Exploiting Implicit Parallelism of Logic Languages with the SAM

Giancarlo Succi

DIST-Università di Genova
Genova, Italia

Abstract

Many researchers have been trying to use the implicit parallelism of logic languages parallelizing the execution of independent clauses. However this approach has the disadvantage of requiring a heavy overhead for processes scheduling and synchronizing, for data migration and for collecting the results. In this paper a different approach is proposed, the data parallel one. The focus is on large collections of data and the core idea is to parallelize the execution of element-wise operations. The target language is SEL, a Subset Equational Language. An abstract machine for it, the SAM (Subset Abstract Machine), is outlined, which, under certain points of views, belongs to the WAM family. The data parallel structure of the SAM and of its instruction set is here explained and some examples of how it works are given. The SAM adds to the WAM structure the Active Memory which subsumes the idea of having available a virtually infinite number of processors. The role of abstract analyzers is then explained and an integrated scheme with process parallelism is outlined. Eventually some conclusions are drawn and it is presented the plan for the future research.

Toward a Parallel Implementations of Logic Languages

One of the biggest appeals of logic programming languages is what is commonly referred to as “Implicit Parallelism”; i.e. the parallelism a compiler can automatically, and easily, identify and exploit, since no explicit constraint on the execution order is posed in the abstract logic evaluation scheme. Several attempts have been made for using this property in order to design and implement a language which can fully exploit the (parallel) architecture of its target machine, possibly without resorting to ad-hoc constructs and/or annotations.

Various forms of parallelism has been evidenced, such as and-, or- and stream- parallelism (and, of course, their combination), but nevertheless the results that have been achieved are not as expected. Probably the initial goals were set too high, but anyway we claim that too much attention has been devoted to the most “expensive” form of parallelism, expensive in terms of time and space overhead required for communications, synchronizations and processes management.

On the other side, very little attention has been given other simple forms of intrinsic parallelism, which could have clearer and simpler definitions, and that can due to more straightforward implementations. This paper analyzes the data parallel approach for this problem.

Data Parallelism in Declarative Programming

We can divide the intrinsic parallelism of logic programs in two classes:

- process parallelism,
- data parallelism.

Process parallelism is the form of parallelism which parallelizes the execution of independent parts of a program, usually with the following approach:

1. identify closures which can be executed independently,
2. decide whether to execute them or not,
3. schedule the ones we have decide to execute on the processors,
4. collect the results.

Using this classification, and-, or- and stream- parallelism fall in this class, despite at a first sight they seem to act quite differently one from another. It is evident that this approach have some intrinsic limitations since it requires quite a lot of communications between processes for exchanging data and synchronizing, a lot of time for forking and joining processes and a lot of bookkeeping to have a consistent execution. Furthermore there is the undeniable problem of how to choose what processes should run on what processors.

A different design can be devised and a good candidate for it seems data parallelism, which is almost...
unexplored within the field of logic programming. It can be described by the following approach:

1. identify the collections of objects globally handled by our program,

2. decide how to spawn them onto the available processors,

3. manipulate them applying, as much in parallel as possible:
   - “element-wise” operators,
   - filters,
   - “folding” operators;

in order to obtain either new collections or just scalar elements.

This approach seems quite appropriate for an implementation on both MIMD parallel architectures and the SIMD ones, like the Connection Machine and it overcomes most of the limitations of process parallelism.

This design requires a language suitable for the representation of collections of objects and an effort of programmers to adapt their mind to this new paradigm. The question is then which kinds of languages and of collections we should use and how we should spawn the collections onto the available processors.

Sets and SEL

The target language for our data parallel approach is SEL, the Subset Equational Language developed by Jayaraman et al. [JN88] at UNC/Chapel Hill and at SUNY/Buffalo. This language handles sets in a clean, neat and simple way. Choosing sets as the core collection has also the advantage that lots of people have experience from many different fields in representing problems as relations between sets.

A SEL program is a sequence of two kind of assertions: 
- equational assertions of the kind \( f(\text{terms}) \equiv \text{expr} \).
- subset assertions of the kind \( f(\text{terms}) \supseteq \text{expr} \).

The meaning of this assertions is:

- equational assertion: the function \( f \) applied to the ground instances of \( \text{terms} \) is equal to the corresponding ground instances of \( \text{expression} \);
- subset assertion: the function \( f \) applied to the ground instances of \( \text{terms} \) contains the corresponding ground instances of \( \text{expression} \).

The language incorporates the collect all assumption for subset assertions, which states that the result of a function application to ground terms is the union of all the subsets obtained by all the subset assertions matching the ground terms with all the possible matching. We do not go into details here, and for a complete description of the language [Suc91] can be consulted.

Some examples of SEL programs can help understanding our approach. The first program we examine is aimed to compute the sets of the squares of a given set.

\[
\text{squareSet}(\{x\}) \text{ contains } \{x \times x\}.
\]

Here it is present a remarkable feature of SEL, i.e., the multiple matching: since no order is imposed over the elements of a set, a matching of the kind \( \{x\} \) produces the matching of \( x \) with all the elements of the argument set; therefore, by the collect all assumption, the result is the set containing the squares of all the elements.

A data parallel implementation on a SIMD architecture, for instance, can perform this operation in just one shot: if the argument set is already distributed among the processors what it is needed is just to ask each processor to square the element stored on it (and this can be done in parallel) plus some extra (constant time) bookkeeping.

In the same way it behaves the cartesian product of two sets:

\[
\text{cartProd}(\{x\}, \{y\}) \text{ contains } \{(x,y)\}.
\]

Here we have two nested set mappings, but the general philosophy is the same, and so can be the implementation.

Also more complicate patterns can be handled in this way, like:

\[
\text{perms}(\{\}) = \{\emptyset\}.
\]

\[
\text{perms}(\{x|t\}) \text{ contains } \text{distr}(x,\text{perms}(t)).
\]

\[
\text{distr}(x,\{t\}) \text{ contains } \{x|t\}.
\]

which determines all the possible permutations of the elements of a set. In this case the computation proceeds first generating all the sets matching the pattern in linear time (assuming to have enough available processors, otherwise we need some sort of virtualization) and then applying to all the sets distr.

Filters can be implemented with this approach too:

\[
\text{filter}(\{x\}) \text{ contains if } p(x) \text{ then } \{x\} \text{ else } \{\}.
\]

The function filter selects the elements of its argument set that satisfy the predicate p; again we can have a data parallel implementation in just one shot, provided that we have enough processors.

The Subset Abstract Machine

The implementation of SEL is divided in two phases [JN88]:
- the development of a compiler targeted to an abstract machine,

- the implementation of the abstract machine on the real architecture.

The abstract machine is called SAM, Subset Abstract Machine. It belongs to the WAM [AK90] family since its general structure resembles quite a lot that of the WAM; however it does not need full unification capabilities, therefore there is no need of the "trail" and faster store and match instructions replace the unify ones. The SAM can be viewed as a sister of the SEL-WAM [Nai88] from which it inherits most of the implementation strategies, which are extended with environment trimming, table of constants and the capability of handling functors.

![Diagram of SAM structure](image1)

**Figure 1: General Structure of the SAM**

Figure 1 outlines the general structure of the SAM: in addition to the standard 4 components - heap, stack, push-down list and processor - there is the Active Memory, a memory whose cells both store data and perform computations. Its aim is to hold the sets so that data parallel operation can be executed on them, as it is explained later.

![Active Memory diagram](image2)

**Figure 2: The Active Memory**

Figures 2 and 3 detail the structure of the AM: it is a multidimensional array of cells where each cell is composed by three elements: a processor, a set of registers and memory. The memory is organized in two parts: a stack, for performing local computations, and a region for keeping set elements, which are stored using binding techniques: when there are some sets differing one-another only for few elements, it may be useful to store in the active memory a superset of all of them, the base, and to represent each of the original sets (called remote sets) with a bitvector (a 1 in its i-th position means that the i-th element of the base belongs to the remote set, and a 0 that it does not).

![Diagram of a Cell structure](image3)

**Figure 3: Structure of a Cell**

![Diagram of set mapping](image4)

**Figure 4: set2 is obtained mapping set1 through f**

![Diagram of remote set computation](image5)

**Figure 5: remote2 is computed filtering remote1 with p**

The SAM takes advantage of three main situations to exploit the data parallelism of problems, viz. when there are mappings of one set into another, when there are filters applied to a set, and when there are foldings of a set in a single element. When there is a map of one set into another, for instance in the set theoretic definition:

$$set2 = \{ f(x) : x \in set1 \}$$

it is possible to compute set2 in one shot applying f to all the elements of set1 in parallel. Likewise if there is a filter like:

$$remote2 = \{ x : x \in remote1, p(x) \}$$
can be computed in constant time. Foldings are definitions of the kind:

\[ f(\{\} \}) = k \]
\[ f(\{x|t\} = z(x, f(t)) \]

Here it is possible to perform a tree-like computation in order to determine the result, as it is shown in figure 6, where \( z \) is applied to \( a \) and \( b \), to \( c \) and \( d \), to \( e \) and \( f \), to \( g \) and \( h \) and to \( i \) and \( l \) at once, and then to each pair of results, and so on, until there is only one element left. Note that this last class of operations is not deterministic, since no order is imposed on the elements of sets, e.g., the (pseudo) function \( \text{nonDet} \):

\[ \text{nonDet}(\{\}) = 0 \]
\[ \text{nonDet}(\{x|t\}) = \text{minus}(x, \text{nonDet}(t)) \]

applied to the set \( \{1, 2, 3\} \) can give 0, 2 or 4 as result, depending on which matching we choose. However [MS89] demonstrated that if the folding function is commutative and associative the result is the same no matter of the matching; hence since \( \text{plus} \) is both associative and commutative, the result of applying \( \text{det} \):

\[ \text{det}(\{\}) = 0 \]
\[ \text{det}(\{x|t\}) = \text{plus}(x, \text{det}(t)) \]

\( \{1, 2, 3\} \) is always 6.

The Role of the Active Memory

Up to few years ago researchers were trying to limit the usage of primary memory as much as possible since its availability was rather scarce; nowadays none would regard it as a big problem toward the design of efficient architecture. The philosophical ideal behind the active memory is that the same situation is likely to happen for processors, and the current trend in massive parallel architectures strongly supports this point of view. Therefore in almost all the paper no concern is placed on the size of the AM, assuming to be able to cope with this problem, possibly with some sort of virtualization. However, the section about abstract analyzers raises the issues on how it is possible to properly allocate AM space and to reuse wasted space; this is not contradictory with the assumption since almost the same happens with the primary memory, which is regarded as infinite during most of the phases of the design of compilers and then is handled by ad hoc modules (such as the garbage collector and again abstract analyzers).

Currently it is under design the implementation of virtualization mechanisms for the Connection Machine implementation of the SAM, which is based on the CM data vault.

Data Parallel Instruction Set

The instruction set of the SAM has been designed along two main guidelines: obviously the first is exploiting the data parallel construct of the language and the second is trying to maintain the global design as simple as possible in order to ease the implementation of the abstract machine on different kinds of architectures. While the instructions devoted to equational assertions has changed only slightly from the one of the SEL-WAM [Jay91], the one concerning subset assertions has been entirely revised.

Instruction Set for Equational Assertions

As we said little has been changed for equational assertions from the SEL-WAM; a minor change has been introduced to speed up the operation of choosing the (equational) clause to apply: it has been devised a kind of clause indexing strategy based on the analysis of not only the structure of the first argument but also of the other arguments and therefore we designed the instruction:

\[ \text{switch on ground expr <expr> var list set functor constant} \]

which on the basis of the kind of <expr> jumps to the proper label or fails.

A remarkable modification has been performed for equational assertions dealing with sets, trying to identify and to take advantages of the implicit data parallelism. In order to clarify the analysis, assertions can be divided in two classes:

- those generating a new set, usually of the kind:

\[ f(\{x|t\}) = \{h(x,t) | m(x,t)\} \]

- those producing a single element (which can be a set, but not necessarily) usually of the kind:
\[ f(\{x|t\}) = h(p(x,t),m(x,t)). \]

Under certain circumstances simple analysis can produce an optimized SAM code for both classes. Consider, for instance, when the former has the shape:

\[ f(\{x|t\}) = \{h(x)!f(t)\}. \]

we can rewrite this pattern as

\[ f(\{x|\}) \text{ contains } \{h(x)\}. \]

which can be efficiently implemented as it is explained in the subset assertion section of this paper, since it is a mapping between two sets through the function \( h \). On the other side when the latter is of the kind:

\[ f(\{\}) = z. \]
\[ f(\{x|t\}) = h(p(x),f(t)). \]

there is the opportunity for folding, as it was explained in section 3. For this purpose it has been designed the fold-like SAM\(^2\) instructions:

\[
\text{fold}_f Zs \ Zr \ h/3 \ f/2 \\
\text{fold} Zs \ Zr \ h/3
\]

which perform the computation on the set \( Zs \) storing the result in \( Zr \) using as zero \( Zs \); the difference between the two is that in \( \text{fold} \ f/2 \) is the identity function.

**Instruction Set for Subset Assertions**

The compilation of subset assertions for the SAM differs from the one targeted to the SEL-WAM since the SAM is aimed to set up a "nice" environment for "simple" porting on (data) parallel machines.

The first difference concerns the collect instruction which is substituted by union, funion, insert and finsert; it is not just a renaming since what they do is quite different:

\[ \text{union} Zs \ Zb \text{ takes the union of two sets storing the result in the first one, i.e., } Zs := Zs \cup Zb \text{ and performs the duplicate check; } \]

\[ \text{funion} Zs \ Zb \ Zf \text{ is like union apart from the fact that the union is performed only if } Zf \text{ is true, i.e., } \text{if } Zf \text{ then } Zs := Zs \cup Zb \]

\[ \text{insert} Zs \ Zb \text{ inserts the element } Zb \text{ in the set } Zs, \text{ i.e., } Zs := Zs \cup \{Zb\} \text{ performing the duplicate check; } \]

\[ \text{finsert} Zs \ Zb \ Zf \text{ is like insert apart from the fact that the insertion is performed only if } Zf \text{ is true, i.e., } \text{if } Zf \text{ then } Zs := Zs \cup \{Zb\}. \]

These four new instructions have a critical role for performing mappings and filters since they can act in parallel on all the AM cells of a set.

Regarding pattern matching, two can be its purposes inside subset assertions:

(i) to identify an element of a given set,

(ii) to iterate over the elements of a set.

It is possible to distinguish at compile time these two situations because in the first case we deal with a ground element, like in:

\[ \text{lookFor3}(\{x|\}) = \text{true}. \]

or an already matched variable, like:

\[ h(x, \{x|\}) \text{ contains } f(x). \]

while in the latter there is a free variable. For the first case we can use the instructions:

\[ \text{match.set.element} Zm \ Zs \ Zb \]
\[ \text{match.set.element.rem} Zm \ Zs \ Zr \ Zb \]

which match the element \( Zm \) against the set \( Zs \) storing in \( Zb \) the boolean result of the matching and in \( Zr \) the remainder of the set (only for \( \text{match.set.element.rem} \)).

**Mapping Instructions**

Most of the operations which iterate over the elements of a set have the shape of generating a new set whose elements are functions of the elements of the original one: this is the standard definition of mappings which was introduced in section 3.

In all the mapping instructions it is present an index which can be used during the computation in case the number of virtual processors available is lower than the number of elements of the set to map; the index is theoretically useless in the hypothesis of an infinite AM, but it is useful to have an efficient single processor implementation and it can turn out to be useful also in the multiprocessor one, therefore the index is kept in all the cases, so that the structure of the SAM is the same in all the implementations.

These mappings can be divided in three categories, depending on the space they need to perform the matching process (which is a multiple matching).

**Constant space**: this is the case of patterns of the kind:

\[ f(\{x|\}) \text{ contains } g(x). \]

in which we obviously need only constant space since we just scan the set; for this case we use the machine instructions

---

\(^2\)Adopting the WAM convention of calling \( Ai \) the argument registers, \( Xi \) the temporary ones and \( Yi \) the permanent ones; furthermore when it is not known whether the registers are temporary or permanent they are called \( Zi \).
map_over Za Zi Zm end
end_map_over Zi Zm start

which have the following behavior:

- map_over Za Zi Zm end: produces the (possibly parallel) analysis of the set pointed by Za using Zi as index, storing in Zm the value pointed by Zi and jumping to end if the set is empty;
- end_map_over Zi Zm start: increments the scanner Zi and updates Zm consistently; if the increment succeeds (i.e., we have not yet examined all the set) the execution jumps back to start else it goes to the next instruction.

**Quadratic space**: when we have patterns of the kind:

\[ f(x|t) \text{ contains } g(x,t). \]

in general we need quadratic space, since we need to build \( n \) copies of the set, being \( n \) the cardinality of the set. The virtual machine instructions for handling this situation are:

map_generating_copy Za Zi Zm Zc end
end_map_generating Za Zi Zm Zc start

which have the following behavior:

- map_generating_copy Za Zi Zm Zc end: here Za is the argument set \( \{x|t\} \), Zi to \( x \) and Zc to \( t \); Za contains what is pointed by Zi. If the argument set is empty the execution jumps to the label end, else a copy of it is generated without an element and the execution goes to the next instruction.
- end_map_generating Za Zi Zm Zc begin: if we have not yet completed the analysis of the argument set, Zi and Zc are differently matched against \( \{x|t\} \), Zm is properly updated and the execution jumps to the label start, else we go to the next instruction; Za is added as parameter to speed up the creation of the new pattern.

**Linear space**: patterns of this latter case need only linear space for matching when the “remainder” of the set \(-t\) is not used in the answer since in this situation we can destructively update it. The instructions are:

map_overriding_copy Za Zi Zm Zc end
end_map_overriding Za Zi Zm Zc start

with the following behavior:

- map_overriding_copy Za Zi Zm Zc end is almost identical to map_generating_copy;
- end_map_overriding Za Zi Zm Zc start differs from end_map_generating only in the destructively modification of the set pointed by Zc.

Simple abstract analyzers can be used to determine when this couple of instructions can be used.

A sample compilation chunk can help understanding this new design. Consider the following code for the set of the squares of a given set:

\[ \text{squareSet}(\{x|\}) \text{ contains } \{x^2\}. \]

the corresponding SAM instructions are:

\[
\text{squareSet}/2:
\begin{align*}
\text{allocate} & \quad \text{get set } A1 Y1 \\
\text{get_variable} & \quad A2 Y2 \\
\text{map_over} & \quad Y1 Y3 Y4 end \\
\text{begin} & \quad \text{put_value} Y4 A1 \\
\text{put_value} & \quad Y4 A2 \\
\text{put_variable} & \quad Y5 A3 \\
\text{call} & \quad \text{mult/3} \\
\text{insert} & \quad Y2 Y5 \\
\text{end_map_over} & \quad Y3 Y4 begin \\
\text{deallocate} & \quad 
\end{align*}
\]

**Filtering Instructions**

Certain classes of subset assertions can be defined as filters, e.g., they select which elements of a given sets belong to another one. Their role in the SAM philosophy was discussed in section 3. Their general form is:

\[ p(\{x|t\}) \text{ contains if } q(x,t) \text{ then } \{g(x,t)\} \text{ else } \. \]

We can define three kind of filterings in a similar way as we did for the mappings apart that here we need a boolean value telling us whether to store the computed element in the set and the usage of insert instead of insert. Therefore the compilation of this assertion:

\[ p(\{x|\}) \text{ contains if } q(x) \text{ then } \{g(x)\} \text{ else } \. \]

is the following:

\[
p/2:
\begin{align*}
\text{... ...} \\
\text{map_over} & \quad Za Zi Zm end \\
\text{begin} & \quad \text{put_value} Zm A1 \\
\end{align*}
\]
Distributed Pattern Matching

The purpose of pattern matching is to identify portion(s) of a structure comparing it against a template. The WAM uses the PD-list to accomplish this task, however this approach is intrinsically sequential, since the list is a FIFO structure. Had this approach been taken also for the SAM, there would have been the need of sequentializing the computation each time a pattern matching would have been required on the elements of a set, consequently creating a critical bottleneck. A completely different mechanism has been therefore devised.

Whenever there is the need of performing a pattern matching on the elements of a set, a template of the pattern is created in the local stack of each AM cell of the set. This template contains a reference to a free register of the cell for each of its free variables. The operation of matching can then be performed locally in the cell operating on the local set element and on the local template and at the end of it either a local fail flag is set or the registers contains the desired values.

Figure 7: Local stack before the distributed pattern matching of the example.

The operations for creating the template are:

\begin{verbatim}
start.match Za
store.temp.variable Za
store.temp.value Za
store.temp.function f/n
store.temp.set Za
store.temp.list
store.temp.constant c
store.temp.unref
\end{verbatim}

\begin{verbatim}
start.match Za places in the register Za the address of the top of the stack, which is the place where the matching process will start. store.temp.variable Za stores in the stack a reference to the register Za, where the result of the pattern matching will be placed. store.temp.value Za store in the stack a reference to the register Za which was already referred to by another store.temp.variable Za in order to handle nested matching (the situation is analogous to the one of [get/put/unify].value of the WAM).
\end{verbatim}
store_temp functor \( T/n \) stores in the stack the functor \( f \) together with its arity \( n \). store_temp.set \( Za \) places a reference to the register \( Za \) which holds a pointer to a set. store_temp.list sets the template to wait for a list while store_temp.constant \( c \) for the constant \( c \) and finally store_temp.unref is used to handle the “don’t care” _...

![Local Stack Diagram](image)

Figure 8: Local stack after the distributed pattern matching of the example.

Taking this approach the clause select_father can be compiled as:

```
select_father/3:
  get_set A1 X3
  get_variable A2 X4
  store.match X5
  store_temp_functor family/2
  store_variable X6
  store_temp.unref
  map_over.matching X3 X7 X5 end
begin:
  insert X4 X6
  end_map_over.matching X3 X7 begin
end:
  proceed
```

The store_temp prepare the pattern to be matched against inside the mapping storing the template in the stack; the store_temp_variable X6 store in the stack a reference to the register X6 so that at the end of the matching process this register will contain the result. Figure 7 shows how the pattern is built on the local stack of an AM cell.

If father(john,paul) has to be matched against the pattern in figure 7, the execution proceeds as follows:

- the functor father/2 is matched against the content of the stack at the location pointed by X5 and the match succeeds;
- john is matched against the next element of the stack: since it is a reference to an unbound register (X6), john is stored in it;
- paul is matched against the next stack element which is a “don’t care”, therefore the matching succeeds;
- nothing else has to be done: the matching has been successfully completed and X6 contains the desired value, as it shown in figure 8.

Extension to N-ary Assertions

Up to now we have always treated functions as if they were only unary; the extension to n-ary functions consists just in analyzing the arguments left-to-right and producing the SAM code thereafter. Therefore for the set_product assertion:

```
set_product([x1..],[y1..]) contains {pair(x,y)}.
```

being pair/2 a functor, we have the following SAM code:

```
set_product/3:
  get_set A1 X4
  get_set A2 X5
  get_variable A3 X6
  map_over X4 X7 end1
begin1:
  map_over X5 X8 end1
begin2:
  store.functor pair/2 X9
  store_const X7
  store_const X8
  insert X6 X9
  end_map_over X5 X8 begin2
end2:
  end_map_over X4 X7 begin1
end1:
  proceed
```

Here the resulting set is pointed by the register X6.

Performance Analysis of the SAM

The fact that the SAM has a very simple global structure has a strong impact in its performances: the SAM has been tested against the SEL-WAM and the results obtained are excellent, there is a reduction in the execution time that ranges from 50% to 75%. The assertions that has been benchmarked are:

- union(s1,s2) contains s1.
- union(s1,s2) contains s2.

- squareSet([x1..]) contains {x*x}.
- perms(()) contains {}.
- perms([x|t]) contains distr(x,perms(t)).
- distr(x,[y1..]) contains {[x|y]}.

The union is taken between two sets, one whose size is 40 and the other whose size is 70. squareSet is applied to a 73 elements set. The execution times are expressed in milliseconds and are computed on a RISC Sun4 with SunOS 4.0.3.
<table>
<thead>
<tr>
<th></th>
<th>SAM</th>
<th>SEL-WAM</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>10.9</td>
<td>33.1</td>
<td>67%</td>
</tr>
<tr>
<td>squareSet</td>
<td>12.6</td>
<td>51.1</td>
<td>76%</td>
</tr>
<tr>
<td>perms</td>
<td>2862.8</td>
<td>5812.2</td>
<td>51%</td>
</tr>
</tbody>
</table>

Obviously this is not a complete and exhaustive test of the SAM: these are just the first results that have been obtained. A complete analysis should also take into account the performances of other logic languages, like the relational based ones, such as prolog: it is encouraging that lots of cases where Prolog need recursion, even not amenable to last call optimization, in SEL are simply treated taking advantage of implicit iteration without any need of recursion at all, such as for the cartesian product of two collection: the Prolog version is:

cartProd([], []).  
cartProd([H1|T1],[H2|T2],X) :-  
append([pair(H1,H2)],Y,Z),  
cartProd([H1|T1],T2,Z),  
cartProd(T1,[H2|T2],W), append(Z,W,X).
while the SEL version (already given above) is simply:

cartProd([{X1,.},{Y1,.}) contains {pair(X,Y)}.  
with no recursion at all.

Abstract Analysis

Abstract analyzers play an important role in the process of building an efficient implementation of a logic language: their aim is to infer important properties of programs analyzing them without executing. Then ad hoc compilers can take advantage of these informations to produce a faster code. There is already a wide usage of abstract analyzers for computing the "mode" of a variable, for determining whether the same object can be shared or need to be copied, and so on. All these techniques can be extended to this framework. However there are more reasons for partial evaluators here.

It has already noted that the usage of the active memory has to be minimized, since its availability is much lower than that of "standard" memories and that architectural constraints may impose to allocate the active memory cells needed at loading time. Hence it is strictly necessary to design abstract analyzers to determine at compile time good approximations of the objects involved in the executions and of their sizes. In this design they are divided in two classes:

- **object size analyzers**, aimed to determine the sizes of the objects,
- **persistence analyzers**, targeted to compute the lifetime of objects.

A full description of them can be found in [SM91].

Integration with Process Parallelism

The fact that this approach is based on data parallelism and not process parallelism does not mean that this paper claims that process parallelism shouldn't be exploited at all. It would be very interesting to try to couple some limited forms of process parallelism with data parallelism in the framework of SEL. Furthermore SEL seems to be a good candidate for such an integration since it seems quite simple to have a conservative or parallelism, parallelizing the execution of multiple subset assertions matching the same head. The design of the SAM does not require big changes to handle this extended framework: the communications and synchronizations can be confined in the `try me then` and in the `union` instructions.

Conclusion

In this paper it is described the instruction set of the abstract machine we are developing for the data parallel execution of logic languages, the SAM. Presently it has been completed the implementation of the SAM on a Risc Sun4 architecture and it is benchmarked. The very first figures seems promising. In the meanwhile the Connection Machine implementation has started and it is in the early stage the mentioned project for the process parallelism integration. Many are the open problems, such as the ones about the best object allocation scheme for sets and the ones about abstract analyzers.

Acknowledgments

The author thanks Prof. Jayaraman, since most of the ideas of this paper are due to his outstanding supervision at SUNY/ Buffalo.

References


[MS89] J. Marino and G. Succi. Data Structure for the Parallel Execution of Functional Languages. In G. Gries and J. Hartmanis, editors,

