term_name(N),
    arg_number(ANO),
    arg_list(AL).
```

| | | | | | | |
|---|---|---|---|---|---|---|
| data plane | *N* | *ANO* | *AL*1 | ••• | *ALn* | Term descriptors are stored in the term plane (Fig. 5) |
| type plane | *TN* | | *TA*1 | ••• | *TAn* | |

For each static data structure the possible argument list elements are represented by:

(*a*) atom — pointer field → Pointer to the Symbol table; *TA* — type field

(*b*) number — Value; *TNO* — type field

(*c*) variable — Var. no.; *TV* — The position of the variable within the clause; type field

(*d*) term — pointer field → Pointer to the Term table; *TT* — type field

### 4.1.2 Dynamic data structures

(1) *Environment stack.* For each node of the search tree the following environment frame is created in the environment stack (see Fig. 5):

```
environment_
    frame(CG,FN,LCN,CP,CVSP.CMSP,CL):-
    current_goal(CG),
    father_node(FN),
    last_choice_node(LCN),
    current_var_stack_pointer(CVSP),
    current_mol_stack_pointer(CMSP),
    current_level(CL).
```

(2) *Variable stack.* For each variable in a procedure a new item is allocated in the variable stack (see Fig. 5), which consists of three planes: data plane, containing the binding value of the variable; type plane, representing the type of the binding value; level plane, giving the binding level in the search tree.

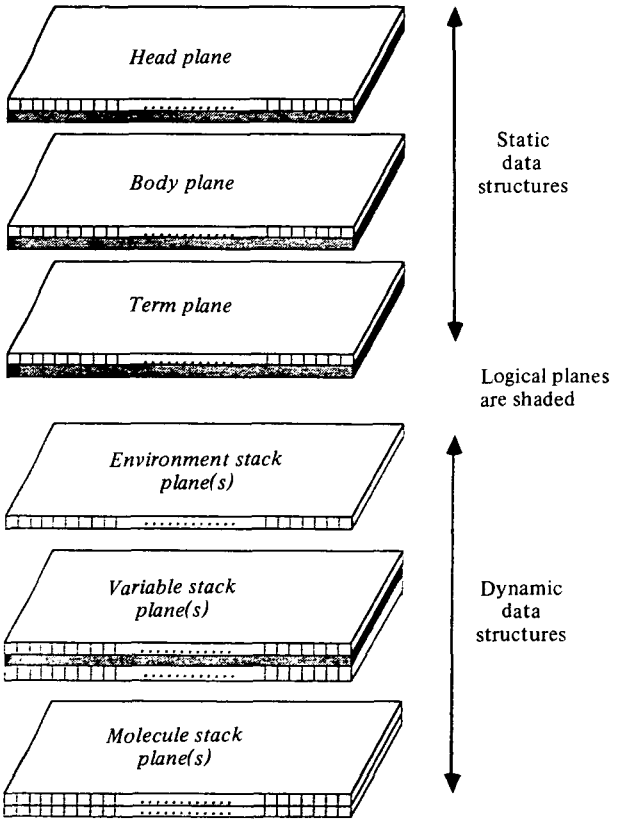



**Figure 5. Representation of the interpreter's data structures.**

(3) *Molecule stack.* When a variable is bound to a compound term, a 'molecule' (as opposed to an 'atom') is created on the molecule stack (see Fig. 5): variable plane, pointer to the variable stack frame from which the variables of the term should be taken; term plane, pointer to the term plane.

### 4.1.3 Interpretation process

The interpretation process is a normal structure-sharing interpreter with one exception: the backtrack undo. The task of the undo process is to unbind variables bound since the latest choice point. This process is conventionally based on a sequential trail stack.

In the DAP Prolog interpreter the whole undo process

is executed in one step independently of the number of variables to be un-instantiated. The trail stack is replaced by a variable stack containing the binding level of each variable. The DAP can associatively find all variables bound below a given node and delete all of them in one step.

### 4.2 Implementation of set mode

#### *4.2.1 Static data structures*

In order to represent a set definition we introduce a new type of clause-head structure which will be more economical in space, and will enable the parallel nature of an SIMD machine to be used in set operations.

All the clauses in a set definition are represented by a common clause-head description in the head plane (Fig. 5)

*clause_head_description(N,ANO,VNO,AL,MP)*:-
  *clause_name(N)*,
  *arg_number(ANO)*,
  *var_number(VNO)*,
  *arg_list(AL)*,
  *mask_pointer(MP)*.

The elements of *AL* are pointers into the set mask planes (Fig. 6). Since the argument sets of a set definition are associated, one common mask plane is sufficient to describe set membership. Notice that the structure of the clause-head descriptor is identical to that for a normal clause, making it easy to handle set definitions in Normal mode.

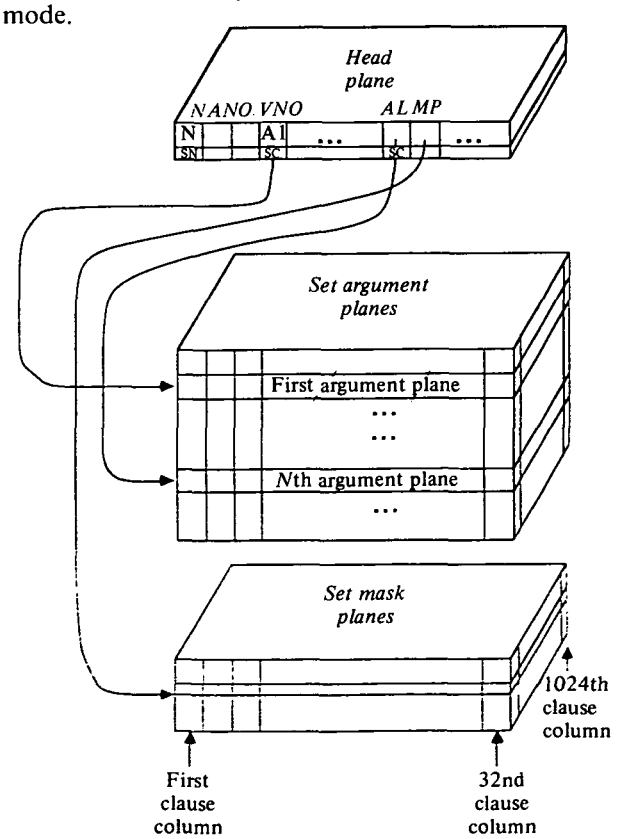



**Figure 6. Representation of set definitions.**

#### *4.2.2 Dynamic data structures*

A new stack, the set descriptor stack, is added to the interpreter's set of stacks. This stack plays the same rôle for sets as the molecule stack plays for functors. Whenever a variable is bound to a set, the interpreter creates a set description record and pushes it on to the set description stack.

*set_description_record(SAP,SMP,SCL,ST)*:-
  *set_arg_pointer(SAP)*,
  *set_mask_pointer(SMP)*,
  *set_creation_level(SCL)*,
  *set_type(ST)*.

where

*set_type(B)*:-
  *base_set*.
*set_type(D)*:-
  *derived_set*.
*set_types(S)*:-
  *sibling_set*.
*set_type(R)*:-
  *restricted_set*.

Administration of relationships among sets is as follows.

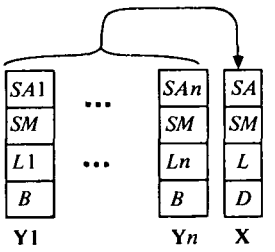
(*a*) When an unbound variable is unified with the argument set of a set definition, the binding set is a 'base set'.

(*b*) *Associated sets.* Associated sets **X** and **Y** are administered automatically through their shared mask: when one set is restricted, the other automatically has the corresponding elements deleted.

(*c*) *Derived sets.* When **X** is derived from *N* associated sets

$$\mathbf{X} \leftarrow \mathbf{Y}_1 \times \mathbf{Y}_2 \times \ldots \times \mathbf{Y}_N : \mathbf{Y}_1 \leftrightarrow \mathbf{Y}_2 \leftrightarrow \ldots \leftrightarrow \mathbf{Y}_N$$

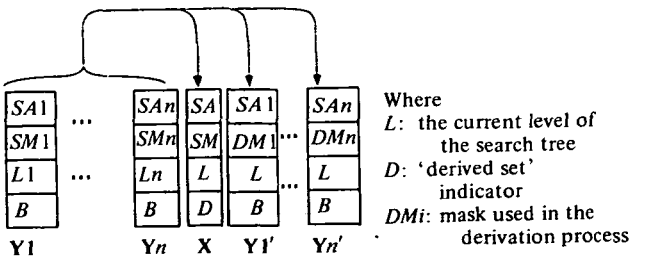
**X** becomes associated with the deriving sets.




When the *N* sets are independent:

$$\mathbf{X} \leftarrow \mathbf{Y}_1 \times \mathbf{Y}_2 \times \ldots \times \mathbf{Y}_N : \mathbf{Y}_1 — \mathbf{Y}_2 — \ldots — \mathbf{Y}_N$$

the set description records of **X** and $\mathbf{Y}_i'$ should be created:

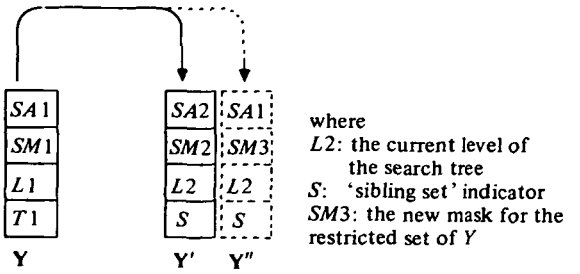



(*d*) *Sibling sets.* A set **Y**′ is the sibling set of **Y** if **Y**′ was created during the unification of **Y** and a head argument. For example, in

*?capital(X,Y),river(Y,Z),...*

**Y** is initially bound to {*rome,vienna,budapest,london,paris*}. Its sibling set **Y**′ is created when **Y** is unified with {*london,rome,budapest,moscow*}. As a result, **Y**′ will be {*london,rome,budapest*} and **Y** should be restricted to **Y**″, {*rome,budapest,london*}. Semantically, of course, the sibling sets are the same object.

The administration of sibling sets is performed by creating a new set description record for Y′, and if Y is restricted, then for Y″ as well:




(*e*) *Restricted sets.* When a set is restricted, a new set description record is created for it, containing the level number and new mask of the set.

Fig. 7 shows the representation of the set descriptor stack.

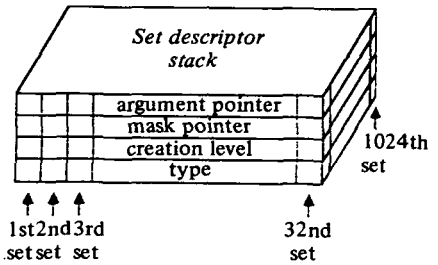



**Figure 7. Representation of the set descriptor stack.**

### *4.2.3 Interpretation process*

The interpretation process consists of four basic operations: set unification, set derivation, set restriction, backtrack.

*Set unification.* There are three cases to consider.

(*a*) *Variable/set.* A new set description record will be created and pushed on to the set descriptor stack. A pointer to this record is associated with the variable in the variable stack.

(*b*) *Two associated sets.* Corresponding elements in the sets where the set mask is 'true' are unified in parallel. The new value of the mask is a 'true' wherever unification succeeded.

(*c*) *Two independent sets.* Given two independent sets **X** with $M$ elements and **Y** with $N$ elements, $M < N$, unification is the following serial/parallel algorithm:

*X_mask = false; Y_mask = false;*
*for* $i = 1, M$ *do*
  *Y_mask = true where unifies*$(X_i, Y)$;
  *X_mask*$[i]$ *= true if any* $Y_j$ *was unifiable*;
*endfor*

The operations in the loop body can be executed in one step on an SIMD machine. Thus we require $2 \times min(M,N)$ steps rather than $M \times N$.

*Set derivation.* Suppose the derivation to be executed is the following:

$\mathbf{Y} \leftarrow f(\mathbf{Y}_1, \mathbf{Y}_2, \ldots, \mathbf{Y}_m)$ where $\mathbf{Y}_1$ has $N_1$ elements, etc.

There are two main cases:

(*a*) The $Y_i$ sets are associated. Each operation of the function 'f' can be executed in 1 step instead of $N_1 \times N_2 \times \ldots N_m$ steps.

(*b*) The $Y_i$ sets are independent. In this case one set – the largest set is most efficient – should be selected for the set operation. Instead of $N_1 \times N_2 \times \ldots \times N_m$ steps only $N_2 \times \ldots \times N_m$ steps are required.

*Set restriction.* Whenever a set **X** is restricted through a restriction condition or unification, all of its associated, derived and sibling sets should also be restricted. Because of the shared mask the restriction of associated sets is automatic. For derived and sibling sets the common-level number stored in the set descriptors can be used associatively. When such a set is found, its mask should be modified in line with **X**'s and a new set description record should be created containing the new mask and the current level of the search tree.

*Backtrack.* The undo mechanism is based on the set descriptor stack. If the last choice point is on level **L**, all set description records whose set creation level is greater than or equal to **L** will be deleted from the stack. In this way all sets created in or after **L** are removed from the stack.

## 5. CONCLUSIONS

DAP Prolog is an extension to Prolog, so the basic ways of thinking and problem-solving techniques are very similar in the two languages. The actual differences are strongly dependent on the application field. The two main application fields of Prolog are the following: AI problems; relational database systems (RDBS).

In the case of AI problems the DAP Prolog programmer uses many of the techniques of the Prolog programmer, but substitutes sets for lists whenever the order of elements is irrelevant to the problem and the structure of the elements is homogeneous.

In the case of RDBSs the DAP Prolog programmer thinks in terms of sets rather than individual binding values, making the difference between the languages a fundamental one is this field.

The introduction of arrays into Prolog makes DAP Prolog a candidate for numerical as well as symbolic programming, where Prolog is inadequate.

Since Prolog is inherently list-oriented – that is, inherently sequential – implementing Prolog on an SIMD machine is not attractive. Implementing DAP Prolog is much more effective, since sets and arrays can be handled in parallel on a machine such as the DAP.

## REFERENCES

1. N. Ito *et al.*, Data-flow based execution mechanism of parallel and concurrent Prolog. *New Generation Computing* 3, 15–41 (1985).
2. P. Kacsuk, Wavefront method for parallel unification in proputer array. 5th Symposium on Microcomputer and Microprocessor Applications, Budapest (1987).
3. S. Taylor, *et al.*, Logic programming using parallel associative operations. *Proceedings of the 1984 International Symposium on Logic Programming* 58–68 (1984).
4. A. Ciepielewski, S. Haridi and B. Hausman, Performance evaluation of a storage model for OR-parallel execution of logic programs, *Symposium on Logic Programming, Salt Lake City* (1986).
5. Y. Sohma, *et al.*, A new parallel inference mechanism based on sequential processing. IFIP TC-10 Work, Conference on Fifth Generation Computer Architecture, Manchester (1985).

6. D. De Groot, Restricted AND-parallelism. *Proceedings of the FGCS 84*, 471–478 (1984).
7. M. V. Hermenegildo, An abstract machine for restricted AND-parallel execution of logic programs. *Proceedings of the 3rd International Conference on Logic Programming, London*, 25–39 (1986).
8. P. Borgwardt. Parallel Prolog using stack segments on shared memory multiprocessors. *Proceedings of the 1984 International Symposium on Logic Programming*, 2–11 (1984).
9. J. Crammond, A comparative study of unification algorithms for OR-parallel execution of logic languges. *IEEE Transactions on Computers*, **34** (10) 911–917 (1985).
10. D. S. Warren, *et al.*, Executing distributed Prolog programs on a broadcast network. *Proceedings of the 1984 International Symposium on Logic Programming*, 198–202 (1984).
11. G. H. Pollard, Parallel execution of horn clause programs. *Ph.D. Thesis*, University of London, Imperial College (1986).
12. S. Umeyama and K. Tamura, A parallel execution model of logic programs. *Proceedings of the 10th Symposium on Computer Architecture*, 349–355 (1983).
13. R. Hasegawa and M. Amamiya, Parallel execution of logic programming based on dataflow. *Proceedings of the ICOT Conference*, 507–516 (1984).
14. P. Kacsuk. Some approaches to parallel implementations of Prolog. *Proceedings of IFIP '86 Congress*, 803–809 (1986).
15. R. W. Hockney and C. R. Jesshope, *Parallel Computers. Adam Hilger, London* (1981).
16. DAP 500: Fortran-Plus Language, *man003.01*. Active Memory Technology Ltd.
17. K. L. Clark, F. McCabe and S. Gregory, IC-Prolog language features. In *Logic Programming*, edited K. L. Clark and S. A. Tarrilund, pp. 253–266. Academic Press, London.
18. M. Ratcliffe and P. Robert, *PEPSy: A Prolog for Parallel Processing*. ECRC Technical Report CA-17 (1986).
19. K. L. Clark and S. Gregory, *PARLOG: Parallel Programming in Logic*. Research Report DOC84/4, Department of Computing, Imperial College, University of London.
20. E. Y. Shapiro. *A Subset of Concurrent Prolog and its Interpreter*. Technical Report TR-003, Institute for New Generation Computer Technology, Tokyo (1983).
21. S. F. Reddaway, The DAP approach. In *Infotech* state of the art Report *Supercomputers*, vol. 2 pp. 309–329 (1979).
22. P. Kacsuk. The design philosophy of DAP Prolog. Third Conference on Vector and Parallel Processors in Computational Science, Liverpool (1987).
23. F. Giannesini, H. Kanoui, R. Pasero and M. van Caneghem, *Prolog*. Addison-Wesley, New York.