Proceedings of the Workshop on Parallelism and Implementation Technologies for (Constraint) Logic Programming Languages

In conjunction with the First International Conference on Computational Logic (CL2000)

London, England

July, 2000

Inês de Castro Dutra
COPPE – Department of Systems Engineering and Computer Science
Federal University of Rio de Janeiro
Brazil

Table of Contents

Introduction

One of the main areas of research in logic programming is the design and implementation of sequential and parallel (constraint) logic programming systems. This research goes broadly from the design and specification of novel implementation technology to its actual evaluation in real life situations. In the continuation of the series of workshops on *Implementations of Logic Programming Systems*, previously held in Budapest (1993), Ithaca (1994), Portland (1995), Bonn (1996), Port Jefferson (1997), Manchester (1998), and Las Cruces, the CL2000 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages provided a forum for ongoing research on the design and implementation of sequential and parallel logic programming systems.

Topics included:

- standard and non–standard sequential implementation schemes (e.g., generalization/modification of WAM, translation to C, etc.);
- implementation of parallel logic programming systems;
- balance between compile–time effort and run–time machinery;
- techniques for the implementation of different declarative programming paradigms based on, or extending, logic programming (e.g., constraint logic programming, concurrent constraint languages, equational–logic languages);
- performance evaluation of sequential and parallel logic programming systems, both through benchmarking and using real world applications;
- other implementation–related issues, such as memory management, register allocation, use of global optimizations, etc.

The workshop was held in conjunction with the First International Conference on Computational Logic (CL2000), which took place in London England, from July 24th to July 28th, 2000.

The workshop lasted for two half day sessions (July 24th-25th), and took place in parallel with the other activities in the conference, thus allowing the participants to become familiarised with recent research on Logic Programming.

We were very fortunate to have quite a few interesting communications, ranging through widely different subjects and giving a broad coverage on current research in both sequential and parallel logic programming. In what regards sequential research in logic programming, the communications focused on compilation of large disjunctions, tabling, and constraint-based inference tools. In what regards parallelism, the focus of interest were on distributed systems and multicast protocols, VRML code generation, answer sets programs, scheduling, granularity analysis tools, new parallel computational models for modern parallel architetcures (clusters of PCs) and performance evaluation.

The workshop organizers would like to thank all authors that chose to submit

their work to this workshop, for their cooperation in making this document possible. We would also like to thank the referees that helped to improve the quality of the final accepted papers. Finally we would like to thank the CL2000 organisers for their support throughout the workshop.

Inês de Castro Dutra, Vítor Santos Costa
COPPE/Department of Systems Engineering and Computer Science
Federal University of Rio de Janeiro, Brazil
Email: {ines,vitor}@cos.ufrj.br

Fernando Silva
LIACC, Universidade do Porto, Portugal
Email: fds@ncc.up.pt

Enrico Pontelli
LLDAP, Department of Computer Science, New Mexico State University, NM, USA
Email: epontell@cs.nmsu.edu

Manuel Carro
Department of Computer Science
Technical University of Madrid, Spain
Email: mcarro@fi.upm.es

Workshop Programme

July 24th, Monday afternoon, 16:30 - 17:50

16:30 - 16:50 *Incremental Stack-Splitting*
K. Villaverde, H. Guo, E. Pontelli, G. Gupta

16:50- 17:10 *A New Tabling Scheme with Dynamic Reordering of Alternatives*
H. Guo, G. Gupta

17:10 - 17:30 *The RAINBOW: Towards a Parallel BEAM*
R. Lopes, F. Silva, V. S. Costa, S. Abreu

17:30 - 17:50 *Parallel Evaluation of Answer Sets Programs: Preliminary Results*
O. El Khatib, E. Pontelli

July 25th, Tuesday afternoon, 14:30 - 16:30

14:30 - 14:50 *GRANLOG: an integrated model to Granularity Analysis in Logic Programming*
C. Geyer, J. Barbosa, P. Kayser, I. Dutra

14:50 - 15:10 *Towards the OAR Language and Computational Model*
S. Abreu

15:10 - 15:30 *Compiling large Disjunctions*
B. Demoen, G. Janssens, H. Vandecasteele

15:30 - 15:50 *Assumption Grammars for Generating Dynamic VRML Pages*
A. Gupta, P. Tarau

15:50 - 16:10 *Multicast Protocols for Jinni Agents*
S. Tyagi, P. Tarau

16:10 - 16:30 *Implementing cTI: a constraint-based left-termination inference tool for LP*
S. Hoarau, F. Mesnard, U. Neumerkel

Incremental Stack-Splitting

K. Villaverde, H. Guo, E. Pontelli, G. Gupta

Laboratory for Logic, Databases and Advanced Programming
Department of Computer Science
New Mexico State University
{kvillave,hguo,epontell,gupta}@cs.nmsu.edu

Abstract

Incremental stack-copying was used to implement OR-parallel Prolog efficiently in the MUSE system. The idea of incremental stack-copying is to only copy the difference between the WAM data areas of two processors, instead of copying them entirely. In order to further reduce the communication during stack-copying and make its implementation efficient on distributed memory machines, a new technique, called stack-splitting, has recently been proposed. In this paper, we describe a scheme to implement incremental stack-splitting for OR-parallelism on distributed memory machines. The incremental stack-splitting scheme presented is based on a procedure which labels parallel choice-points and then compares the labels to determine the incremental WAM areas to be copied. Furthermore, we describe a scheduling scheme for the incremental stack-splitting strategy. Both the incremental stack-splitting and the scheduling schemes described are currently being implemented in the PALS system. Preliminary results from this implementation are also included.

1 Introduction

Incremental stack-copying has been used to implement OR-parallel Prolog efficiently in the MUSE system. The idea of incremental stack-copying is to only copy the difference between the WAM data areas of two processors, instead of copying them entirely. In order to further reduce the communication during stack-copying and make its implementation efficient on distributed memory machines, a new technique, called stack-splitting, has recently been proposed [10]. In this paper, we describe a scheme to implement incremental stack-splitting for OR-parallelism on distributed memory machines. The incremental stack-splitting scheme is based on a procedure which labels parallel choice-points and then compares the labels to determine the incremental WAM areas to be copied. Furthermore, we describe a scheduling scheme which uses the incremental stack-splitting scheme. Both the incremental stack-splitting and the scheduling schemes described are currently being implemented in the PALS system.

The paper is divided into eight sections. Section 2 is a review of the stack-splitting technique. Sections 3 and 4 describe the data structures and the procedure that performs the labeling of parallel choice-points to determine the incremental part to be copied during incremental stack-splitting. Section 5 describes three issues that are fundamental for the correct implementation of incremental stack-splitting. Section 6 describes the scheduling scheme, while section 7 presents preliminary performance figures. Section 8 presents our conclusions.

2 Stack-splitting

Most research related to Or-parallel implementation of logic programming so far has focused on techniques aimed at shared-memory multiprocessors (SMMs). Relatively fewer efforts [9,2,7,6,5] have been devoted to implementing LP systems on distributed memory multiprocessors (DMMs). Out of these efforts only a small number have been implemented as working prototypes, and even fewer have produced acceptable speedups. Existing techniques developed for SMMs are mostly inadequate for the needs of DMMs. In fact, most implementation methods require sharing of data and/or control stacks to work correctly. Even if the need to share data stacks is eliminated—as in *stack-copying*—the need to share the control stack still exists. The control stack can be large, thus degrading the performance on DMMs.

Stack-splitting is a modification of the stack-copying method and is designed to support *Or-Parallelism (OP)* on DMMs. Stack-splitting allows one to avoid sharing of the control stack, thus allowing efficient SMM techniques devised to be implemented on DMMs without excessive performance degradation. Stack-splitting has the potential to improve locality of computation, reduce communication between parallel threads, and improve cache behavior. Furthermore, it allows better scheduling strategies—specifically scheduling on bottom-most choice-point—even in DMM implementations of OP. The fact that stack-splitting is based on simple modifications of efficient techniques that have proved to work on SMMs leads us to believe that it will work better than many approaches previously proposed. A prototype implementation (on SMMs) as well as simulation studies [11] further support our belief. For comparisions with earlier distributed Or-parallel systems, see the original stack-splitting paper [10].

Stack-splitting can be efficiently implemented on SMMs as well as DMMs. On SMMs we anticipate that the method's performance will be comparable to stack-copying [1]—an already developed prototype shows that this method's performance is, in fact, better [10]. The real benefit of the method will be apparent on DMMs, on which efficient implementation of OP becomes possible. The ability to reuse the same technology on both SMMs and DMMs is also a key component to allow the development of LP systems on *Clusters of SMMs*, i.e., distributed architectures whose individual nodes are SMMs.

2.1 The Need for a Different Stack-Copying Model

In the traditional stack-copying technique, as implemented in MUSE, backtracking on a shared choice-point requires acquiring exclusive access to the corresponding *shared frame*. Shared frames are associated to each copied choice-point and used to maintain a shared representation of the alternatives available in such choice-point. This solution works fine on SMMs—where mutual exclusion is easily implemented using *locks*. However, on a DMM this process is a source of overhead—access to the shared area becomes a bottleneck [3]. Sharing of information in a DMM leads to frequent exchange of messages and hence considerable overhead. Centralized data structures, such as shared frames, are expensive to realize in a distributed setting. Nevertheless, stack-copying has been recognized as the best representation methodology to support OP in a DMM setting [7,9,2,6,5]. This is because, while the choice-points are shared, at least other data-structures, such as the environment, the trail, and the heap are not. Other environment representation schemes, e.g., the Binding Arrays scheme, have been adapted to implement distributed systems [12]. To avoid the problem of sharing choice-points in distributed implementations, many implementors have reverted back to the *scheduling on top-most choice-point* strategy [7,9]. The reasoning is that untried alternatives of a choice-point created higher up in the or-tree are more likely to generate large subtrees as well as minimize the amount of computation "shared" by different processors. However, if the granularity does not turn out to be large, then another untried alternative has to be picked and a new copying operation performed. In contrast, in scheduling on bottom-most more work could be found via backtracking, since more choice-points are copied during the same sharing operation. Additionally, scheduling on bottom-most is closer to the depth-first search strategy used by sequential systems, and facilitates support of Prolog semantics. Research done on comparing scheduling strategies indicates that scheduling on bottom-most is superior to scheduling on top-most [4]. This is especially true for stack-copying because: (i) the number of copying operations is minimized; and, (ii) the alternatives in the choice-points copied are "cheap" sources of additional work, available via backtracking. However, the shared nature of choice-points is a major drawback for a DMM implementation of stack-copying. The question we consider is: can we avoid sharing of choice-points while keeping scheduling on bottom-most? The answer is affirmative, as is discussed next.

2.2 Split Choice-point Stack Copying Model

In the stack-copying approach, the primary reason why a choice-point has to be shared is because we want to serialize the selection of untried alternatives, so that no two processors can pick the same alternative. The shared frame is locked while the alternative is selected to achieve this effect. However, there are other simple ways of ensuring the same property: perform a splitting of the choice-points, i.e., each processor is given all the alternatives of alternate choice-points (See Figure 1). In this case, the list of choice-points is split between the two processors. We call this operation *choice-point stack splitting* or simply *stack-splitting*.

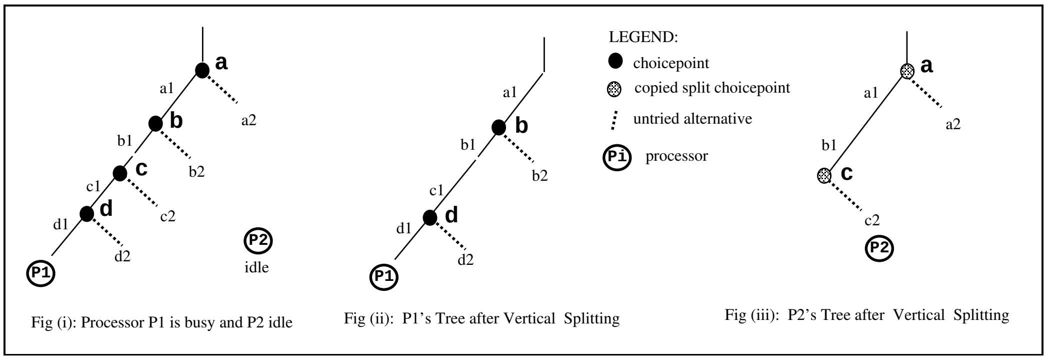


Fig. 1. Splitting of Choice-points

Stack-splitting will ensure that no two processors pick the same alternative. The need for a shared frame, as a critical section to protect the alternatives from multiple executions, has disappeared, as each stack copy has a different choice-point. All the choice-points can be evenly split in this way during the copying operation. The major advantage of stack-splitting is that scheduling on bottom-most can still be used without incurring huge communication overheads. Essentially, after splitting, the different or-parallel threads become fairly independent of each other, and hence communication is minimized during execution. This makes the stack-splitting technique highly suitable for DMMs.

The shared frames in the stack-copying technique are used to maintain global information related to scheduling. The shared frames provide a global description of the or-tree, and each shared frame records which processor is working in which part of the tree. This last piece of information is needed to support scheduling in stack-copying systems—work is taken from the processor that is "closer" in the or-tree, thus reducing the amount of information to be copied. The shared frames ensure accessibility of this information to all processors, providing a consistent view of the computation. However, under stack-splitting the shared frames no longer exist; scheduling and work-load information will have to be maintained in some other way. They could be kept in a global shared area similar to the case of SMMs—e.g., by building a representation of the or-tree—or distributed over multiple processors and accessed by message passing in case of DMMs. The maintenance of global scheduling information represents a problem which is orthogonal to the environment representation. This means that scheduling management in a DMM will anyway require communication between processors.

Shared frames are also employed in MUSE [1] to detect the Prolog order of choice-points, needed to execute order-sensitive predicates (e.g., side-effects, extra-logical predicates) in the correct order. As in the case of scheduling, some information regarding global ordering of choice-points needs to be maintained to execute order-sensitive predicates in the correct order. Thus, stack-splitting does not completely remove the need of a shared description of the or-tree. The use of stack-splitting can mitigate the impact of accessing shared resources—e.g., stack-splitting allows scheduling on bottom-most which, in general, leads to a reduction of the number of calls to the scheduler.

3 Incremental Stack-splitting: Data Structures

During stack-splitting, all WAM data areas except for the code area are copied from the processor giving work to the idle one. Next, the parallel choice-points are split between the two processors. Blindly copying all the stacks every time a processor shares work with another idle processor can be wasteful, since frequently the two processors already have parts of the stacks in common due to previous copying. We can take advantage of this fact to reduce the amount of copying by performing *incremental copying*. In incremental copying, only the difference between the stacks of the processor that is giving work and the idle processor is copied on to the memory areas of the idle processor. Incremental copying has been implemented in the MUSE system with relatively little overhead. This is primarily because all the information needed for performing incremental copying efficiently can be found in the shared frames—the use of shared frames is essential to determine the bottom-most common choice-point between the two processors. In our stack-splitting scheme, there are no shared frames, hence performing incremental stack-copying will incur more overhead due to the communication overhead involved.

In order to figure out the incremental part that only needs to be copied during incremental stack-splitting, parallel choice-points will be *labeled* in a certain way. The labeling procedure is described next.

To perform labeling, each processor maintains a counter. Initially, the counter in each processor is set to *1*. The counter is increased by 1 every time the labeling procedure is performed. When a parallel choice-point is copied for the first time, a label for it is created. The label is composed of three parts:

- processor rank,
- counter, and
- choice-point address.

The processor rank is the rank (i.e., id) of the processor which created the choice-point. The counter is the current value of the counter for the processor. The choice-point address is the address of the choice-point which is being labeled. The labels for the parallel choice-points are recorded in a *label stack* in the order they are created. Also, when a parallel choice-point is removed from the stack, its corresponding label is also removed from the label stack. Initially, the label stack in each processor is set to *empty*. Intuitively, the label stack keeps a record of changes done to the stacks since the last stack-splitting operation. Observe that the choice of maintaining labels in a stack—instead of associating them directly to the corresponding choice-points—has been dictated by efficiency reasons.

Let us illustrate the stack-splitting accompanied by labeling with an example. Suppose process A has just created two parallel choice-points and process B is idle. Process A and B have their counters set to *1* and their label stacks set to *empty*. Then Process B requests work from process A. Process A first creates labels for its two parallel choice-points. These labels have their rank and counter parts as *A:1*. Process A then pushes these labels into its label stack. See Figure 2 (for simplicity in our figures, we do not show the label stack explicitly but show each label rank and counter parts inside the parallel choice-point being labeled). Notice that process A incremented its counter to 2 after the labeling procedure is over.

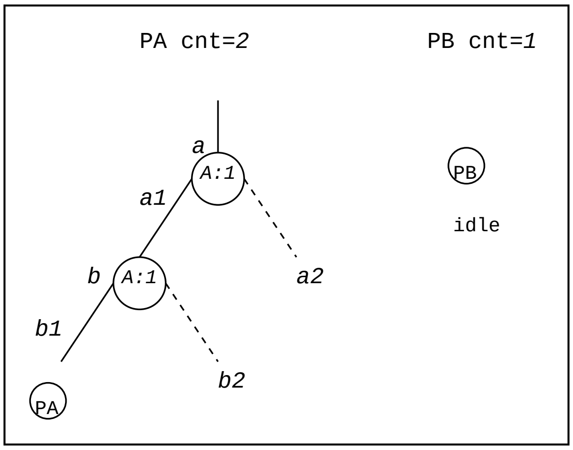


Fig. 2. Process A labels its two parallel choice-points

Stack-copying is done next. Process B gets all the parallel choice-points of process A along with process A label stack. Then, stack-splitting takes place: process A will keep the alternative *b2* but not *a2*, and process B will keep the alternative *a2* but not *b2*. We have designed a new WAM scheduling instruction which is placed in the next alternative field of the choice-point above which there is no more parallel work. This scheduling instruction implements the scheduling scheme described in section 6. Process A keeps the alternative *b2* of choice-point *b* and changes the next alternative field of choice-point *a* to new WAM instruction *schedule* which will take process A into scheduling. Process B changes the next alternative field of choice-point *b* to WAM instruction *trust_fail* to avoid taking the original alternative of this choice-point, keeps the alternative *a2* of choice-point *a*, and changes the next alternative field of the choice-point above *a* to the *schedule* instruction. See Figure 3. Afterwards, process B backtracks, removes choice-point *b* along with its corresponding label in the label stack, and then takes alternative *a2* of choice-point *a*.

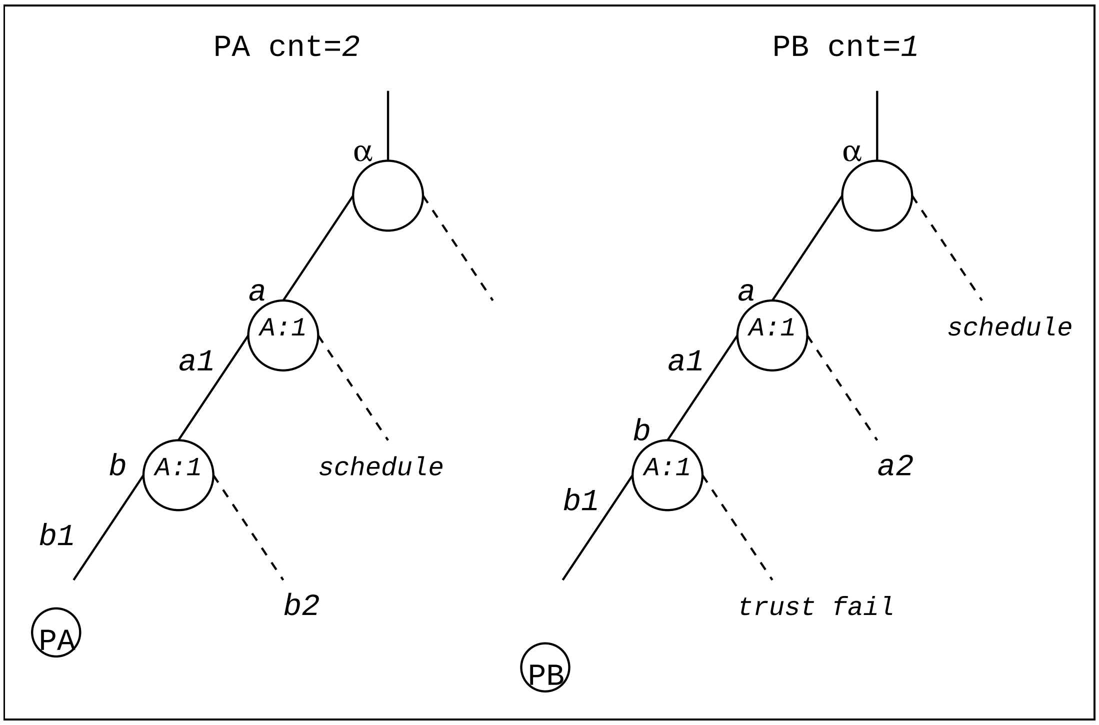


Fig. 3. Process A gave work to process B

Suppose now that process B creates two parallel choice-points and process C is idle. Process C, with its counter set to *1* and its label stack set to *empty*, requests work from process B. Process B first creates labels for its two new parallel choice-points. These labels have their rank and counter parts as *B:1*. Process B then pushes these labels into its label stack. See Figure 4. Notice that process B incremented its counter to *2*.

Now, stack-copying takes place. Process C gets all the parallel choice-points of process B along with process B label stack. Stack-splitting takes place: process B will keep the alternative *d2* but not *c2*, and process C will keep the alternative *c2* but not *d2*. We have modified stack-splitting so that this activity takes place only in the recently created, labeled, and never split parallel choice-points of the process giving work – e.g., *c* and *d*. In other words, the process receiving work should only get new work from these parallel choice-points. Process B keeps the alternative *d2* of choice-point *d* and changes the next alternative field of choice-point *c* to WAM instruction *trust_fail* to avoid taking the original alternative of this choice-point. Process C changes the next alternative field of choice-point *d* to WAM instruction *trust_fail* to avoid taking the original alternative of this choice-point, keeps the alternative *c2* of choice-point *c*, and changes the next alternative field of choice-point *a* to *schedule*. This is illustrated in Figure 5. Process C backtracks, removes choice-point *d* along with its corresponding label in the label stack, and then takes alternative *c2* of choice-point *c*.

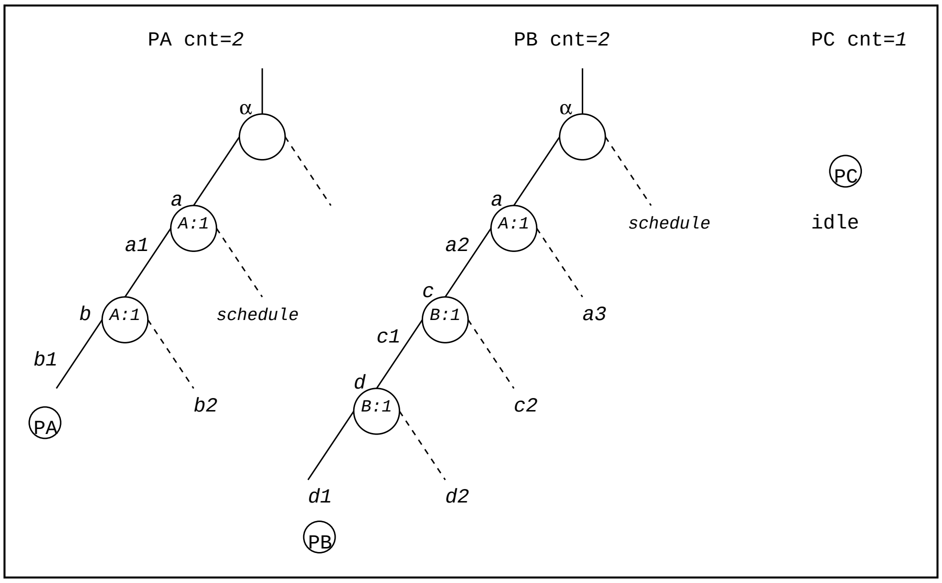


Fig. 4. Process B labels its two new parallel choice-points

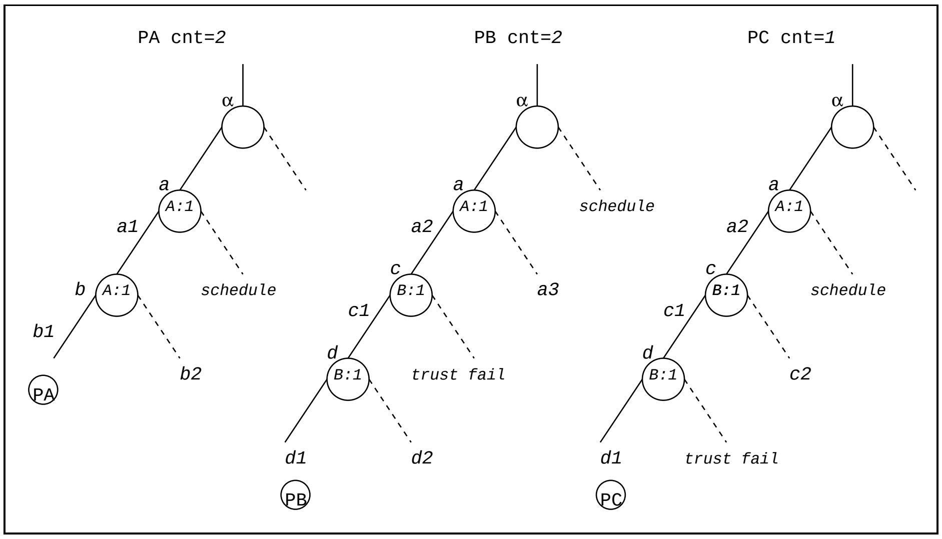


Fig. 5. Process B gives work to process C

4 Incremental Stack-splitting: The Procedure

In this section we describe how the label stacks are used to compute the incremental part to be copied. Assume process W is giving work to process I. Process W will label all its parallel choice-points which have not been labeled before and will push them into its label stack. Process W then increments its counter.

If process I label stack is empty, then stack-copying will need to be performed followed by stack-splitting. Process W sends its complete choice-point stack and its complete label stack to process I. Then stack-splitting will only be performed on the recently created, labeled, and never split parallel choice-points. Process I then tries its new work via backtracking.

However, if process I label stack is not empty then process I sends its label stack to process W. Process W compares its label stack against the label stack from I. Comparison will go from the first label entered in the stacks to the last label entered in the stacks. The objective is to find the last choice-point *ch* with a common label. In this way, processes W and I are guaranteed to have the same computation *above* the choice-point *ch*, while their computations will be different below such choice-point.

If the choice-point *ch* does not exist, then (non-incremental) stack-copying will need to be performed followed by stack-splitting just as described before. However, if choice-point *ch* does exist, then process I backtracks to choice-point *ch*, and performs incremental-copying. Process W sends its choice-point stack starting from choice-point *ch* to the top of its choice-point stack. Process W also sends its label stack starting from the label corresponding to choice-point *ch* to the top of its label stack. Stack-splitting is then performed on the recently created, labeled, and never split parallel choice-points. Afterwards, processor I tries its new work by backtracking.

We illustrate the above procedure by continuing with our example. Suppose process A has created two more parallel choice-points, and process C which has backtracked to choice-point *a* requests work from process A. At this point, process A creates labels for its two new parallel choice-points. These labels have their rank and counter parts as *A:2*. Process A then pushes these labels into its label stack and increments its counter to *3*. Afterwards, process C sends its label stack to process A. Process A compares its label stack against the label stack of process C and finds the last choice-point *ch* with a common label. Above choice-point *ch*, the Prolog trees of processes A and C are equal. Below choice-point *ch*, the Prolog trees of processes A and C differ. See Figure 6.

Now, process C backtracks to choice-point *ch*. Incremental stack copying can then take place. Process A sends its choice-point stack starting from choice-point *ch* to the top of its choice-point stack. Process A also sends its label stack starting from the label corresponding to choice-point *ch* to the top of its label stack. Then, stack splitting takes place. See Figure 7. Process C backtracks to choice-point *e* and takes alternative *e2*.

5 Incremental stack-splitting issues

Three issues that were not discussed above and which are fundamental for the correct implementation of the incremental stack-splitting scheme presented are discussed below.

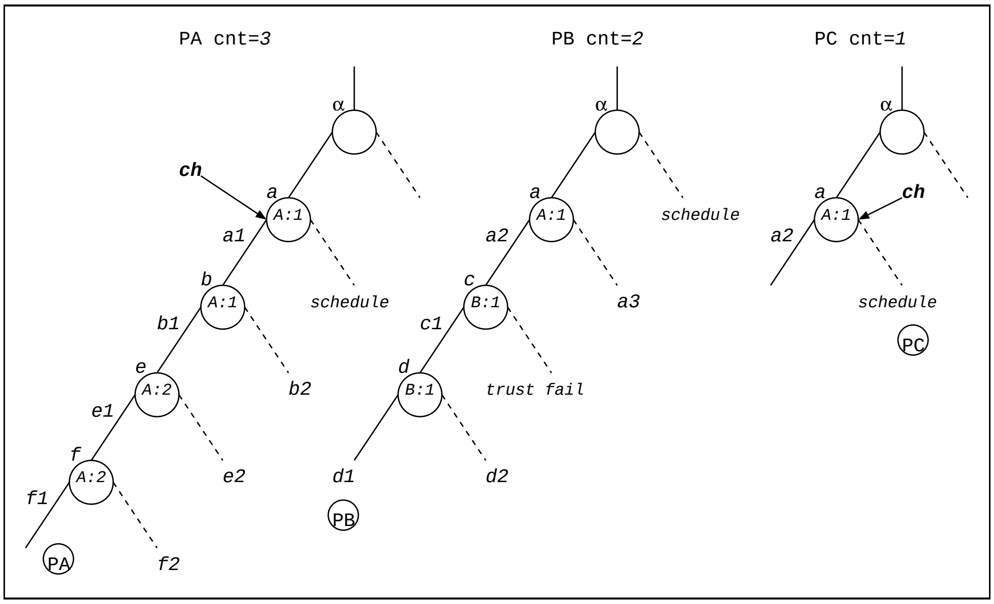


Fig. 6. Process A labels its two new parallel choice-points and compares labels with process C

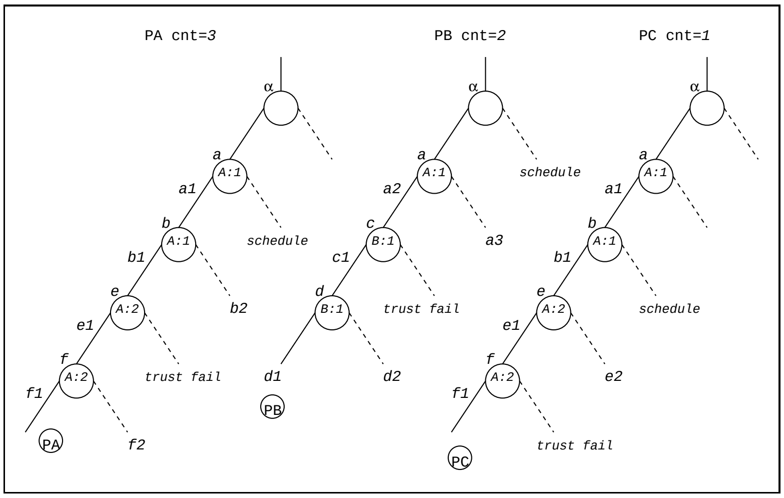


Fig. 7. Process A gave work to process C

5.1 Sequential Choice-points

The first issue has to do with the sequential choice-points that are located among the parallel choice-points that will be split between two processors. If the alternatives of these choice-points are kept in both processes, we may have repeated, useless or wrong computations. Hence, the alternatives of these choice-points should only be kept in one process (e.g., the process that is giving work), or these choice-points should also be splitted. If the alternatives are kept in the process giving work, then the process that is receiving work should change the next alternative field of all these choice-points to the WAM instruction *trust_fail* to avoid taking the original alternatives of these choice-points.

5.2 *Installation process*

The second issue has to do with the bindings of conditional variables which need to be copied too as part of the incremental stack-splitting process. For example, suppose that in our last example before process A gives work to process C, process A created variable *X* before choice-point *ch* was created and instantiated it after choice-point *ch* was created. See Figure 8. We can see that the binding for *X* was not copied during incremental stack-splitting. This is because *X* is a conditional variable which was created before choice-point *ch* and the incremental part of the heap or environment stack that was copied did not contain its binding.

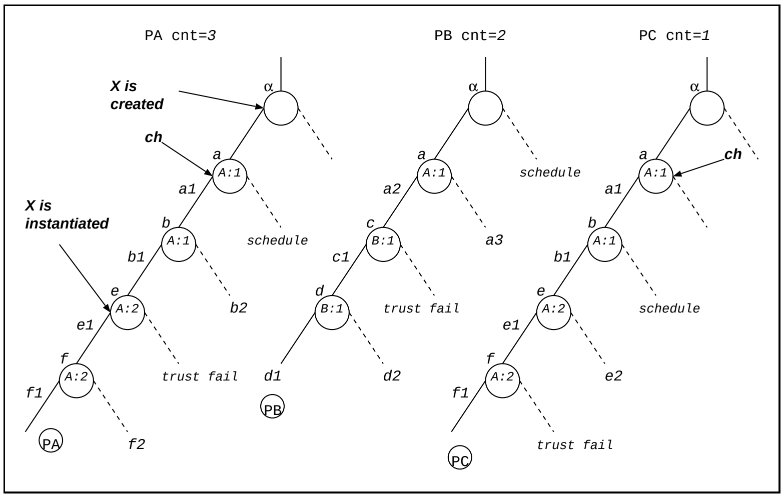


Fig. 8. The binding of conditional variable *X* needs to be copied

Therefore, we need to modify our labeling procedure so that it also performs the installation of the bindings of all conditional variables. This can be done by having a separate valued trail. Every time we trail a (conditional) variable, we store its binding as well in the valued trail. Then when incremental stack copying takes place, we send the incremental part of the valued trail. The process receiving the valued trail will then perform the installation of the bindings.

5.3 *Garbage collection*

The third issue arises when garbage collection takes place. When this happens, relocation of choice-points may also take place. Hence, the labels in our label stack may no longer label the correct parallel choice-points. Therefore, we need to modify our labeling procedure so that when garbage collection on a processor takes place, the label stack of this processor is invalidated. This can be done by just setting its label stack to empty. The next time this process gives work, full stack copying will have to take place. This solution is analogous to the one adopted in the original implementation of the MUSE system [1].

6 Scheduling

Scheduling is an important aspect of any parallel system. The scheduling strategy adopted largely determines the level of speed-up obtained for a particular parallel execution. The main objective of a scheduling strategy is to balance the amount of parallel work done by different processors. Additionally, work distribution among processors should be done with a minimum of communication overhead. These two goals are somewhat at odds with each other, since achieving perfect balance may result in a very complex scheduling strategy with considerable communication overhead, while a simple scheduling strategy which re-distributes work less often will incur low communication overhead but poor balancing of work. Therefore, it is obvious that there is an intrinsic contradiction between distributing parallel work as even as possible and minimizing the distribution overhead. Thus our main goal is to find a trade-off point that results in a reasonable scheduling strategy.

We adopt a simple and fair distributed algorithm to implement a scheduling strategy in the PALS system. We must say here that the scheduling scheme presented is preliminary, approximate, but seems to work on most of the cases. In the future, we will work on implementing a more efficient and effective scheme. Now, we go on to describe this scheduling scheme. A new data structure—the *load vector*—is introduced to indicate the work loads of different processors. The work load of a processor is approximated by the number of parallel choice-points present in its local computation tree. Each processor keeps a work load vector V in its local memory, and the value of $V[i]$ represents the work load of the processor with rank i. Based on the work load vector, an idle processor can request parallel work from other processor with the greatest work load, so that parallel work can be fairly distributed. The load vector is updated at runtime. When stack-splitting is performed, a `Load_Info` message with updated load information will be broadcasted to all the processors so that each processor has the latest information of work load distribution. Additionally, load information is attached with each incoming message. For example: when a `Request_Work` message is received from processor P_1, the value of P_1's work load, 0, can be inferred.

Based on its work load each processor can be in one of two states: *scheduling* state or *running* state. When a processor has some work to do, it is in a running state, otherwise, it is in a scheduling state. A processor that is running, occasionally checks whether there are incoming messages. Two possible types of messages are checked by the running processor: one is `Request_Work` message sent by an idle processor, and the other is `Load_Info` message, which is sent when stack-splitting occurs. The idle processor in scheduling state is also called a scheduling processor. An idle processor wants to get work as soon as possible from another processor, preferably the one that has the largest amount of work. The scheduling processor searches through its local load vector for the processor with the greatest work load, and then sends a Request_Work message to that processor asking for work. If all the other processors have no work, then the execution of the current query is finished and the processor halts. When a running processor receives a Request_Work message, stack-splitting will be performed if the running processor's work load is greater than the splitting threshold, otherwise, a Reply_Without_Work message with a positive work load value will be sent as a reply. If a scheduling processor receives a Request_Work message, a Reply_Without_Work message with work load 0 will be sent as a reply.

The distributed scheduling algorithm mainly consists of two parts: one is for the scheduling processor, and the other is for the running processor. The running processor's algorithm can be briefly described as follows:

```
while (any incoming message) {
   status = the message's status;
   switch (status.TAG) {
   case Send_LoadInfo:
        update the corresponding processors' work load.
        break;
   case Request_Work:
        if (local work load > Splitting Threshold) {
           stack-splitting with the status.SOURCE processor;
           broadcast the updated work load to all the processors;
        }
        else
           reply a message with the tag Reply_Without_Work
                 and the value of its own work load;
        break;
   }
}
```

The scheduling processor's algorithm can be briefly described as follows:

```
while (1) {
   D = the rank of the processor with the greatest work load;
   if (D's work load == 0) exit;   /* The whole work is done */
   Send a Request_Work message to D;
   matched = 0;
   While (!matched) {
      status = the message's status;
      switch (status.TAG) {
         case Reply_With_Work:
              stack-splitting with the status.SOURCE processor;
              update the corresponding work load;
              simulate failure and go to execute the split work;
```

```
                break;
            case Reply_Without_Work:
                if (status.SOURCE == D) matched = 1;
                V[status.SOURCE] = work load of status.SOURCE;
                break;
           case Request_Work:
                reply a message with the tag Reply_Without_Work and
                     its work load 0 to status.SOURCE;
                V[status.SOURCE] = 0;
                break;
            case Send_LoadInfo:
                update the corresponding processors' work load;
                break;
        }
    }
```

7 Implementation and Performance

The stack-splitting procedure has been implemented on top of the commercial ALS Prolog system using the MPI library [8] for message passing. The whole system runs on a Sun Enterprise shared memory machine. While the Sun Enterprise is an SMM, it should be noted that all communication—during scheduling, copying, splitting, etc.—is done using messages. Migration of the system to a truly distributed Beowulf machine (a network of Pentium II nodes connected by a Myrinet Switch) is currently under way.

The results from our stack-splitting system, which has just become operational, are presented in Table 1. This system is based on incremental stack-splitting and uses the scheduling strategy described above. The table shows the timing results in seconds along with speedups (obtained) in parenthesis. Although these results are very encouraging, it should be noted that they are very preliminary since no performance tuning has been done. Note also that for benchmarks for substantial running time (10-Queens and Map-coloring), the speed-ups are quite good, while for programs with smaller running time (8-Queens and Hamilton) the speed-ups deteriorate. This is consistent with our belief that distributed memory implementations should be used for parallelizing programs with substantial running times. For programs with small-running times, there is not enough work to offset the cost of exploiting parallelism using a distributed communication model. Nevertheless, our system is reasonably efficient, given that even for small benchmarks it can produce reasonable speedups.

The system, although still not optimized, introduces a reasonably small overhead with respect to the original sequential ALS Prolog system. Our PALS system, on a single processor, is on average 30% slower than the sequential ALS system.

Benchmark	# Processors			
	1	2	4	8
8-Queens	0.48	0.30 (1.6)	0.18 (2.67)	0.16 (3.0)
10-Queens	9.30	4.98 (1.87)	2.56 (3.63)	1.43 (6.50)
Map Coloring	7.17	4.46 (1.61)	2.28 (3.14)	1.24 (5.78)
Hamilton	1.10	0.70 (1.57)	0.38 (2.89)	0.29 (3.79)

Table 1
Performance Evaluation

8 Conclusions and Future Work

In this paper we proposed a novel scheme to implement incremental stack-splitting for OR-parallelism on distributed memory machines. The incremental stack-splitting scheme presented is based on a procedure which labels parallel choice-points and then compares the labels to determine the incremental WAM areas to be copied. Furthermore, we also describe a simple scheduling strategy for the incremental stack-splitting based implementation. The incremental stack-splitting scheme as well as the scheduling strategy has been implemented on top of the commercial ALS Prolog system, and performance results from this preliminary implementation were reported. Future work includes migrating the system to a Beowulf distributed memory machine, designing a more efficient and effective scheduling strategy, performance tuning of the system, and providing support for side-effects and cuts.

Acknowledgments

We are grateful to Ken Bowen and Charles Houpt for providing us access to the ALS system, and for explaining its intricate details. This work has been partially supported by NSF grants CCR9875279, CCR9900320, CDA9729848, EIA9810732, and HRD9906130, and by a graduate fellowship from the US Department of Education.

References

[1] K.A.M. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in MUSE. *International Journal of Parallel Programming*, 1991. 19(6):445–475.

[2] L. Araujo and J. Ruz. A Parallel Prolog System for Distributed Memory. *J. Logic Programming*, 1998.

[3] H. Babu. Porting muse on ipsc860. Master's thesis, New Mexico State Univ., 1996.

[4] T. Beaumont and D.H.D. Warren. Scheduling Speculative Work in Or-Parallel Prolog Systems. In *Int'l Conf. on Logic Programming*, MIT Press, 1993.

[5] V. Benjumea and J.M. Troya. An OR Parallel Prolog Model for Distributed Memory Systems. *Procs. PLILP*, Springer Verlag, 1993.

[6] J. Briat et al. OPERA: Or-Parallel Prolog System on Supernode. In *Implementations of Distributed Prolog*, J. Wiley & Sons, 1992.

[7] L.F. Castro et al. DAOS – Scalable And-Or Parallelism. *5th International Euro-Par Conference*, Lecture Notes in Computer Science, Vol. 1685, Springer, 1999.

[8] J. Dongarra et al. MPI: A Message Passing Interface Standard. University of Tennessee, Knoxville, Technical Report, 1995.

[9] W-K. Foong. *Combining and- and or-parallelism in Logic Programs: a distributed approach*. PhD thesis, University of Melbourne, 1995.

[10] G. Gupta and E. Pontelli. Stack-splitting: Or-/And-parallelism on Distributed Memory Machines. International Conference on Logic Programming. MIT Press, 1999.

[11] E. Pontelli and G. Gupta. A Simulation Study of Distributed Execution of (Constraint) Logic Programs. Internal Paper, 1999.

[12] F. Silva and P. Watson. Or-Parallel Prolog on a Distributed Memory Architecture. *Journal of Logic Programming*, 2000. 43(2):173-186.

A New Tabling Scheme with Dynamic Reordering of Alternatives [1]

Hai-Feng Guo and Gopal Gupta

Laboratory for Logic, Databases, and Advanced Programming
Department of Computer Science
New Mexico State University
Box 30001/CS, Las Cruces
New Mexico, USA 88003
http://www.cs.nmsu.edu/lldap

Abstract

Traditional tabled logic programming systems are implemented by maintaining a *forest* of SLD trees along with suspension of goals. This results in a considerably complex implementation as well as significant runtime overhead. We present a simple alternative technique that *dynamically reorders* clauses containing variant calls at runtime. Our simple technique allows tabled evaluation to be performed with a single SLD tree and without the use of complex operations such as stack-freezing. It can be incorporated relatively easily in an existing logic programming system. We also propose a scheme to handle the completion problem in tabled evaluation. Finally, our scheme facilitates exploitation of or-parallelism from tabled-systems.

1 Introduction

Traditional logic programming systems use SLD resolution [3]. In Prolog, subgoals of a resolvent are tried from left to right and clauses that match a subgoal are tried in the textual order they appear in the program. It is well known that SLD resolution may lead to non-termination for certain programs, even though an answer may exist. In fact, this is true of any "static" computation rule that is adopted. That is, given any static computation rule, one can always produce a program that will not be able to find the answers due to non-termination even though finite solutions may exist. In case of Prolog, certain types of left-recursive programs are examples of such problems.

To get around this problem, researchers have suggested computation rules that are "dynamic" in nature, that is, the decision regarding which clause to use next for resolution is taken based on runtime properties, e.g., the nature and type of goals in the current resolvent. OLDT [6] and SLDT [8] are two such computation rules. In OLDT resolution, when a subgoal is encountered again during the computation—such a subgoal may possibly lead to non-termination if SLD resolution is used—the

[1] The authors have been supported by NSF grants CCR 9900320, EIA 9810732, and CDA 9729848.

OLDT resolution strategy will check its already-computed solutions (called *tabled* solutions) first, before trying program clauses. If any solutions are found in the table, they are consumed one by one just as a list of fact clauses. After consuming, the computation of the subgoal is suspended until some new solutions appear in the table. A new computation is simultaneously started to try the alternative branches of the original call and to compute solutions that will be added to the table.

OLDT is implemented by a combination of suspension and maintaining a forest of SLD trees [5]. Certain predicates are declared to be *tabled*. Every time a tabled call is encountered, a new SLD computation is begun. This SLD computation *produces* answers that will be *consumed* by another SLD tree waiting for answers. Due to this maintenance of forest of SLD trees, and suspensions, implementing OLDT in this way is quite complex and overhead prone.

Recently, SLDT has been proposed as an alternative scheme to OLDT. The main idea of SLDT is that when a tabled subgoal is encountered again, instead of suspending it and waiting for solutions to be computed by the previous variant call, its execution is continued by *stealing* the backtracking point of the previous variant tabled call. SLDT is relatively easy to implement on an existing logic programming system without goal suspension. However, because backtracking points are "stolen," some matching alternatives of tabled calls may be skipped and not explored, requiring recomputation of the variant call to ensure that all solutions are computed. Furthermore, recomputing only once cannot guarantee that a fixpoint will be reached; recomputation may have to be repeated several times.

In this paper we argue that traditional tabled logic programming systems involve some drawbacks [1] such as stack freezing, computation suspension, and maintaining a forest of SLD trees. We present a simple technique, based on *dynamic reordering of alternatives with variant calls*, for efficiently incorporating tabling in an existing logic programming system. Our scheme works with a single SLD tree without requiring suspension of goals and freezing of stacks. In our dynamic alternative reordering strategy, not only are the solutions to tabled calls tabled, the alternatives leading to variant calls are also memorized in the table (these alternatives, or clauses, containing variant calls are called *looping alternatives* in the rest of the paper). A tabled call can repeatedly try its looping alternatives until it reaches a fixpoint. We believe that our tabling technique based on dynamically reordering looping alternatives is quite efficient and can lead to a more efficient implementation than traditional techniques.

The rest of the paper is organized as follows: the next section introduces the general ideas of SLD and OLDT resolutions. Section 3 discusses how OLDT is implemented in the XSB tabled Prolog system; it also discusses drawbacks of SLDT. Section 4 proposes a new tabling scheme based on reordering looping alternatives dynamically. Section 5 explains how we solve the *completion problem* in tabled evaluation. Section 7 describes some data structures and algorithms to handle the dynamic reordering scheme. Section 8 discusses how our scheme is more amenable to exploiting parallelism from tabled logic programming systems. Section 9 presents our conclusions.

2 SLD and OLDT Resolution

Prolog was initially designed to be a declarative programming language [3], that is, a logic program with a correct declarative semantics should also get the same results through procedural semantics. However, the operational semantics of standard Prolog systems that adopt SLD resolution (leftmost-first selection rule and a depth-first search rule) is not close to their declarative semantics. The completeness of SLD resolution ensures that given a query, the solution implied by the program, if it exists, can be obtained through a computation path. However, standard Prolog systems with a pre-fixed computation rule may only compute a subset of these due to problems with non-termination.

Example 2.1 *Consider the following program:*

```
r(X, Y) :- r(X, Z), r(Z, Y).        (1)
r(X, Y) :- p(X, Y), q(Y).           (2)
p(a, b).   p(a, d).   p(b, c).
q(b).      q(c).
:- table r/2.
:- r(a, Y).
```

Following the declarative semantics of logic programs, the example program 2.1 above should produce two answers *Y=b* and *Y=c*. However, standard Prolog system will go into an infinite loop for this program. It is Prolog's computation rule that causes the inconsistency between declarative semantics and procedural semantics. Even though there may exist some computation paths to the solutions, with the leftmost-first selection rule and depth-first search rule, Prolog systems are trapped in an infinite loop in the SLD-tree. It seems that breadth-first search strategy may solve the problem of infinite-looping, and it does help in finding the first solution. However, if the system is required to find all the solutions and terminate, breadth-first search is not enough.

To get around this problem, a tabled evaluation strategy, *OLDT*, is introduced in tabled logic programming systems. OLDT is a variation of SLD. In OLDT calls that may potentially lead to infinite loops are detected and not explored. In OLDT, when a call to a tabled predicate is encountered for the first time, a new SLD tree is built to compute the answers to this tabled call. The new tree is called a *generator*, while the old tree (which led to the tabled call) is called a *consumer* corresponding to the new tabled call. When a call that is a variant of a previous call, and that may cause infinite loop if SLD is applied, is encountered OLDT checks and then consumes the tabled solutions of that call (i.e., solutions that have already been computed by the previous call). If all the tabled solutions have been exhausted, the current call is suspended until some new answers are available in the table.

Simultaneously, a new computation is started to try the alternative branches of the last choice point.

The OLDT resolution forest for example 2.1 is shown in figure 1.

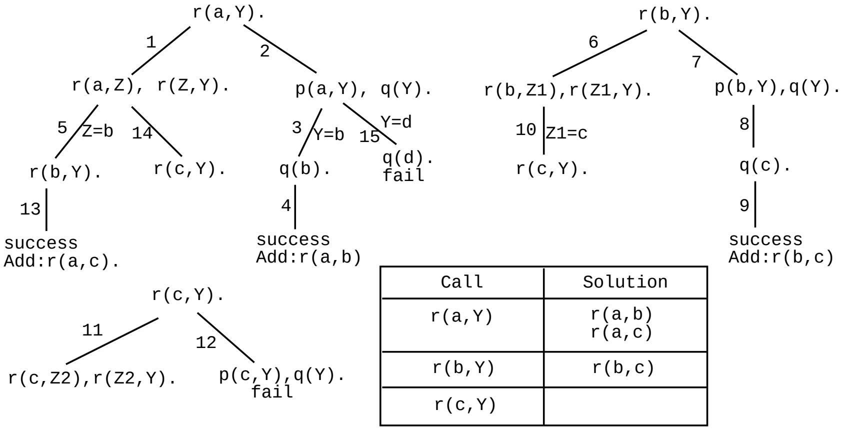


Fig. 1. An OLDT Tree

Compared with SLD, OLDT has several advantages described as follows:

- A tabled Prolog system avoids redundant computation by memoing the computed results. In some cases, it can reduce the time complexity of a problem from exponential to polynomial.
- A tabled Prolog system terminates for all queries posed to programs that have a finite least fixpoint.
- Tabled Prolog keeps the declarative semantics and procedural semantics of definite Prolog programs consistent.

3 Implementation of Tabling

The most mature implementation of tabling is the XSB [5] system from SUNY Stony Brook. The XSB system implements OLDT by developing a forest of SLD trees. In the XSB system certain predicates are statically designated as *tabled* by the user. Tabled evaluation is used only for these tabled predicates, all other predicates are evaluated using normal SLD resolution. When a tabled predicate is encountered, its execution is done separately from the current computation (call the current computation $\mathcal{C}$, and the side computation $\mathcal{S}$). The idea is that this side computation will produce a number of solutions to the call to the tabled predicate. These solutions, once produced by $\mathcal{S}$, will be "consumed" by computation $\mathcal{C}$, i.e., the solutions produced by $\mathcal{S}$ will be used to resolve calls to the tabled predicate in $\mathcal{C}$. The solutions produced by $\mathcal{S}$ are entered in a memo-table, so that they are computed only once. Other "side computations" may be created during the computation $\mathcal{S}$, if more tabled calls are encountered in $\mathcal{S}$. As a result, we may have a forest of SLD trees that are simultaneously active. In a sequential implementation, all but one of the trees are suspended. Considerable book-keeping is required to manage the information flow between the different SLD trees. In the XSB system, these multiple SLD trees are implemented by suspending a computation when a tabled call is encountered. A computation is suspended by freezing its stacks. At any given moment a number of such frozen computations may exist. The stack can be unfrozen only after the corresponding tabled call is completely evaluated.

A new data structure, called *completion stack* [5], was introduced in XSB to help detect whether a tabled call has been completely evaluated. A tabled call may depend on some other tabled calls to generate new solutions. So a tabled call is *completely evaluated* only if its corresponding generator has explored all the choicepoints and the other tabled calls it depends on are also completely evaluated. Only when the tabled call is completely evaluated, its corresponding stack areas that were frozen can be reclaimed.

In the XSB system, OLDT is implemented by a combination of stack freezing, computation suspension, maintaining a forest of SLD trees, and completion check. Implementing OLDT in this way is quite complex and overhead prone.

Recently, another implementation of a tabled Prolog system based on SLDT has been reported [8]. This implementation has been done on top of the existing Prolog system called B-Prolog [7]. The main idea behind this implementation is as follows: when a variant is recursively reached from a tabled call, the active choicepoint of the original call is transferred to the call to the variant (the variant *steals* the choicepoint of the original call, using the terminology in [8]). Suspension is thus avoided (in XSB, the variant call will be suspended and the original call will produce solutions via backtracking) and the computation pattern is closer to SLD. However, because the variant call avoids trying the same alternatives as the previous call, the computation may be incomplete. Thus, *recomputation* [8] is required to make up for the drawback of SLDT and to make sure that the fixpoint is reached. B-Prolog does not propose a complete theory regarding when a tabled call is completely evaluated. Additionally, if there are multiple clauses containing recursive variant calls, the variant calls may be encountered several times in one computation path. In the linear tabling mechanism [8], each variant call executes from the backtracking point of a former variant call, that is, each variant call skips the previous alternatives containing variant calls which it should explore. The solutions that may potentially be lost by skipping these alternatives are found by recomputation. However, in the case of multiple clauses with variant calls, recomputation will be quite complicated because new solutions can be possibly generated by exploring the multiple alternatives in different combinations. However, stealing the backtracking point of a former variant prevents those combinations to be tried in which the alternatives that occur textually-lower are tried first and those occurring textually-higher are tried later. Thus, the whole computation has to be re-explored one or more times from the first tabled call to make sure all the solutions have been obtain and none missed. Such kind of recomputation has a significant overhead.

4 Dynamic Alternative Reordering

We present a simple technique for implementing tabling that is based on *dynamic reordering of looping alternatives* at runtime, where a *looping alternative* refers to a clause that matches a tabled call containing a recursive variant call. In our scheme, a tabled call can be in one of three possible states: *normal state*, *looping state*, or *complete state*. The state transition graph is shown in figure 2.

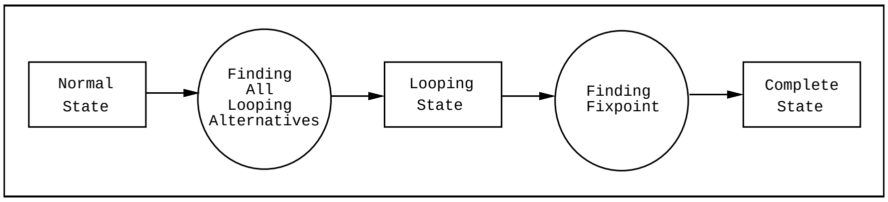


Fig. 2. State Transition Graph

Consider any tabled call C, normal state is initially entered when C is first encountered during the computation. This first occurrence of C is allowed to explore the matched clauses as in a standard Prolog system. Other occurrences of this call later will consume the tabled solutions instead. In normal state, while exploring the matching clauses, the system tables all the solutions of the call C and also checks for variants of C. If a variant is found, the current clause that matches the original call to C will be memorized as a *looping alternative*. Essentially, the alternative corresponding to this call will be reordered and placed at the end of the alternative list in the choicepoint. This call will not be expanded at the moment because it can potentially lead to an infinite loop. A failure will be simulated and the alternative containing the variant will be backtracked over. After exploring all the matched clauses, C goes into its looping state. From this point, tabled call C keeps trying its tabled looping alternatives repeatedly until C is completely evaluated, that is, its fixpoint is reached. If no new solution is added to C's tabled solution set in any one cycle of trying its tabled looping alternatives, then we can say that C reaches its fixpoint.

C enters its *complete state* after it reaches its fixpoint. During its complete state, all other variants of the tabled call C only need to consume C's solution set one after another as if trying a list of facts.

Example 4.1 *Consider resolving the following program with the dynamic reordering strategy:*

```
r(X, Y) :- r(X, Z), p(Z, Y).     (1)
r(X, Y) :- p(X, Y).              (2)
p(a, b).   p(b, c).   p(c, d).
:- table r/2.
:- r(a, Y).
```

Figure 3 shows the whole evaluation using our dynamic reordering strategy. Two new actions, *s_add* and *a_add*, are introduced in the figure, where *s_add* and *a_add* add a new solution and a new looping alternative to the corresponding tabled call in the table respectively. Consider the query r(a, Y). Because r(a, Y) belongs to the tabled call r/2, it is inserted into the table as an entry. Then the first matched

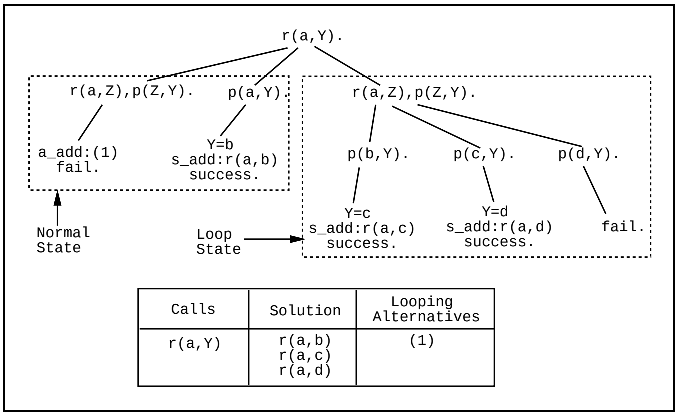


Fig. 3. Example 4.1

clause, clause (1), is explored. Immediately, r(a, Z), a variant of the tabled call r(a, Y), is met, so the current alternative of r(a, Y) is added to the set of looping alternatives. The variant call r(a, Z) starts to consume the tabled solutions instead of trying any matching clauses. Since there is no solution so far under the call r(a, Y) in the table, the call r(a, Z) just fails. Then r(a, Y) tries another matched clause, clause (2), to find a solution *Y=b*. So a new solution r(a, b) is added into the table under the tabled call r(a, Y). After all the matched clause, clauses (1) and (2), have been explored, the tabled call r(a, Y) enters its looping state. In the looping state, only tabled looping alternatives will be tried repeatedly till the fixpoint is reached. Thus, the tabled looping alternative, clause (1), is tried, and the variant call r(a, Z) consumes the only tabled solution r(a, b). After consuming the solution r(a, b), the variant call r(a, Z) finds another new solution r(a, c) tabled, so r(a, Z) continue to consume the new tabled solution, and so on, until there are no more solutions generated.

Example 4.1 gives the basic framework for how dynamic alternative reordering works. Now let's consider another example with multiple looping alternatives, where we can get a better impression of how the tabled call reaches its fixpoint.

Example 4.2 *Consider resolving the following program with the dynamic reordering strategy:*

```
r(X, Y) :- r(X, Z), p(Z, Y).     (1)
r(X, Y) :- p(X, Y).              (2)
r(X, Y) :- r(X, Z), q(Z, Y).     (3)
p(a, b).   p(b, c).   q(c, d).
:- table r/2.
:- r(a, Y).
```

Figure 4 gives the whole computation tree by applying the dynamic reordering strategy on example 4.2. Both clause (1) and clause (3) need to be tabled as looping alternatives for the tabled call r(a, Y). When the call r(a, Y) enters its looping state, it keeps trying the looping alternatives repeatedly until the solution set does not change any more, then it is said that r(a, Y) is completely evaluated.

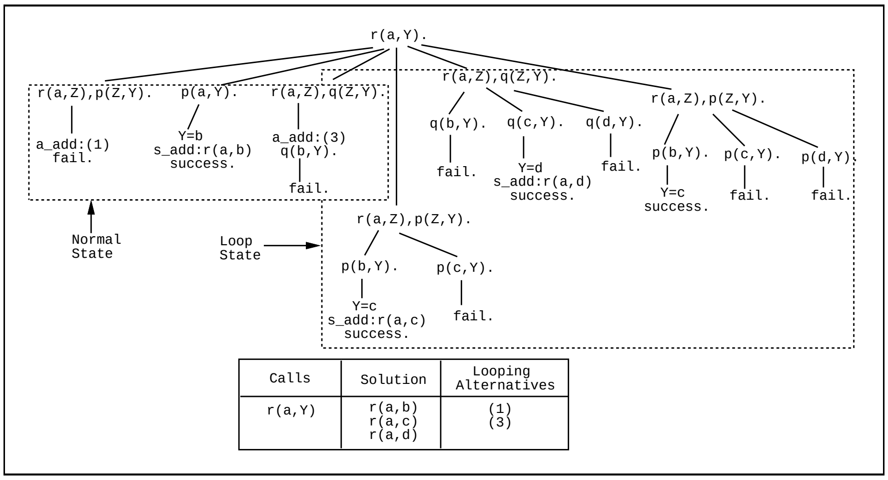


Fig. 4. Example 4.2

The tabling scheme with dynamic reordering follows the bottom-up computation of logic programs. Tabled evaluation is used only for the tabled predicates, and all other predicates are evaluated using normal SLD resolution, whose correctness and completeness have been proven in [3]. For any tabled call $\mathcal{C}$, consider all the matched clauses of $\mathcal{C}$ as a function $\mathcal{F}$. Initially the solution set of $\mathcal{C}$ is empty. Then the tabled evaluation of $\mathcal{C}$ can be described as applying the function $\mathcal{F}$ on the solution set iteratively until the fixpoint of $\mathcal{F}$ is reached, which mimics computing the bottom-up semantics of logic programs.

5 Detecting Completion

When there are multiple tabled calls occurring simultaneously during the computation, and results produced by one tabled call may depend on another's, then knowing when the computation of a tabled call is complete (i.e., all solutions have been computed) is quite hard.

Definition 5.1 *Consider two tabled calls $\mathcal{A}$ and $\mathcal{B}$, if $\mathcal{A}$ exists in one of the computation paths of $\mathcal{B}$, then we say the tabled call $\mathcal{B}$ is* **dependent** *on $\mathcal{A}$.*

Definition 5.2 *A tabled call is* **completely evaluated** *only if all its looping alternatives cannot generate new solutions any more and its dependent tabled calls are* **completely evaluated**.

Definition 5.3 *A tabled call is in* **complete state** *when it is completely evaluated.*

In order to formalize the completion problems of tabled calls, we introduce a model, called *completion dependency graph*, to help describe the dependencies between tabled calls.

Definition 5.4 *A* **Completion Dependency Graph** *G is a pair (V, E), where V is a finite set of the tabled calls and E is a binary dependent relation on V. The elements of V are called* **vertices***, which are represented by circles, and the elements of E are called* **edges***, which are represented by arrows in the graph G. The dependent relation E is transitive, that is, aEb and bEc implies aEc, where a, b, and c are three vertices and E is the binary relation of edges. Self-loops are ignored in G, since every call is always dependent on itself.*

Example 5.1 *Consider resolving the program in example 2.1 with the dynamic reordering strategy.*

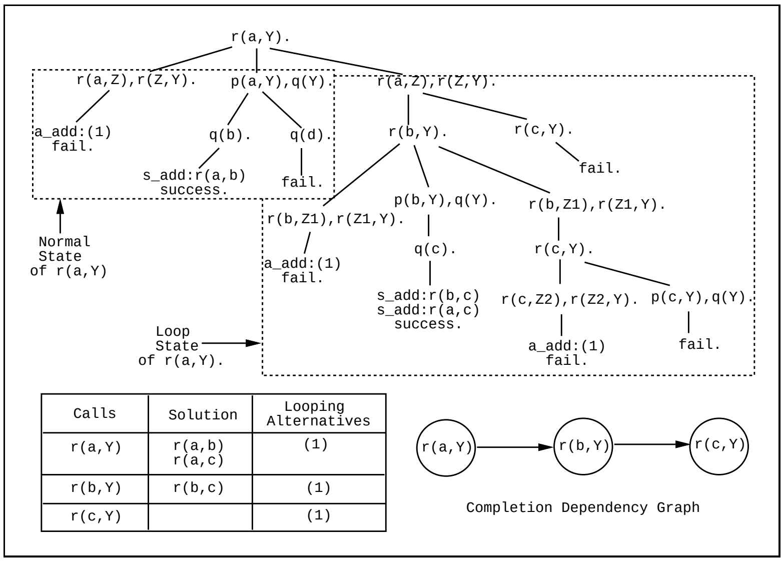


Fig. 5. Example 5.1

A typical example of completion dependency graph has been shown in the figure 5 for example 5.1. In that graph, each node represents a tabled call and a directed edge represents dependency between tabled calls. So r(b, Y) is completely evaluated only if its dependent call r(c, Y) is completely evaluated, and r(a, Y) is completely evaluated only if its dependent calls, r(b, Y) and r(c, Y), are completely evaluated. Due to the depth-first search rule, r(c, Y) always enters its complete state ahead of r(b, Y), and r(b, Y) ahead of r(a, Y), The depth-first strategy with alternative reordering guarantees for such graphs (i.e., graphs with no cycles) that dependencies can be satisfied without special processing during the computation.

Lemma 5.5 *Given a completion dependency graph without any cycles for a tabled predicate, the depth-first strategy with dynamic alternative reordering will compute all the solutions to a call to this tabled predicate.*

Proof Let P be a program and G a goal. Suppose C_1 and C_2 are two tabled calls during the computation of $P \cup \{G\}$ and C_1 is dependent on C_2. According to the Definition 5.1, C_2 must exist in one of the computation paths of C_1. Then based on depth-first strategy and dynamic reordering strategy, C_2 enters its complete state ahead of C_1. □

Completion checking becomes harder when the dependency graph contains a cycle, that is, a tabled call depends on itself indirectly. In such a case, we handle the completion problem a little differently.

Definition 5.6 *A directed graph is* **strongly connected** *if every two vertices are reachable from each other. The* **strongly connected components (SCCs)** *of a graph are the equivalent classes of vertices under the "are mutually reachable" relation.*

Lemma 5.5 is not enough for the correctness of completion if two or more tabled calls depend on each other, that is, there exist SCCs in the dependency graph. In presence of SCCs, it is not clear which tabled call is completely evaluated first. Therefore, in such a case, a proper semantics can be given to the program only if all tabled calls in a SCC reach their complete state simultaneously.

Lemma 5.7 *Let G be a completion dependency graph, and S a strongly connected component in G. S is* **completely evaluated** *if and only if all tabled calls in S are* **completely evaluated** *and those tabled calls must reach their complete states simultaneously.*

Then we get the following corollary from Lemma 5.7:

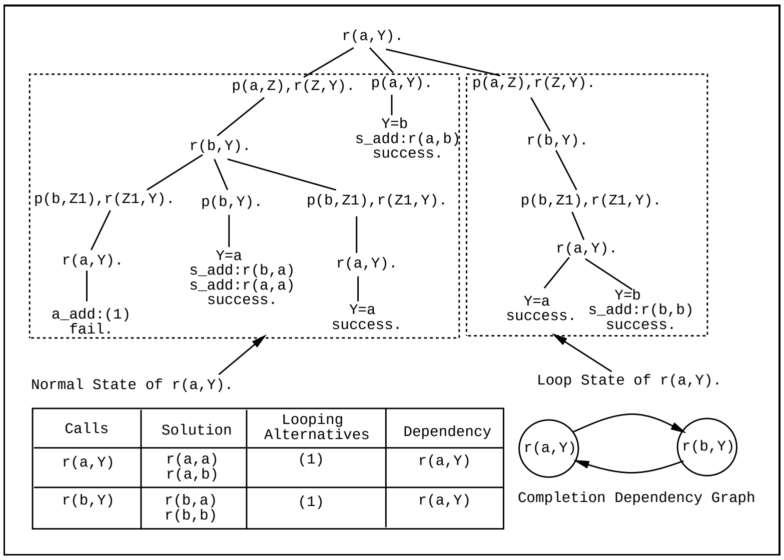
Corollary 5.8 *Let G be a completion dependency graph, S a strongly connected component in G, and C any tabled call in S. If C is completely evaluated, then S is also completely evaluated.*

The corollary 5.8 provides us a good way of handling cyclically dependent tabled calls: If we can ensure that one tabled call in the SCC is completely evaluated, then we can say the SCC is completely evaluated. By combining the depth-first strategy, we can focus on checking the fixpoint of the least deep tabled call (i.e., the tabled call that is encountered first during computation) for each SCC. The state of the rest of the tabled calls in the SCC cannot be set to "complete" until the state of the least deep one is set to "complete."

Example 5.2 *Consider resolving the following program with the dynamic reordering strategy:*

```
r(X, Y) :- p(X, Z), r(Z, Y).     (1)
r(X, Y) :- p(X, Y).              (2)
p(a, b).   p(b, a).
:- table r/2.
:- r(a, Y).
```

Figure 6 shows the complete computation tree of example 5.2. In this example, two tabled calls, r(a, Y) and r(b, Y), are dependent on each other, forming a SCC in the completion dependency graph. A new column, “Dependency”, is introduced in the Table structure, which is used to record the least deep tabled call of each SCC, so that the remaining calls in the SCC can tell whether they are in the complete state by checking the state of the least deep call. The state of a tabled call can be set to “complete” only after its corresponding least deep call is in a complete state. There are two occurrences of r(b, Y) during the computation. In its first occurrence, r(b, Y) can not be set to “complete” even though it reachs a *temporary* fixpoint after exploring its looping alternative, because it depends on the tabled call r(a, Y), which is not completely evaluated yet. Otherwise, a solution r(b, b) will be lost. Only after the tabled call r(a, Y) is completely evaluated during its looping state, then the tabled call r(b, Y), within the same SCC with r(a, Y), can be set to complete state.



Calls	Solution	Looping Alternatives	Dependency
r(a,Y)	r(a,a) r(a,b)	(1)	r(a,Y)
r(b,Y)	r(b,a) r(b,b)	(1)	r(a,Y)

Fig. 6. Example 5.2

6 Recomputation Issues

The dynamic reordering scheme repeatedly tries looping alternatives until a fixpoint is reached. Retrying these reordered looping-alternatives may cause redundant computations. Consider the following clause:

```
r(X, Y) :- p(X, Z), r(Z, Y).     (1)
```

Suppose that only r/2 is tabled, then clause (1) is obviously a looping alternative for r/2. When this alternative is repeatedly tried in the looping state, two types of redundant computation may be done: (i) the subgoal p(X, Z) will be recomputed everytime alternative (1) is tried; (ii) the variant call r(Z, Y) will reconsume all tabled solutions. The former case of recomputation can be eliminated by tabling the predicate p/2, so that p(X, Z) can avoid recomputation by consuming its tabled solutions. The latter case can also be avoided if there is only one variant subgoal found in the current alternative, such as clause (1) above which has a single variant r(Z, Y). However, the latter case cannot be always avoided. For example, consider:

```
r(X, Y) :- r(X, Z), r(Z, Y).     (2)
```

Both r(X, Z) and r(Z, Y) are variants of r(X, Y). Some solutions may be generated by combining an old tabled solution consumed by r(X, Z) and a new tabled solution consumed by r(Z, Y). If only new tabled solutions are tried, then some solutions may be lost.

However, in spite of recomputation, our dynamic reordering scheme is not worse than traditional SLD resolution (i.e., it will certainly not perform more computation than SLD). For any recomputed branch in the dynamically reordered computation tree, we can always find a corresponding computation path in SLD-tree, and along that path some alternatives must be recomputed also. Furthermore, our dynamic reordering scheme is considerably better than SLD, because it avoids a lot of recomputation by using tabling.

The XSB system which is based on goal suspension and stack freezing will show better behavior as far as the two problems related to recomputation mentioned above are concerned. Goal suspension and stack freezing effectively memo the computation environment for the goals occuring before the variant subgoal in the looping alternative. Thus, in an implementation based on subgoal suspension, any subgoals occuring before the variant subgoal are implicitly and unintentionally tabled. That is, not only explicitly tabled predicates are never recomputed, but some non-tabled predicates are also never recomputed. Thus, tabling the goal p/2 to avoid recomputation in our dynamic reordering scheme is quite reasonable (since effectively it will be tabled in suspension based systems such as XSB as well).

In fact, the problem of recomputation cannot be avoided unless we resort to an XSB style implementation based on suspension and freezing. However, a suspension based implementation has its own problems: more complex memory management and implementation.

Our recomputation scheme is better than that of B-Prolog. In dynamic reordering scheme, for any tabled call, the recomputation is performed only during its looping state. One advantage is that only looping alternatives are recomputed. Another important fact is that only the first occuring tabled call can generates new solutions, while the later occuring tabled calls only consume the tabled solutions. Thus, the depth of the tree created for any tabled call will be very limited because the tabled calls consuming the tabled solutions will not call themselves any more. Additionally, it is relatively easy to find the fixpoint whenever all the looping alternatives cannot generate new solutions.

7 Implementation

Dynamic reordering scheme can be easily implemented on an existing Prolog system. Memory management and execution environment can be kept the same as in a regular Prolog engine. However, a table structure has to be introduced that records the information stored in the table at runtime:

```
struct Table{
   item "Tabled Call",
   item "Solution Set",
   item "Looping Alternative Set",
   item "State"
   item "Dependency"
}
```

where "Tabled Call" is the name of the call being tabled; "Solution Set" stores the solutions of the tabled call collected during the normal and looping state; "Looping Alternative Set" stores the tabled call's looping alternatives during the normal state (a looping alternative is represented by the starting address of its WAM code); "State" records the current state of the tabled call whether *normal, looping,* or *complete*; and the "Dependency" field is used to remember the least deep tabled call of the SCC which contains the "Tabled Call", so that each tabled call in the SCC can tell whether it has reached the complete state by checking the state of the least deep tabled call.

Basically, tabled evaluation follows a two-step precedure for each tabled call as shown in figure 2. The operation "Find All Looping Alternatives" from normal state to looping state is described through the following procedure:

```
Procedure Finding_All_Looping_Alternatives
    set "Dependency" self;
    while (matched clauses are not exhausted) {
        try the next matched clause with Dynamic Reordering Strategy;
        if (a variant is found)
           Table the current clause as a looping alternative;
        if (a new tabled call is found) {
           D = the "Dependency" item of the tabled call after executing
               its Finding_All_Loop_Alternatives and Finding_Fixpoint;
           replace "Dependency" with D if D is less deep than the "Dependency";
        }
        if (a variant of some other tabled call) {
           if (that tabled call is already complete)
              consume its solutions;
           else
              replace "Dependency" With that tabled call if the tabled
              call is less deep than than the current "Dependency";
        }
```

```
        Table all the solutions of the tabled call;
    }
    set state to "looping State";
```

After entering the looping state, the tabled call will only try the tabled looping alternatives repeatedly until the fixpoint is reached. Two new variables, Current_Alternative and Last_Fixed_Alternative, are introduced to help find the fixpoint. Current_Alternative means the current looping alternative being tried, and Last_Fixed_Alternative represents the last looping alternative that resulted in a new solution being added to the table. Suppose there are n looping alternatives: $A_0, A_1, ..., A_{n-1}$, the algorithm for the operation "Finding the Fixpoint" is shown below:

```
Procedure Finding_Fixpoint
    if ("Dependency" != self && the state of "Dependency" is complete) {
       Set its state "Complete State";
       return;
    }
    Current_Alternative = 0;
    Last_Fixed_Alternative = n - 1;
    Fixpoint = false;
    while (!Fixpoint) {
        Try the current alternative with Dynamic Reordering Strategy;
        if (any new solutions)
           Last_Fixed_Alternative = Current_Alternative;
        else
           if (Current_Alternative == Last_Fixed_Alternative)
              Fixpoint = true;
        Current_Alternative = (Current_Alternative + 1) % n;
    }
    if ("Dependency" == self) set state to "Complete State";
```

8 Table-parallelism

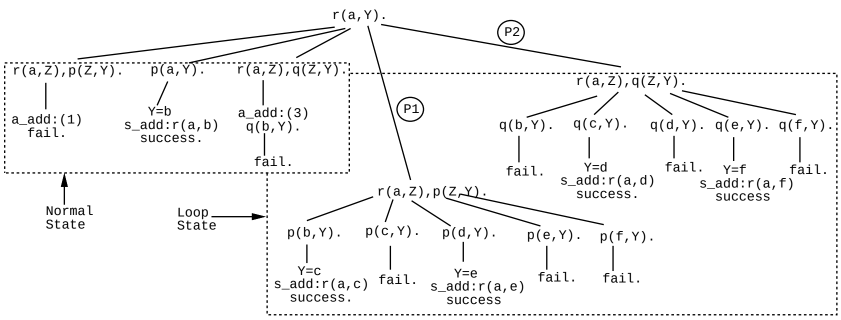
The main idea in our tabling strategy is to collect the looping alternatives, which can be applied on the tabled solutions repeatedly to generate new solutions until the fixpoint is reached. Let G be a tabled call. Suppose there are multiple looping alternatives found corresponding to G, then according to strategy of dynamic reordering, the looping alternatives will be tried repeatedly. For a pratical program, it is possible that all looping alternatives may have to be tried several times to reach the fixpoint. The multiple looping alternatives can be effectively parallelized using different processes where each process is responsible for executing one looping alternative. The communication needed among different processes can be done through the table. We call this kind of parallelism—an instance of or-parallelism where the threads communicated through the table—*Table-parallelism* [2].

Example 8.1 *Consider resolving the following program using our dynamic re-*

ordering strategy:

```
r(X, Y) :- r(X, Z), p(Z, Y).     (1)
r(X, Y) :- p(X, Y).              (2)
r(X, Y) :- r(X, Z), q(Z, Y).     (3)
p(a, b).   p(b, c).   p(d, e).
q(c, d).   q(e, f).
:- table r/2.
:- r(a, Y).
```

Table-parallelism is illustrated as in figure 7. There are two looping alternatives, clause (1) and clause (3), for the tabled call r(a, Y). Processors P1 and P2 are assigned to the looping alternatives (1) and (3) respectively. Then P1 and P2 execute the different looping alternatives in their own local environments communicating with each other through the table. The tabled call r(a, Y) is completely evaluated when both P1 and P2 reach their fixpoints.



Calls	Solution	Looping Alternatives
r(a,Y)	r(a,b) r(a,c) r(a,d) r(a,e) r(a,f)	(1) (3)

Fig. 7. Example 8.1

Since the dynamic reordering strategy works on a single SLD tree, it is also feasible to apply existing techniques for and/or-parallelism [1] to execute tabled logic programs in parallel.

9 Conclusion and Future Work

The advantages of our tabling scheme with dynamic reordering can be listed as follows: (i) Our scheme closely mimics bottom-up semantics of logic programs. Its correctness and completeness is easy to prove; (ii) It can be easily implemented based on an existing Prolog system without modifying the kernel of WAM engine in any major way; (iii) It works with a single SLD tree without suspension of goals and freezing of stacks; (iv) Unlike SLDT, it avoids blindly recomputing subgoals (to ensure completion) by tabling looping alternatives. Perhaps the greatest advantage of our scheme is that it can be easily incorporated in existing Prolog systems with minimal modification [2].

Our alternative reordering strategy can be thought of as a dual [4] of the Andorra-principle [9]. In the Andorra principle, goals in a clause are reordered leading to a considerable reduction in search space and better termination behavior. The reordering of subgoals is done based on runtime properties. Likewise, our tabling scheme based on reordering alternatives also reduces the size of the computation (since solutions for tabled call once computed are remembered) and results in better termination behavior.

Our scheme is quite simple to implement. We were able to implement it on top of an existing Prolog engine in about one week of work. Performance evaluation of this implementation is in progress [2].

References

[1] Hai-Feng Guo. *A Survey on Distributed Parallel Logic Programming and Tabled Logic Programming*. Comprehensive Paper, New Mexico State Univ., Apr. 2000.

[2] Hai-Feng Guo. *High Performance Logic Programming*. Ph.D. thesis. *forthcoming*.

[3] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[4] E. Pontelli, G. Gupta. "On the Duality between Or-parallelism and And-parallelism," In *European Conference on Parallel Processing '95*, Stockholm, Sweden, Springer Verlag Lecture Notes 966, pp. 43-54.

[5] Terrance Swift and David S. Warren. An Abstract Machine for SLG Resolution: Definite Programs. *SLP* 1994: 633-652.

[6] David S. Warren. Programming in Tabled Prolog (very) DRAFT. Dept. of CS, SUNY@Stony Brook, 1995.

[7] Neng-Fa Zhou. Parameter Passing and Control Stack Management in Prolog Implementation Revisited. *ACM Transactions on Programming Languages and Systems*, Vol. 18, No. 6, 752-779, 1996.

[8] Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan and Jia-Huai You. Implementation of a Linear Tabling Mechanism. *PADL* 2000: 109-123.

[9] V. Santos Costa, D. H. D. Warren, R. Yang. Andorra-I: A Parallel Prolog system that transparently exploits both And- and Or-Parallelism. In *Proceedings of Principles & Practice of Parallel Programming*, ACM Press, Apr. '91, pp. 83-93.

The RAINBOW: Towards a Parallel BEAM

Ricardo Lopes and Fernando Silva

LIACC, DC-FCUP
Universidade do Porto
Portugal
{rslopes,fds}@ncc.up.pt

Vítor Santos Costa

COPPE/Sistemas, UFRJ
Brazil
vitor@cos.ufrj.br

Salvador Abreu

Departamento de Informática
Universidade de Évora
Portugal
spa@di.uevora.pt

Abstract

Logic programming provides a high-level view of programming that gives implementors a vast latitude in what techniques to research towards obtaining the best performance for logic programs. The Extended Andorra Model was designed towards achieving reduction of the search space whilst exploiting all the available parallelism in the application. The BEAM is a first sequential implementation for the Extended Andorra Model with Implicit Control, that has been shown to obtain good results. In this work we propose the RAINBOW, a parallel execution model for the BEAM. We present a general overview of how to distribute work, propose alternative approaches towards addressing the binding problem for the EAM, and present a scheduling strategy.

1 Introduction

Logic programming provides a high-level view of programming where programs are fundamentally seen as a collection of statements that define a model of the intended problem, and questions may be asked against this model. Prolog is the most popular example of a logic programming language, and has been successfully used in applications such as artificial intelligence, database programming, circuit design, genetic sequencing, expert systems, compilers, simulation and natural language processing.

Prolog uses a straightforward left-to-right selection function and depth-first search rule. This computation rule is simple to understand and efficient to implement but, unfortunately, it is not the ideal rule for every logic program. It is well known that for many programs the left-to-right selection function is not effective at constraining the search space and may in fact lead to looping. Moreover, the Prolog rule does not exploit implicit parallelism. To improve Prolog, novel computational models thus must achieve these goals, presented by David H. D. Warren [26] as, in order of priority:

- *Minimum number of inferences*: this is achieved by trying never to repeat the same execution step of one inference in different locations of the execution tree.
- *Maximum parallelism*: this is achieved by allowing goals to be executed as more independently as possible, and by doing the combination of all solutions as late as feasible.

Co-routining is an example of an optimisation that can improve the search space for Prolog programs. This improvement on the selection function has been discussed since the initial days of logic programming and was implemented in early systems such as Prolog-II [5], IC-Prolog [4], Epilog [20], and MU-Prolog [18]. In contrast, several systems have tried to achieve parallelism while keeping close to Prolog semantics. These systems have successfully exploited distinct forms of parallelism such as or-parallelism [14,2], restricted and-parallelism [7,9], or producer consumer and-parallelism [23,19]. The Basic Andorra Model (BAM) [24] is a very well known proposal where co-routining between determinate goals both constrains the search space and generates and-parallelism, while or-parallelism is achieved through exploiting alternatives of non-determinate goals. The Andorra-I prototype [22] demonstrated that the BAM can successfully exploit two forms of parallelism, or-parallelism and dependent and-parallelism, and that it can reduce the search space for a large category of programs. On the other hand, Andorra-I depends on finding determinate applications. If determinacy cannot be found efficiently, applications with strong parallelism may need to be run sequentially.

The limitations of the BAM motivated the development of the Extended Andorra Model [25]. The key ideas for this model can be described as:

- Goals can execute (in parallel) as long as they do not need to bind external variables;
- Ultimately, completeness requires some form of enumeration: the computation will then *fork*.

The EAM has a more complex structure than the original BAM, and was thus formalised through the concept of rewriting and- and or-boxes. Research then evolved in two directions. David H. D. Warren and researchers at Bristol concentrated on the *Extended Andorra Model with Implicit Control* [26], where the goal was to obtain the advantages of the Extended Andorra Model with least effort from the programmer. Haridi, Janson and researchers at SICS concentrated on the AKL [10], the Andorra Kernel Language, based on the principle that the advantages of the Extended Andorra Model justified a new programming paradigm that could subsume both traditional Prolog and the committed choice languages.

Gopal Gupta was the first author to show that the EAM with implicit control was indeed feasible [8]. He used a Prolog interpreter to show how the model would perform on several applications and uncovered several problems in the initial proposal. Warren and Gupta thus proposed solutions such as lazy copying, eager producers, and modes to give finer control over search when required. The BEAM [11] is the first actual (sequential) implementation of the EAM with implicit control. We have demonstrated that the BEAM can achieve quite reasonable performance, indeed similar to the AKL prototype, and that we can run large applications on these systems. The BEAM was designed to extend Prolog systems, so that it can be used as a sophisticated query processor for systems that generate huge numbers of complex queries, such as ILP systems [21].

Ultimately, the full advantages of the EAM can only be achieved through parallelism. Parallelism for the EAM was originally studied in systems such as ParAKL [17] and Penny [15]. Penny used copying and was shown to obtain good results: the BEAM is indeed a natural form of parallelism. Penny also demonstrated some dangers in the implementation of parallelism. For instance, garbage collection may be a limiting factor on the total amount of parallelism, suggesting that a distributed approach may be ideal [16].

The natural match between parallelism and the EAM motivated our initial proposal for the *RAINBOW*: *R*apid *A*nd-parallelism with *I*ndependence *B*uilt upon *O*r-parallel *W*ork. In the sequence we present the Extended Andorra Model in some more detail, describe the BEAM, and describe how we propose to tackle our road to RAINBOW. We present a general overview of how to distribute work, propose alternative approaches towards addressing the binding problem for the EAM, and present a scheduling strategy.

2 The Extended Andorra Model

The EAM was designed to generate all answers to problems coded as Horn clauses with a minimum number of inferences. The EAM tries to achieve this goal by performing as many steps in parallel as possible.

2.1 Execution Principles for the EAM

The EAM is formally defined using rewrite rules that manipulate and-or-trees. These are represented through *and*, *or* and *choice-boxes*.

- **And-box:** $[\exists X_1,...,X_m : \sigma \& G_1 \& ... \& G_n]$
  And-boxes are created when launching a new clause with goals G_1 to G_n. The variables X_1 to X_m represent the variables created in the box and σ represents the constraints on external variables imposed by the and-box.
- **Or-box:** $\{C_1 \vee ... \vee C_n\}$
  Or-boxes are created when launching a new goal, each C_i represents an alternative clause for the goal.
- **Choice-box:** $\{G_1 \% C_1 \vee ... \vee G_n \% C_n\}$
  Choice-boxes are special or-boxes created when the clauses defining a procedure include a pruning operator, %. Each G_i represents the guard of the goal and each C_i represents the remainder of the clause. The original EAM supports two pruning operators, cut, !, and commit, |. AKL further supports a sequential conjunction, :.

We next concentrate on cut-free programs. In this case, there are four main operations in the Extended Andorra Model:

(i) **Reduction:** $G \rightarrow \{[\exists Y_1 : \sigma_1 \& C_1] \vee \ldots \vee [\exists Y_n : \sigma_n \& C_n]\}$
Reduction expands a goal G in an and-box into an or-box.

(ii) **Promotion:**
$[\exists X, Z : A \& [\exists Y, W : Z = \theta(W) \& C] \& B] \rightarrow [\exists X, W, Z : Z = \theta(W) \& A \& [\exists Y : C] \& B]$
Promotion promotes constraints from a inner-box to the outer and-box.

(iii) **Substitution:** $[\exists X, Z : Z = \theta(W) \& C] \rightarrow [\exists X : C\theta]$
Substitution applies a constraint on a variable.

(iv) **Forking:** $[\exists X : \{(C_1 \vee ... \vee C_n) \ \& \ G] \rightarrow \{[\exists X : C_1 \& G] \vee ... \vee [\exists X : C_n \ \& \ G]\}$
Forking distributes a conjunction across a disjunction.

Several simplification rules are also implemented. Bracket removal rules normalise nested and-boxes and or-boxes . Failure propagation rules simplify and-boxes of the form $\ldots \wedge$ `false` $\wedge \ldots$, and success propagation rules simplify or-boxes of the form $\ldots \wedge$ `true` $\wedge \ldots$. Check [13] for a more detailed description of these rules and for a description of the cut rules.

3 Execution control in BEAM

The BEAM uses a control scheme where an and-box suspends if it constrains an external variable. Fig. 1 shows the control scheme used in the BEAM. The system first tries to apply a reduction or a promotion step. If it succeeds, it then applies a sequence of *simplification* steps to maintain the data structures (these are detailed in [13]). If no reduction or promotion steps were possible, it does a forking step.

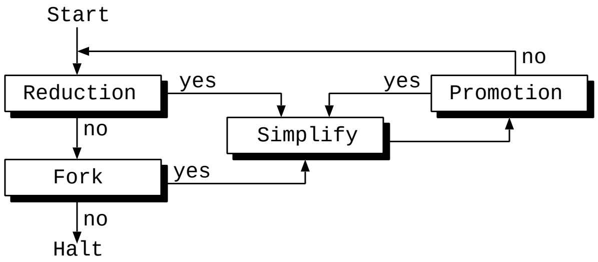


Fig. 1. Execution control in the BEAM.

Forking is the operation that can lead to duplication of work, and thus is only applied when all the other operations cannot be applied further. In the BEAM, if several goals can produce a non-determinate binding for a variable, the leftmost one will be chosen to fork. Notice that a binding for a variable X has to be generated by one of the C_i before forking is initiated. After forking, this binding can be promoted in the newly created and-box. In the BEAM the forking step is always followed by a promotion. A forking step followed by a promotion is called *global forking*.

Favoring reduction steps over forking steps in the EAM allows early execution of determinate goals. In some cases, this may create speculative and-work, as a goal to the left might fail. In the worst case, speculation may result in a larger search space or even lead to non-termination, as the EAM allows any computation to go ahead as long as it does not bind variables. Ideally, one would like to provide an implicit control mechanism to prevent this problem. Unfortunately, detecting speculative work depends on knowing a priori whether the computation will terminate, an undecidable problem. On [11] we discuss how tabling can be used in a EAM model to prevent non-termination.

4 BEAM Memory Areas

Memory usage is much more complex on the BEAM than in standard WAM implementations due to the fact that the EAM scheduler does not necessarily follow a stack discipline. In contrast, WAM-based Prolog manages a simple stack where space can be recovered after backtracking.

The external state of the BEAM, stored in memory, is divided into two main areas (see figure 2): the *Code_Space* and the *Global_Memory*. The *Code_Space* holds the data-base, including the compiled logic program, information on predicates, and a symbol-table. The use of this area is straightforward, being very similar to traditional Prolog systems.

The *Global_Mem* maintains the and-or-tree that will be created during the execution of the Prolog programs. This area subdivides into the *Heap* and the *Box Memory*.

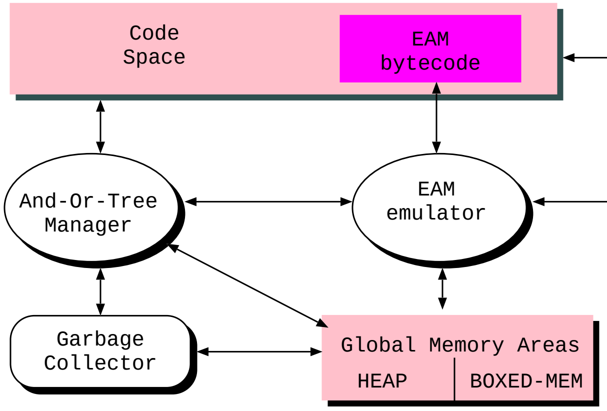


Fig. 2. General area memory usage.

The *Heap* holds lists and structures, that is Prolog's compound terms. The use of the Heap is very similar to the WAM's Heap, with the difference that on the BEAM Heap memory can not be recovered after backtracking. A Garbage Collector is thus necessary to recover space in this area (see [12]).

The *Box Memory* satisfies memory requests for the creation of and-boxes, or-boxes, local variables, external variables, and suspension structures. This memory must both deal with intensive requests efficiently, but we must be able to recover memory after the boxes being removed. The BEAM uses three routines to manage the *Box Memory*: `initialization`, `memory_request` and `memory_release`. A more detailed explanation on these routines can be found at [12].

4.1 BEAM: Data structures

The *Box Memory* stores the and-or-tree and is the main novel data-structure in the BEAM. In this section we briefly describe how the BEAM represents or-boxes, and-boxes, local variables and external references to local variables.

Or-boxes correspond to goals with open alternatives, represented as an array of *alternatives* (see figure 3). Each alternative includes a pointer to an array of arguments, code to execute, a *state*, and a pointer to the and-box for the clause, if it exists. The or-box also points back to the and-box and to the goal that created this or-box.

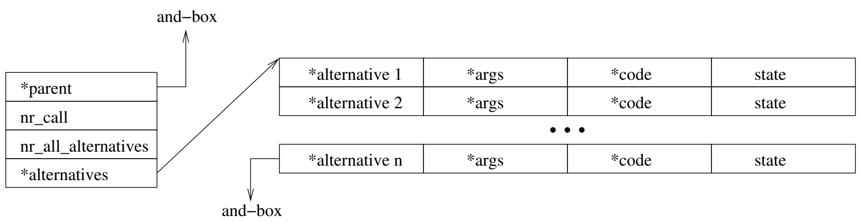


Fig. 3. Or-Box representation.

And-boxes represent clauses. Clauses have *local* variables, and subgoals (see figure 4). Subgoals may point to an or-box, and always include code to execute and state information. And-boxes also maintain a depth-counter (`level`) that is used to classify variables. And-boxes also maintain a list of `external` variables and a pointer back to the parent alternative.

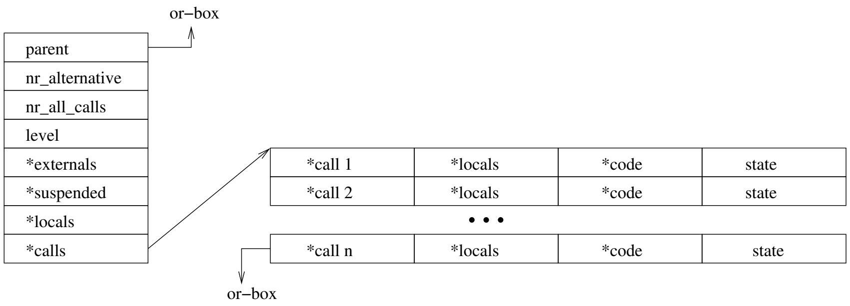


Fig. 4. And-Box representation.

Local Variables may belong to a single sub-goal, or may be shared by several goals (see figure 5). The *value* field is used to store the current value for a variable. Terms are represented using a copying based scheme, as in the WAM. Unbound variables are represented as self-pointers. Variables also point to their *home* and-box, and maintain a list of and-box *suspensions* on them.

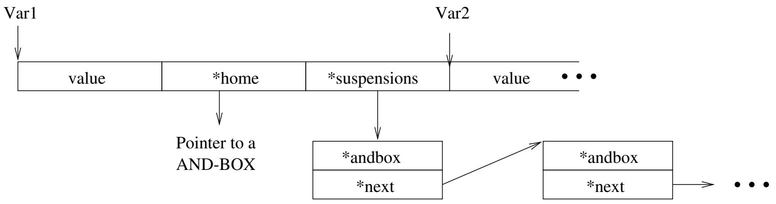


Fig. 5. Local Variables representation.

External Variables store bindings for variables external to the current and-box (see figure 6). Each such binding is represented as an external variable data-structure that includes a pointer to the variable definition, *local_var*, and the new value, *value*.

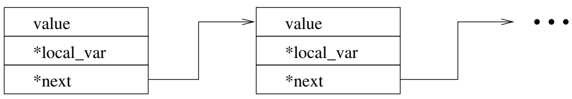


Fig. 6. External Variables representation.

Whenever a goal binds an external variable, the assignment is recorded in *both* the external variable structure and in the local variable itself. This way, whenever a descendent and-box wants to use the value of the external variable, it can do so simply by accessing the local variable directly. The external_variable data structure therefore works as the Prolog's trail, allowing the unwinding and rewinding of bindings performed at the current and-box.

5 Towards the RAINBOW

One of the aims of the EAM model is to exploit the all the major forms implicit parallelism available in logic programs, including or-parallelism, independent-and and dependent-and parallelism. Therefore, only a parallel implementation of the BEAM can take full advantage of the EAM model. We have designed a shared memory parallel computational model for BEAM, called RAINBOW, which we describe in detail here.

We describe the RAINBOW computational model and all the changes that are necessary to make to the BEAM in order to support parallelism. Then we describe how the RAINBOW memory organisation and how the data-structures in the BEAM must be changed. Next, we discuss scheduling strategies. Last, we study how parallelism affects the use of the BEAM computational rules.

5.1 The computational model

In the RAINBOW a set of workers, or processes, execute in parallel trying to exploit a shared and-or execution tree. One of the workers, we shall name it the *chair*, initiates the computation, as in BEAM, by creating the initial and-box. Parallelism is made available by marking this and-box as *shared*. The RAINBOW thus splits work into shared and non-shared (this is similar to the notion of public work in or-parallel systems [14]).

All workers but the chair start execution in scheduling mode, looking for their first work assignment. In our design, workers, when looking for work, try to find a worker publishing a the topmost marker for a box with possibly shared work. To get work from a shared box, the workers, and note this is true even for the box owner, have to ensure mutual exclusion so that no workers pick the same task. Therefore working on shared work is slower than working in non-shared worker.

We shall assume *demand-driven* scheduling in the RAINBOW: from time to time, each worker checks its marker to determine whether it needs to be updated to a box in its subtree with shared work. The termination condition is given thus:

(i) all the workers find themselves as idle looking for work;

(ii) all the worker's markers indicate that there is no shared work then computation terminates

5.2 Representing local variables

With multiple workers running there is always the possibility of more than one worker wanting to bind a value to a variable belonging to a shared and-box. To deal

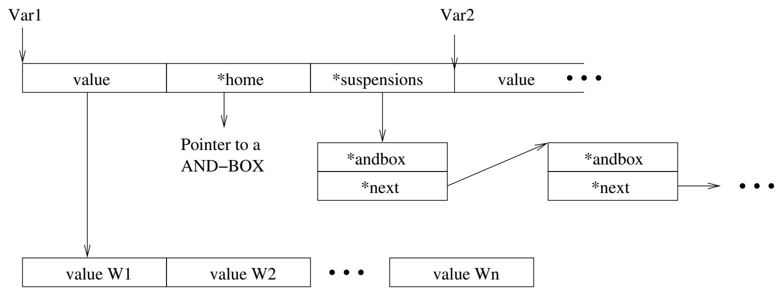


Fig. 7. Local Variable.

with this situation, local variables need special representation in a parallel systems. We believe two alternative proposals to be the most interesting for RAINBOW: full sharing and copying plus sharing.

5.2.1 Full Sharing

In this solution the memory data areas are organized in a way similar to the BEAM sequential implementation: the whole memory is shared by all the workers. The *Heap* and the *Box Memory* data areas are equally divided among the workers. Each worker manages its own memory areas. During normal worker execution workers behaves as in sequential BEAM, hence workers allocate memory in their own memory areas without having to synchronize with other workers. Locally allocated blocks will be seen by other workers whenever those blocks become part of the shared part of the execution tree.

We suggest that the binding problem for the shared RAINBOW can be addressed by using *version vectors* to store alternative bindings [3] or by using the Sparse Binding Array (SBA) [6].

If we use version vectors we have associated with each local variable a vector indexed by the worker-id. The `value` field of the local variable structure is initialized to point to the base of the vector. For a worker to assign a binding to a local variable it must add its worker-id as an offset to the base address of the vector stored in the `value` field for that variable. When the binding of a local variable becomes deterministic, we optimise by storing it directly in the `value` field associated with the variable, hence freeing the initial vector. This optimization may take place during a promotion phase. The and-boxes that were suspended on the local variable, just found to be deterministic receive a wake signal to force an environment check (as in the sequential BEAM).

If we use the SBA each cell in the box memory will have a private copy, the SBA: bindings to shared local variables are stored in this area.

One advantage of version vectors is that we may create the vectors lazily, that is, only when we first access a variable local to a shared and-box. To do so we must first lock the variable, as otherwise we could have a race between several workers trying to create their own vectors. In the RAINBOW this is not a problem as locking is necessary anyway, whereas in traditional or-parallel systems this would be a significant overhead. The SBA logically reserves sufficient space for the whole stacks, but it does have the advantage of avoiding the extra overhead of vector allocation.

An alternative to these two proposals is to use hash vectors, as suggested for ParAKL [17]. In our case, it was the complexity of the BEAM design made us feel more inclined towards the advantages of simpler data-structures such as version vectors and the SBA.

5.2.2 Copying Plus Sharing

An alternate solution we are considering is an hybrid that can use both copying and sharing. The key ideas are:

(i) workers generate work in fully private stacks;

(ii) shared work is copied to the *shared* work area;

(iii) the shared work area is available to every worker; the binding problem is address through the SBA.

We call this solution the COSMOS model (*C*opy-*O*nce, *S*hare-*M*any *O*bject *S*tore). Note that our design is such that shared work must not directly refer to private work.This solution has the following advantages:

(i) Private work is clearly separated from shared work: this should facilitate memory management.

(ii) The SBA only applies to shared space, resulting in much smaller SBAs per worker.

(iii) Accesses to the shared area will be most often read-only. The system should thus nicely support distributed shared memory architectures, such as OAR [1]

It also has a few drawbacks, namely:

(i) The advantages of COSMOS will quickly disappear if we need to store too many boxes in the shared areas;

The sequential BEAM has obtained good performance without using copying. Although copying has worked well for Penny, we believe that a shared-all approach or an hybrid approach such as COSMOS may avoid the large overheads that can be incurred with general logic programs if one uses copying.

5.2.3 Garbage Collection

Both memory organizations are designed to allow the implementation of a distributed garbage collector that does not need to stop all the workers to recover space. The key idea is that during execution, each worker must be able trigger its copy based garbage collector (GC) to work on its private memory areas in a similar way as in the BEAM sequential implementation (see [12]). By avoiding pointers from the private to the shared area, COSMOS guarantees that garbage collection can be performed independently. The all-shared approach requires a bit more care in this regard. Shared boxes will require reference counters, and we must establish some communication when shifting shared boxes.

5.3 Distributing Work

We will now describe in more detail data structures that we suggest towards supporting work distribution in RAINBOW.

The main data-structure is *Worker-Info*, a shared data structure, used to guide workers to available work in the system. It is an array, indexed by worker-id, in which each entry determines the marker of a worker. Markers identify, directly in the execution tree, a box with the topmost work in a worker's subtree (this contrasts with queue-based approach used in Penny [15]). Each entry of the worker-info data structure includes:

- `box_pointer`: a pointer to the worker's box marked as having shared tasks.
- `box_type`: indicates the type of the marked box: and-box or or-box.
- `depth_level`: specifies the depth-level of the marked box within the execution tree. This field is a copy of the same field as present in every and-box, however, we have included it to simplify the work search algorithm.
- `work_available`: the number of tasks still left for this box.

Workers, when looking for work, search the *worker_info* structure to determine the worker with the topmost marker still with shared work. Ideally, this worker is given as $Max(0, work\_available_0, \ldots, work\_available_{n-1})$.

The markers also separate private work from shared areas of a worker's execution. A private area for a worker is defined by its subtree below the box pointed by its marker. Within the private area the worker executes sequentially without having to synchronise.

There are two situations in which a worker must update its marker:

(i) *No shared work*: whenever a marker points to a box with no work left (i.e. the *available_work* entry is zero), it is updated by the owner worker to the root of its private subtree.

(ii) *No local work left*: whenever a worker exhausts its private subtree, it returns to its marked box (in the shared area). If there is no work left at this box, it goes into scheduling mode looking for work from another worker. If it succeeds in finding a new task, then when starting executing it, the worker creates a box and updates its marker to point to it.

5.3.1 An example

To illustrate the executional model, consider the situation described in figure 8a. It shows three workers, W_1 and W_2 actively working below, respectively, and-box `a21` and or-box `o31`, while worker W_3 is looking for work. Currently, the worker-info array shows that the topmost box with shared work is `o11` of worker W_1. Suppose, W_3 succeeds in grabbing a task from that box and starts executing it. Then, it naturally creates an and-box `a22`, and updates its entry in the worker-info array. Meanwhile, the box `o11` has no more work left and therefore worker W_1 must update its marker to the root of its private subtree. This is illustrated in figure 8b.

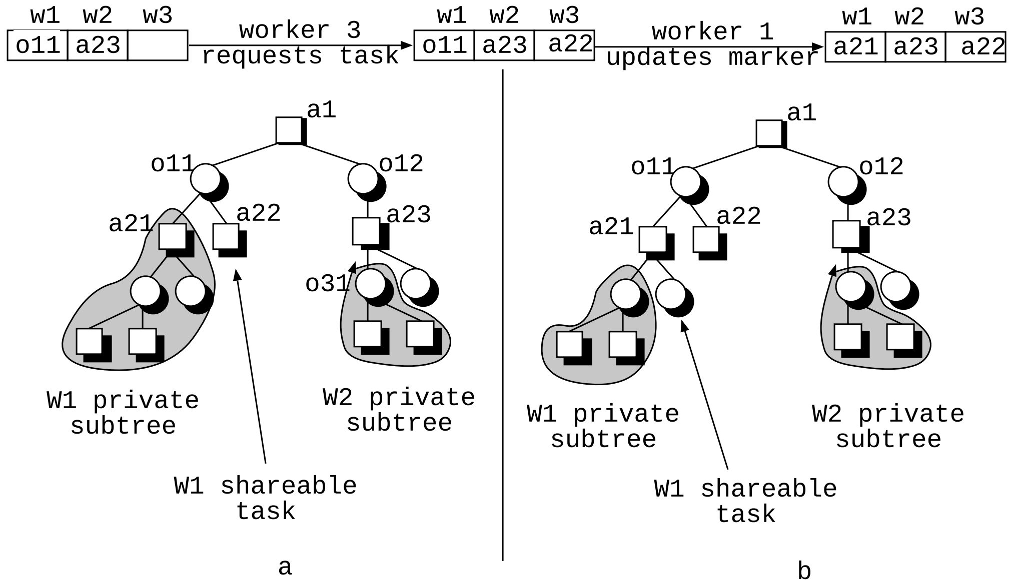


Fig. 8. Execution Tree in a paralell enviroment.

5.4 Extending And-Or-Boxes

In RAINBOW there are several workers exploiting the and-or execution tree. It is therefore necessary to be able to locate the workers in the tree, in order to share work and to avoid having multiple workers visiting the same branches. This is accomplished by extending the and-or boxes with two extra fields, the `lock` and the `workers` fields.

The `lock` field is used by workers to ensure mutual exclusion when changing the box contents.

The `workers` bitmap field is used to identify the workers that are working through the current box. A worker is known to be working on a box if its bit number is set to the active state. This field is used in situations such as forking or box simplification where it is necessary to determine all the workers working within a box or determine if a worker is unique within it. When a worker leaves a box, it unsets its bit in the `workers` field and checks whether the value of this field is zero. If that is the case, the worker checks the box `state` field to decide whether the box may be deleted.

5.5 Searching for work

The scheduling strategy adopted by the workers follows from our initial description of the execution model. Basically we distinguish two cases:

Local work search: workers during their execution always try first to get work from their private area, starting from their `current_box`. If they are not successful in the current box they move upwards in the tree, in which case they may eventually reach their `marker_box`. Getting work from a box in the private part of the tree is realized as in sequential BEAM whilst getting work from the `marker_box` has to be protected by the box's lock. Figure 9 shows the pseudo-code for the function `get_local_work()` that implements the strategy just outlined.

The `task` data structure includes all the necessary information for a worker to start computing a new task. It also includes a field, `found`, to mark whether the new task is valid or not.

Remote work search: whenever a work fails to find local work in its private tree area or in its marker box, it directs its work search towards the worker in the system that has the topmost marker box. That box is found off the `worker-info` array where it is registered the markers for each work with shared work and the depth level associated with the markers. The worker that has the marker with the least depth level value is chosen as a target for work sharing. This algorithm is given in pseudo-code in function `get_remote_work()` in figure 9.

5.6 Implications on the rewrite rules

In the context of a parallel execution some of the basic EAM rewrite rules need to be modified as well as their application may have to be thought carefully. We will address each in turn:

- `reduction`: the reduction rule can still be applied with no modifications, since it only changes the status on the worker's current box (i.e. it only affects the private tree area).
- `fork`: since in our computational model this rule is only applied when no other rule can, it does not pose any difficulties in a parallel setting. A worker is only allowed to perform a fork operation in a box, if there are no other workers installed in a subtree off that box. If there were workers below the current box, then most likely they would be work to be explored from the boxes pointed by the markers of those workers.
- `promotion`: in the parallel context this application of this rule becomes more

```
task get_local_work( )
{
  if (Is_Private(current_box)) {
    newtask=get_work_private(current_box);
    if (newtask.found==true)
      return newtask;
    current_box=current_box->parent; /* move current_box ptr upwards */
    return get_local_work();         /* and call get_work() again    */
  }
  /* get work from public area since current_box=marker_box */
  if (worker_info[workerID].work_flag>0) {
    LOCK(current_box->lock) {
    newtask=get_work_private(current_box);
    UNLOCK(current_box->lock);
    if (newtask.found)
      return newtask;
  }
  return get_remote_work();
}
task get_remote_work()
{
  targetWID= find_worker_with_topmost_marker();
  current_box= worker_info[targetWID].box_pointer;
  if (worker_info[targetWID].work_flag>0) {
    LOCK(current_box->lock) {
    newtask=get_work_private(current_box);
    if (newtask.found) {
      set_bitmap(current_box->workers, workerID);
      UNLOCK(current_box->lock);
      return newtask;
    }
    UNLOCK(current_box->lock);
  }
  return get_remote_work();
}
```

Fig. 9. Scheduler Algorithm.

complex. For the rule to be applied, one has to ensure that the parent and-box does not have any other active workers in it. This requires mutual exclusion on the parent and-box.

Promotion is a deterministic rule widely used, hence having to lock boxes to realize it may pose a serious overhead to the execution. We address this problem by disallowing an and-box to be fully promoted. Instead, we allow the constrains to be promoted to the parent and-box but the and-box itself is kept separate. Although, the and-boxes are kept separate, any variable that is local to the current and-box also becomes local to the parent and-box. This is achieved by making the current and-box `Depth-level` parameter equal to that of the parent and-box.

6 Conclusion

We have presented our first design for RAINBOW, a parallel implementation of the BEAM. RAINBOW builds on the good results we have obtained with our implementation of the Extended Andorra Model with Implicit Control and on previous experience with parallel systems such as ParAKL and Penny.

We propose several alternative strategies for environment representation in RAINBOW. Version vectors are a possible solution that although not very popular for Prolog may be quite more appealing in our context. An alternative is COSMOS, a solution where only shared work is stored in shared memory. We also propose a work distribution and scheduling strategy for RAINBOW based on minimising the amount of sharing in the system.

We plan to implement RAINBOW using the BEAM and shared-memory architectures. Our work is also designed to support parallel execution in Abreu's OAR computational framework.

Acknowledgments

We would like to gratefully acknowledge the contributions we received from Gopal Gupta, Enrico Pontelli, Kish Shen, and Rong Yang. This work is partially supported by the projects Melodia (JNICT PBIC/C/TIT/2495/95), Dolphin (PRAXIS/2/2.1/TIT/1577/95), CLoP (CNPq), PLAG (FAPERJ) and by Fundação para a Ciência e Tecnologia.

References

[1] S. Abreu. Towards the OAR Language and Computational Model. In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Languages*, July 2000. to appear.

[2] K. A. M. Ali and R. Karlsson. The Muse approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162 (or 129–160??), Apr. 1990.

[3] A. Ciepielewski and S. Haridi. A formal model for or-parallel execution of logic programs. In *IFIP 83 Conference*, pages 299–305. North Holland, 1983.

[4] K. L. Clark, F. G. McCabe, and S. Gregory. IC-PROLOG – language features. In K. L. Clark and S. A. Tärnlund, editors, *Logic Programming*, pages 253–266. Academic Press, London, 1982.

[5] A. Colmerauer. Theoretical Model of Prolog II. In M. van Caneghen and D. H. D. Warren, editors, *Logic Programming and its Applications*, pages 3–31. Ablex Publishing Corporation, 1986.

[6] M. E. Correia, F. Silva, and V. Santos Costa. The SBA: Exploiting orthogonality in OR-AND Parallel Systems. In *Proceedings of the 1997 International Logic*

Programming Symposium, pages 117–131. MIT Press, October 1997. Also published as Technical Report DCC-97-3, DCC - FC & LIACC, Universidade do Porto, April, 1997.

[7] D. DeGroot. Restricted and-parallelism. In H. Aiso, editor, *International Conference on Fifth Generation Computer Systems 1984*, pages 471–478. Institute for New Generation Computing, Tokyo, 1984.

[8] G. Gupta and D. H. D. Warren. An Interpreter for the Extended Andorra Model. Internal report, University of Brsitol, 1991.

[9] M. V. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[10] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the International Logic Programming Symposium*, pages 167–186. MIT Press, October 1991.

[11] R. Lopes and V. Santos Costa. *Parallelism and Implementation of Logic and Constraint Logic Programming*, chapter The BEAM: Towards a first EAM Implementation, pages 87–106. Nova Science, Inc., 1999.

[12] R. Lopes and V. Santos Costa. A Performance Analysis of the BEAM Memory Manager. In *CL2000 First Workshop on Memory Management in Logic Programs*, Technical Report of Dept. Comp. Science, K.U.Leuven, July 2000.

[13] R. Lopes and V. Santos Costa. The BEAM: A first EAM Implementation. Technical report, DCC-FC & LIACC, Universidade do Porto, 2000.

[14] E. Lusk, R. Butler, T. Disz, R. Olson, R. A. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[15] J. Montelius and K. A. M. Ali. An And/Or-Parallel Implementation of AKL. *New Generation Computing*, 14(1), 1996.

[16] J. Montelius and P. Magnusson. Using SIMICS to evaluate the Penny system. In J. Małuszyński, editor, *Proceedings of the International Symposium on Logic Programming (ILPS-97)*, pages 133–148, Cambridge, Oct. 13–16 1997. MIT Press.

[17] R. Moolenaar and B. Demoen. A parallel implementation for AKL. In *Proceedings of the Programming Language Implementation and Logic Programming: PLILP '93, Tallin, Estonia*, pages 246–261, 1993.

[18] L. Naish. *Negation and Control in Prolog*. Lecture notes in Computer Science 238. Springer–Verlag, 1985.

[19] E. Pontelli and G. Gupta. Dependent and Extended Dynamic Dependent And-parallelism in ACE. *Journal of Functional and Logic Programming*, 1999(Special Issue 1), April 1999.

[20] A. Porto. Epilog: A language for extended programming in logic. In J. Campbell, editor, *Implementations of Prolog*, pages 268–278. Ellis Horwood, 1984.

[21] V. Santos Costa, A. Srinivasan, and R. Camacho. A note on two simple transformations for improving the efficiency of an ILP system. In *Proceedings of ILP'2000*, July 2000. Proceedings to appear in Springer-Verlag LNAI series.

[22] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[23] K. Shen. Exploiting And-parallelism in Prolog: the Dynamic Dependent And-parallel Scheme (DDAS). In *Proceedings of the 1992 Joint International Conference and Symposium on Logic Programming*, 1992.

[24] D. H. D. Warren. The Andorra model. Presented at Gigalips Project workshop, University of Manchester, March 1988.

[25] D. H. D. Warren. Extended Andorra model. PEPMA Project workshop, University of Bristol, October 1989.

[26] D. H. D. Warren. The Extended Andorra Model with Implicit Control. Presented at ICLP'90 Workshop on Parallel Logic Programming, Eilat, Israel, June 1990.

Parallel Evaluation of Answer Sets Programs Preliminary Results

O. El Khatib and E. Pontelli

Laboratory for Logic, Databases, and Advanced Programming Dept. of Computer Science New Mexico State University {okhatib,epontell}@cs.nmsu.edu

Abstract

Answer Sets Programming is a novel programming paradigm that has recently gained popularity within the non-monotonic reasoning community. The paradigm builds on the notion of answer sets (aka stable models) and views programs as collection of constraints on a domain of models. In this work we propose a preliminary investigation aimed to develop technique for parallelizing execution of ASP programs. We present a first prototype of a parallel ASP engine and provide preliminary performance figures.

1 Introduction

It is becoming increasingly clear that, to fully realize the potential of computer revolution, computer scientists need to develop a *systematic methodology* for the design and construction of advanced software systems. These systems should be capable of basing their behavior on knowledge about their environment and on the exchange of complex information and services with other systems as well as with humans. This realization led to recent advances in the development of programming paradigms centered around the notion of *knowledge*. These paradigms strive to reduce a substantial part of the programming process to the description of objects comprising the domain of interest and relations between these objects. The resulting description can be queried to establish truth or falsity of statements about the domain, or to find objects satisfying various properties. Knowledge can also be updated, modified, and used to make decisions on actions which need to be taken in a given situation to achieve certain goals. Various languages, e.g., database query languages (e.g., SQL), Prolog, Frame, Rule, and Graph based systems, support this approach to programming. Normally, they come together with *inference engines*, tailored to a particular form of syntax in a given language. Though different in style, all these languages are based on *classical logic*, developed to model mathematical reasoning. This severely limits their ability to represent common-sense knowledge and reasoning which is crucial for development of sophisticated knowledge based systems. In the last ten years we witnessed a rapid development of alternative logical systems, called *non-monotonic logics* [6]—which allow new axioms to retract existing theorems and result to be more adequate for common-sense reasoning.

In the last couple of years a novel programming paradigm has arisen, called *Answer Sets Programming (ASP)*, which builds on the mathematical foundations of logic programming and non-monotonic reasoning. ASP offers novel and very declarative solutions in a number of *well-defined application areas*, including intelligent agents, planning, and software verification. In spite of the considerable interest that ASP has recently attracted, it still lacks some of the fundamental components needed to assert it as a practical and widely usable programming paradigm. ASP benefits from solid and well-developed mathematical foundations, but its programming aspects still require considerable research. In particular, we need to

- Develop efficient implementations of inference engines for ASP, and
- Develop methodology for representing and manipulating knowledge.

The goal of this project is to tackle these issues, using constraint solving and parallel execution technology. In particular, in this paper we present some preliminary ideas that have been used to address the first issue.

2 Answer Sets Programming

2.1 Answer Sets Semantics

Answer Sets Semantics (AS) [15] (a.k.a. *Stable Models* semantics) was designed in the mid 80s as a tool to provide semantics for logic programming with *negation* [5]. Traditional logic programming [24] provides the ability to derive only *positive consequences* from a program. However, in a large number of cases it is useful to reason also about negative consequences, by allowing negative knowledge to be inferred and allowing negative assumptions in the rules of the program. The presence of negation leads to a natural support for non-monotonic reasoning, and the availability of efficient computational mechanisms provides a natural setting for the study of proof systems for non-monotonic reasoning.

The introduction of negation in logic programming leads to various complications; in particular, it leads to the loss of a key property of logic programming, the existence of a *unique* intended model for each program—i.e., there is no ambiguity in what is true and what is false w.r.t. logic programming programs. Two classes of proposals have been developed to tackle the problem of providing semantics to logic programming with negation. The first class [4,40] attempts to reconstruct a *single intended model*, either by narrowing the class of admissible programs (e.g., *stratified programs* [37]) or by switching to 3-valued logic—admitting that formulae can be not only true or false, but also undefined. The second class instead admits the existence of a *collection* of intended models (*answer sets*) for a program [11,15]. *Answer Sets* is the most representative approach in this class, and has been recognized to provide the "right" semantics for logic programming with negation—e.g., it subsumes the semantics of standard logic programming and the intended model of stratified programs [5].

AS relies on a simple definition. Given a program P (for simplicity let us assume it does not contain variables) and given a tentative model M, we can define a new program P^M (the *reduct* of P w.r.t. M) by:

- removing all rules containing negative elements which are contradicted by the model M;
- removing all negative elements from the remaining rules.

Thus, P^M contains only those rules of P that are applicable given the model M. Furthermore, P^M is a standard logic program, without negation, which admits a unique intended model M' [24]. M is an *answer set* (a.k.a. *stable model*) if M and M' coincide. Intuitively, an answer set contains all and only those atoms which have a justification in terms of the applicable rules in the program. These models can be proved to be *minimal*. In general, a program with negation may admit different answer sets.

Example 2.1 *If we have a database indicating people working in different departments*

```
dept(hartley,cs).            dept(pfeiffer,cs).
dept(gerke,math).            dept(prasad,ee).
```

and we would like to select the existing departments and one (arbitrary) representative employee for each of them:

```
depts_employee(Name,Dep) :- dept(Name,Dep), not other_emps(Name,Dep).
other_emps(Name,Dep)     :- dept(Name1,Dep), depts_employee(Name1,Dep),
                            neq(Name , Name1).
```

The rules assert that `Name/Dep` *should be in the solution only if no members of the same department have been selected. AS produces 2 answer sets:*

$$\{\langle \mathtt{hartley}, \mathtt{cs}\rangle, \langle \mathtt{gerke}, \mathtt{math}\rangle, \langle \mathtt{prasad}, \mathtt{ee}\rangle\}$$

$$\{\langle \mathtt{pfeiffer}, \mathtt{cs}\rangle, \langle \mathtt{gerke}, \mathtt{math}\rangle, \langle \mathtt{prasad}, \mathtt{ee}\rangle\}$$

2.2 Answer Sets Programming: a Novel Paradigm

As recognized by a number of authors [25,27], the adoption of AS requires a *paradigm shift* to reconcile the peculiar features of AS with the traditional program view of logic programming. In first place, under AS, each program potentially admits more than one intended model. This ends up creating an additional level of *non-determinism*—specifically a form of *don't know* non-determinism—

on top of the non-determinism typically identified in traditional logic programming (and directly ensuing from the use of resolution). The presence of multiple answer sets complicates the framework in two ways. First of all, we need to provide programmers with a way of handling multiple answer sets. One could attempt to restore a more "traditional" view, where a single "model" exists. This has been attempted, for example, using *skeptical semantics* [25], where an atom is considered entailed from the program only if it is true in *each* answer set. Nevertheless, skeptical semantics is often inadequate—e.g., in many situations it does not provide the desired result (see Example 2.1), and in its general form provides excessive expressive power [25,26]. The additional level of non-determinism—removed by skeptical semantics—is indeed a real need for a number of applications. It is also possible to see some similarities between this and some of the proposals put forward in other communities—such as the *choice* and *witness* constructs used in the database community [22,1,34].

The presence of multiple answer sets, in turn, leads to a new set of requirements on the *computational mechanisms* used. Given a program, the goal of the computation is not to provide a goal-directed tuple-at-a-time answer (i.e., a true/false answer or a substitution), as in traditional logic programming, but the goal is to return *whole answer sets*. The traditional resolution-based control used in logic programming is largely inadequate, and should give place to different forms of control and execution mechanisms.

In this project we embrace a different view of logic programming under AS, interpreted as a *novel programming paradigm*—that we will refer to as *Answer Sets Programming (ASP)*. This term was originally created by Lifschitz, and nicely blends the notion of *programming* with the idea that the entities produced by the computation are *answer sets*. The notion of ASP has been advocated by others during the last two years: Niemela has proposed AS as a *constraint programming paradigm* [27], while Marek & Truszczynski have coined the term *Stable Logic Programming* [25] to capture the notion we are describing. In simple terms, the goal of an ASP program is to identify a *collection of answer sets*—i.e., each program is interpreted as a specification of a collection of *sets of atoms*. Each rule in the program plays the role of a *constraint* [27] on the collection of sets specified by the program: a generic rule

$$Head :- B_1, \ldots, B_n, not\, G_1, \ldots, not\, G_m$$

requires that whenever $B_1, \ldots, B_n$ are part of the answer set and $G_1, \ldots, G_m$ are not, then *Head* has to be in the answer set as well. The shift of perspective from logic programming to ASP is very important. The programmer is lead to think about writing programs as manipulating sets of elements, and the outcome of the computation is going to be a collection of sets. This perspective comes very natural in a large number of application domains—graph problems deal with set of nodes/edges, planning problems deal with sets of actions or states, etc.

Example 2.2 *[23] The following simple ASP program computes the hamiltonian cycles of a graph.*

```
%% Graph:
node(0). node(1). node(2). node(3).
edge(0,1). edge(0,2). edge(1,3). edge(1,2). edge(2,3). edge(3,0).

%% choice rules to select one edge
in(U,V)      :-   node(U), node(V), edge(U,V), not nin(U,V).
nin(U,V)     :-   node(U), node(V), edge(U,V), not in(U,V).

%% each node is traversed once
:- node(U), node(V), node(W), neq(V,W), in(U,V), in(U,W).
:- node(U), node(V), node(W), neq(U,V), in(U,W), in(V,W).
reachable(U) :- node(U), in(0,U).
reachable(V) :- node(U), node(V),reachable(U), in(U,V).

%% Guarantees that each node is reachable
:- node(U), not reachable(U).
```

For the graph in the example, the program admits a single answer set:

$$\{in(0,1), in(1,2), in(2,3), in(3,0)\}$$

ASP has received great attention in knowledge representation and deductive databases, as it enables to represent default assumptions, constraints, uncertainty and nondeterminism *in a direct way* [6]. The automation of nonmonotonic reasoning may well rely upon automatic ASP, through the well-studied equivalences with, e.g., autoepistemic logic. ASP is related both ideally and through formal equivalences to the algorithmic study of satisfaction of boolean formulas (SAT). It is believed that ASP encoding of traditionally hard problems should be more compact than SAT encoding. For instance, [25] argue that ASP encodings of the Hamiltonian cycle problem are *asymptotically* more concise than SAT ones. This implies that, other things being equal, ASP interpretations can be as efficient as satisfiability and even as constraint satisfiability systems. For example, [12] reports ASP solutions of planning problems in time comparable to ad-hoc planning algorithms. Finally, ASP syntax corresponds to *DATALOG*$^{\neg}$ of deductive databases, and should make database access transparent and straightforward.

3 Sequential Execution of ASP Programs

Various execution models have been proposed in the literature to support computation of answer sets and some of them have been applied as inference engines to support ASP systems [8–10,27,14].

In this project we propose to adopt an execution model which is built on the ideas presented in [27] and effectively implemented in the popular *Smodels* system [28]. The choice is dictated by the relatively simplicity of this execution model and its apparent suitability to exploitation of parallelism. The overall structure of the system is illustrated in Figure 1. The system consists of two parts, a compiler—we are currently using the `lparse` system [39] to carry on the compilation part—which is in charge of creating atom tables and performing program grounding, and an engine, which is in charge of computing the answer sets of the program. Our interest is focused on the engine component. Note that the whole system is implemented as a C program (we are not using a meta-interpreter approach to solve the problem).

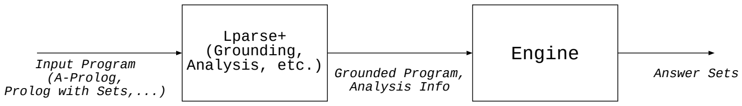


Fig. 1. Overall Structure of the System

A detailed presentation of the structure of the smodels execution model [38] is outside the scope of this paper. In the rest of this section we propose an intuitive overview of the basic execution algorithm. [1]

Figure 2 presents the overall execution cycle for the computation of stable models.

```
function compute (Π : Program, A : LiteralsSet)
begin
    B := expand(Π, A) ;
    while ( (B is consistent) and (B is not complete) )
         l := choose_literal(Π, B);
         B := expand(Π , A ∪ { l }) ;
    endwhile
    if (B stable model of Π) then
         return B;
end
```

Fig. 2. Basic Execution Model for ASP

As from Figure 2, the computation of stable models can be described as a non-deterministic process—needed since, in general, each program Π may admit multiple distinct stable models. Non-determinism originates from the execution of $\texttt{choose\_literal}(\Pi, B)$, which selects a literal l which satisfies the following properties:

- the literal l corresponds to an atom which appears negated in the program Π

[1] The presentation has been simplified and does not have any pretense of completeness.

- neither l nor its opposite are present in B

Each non-deterministic computation can terminate in three different ways:

(i) successfully when B is indeed a stable model of the original program Π

(ii) unsuccessfully because a conflict has arisen—i.e., there exists an atom a such that both a and *not* a have been introduced in B

(iii) unsuccessfully because B has been completely expanded (i.e., the procedure `choose_literal` cannot produce any literal), it is consistent, but it does not represent a stable model of the program Π. This situation typically occurs when a positive literal a is introduced in B (e.g., it is selected by `choose_literal`) but the rules of the program do not provide a support for the presence of a in a stable model.

As in traditional execution of Prolog, non-determinism is handled via backtracking.

The `expand` procedure mentioned in the algorithm in Figure 2 is intuitively described in Figure 3. This procedure proceeds by repeatedly applying expansion rules to the given set of literals until no more changes are possible. The expansion rules are derived from the program Π and allow to determine the truth status of literals. The following rules have been considered for the computation:

(i) *forward rule:* if the rule $h \leftarrow a_1, \dots, a_n, not\, b_1, \dots, not\, b_m$ is in Π and all the literals in the body of the rule are in B, then also h can be added to B

(ii) *nullary rule:* if there are no rules having the atom a as a head, then $not\, a$ can be added to B

(iii) *single positive rule:* if the atom h is in B and there is a single rule $h \leftarrow a_1, \dots, a_n, not\, b_1, \dots, not\, b_m$ in Π having h as head, then all the elements of the body of the rule can be added to B

(iv) *negative rules:* if the literals $not\, h, l_1, \dots, l_{i-1}, l_{i+1}, \dots, l_m$ are in B and the rule $h \leftarrow l_1, \dots, l_m$ is in Π, then $\bar{l}_i$ can be added to B, where $\bar{l}$ indicates the complement of the literal l

```
function expand (Π : Program, A : LiteralsSet)
begin
    B := A ;
    while ( B ≠ B' ) do
         B' := B;
         B := apply_rule(Π, B);
    endwhile
    return B ;
end
```

Fig. 3. Expand procedure

Efficient implementation of this procedure requires considerable care to avoid unnecessary steps, e.g., by

- dynamically transforming the program Π by removing rules that cannot be applied
- using heuristics in the implementation of the `choose_literal` procedure

4 Parallel Execution of ASP Programs

The structure of the computation of answer sets illustrated previously can be easily seen as an instance of a constraint-based computation, where

- the application of the expansion rules (in the `expand` procedure) represents the propagation step in the constraint computations
- the selection of a literal in the `choose_literal` procedure represents a labeling step

From this perspective, it is possible to identify two traditional sources of non-determinism in the computation:

- *horizontal non-determinism:* which arises from the choice of the next expansion rule to apply
- *vertical non-determinism:* which arises from the choice of the literal to add to the partial answer set

These two forms of non-determinism bear strong similarities respectively with the *don't care* and *don't know* non-determinism traditionally recognized in constraint and logic programming.

The goal of this project is to explore avenues for the exploitation of parallelism from these two sources of non-determinism. In particular, we will use the terms

- *vertical parallelism* to indicate a situation where separate threads of computation are employed to alternatives arising from vertical non-determinism
- *horizontal parallelism* to indicate the use of separate threads of computation to concurrently apply different expansion rules to a given set of literals

Preliminary experimental considerations have underlined the difficulty of this task. In particular

- considerable research has been invested in the design of algorithms and heuristics to provide fast computation of answer sets; realistically, we desire to maintain as much as possible of this technology;
- the structure of the computation (seen as a search tree where the points of non-determinism correspond to the nodes of the tree) can be extremely irregular and ill-balanced.
- neither of the two forms of non-determinism dominates on the other—this means

that certain ASP programs performs very few choices of literals (i.e., calls to `choose_literal`), while spending most of the time in doing expansions, while other programs perform a large number of choices. The structure of the computation is heavily dependent on the nature of the programs. There are programs which lead directly to answer sets with little or no choices (e.g., positive programs), other which invest most of their time in searching through a large set of literals for compatible truth assignments.

- branches of the search tree are frequently very short

This leads to the following conclusions:

- granularity control is a serious issue—techniques have to be adopted to collapse branches of the search tree and produce parallel computations of adequate grain size
- both vertical and horizontal parallelism need to co-exist within the same system; although it is unclear whether we need to explore the issue of concurrent exploitation of both forms of parallelism, it is instead clear that both vertical and horizontal parallelism may need to be (alternatively) employed during the execution of a single program.

4.1 Horizontal Parallelism

A first direction for exploitation of parallelism from the execution of ASP programs can be achieved by parallelizing the steps occurring along *one* branch of the search tree. This implies, as in Fig. 2, the parallel evaluation of the individual executions of `expand`. `expand` determines what are the literals whose truth value is immediately determined by the partial answer set B. This is achieved by applying in various ways the program rules (e.g., forward and backward chaining), to expand the partial answer set, without performing any choice. Each rule can provide a different contribution, depending on the partial answer set. *Horizontal parallelism* can be achieved by allowing concurrent application of different rules to expand the partial answer set. Our approach involves performing a partitioning of the set of program rules between different processors (*and-agents*—to stress the similarity between this form of parallelism and and-parallelism in traditional parallel logic programming), so that each agent is in charge of applying a given set of rules (*program fragment*) to the partial answer set. In spirit, this is close to the work proposed for parallelizing computation in the context of consistency-based constraint solving [7,35]. In a *Shared Memory System (SMP)*, the different agents manipulate a common representation of the partial answer set, and apply rules belonging to their fragment to expand the answer set. We will consider 3 approaches for computing fragments.

(1) Static Fragmentation: the partitioning of the program in fragments is performed statically, by the preprocessor. The preprocessor receives in input information con-

cerning the number of and-agents in the system, and creates *one fragment for each agent*. The partitioning is done based on the dependency graph of the program. Partitioning is aimed at

- maintaining a good job balancing between agents
- maintaining a good locality of the computation

It is important to observe that there are large classes of regular programs (e.g., stratified programs [37]) for which achieving a good partitioning is relatively simple.

(2) Dynamic Fragmentation: dynamic fragmentation relies on complete dynamic scheduling of horizontal parallel work. Each and-agent maintains a job queue, which is accessible by all other agents. During the execution of `expand`, each and-agent detects rules which are triggered for application (this can be done efficiently, using known run-time program representations [13]), and place them in the job queue. Each agent can take unexplored rules from the job queues and apply them to the current partial answer set. This approach attempts to maximize the time agents spend executing rules for extending the partial answer set.

(3) Hybrid Fragmentation: this approach attempts to create a balance between static and dynamic fragmentation. The preprocessor is in charge of fragmenting the program according to the previously described criteria (e.g., locality), but without any regard for the number of and-agents. The emphasis is on creating fragments which can potentially provide work of reasonable grain size and which interact with other fragments only in a well-behaved fashion (e.g., only one fragment can generate literals belonging to a certain class). Perfect behavior is hard to accomplish, since rules can be used both in forward and backward fashion, but the freedom in creating different fragments allows to design a representation of the answer set which reduces the risk of contention. In hybrid fragmentation, dynamic scheduling is not used on individual rules (as in dynamic fragmentation), but it is used at the level of fragments—fragments which contain rules activated by a new literal in the answer set are deposited in a job queue. Each and-agent has the ability of selecting a fragment from the job queue and starting its execution.

It is unclear which scheme will provide better performance. Each one provides a different balance between flexible scheduling and run-time overhead. We are currently exploring these issues. We are in particular trying to integrate in the preprocessor algorithms to analyze the dependence structure of the program and decide whether one of the strategies is likely to provide better performance.

4.2 Vertical Parallelism

Alternative choices of literals during the derivation of answer sets (`choose_literal` in Figure 2) are *independent* and can be concurrently explored, generating separate threads of computation, each potentially leading to a distinct answer set. Thus, *vertical parallelism* parallelizes the computation of *different* answer sets.

As ensues from research on parallelization of search tree applications and non-deterministic programming languages [33,17,3,20], the issue of designing the appropriate data structures to maintain the correct state in the different concurrent branches, is essential to achieve efficient parallel behavior. It must be observed that straightforward solutions to similar problems have been proved to be ineffective, leading to unacceptable overheads [33,31].

The architecture for vertical parallel ASP that we envision is based on the use of a number of ASP engines (*or-agents*) which are concurrently exploring the search tree generate by the search for answer sets—specifically the search tree whose nodes are generated by the execution of the `choose_literal` procedure. Each or-agent explores a distinct branch of the tree; idle agents are allowed to acquire unexplored alternatives generated by other agents. The major issue in the design of such architecture is to provide efficient mechanisms to support this sharing of unexplored alternatives between agents. Two major approaches can be considered to tackle this class of problems [19,36]. *Trailing* relies on the use of a *shared representation* of the search tree and on the use of separate trail areas to keep track of the differences between different branches. *Copying* relies instead on the duplication of common parts of the search tree, to allow each individual agent to have a *private* view of its branch. These two schemes have provided different degrees of benefit to different classes of problems relying on search trees—e.g., trailing has been frequently used in constraint programming systems [2,21,32,36], while copying has been proved effective in parallel logic programming systems [3,18]. We propose to compare these two schemes in the context of ASP. Observe also that ASP computations generate *irregular* parallel computations, with potentially unbalanced search trees. As a consequence, any efficient parallel ASP engine requires the ability to perform *dynamic scheduling*. This justifies the choice of an engine design where different agents are capable of traversing a shared representation of the search tree to detect and acquire unexplored alternatives. We will explore a number of optimization schemes to improve efficiency of these mechanisms, via run-time transformations of the search tree [16,30]—e.g., flattening the tree to facilitate the sharing of work.

5 Implementation

This project has recently started. We are currently focusing on two directions, which include:

- performance analysis of existing sequential ASP systems: the objective is to study what are the "typical" patterns arising from execution of ASP programs, in order to determine which parts of the computation can take better advantage from the exploitation of parallelism.
- development of an inference engine exploiting vertical parallelism during computation of ASP programs.

In the rest of this paper we discuss the preliminary results achieved from the development of the vertical parallel inference engine.

5.1 Program Representation

The inference engine receives as input a ground representation of the ASP program—for the sake of simplicity in this first part of the project we rely on the use of a standard grounding pre-processor, specifically the `lparse` program [39] used by many existing ASP engines (e.g., smodels and DeRes).

Programs are internally represented using cross-linked data structures. These data structures are derived from the program representation described in [13] which allow linear-time computation of declarative closures of logic programs.

As done in the implementation of smodels [38], each rule is represented as in Figure 4, where

- Head is a pointer to the atom representing the head of the rule
- Positives and Negatives are lists of pointers to the atoms which respectively appear positive and negative in the body of the rule
- pos and neg respectively counts the number of positive and negative literals in the body of the rule which are not covered by the current model
- done is a flag indicating whether the rule has already been used in the computation of the answer set
- active is a flag indicating whether the rule is active or inactive with respect to the current answer set

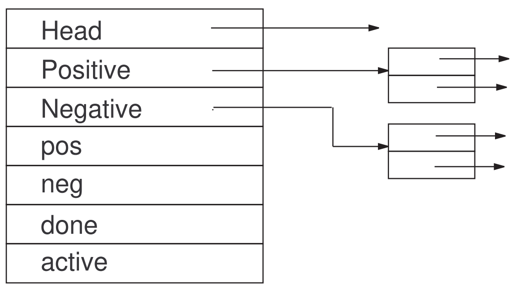


Fig. 4. Representation of a Rule

A separate list maintains all the atoms that are present in the program; the representation of each atom includes information such as:

- truth value of the atom (if the atom is already present in the answer set)
- pointers to the rules which have such atom as head
- pointers to the rules which have such atom as a positive (negative) literal in their body

5.2 Exploiting Vertical Parallelism

5.2.1 Introduction

One of the goal of this project is to determine which program organization is more suitable to support efficient exploitation of vertical parallelism. Traditionally two approaches have been considered in the development of constraint-based systems [36]:

- *copying* which involves creation of separate copies of the computation state at each choice point; the copies are used to restore the appropriate state whenever backtracking is performed.
- *trailing* requires keeping track of the modifications which are made to the computation state (using a *trail* structure). Whenever backtracking occurs, the content of the trail is used to restore the appropriate state.

Research in comparing these two approaches for the development of constraint-based systems is rather scarce ([36] provides a nice discussion on these issues). In this project we propose to compare both these approaches. It is in fact unclear which of the two methods better suits the needs of

- computation of answer sets
- support for exploitation of vertical parallelism

In particular, it is important to observe that

- smodels, an efficient sequential engine for the execution of ASP programs, relies on a technology which resembles a trailing scheme;
- in traditional logic programming, copying technology has proved highly suitable to support or-parallelism (which is strongly related to vertical parallelism).

We have thus started with the development of a copying-based implementation of ASP and we propose to continue with an analysis of its performance against a trailing-based implementation. We expect to achieve the best performance and exploitation of parallelism through an accurate balance of copying and trailing (as discussed in [29]).

5.2.2 Current Implementation

The current systems has been developed using Posix threads on a Sun Enterprise multiprocessor.

Each thread (*agent*) implements an independent inference engine for ASP computation. A single shared task stack has been introduced to support exchange of work between the different agents. Entries in the task stack replace the creation of choice points during execution of the program. Each time an agent executes `choose_literal` and selects a literal ℓ, a new entry is created in the task stack, containing a copy of the current partial answer set and the literal $\bar{\ell}$. Idle threads can extract an entry $\langle B, \ell \rangle$ from the task stack, install the partial answer set B, and continue the computation by expanding B with the new literal ℓ.

Figure 5 presents the overall structure of the algorithm executed by each agent.

```
procedure agent(B, atom x);
  add x to B to get NB
  store <B, not x> in the task stack
  A = expand(NB)
  if A is inconsistent then
        return no solution
        agent gets idle
  if all atoms are covered by A then
        A is the solution
        agent gets idle
  else
     subtract atoms_of(A) from atoms_set to get left_atoms
     pick first atom from left_atoms and call it x
     call agent(A, atom x)

  if agent idle then
      select <B,x> from task stack
      call agent (B, x)
```

Fig. 5. Structure of an Agent

The solution adopted is rather naive and is mostly aimed at getting a first feeling for the complexity of the problem. The creation of separate copies of the partial answer set can have a potential impact on the overall performance—observe that answer sets can grow very large as the computation proceeds. A solution which uses separate task stacks and provides incremental copying of partial answer sets is currently under implementation and is expected to provide considerable lower overhead.

5.2.3 Preliminary Results

We have just obtained a first running version of our engine with vertical parallelism. The results are still not satisfactory and considerable work has yet to be done to make it efficient and competitive. Table 1 presents some preliminary performance results obtained on four benchmarks. Benchmarks 1, 3, and 4 are drawn from the test pool which is part of the standard distribution of the smodels system, while benchmark 2 is a synthetic benchmark automatically generated using random functions. As can be seen the speedups achieved are relatively small, and this is due to two major factors:

- the relatively rough way in which the sharing of work between threads is currently performed—which leads to copying of excessively large amounts of data (repeated copying of the same parts of answer sets, which are repeated in the task stack);

- the complete lack of any form of granularity control, which leads to excessive task switching and contention on the locks which protects the access to the task stack.

We expect to obtain considerably better results once these two issues are properly tackled. On the other hand, it is important to observe that the basic engine we have developed, constructed using constraint programming technology and yet unoptimized, is considerably more efficient than the current distribution of smodels, as indicated in table 2. We believe this is due in part to the overhead of smodels in handling alternative kind of rules (our system currently handles exclusively standard logic programming rules).

Another aspect that is important to observe regarding these benchmarks is the actual distribution of effort during the computation. The actual implementation of the system includes a pre-processing phase, which is performed during the input of the ASP program. The pre-processing phase is aimed at detecting atoms which cannot be part of an answer set under any circumstances—thus, these atoms can be immediately set to false and removed from the program. This pre-processing requires computing a declarative closure of the program (i.e., a fixpoint using the positive part of each rule) and can require a relevant amount of time for programs with large sets of rules. This phase does not perform any choice and it is not amenable to parallelization using vertical parallelism. Thus, in benchmarks like Test2, it is not surprising that speedups are not particularly good, since only a fraction of the total computation takes advantage of vertical parallelism.

Benchmark	**No. of**	**No. of Threads**			
	Answer Sets	1	2	4	6
Test1	64	3.4412	2.1528	2.0099	2.1135
Test2	2	3.9257	2.7307	2.8506	3.0846
Test3	4092	39.118	20.411	13.175	11.835
Test4	81	3.3669	2.2516	2.2213	2.3616

Table 1
Execution Times (in sec.) and Speedups

6 Conclusions

In this paper we have reported some very preliminary considerations concerning the parallel evaluation of ASP programs. ASP is a novel programming paradigm

system	Test1	Test2	Test3	Test4
smodels	9.6631	11.5915	73.773	3.9865
our system	3.4412	3.9257	39.118	3.3669

Table 2
Comparison with smodels

which has become quite popular within the non-monotonic reasoning community and has good hopes of finding applications in a number of application areas (planning, intelligent agents, software specification and verification).

Applications of ASP have been severely limited by the time-consuming nature of ASP computations. In spite of the development of efficient sequential inference engines and optimization techniques, it is not unusual to encounter ASP executions capable of generating answer sets with thousands of elements and requiring hours for their complete development.

In this paper we have presented some preliminary results of a study aimed at exploring the use of parallelism to speedup the execution of ASP programs. We have abstracted the execution mechanisms lying at the base of one of the most efficient sequential ASP engines (the smodels system) as a constraint propagation procedure and we have proceeded to its parallelization. In particular, we have identified two sources of parallel work (horizontal and vertical non-determinism) from this algorithm. Our preliminary prototype makes use of a stack copying technique to extract vertical parallelism from the execution of ASP programs. The performance of the resulting system is sequentially very good—the simplification of the structure of basic algorithms lead to a faster engine than smodels—but is only beginning to produce speedups when run in parallel. These preliminary results are anyway very encouraging and demonstrate that parallelism can be adopted to improve performance of ASP engines.

Acknowledgments

The authors wish to thank M. Gelfond, G. Gupta, I. Niemela, and S. Tran for their help in this project. This work is partially supported by NSF grants CCR9875279, CCR9900320, CDA9729848, EIA9810732, and HRD9906130.

References

[1] S. Abiteboul and V. Vianu. Fixpoint Extensions of First-order Logic and Datalog-like Languages. In *Symposium on Logic in Computer Science*, pages 71–89. IEEE Computer Society, 1989.

[2] A. Aggoun and N. Beldiceanu. Overview of the CHIP Compiler System. In *Constraint Logic Programming: Selected Research*, pages 421–437. MIT Press, 1993.

[3] K.A.M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *1990 N. American Conf. on Logic Prog.*, pages 757–776. MIT Press, 1990.

[4] K. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1989.

[5] K.R. Apt and R.N. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19/20, 1994.

[6] C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.

[7] B. Baudot and Y. Deville. Analysis of Distributed Arc-Consistency Algorithms. Technical report, Université Catholique de Louvain, 1997.

[8] C. Bell, A. Nerode, R. Ng, and V.S. Subrahmanian. Implementing Stable Semantics by Linear Programming. In *Logic Programming and Non-monotonic Reasoning*, pages 23–42. MIT Press, 1993.

[9] W. Chen and D.S. Warren. Computation of Stable Models and its Integration with Logical Query Processing. *Transactions on Knowledge and Data Engineering*, 8(5):742–757, 1996.

[10] P. Cholewinski, V.W. Marek, and M. Truszczynski. Default Reasoning System DeReS. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 518–528. Morgan Kauffman, 1996.

[11] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*. Plenum, 1978.

[12] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding Planning Problems in Nonmonotonic Logic Programs. In *European Workshop on Planning*, 1997.

[13] W.F. Dowling and J.H. Gallier. Linear-time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming*, 3:267–289, 1984.

[14] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 406–417, 1998.

[15] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. In *International Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[16] G. Gupta and E. Pontelli. Optimization Schemas for Parallel Implementation of Nondeterministic Languages and Systems. In *International Parallel Processing Symposium*. IEEE Computer Society, 1997.

[17] G. Gupta and E. Pontelli. Efficient Techniques for Distributed Implementation of Search-based AI Systems. In *International Conference on Parallel Processing*. IEEE Computer Society, 1999.

[18] G. Gupta and E. Pontelli. Stack-splitting: A Simple Technique for Implementing Or-Parallelism and And-Parallelism on Distributed Machines. In *International Conference on Logic Programming*. MIT Press, 1999.

[19] G. Gupta, E. Pontelli, K.M. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. Technical report, New Mexico State University, 1999.

[20] B. Hausman, A. Ciepielewski, and S. Haridi. OR-Parallel Prolog Made Efficient on Shared Memory Multiprocessors. In *IEEE Symposium on Logic Programming*, 1987.

[21] ILOG. *ILOG Solver*, 4.0 edition, 1997.

[22] R. Krishnamurthy and S.A. Naqvi. Non-deterministic Choice in Datalog. In *International Conference on Data and Knowledge Bases*, pages 416–424. Morgan Kaufmann, 1988.

[23] V. Lifschitz. Answer Set Planning. In *Logic Programming and Non-monotonic Reasoning*, pages 373–374. Springer Verlag, 1999.

[24] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[25] V.W. Marek and M. Truszczynski. Stable Models and an Alternative Logic Programming Paradigm. In K.R. Apt, V.W. Marek, M. Truszczinski, and D.S. Warren, editors, *The Logic Programming Paradigm*. Springer Verlag, 1999.

[26] W. Marek, A. Nerode, and J.B. Remmel. The Stable Models of Predicate Logic Programs. In *Joint International Conference and Symposium on Logic Programming*. MIT Press, 1992.

[27] I. Niemela. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and AI*, (to appear).

[28] I. Niemela and P. Simons. Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. In *Logic Programming and Non-monotonic Reasoning*, pages 421–430. Springer Verlag, 1997.

[29] E. Pontelli and M. Gelfond. Research Directions in Efficient Execution of Answer Set Programs. Technical report, New Mexico State University, 2000.

[30] E. Pontelli, G. Gupta, D. Tang, M. Carro, and M. Hermenegildo. Efficient Implementation of Independent And Parallelism. *Computer Languages*, 22(2/3), 1996.

[31] E. Pontelli, D. Ranjan, and G. Gupta. On the Complexity of Parallel Implementation of Logic Programs. In *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 123–137. Springer Verlag, 1997.

[32] J.F. Puget. A C++ Implementation of CLP. Technical report, ILOG SA, 1995.

[33] D. Ranjan, E. Pontelli, and G. Gupta. On the Complexity of Or-Parallelism. *New Generation Computing*, 17(3):285–308, 1999.

[34] D. Saccà and C. Zaniolo. Stable Models and Non-determinism in Logic Programs with Negation. In *Symposium on Principles of Database Systems*, pages 205–217. ACM Press, 1990.

[35] A. Samal and T. Henderson. Parallel Consistent Labeling Algorithms. *International Journal of Parallel Programming*, 10(5):341–364, 19987.

[36] C. Schulte. Compairing Trailing and Copying for Constraint Programming. In *International Conference on Logic Programming*, pages 275–289. MIT Press, 1999.

[37] J.C. Shepherdson. Negation in Logic Programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1989.

[38] P. Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, 2000.

[39] T. Syrjanen. Implementation of Local Grounding for Logic Programs with Stable Model Semantics. Technical Report B-18, Helsinki University of Technology, 1998.

[40] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

GRANLOG: An Integrated Granularity Analysis Model for Parallel Logic Programming

Jorge Luis Victória Barbosa [a,b], Patrícia Kayser Vargas [b], Cláudio Fernando Resin Geyer [b] and Inês de Castro Dutra [c]

[a] *Universidade Católica de Pelotas, Escola de Informática*
Pelotas, RS, Brazil
barbosa@atlas.ucpel.tche.br

[b] *Universidade Federal do Rio Grande do Sul, Instituto de Informática*
Porto Alegre, RS, Brazil
{barbosa,kayser,geyer}@inf.ufrgs.br

[c] *Universidade Federal do Rio de Janeiro, COPPE/Sistemas*
Rio de Janeiro, RJ, Brazil, ines@cos.ufrj.br

Abstract

This work presents a new granularity analysis model for logic programming, the GRANLOG model. GRANLOG can generate granularity information for different sources of parallelism, and this information can be used by different parallel logic programming systems. GRANLOG was designed and implemented in modules that facilitate the inclusion and modification of any part of the system. GRANLOG employs several static analysis techniques based on other global analysis systems such as MEL and UDG and extends other tools used for granularity analysis such as CASLOG. We show that GRANLOG can effectively help logic programming systems to exploit parallelism in a more efficient way exhibiting some preliminary results obtained with the utilization of GRANLOG.

Key words: Logic programming, parallelism, granularity analysis

1 Introduction

The exploitation of implicit parallelism, that is relatively easier to detect in logic-based languages than in imperative languages, is considered an alternative that simplifies the parallel machines programming and which improves the logic programs performance. Logic programming allows the exploitation of two common forms of implicit parallelism: and-parallelism, where several goals in a conjunction can be selected to execute in parallel; and or-parallelism where several candidate clauses for a goal can be evaluated in parallel.

Several systems exploit only or-parallelism such as PloSys [8], Muse [14], Aurora [15] and YapOR [16]. Other systems exploit only and-parallelism such as &-Prolog [17], DASWAM [18], &-ACE [27] and Reform Prolog [24]. Yet other systems can exploit both kinds of parallelism such as Andorra-I [6,7], Penny [19] or ParAKL [20]. Such systems generate quite irregular parallel execution patterns that demand dynamic load balancing in order to maintain all processors busy most of the execution time. Most schedulers in these systems which distribute or-work, and-work or and-or work [25] make scheduling decisions based on runtime information and use different heuristics to estimate the amount of parallel work available in the execution tree. Parallel execution can be improved by a combination of runtime decisions and information generated at compile-time since the scheduler could have more precise information about amount of parallelism to help it to distribute work.

Most work developed in the area of compile-time analysis has concentrated on determinacy, mode and argument type analysis for generating optimized code mainly to sequential logic programming systems. In this work we develop a framework for compile-time granularity analysis in order to generate information for parallel logic programming systems that exploit and-parallelism, or-parallelism and and-or-parallelism. Compile-time granularity analysis consists of evaluating sizes of goals and clauses and amount of potential parallelism. Research has been done in compile-time granularity analysis for concurrent logic languages such as FGHC [21] and for parallel logic programming systems that exploit independent [22] and dependent [28] and-parallelism. Besides, Debray et al have developed the CASLOG system to perform complexity analysis of goals and clauses [4][5].

In this work we present a more complete framework for compile-time granularity analysis and its implementation, GRANLOG (GRanularity ANalyzer for LOGic programs).

GRANLOG performs automatic granularity analysis of Prolog programs and generates annotated programs. It uses several advanced static analysis techniques such as Global Analysis via Abstract Interpretation [23][1][2] and Complexity Analysis [3][4][5]. The information generated by the system can be used in applications such as scheduling decisions and program execution simulation. In order to demonstrate the usefulness of GRANLOG, scheduling strategies based on information generated by the system were implemented. These strategies were applied in two systems that exploit parallelism in logic programs: Andorra-I [6][7] and PloSys [8]. They were also used in the DSLP simulator [9] which is a prototype used to evaluate the DSLP distributed scheduler.

The remainder of the paper is organized as follows. In section 2, the basic principles of the system are presented. Section 3 shows an overview of GRANLOG focussing on its modular structure. Sections 4, 5 and 6 describe the modules that compose the system: the Global Analyzer, the Grain Analyzer and the Complexity Analyzer. In section 7, we present the improvements that GRANLOG can produce on Andorra-I, PloSys and DSLP. Finally, section 8 draws conclusions and discusses future work.

2 Design Decision

We designed and implemented the GRANLOG system having in mind the following principles:

Principle 1: *Independence of the parallel system that will use the information*. The granularity information, which is obtained through the analysis performed by GRANLOG does not depend on the parallel system (parallel architecture and runtime environment) on which the program will be executed. In this way, flexibility and portability are achieved.

Principle 2: *Automatic detection of parallelism (implicit parallelism)*. The system implements the automatic detection of parallelism, concerning logic programs. Therefore, end-programmers do not get involved with parallelism exploitation and the programming language keeps unchanged. This principle allows reuse of already developed programs and frees end-programmers from managing parallel development tasks.

Principle 3: *Optimal exploitation of parallelism in logic programs*. GRANLOG generates information on granularity analysis related to or-parallelism and and-parallelism. GRANLOG tries to obtain the maximum amount of parallelism information arising from the most common sources of parallelism found in logic programs to allow an optimum exploitation of parallelism.

3 GRANLOG Structure

GRANLOG is composed by three modules: the Global Analyzer (GLA), the Grain Analyzer (GRA) and the Complexity Analyzer (CA). Figure 1 presents the basic organization of the system. The input is a Prolog program and the output is a granulated program. The *granulated program* is composed by the original program plus *granularity annotation*. This annotation expresses the information produced by the granularity analysis process.

GLA employs global analysis to determine modes, types and argument sizes (named measures in the figure) for each predicate. Modes establish the direction of the argument bindings (*input*, *output*, *input/output*). Argument sizes are used during the complexity analysis to determine the influence of each argument in the complexity of the predicate. Types determine the binding pattern of the arguments (*list*, *integer*, *atom*, etc). Moreover, GLA analyzes accurately the dependencies between clause literals. Based on this analysis, the Grain Analyzer module establishes the grains of a clause body, i.e., which are the tasks that can be executed in parallel. The information inferred by GRA is annotated in the original program. The resulting program receives the name of *partitioned program*.

The Complexity Analyzer evaluates the cost involved in the execution of the grains (granularity) detected by the Grain Analyzer. In addition, it determines the complexity of the *pending resolvent* for each literal in a clause. The pending resolvent consists of all continuation literals that follow a clause literal [3]. It is used to get the complexity of a search tree branch during the execution of programs (*OR complexity* [3]). The CA module also determines the relationship between the input and output sizes of each grain (evaluation of communication costs). The information inferred by CA is annotated in the partitioned program. The final product is the granulated program. In the following sections we describe each module in more detail.

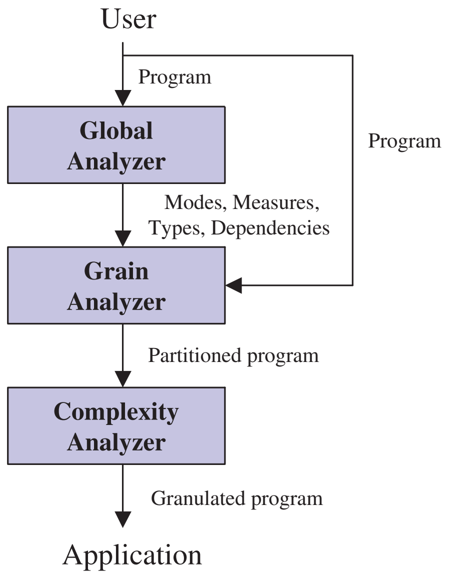


Fig. 1. GRANLOG structure

4 The Global Analyzer Module

The Global Analyser (GLA) is the first module in the GRANLOG system. It was designed in two parts that were implemented as two independent modules: *Party* (PARrallel TYpes analyzer [10]) and *MODA* (MOdes and Dependencies Analyzer [1]). ParTy is an abstract interpreter that uses *abstract compilation* [2] to determine the types of arguments in a Prolog program. On the other hand, MODA is an abstract interpreter of modes and dependencies. Party and MODA are independent systems that were integrated to create GLA [1]. Figure 2 presents the internal structure of the Global Analyzer. GLA receives a Prolog program increased with an *entry query* (or entry point) that describes basic characteristics of the predicate used to call the program. An entry query indicates the most used query pattern for a program with information that makes global analysis possible. Detailed description of the entry query will not be discussed in this paper due to space restrictions. After the analysis, the module produces information about the abstract execution of the program.

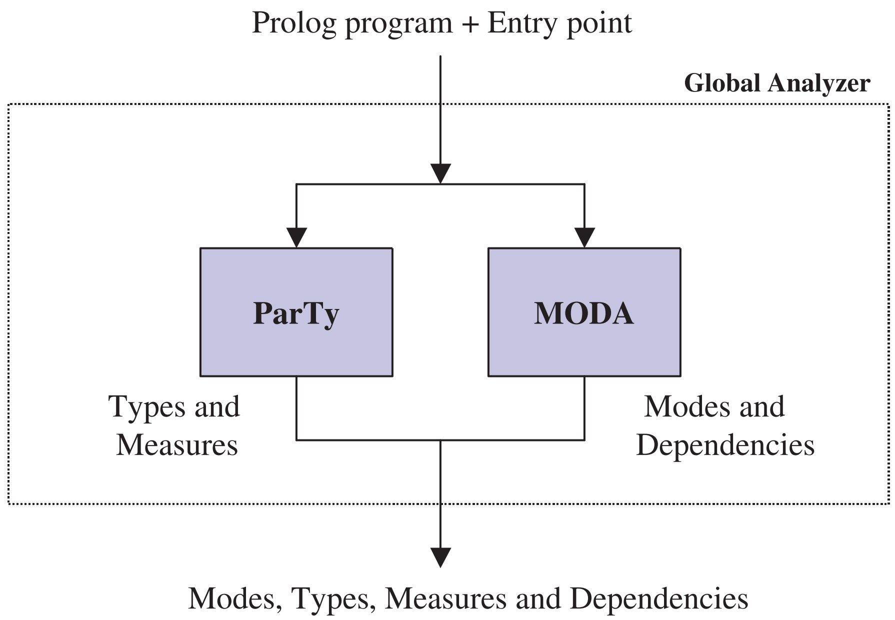


Fig. 2. Global Analyzer structure

Figure 3 shows the program *Hanoi* that will be used to explain the results of Global Analysis. This example program is composed by two predicates: `hanoi/5` and `append/3`. As a result of global analysis of this program, information related to each of the clauses of both predicates is generated. As an example, figure 4 presents the output pattern generated by MODA to the second clause of the predicate `hanoi/5`. In the figure, modes and dependencies are organized in two groups. The first group (first four lines) contains information about modes of each body literal (information is generated only for user predicates). Each line identifies the literal and presents a list with modes for all arguments (*i* indicates *input* and *o* indicates *output*).

The representation is `predicatename/arity/clausenumber/goalnumber`. The second group (last line) describes the dependencies among the clause literals. The information is structured as an annotation that identifies the clause and presents a list with pairs of literal dependencies (*I* and *O* are used to indicate the head).

This notation was designed without using any specific system as a model. The great contribution of this module is to use several important and consolidated concepts in a sole new output notation. For example, the notion of output and input modes in Prolog has been studied for a long time [2,4,5] and the dependency graph was used in early models as Conery's AND/OR Process Model [26].

Figure 5 presents the graph dependency created by the global analysis for this example.

```
hanoi(1,A,B,C,[mv(A,C)]).
hanoi(N,A,B,C,M) :-
   N > 1, N1 is N - 1,
   hanoi(N1,A,C,B,M1), hanoi(N1,B,A,C,M2),
   append(M1,[mv(A,C)],T), append(T,M2,M).

append([],L,L).
append([H|L],L1,[H|R]) :- append(L,L1,R).
```

Fig. 3. Program Hanoi

```
hanoi/5/2/3 [i,i,i,i,o]
hanoi/5/2/4 [i,i,i,i,o]
hanoi/5/2/5 [i,i,o]
hanoi/5/2/6 [i,i,o]
:-depend(hanoi/5/2,[(I,1),(I,2),(2,3),(2,4),(3,5),(4,6),(5,6),(6,O)]).
```

Fig. 4. Output generated by MODA

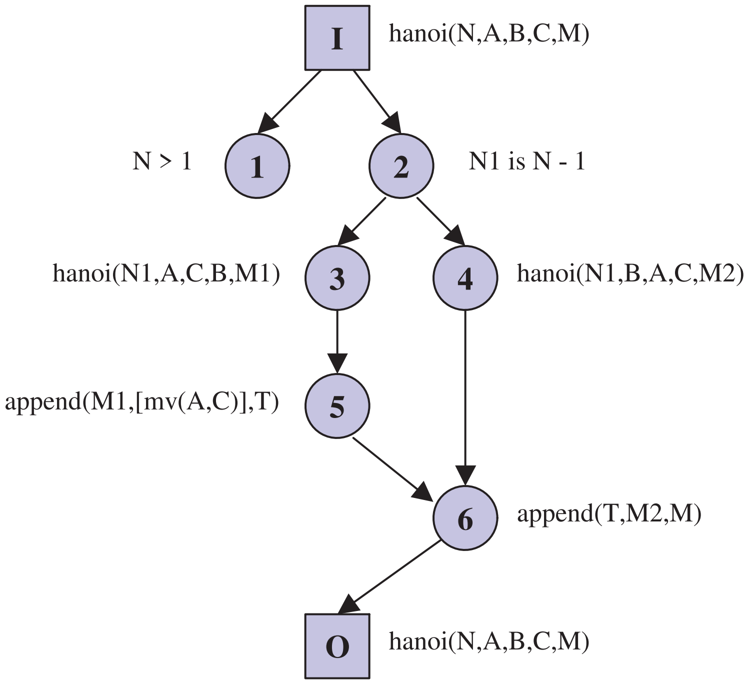


Fig. 5. Graph dependency generated by MODA

Following the example, figure 6 shows a possible output generated by ParTy for the predicate `hanoi/5`. The first line presents the type of each argument that was inferred considering one specific entry query. The first argument of the hanoi predicate was found to be of integer type. The second, third and fourth arguments are all atoms whose size is found to be three characters. The last argument is a list of unknown size (in the figure, represented by "?") but with known element type: structure whose name has two characters (mv in the program) with arity equals 2 and both sub-arguments of type *atom(3)*. When all sub-arguments have the same type, this is presented only once, otherwise, a complete list of types is included in the type annotation. The second line keeps the measures that must be used during the complexity analysis. The first argument must be measured like an integer (*int*) and the last must be measured considering the number of list elements (*length*). The first integer controls the hanoi recursion while the list size determines the output size. The other arguments are not used by the complexity analysis (*void*) since they do not influence the complexity of this program .

```
:- type(hanoi/5,[int,atom(3),atom(3),atom(3),list(?,[struct(2,2,
   [atom(3)])])]).
:- measure(hanoi/5,[int,void,void,void,length]).
```

Fig. 6. Output generated by Party

The combination of modes and types defines *communication channels*. Each argument defines a channel that can be used to predict the flow of information exchanged by processors during the parallel execution. Using this prediction, it is possible to calculate the size of messages (communication costs) in a distributed memory system. For example, figure 7 shows a possible configuration of channels during the parallel execution of hanoi. Modes define the directions and types determine the kind of information in the message. In this example, there are four input (mode *i*) and one output channels (mode *o*). Input messages have well defined types. Therefore, sizes and costs can be precisely evaluated. On the other hand, the information about the output message is incomplete, because the list size is unknown. The list size depends on the first argument which value will be sent in the first input channel during the execution. It is important to consider these channels as logical channels. None efficient implementation would use five different messages for sending and receiving arguments.

5 Grain Analyzer

The Grain Analyzer (GRA) module uses the dependencies information generated by the Global Analyzer to create the *partitioned program*. GRA inserts in the original program an annotation that expresses the potential parallelism. This annotation identifies the grains and describes their characteristics. In the GRANLOG context, a grain is defined as a task that can be executed in parallel with other tasks.

Figure 8 shows the same program example, hanoi, after annotated by the GRA module. The symbol & is used to delimit the grains in the body of each clause.

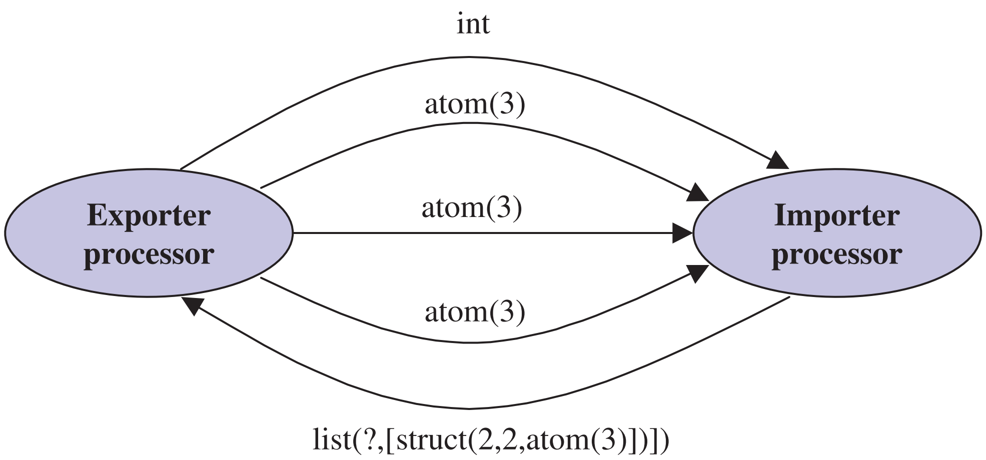


Fig. 7. Communication channels using modes and types

Before the original source program, GRA introduces *grain annotations* that describe the characteristics of each grain. There are three kinds of grain annotation: *grain_clause*, *grain_goal* and *grain_goals*. The *grain_clause* annotation describes the characteristics of a grain related to a clause. The *grain_goal* annotation describes the characteristics of a grain related to a body clause goal. Finally, the *grain_goals* annotation describes the characteristics of a grain related to the goals of a body clause.

```
:- grain_clause(hanoi/5,g1,[i,i,i,i,o],[int,void,void,void,length]).
:- grain_clause(hanoi/5,g2,[i,i,i,i,o],[int,void,void,void,length]).
:- grain_goals(hanoi/5,g2_3,[N1,A,B,C,T],[i,i,i,i,o],[int,void,void,
   void,length],[int,atom(3),atom(3),atom(3),list(?,[struct(2,2,
   [atom(3)])])]).
:- grain_goal(hanoi/5,g2_3_1,[i,i,i,i,o],[int,void,void,void,length],
   [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])]).
:- grain_goal(hanoi/5,g2_3_2,[i,i,o],[length,void,length],
   [list(?,[struct(2,2,[atom(3)])]),list(1,[struct(2,2,[atom(3)])]),
   list(?,[struct(2,2,[atom(3)])])]).
:- grain_goal(hanoi/5,g2_4,[i,i,i,i,o],[int,void,void,void,length],
   [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])]).
:- grain_goal(hanoi/5,g2_5,[i,i,o],[length],[list(?,[struct(2,2,
   [atom(3)])])]).
hanoi(1,A,B,C,[mv(A,C)]).
hanoi(N,A,B,C,M) :-
   N > 1, N1 is N - 1,
   (hanoi(N1,A,C,B,M1),append(M1,[mv(A,C)],T)) &
   hanoi(N1,B,A,C,M2), append(T,M2,M).
```

Fig. 8. Predicate hanoi after grain analysis

The parameters of *grain_clause* are the predicate name and arity, the grain identifier and the arguments modes and measures. The *grain_goal* annotation has an additional parameter, which contains arguments types. The *grain_goals* annotation has all parameters used by *grain_goal* plus one extra parameter that contains the argument names (input and output grains). The Grain Analyzer evaluates the grains associated to a set of goals in the body clause and identifies their inputs and outputs.

The Grain Analyzer indicates the potential parallelism available from clauses of a predicate (OR parallelism) and from each clause body (AND parallelism). OR parallelism is exploited using the information provided by the *grain_clause* annotation. AND parallelism is exploited using the symbol & and the annotations *grain_goal* and *grain_goals*. Further, GRA reorganizes goals in the body clause in order to exploit as much parallelism as possible. Figure 8 exemplifies this situation where the first call to the `append/3` goal in the second hanoi clause of the original source program was replaced with the second call to `hanoi/5`.

6 Complexity Analyzer

The Complexity Analyzer module evaluates the computational costs involved in the execution of the grains identified by the Grain Analyzer. CA is composed by two modules: *ORCA* (OR Complexity Analyzer [3]) and *CASLOG* (Complexity Analysis System for LOGic [4][5]). Both modules generate information about the complexity of logic programs.

ORCA produces simplified information used to determine the OR complexity. It is based on Tick's algorithm [11] with some modifications to increase precision as presented in [3]. CASLOG was developed by Lin and Debray and generates highly accurate and complex recurrent expressions that can be used to determine the complexity of each predicate in a program (AND complexity). The information generated by ORCA is called simplified because it does not consider actual recursion and gets all information at compile time. In contrast, CASLOG generates expressions which can be evaluated at run time and give really precise complexity information.

Figure 9 shows the internal structure of the Complexity Analyzer module.

The Complexity Analyzer includes *complexity annotations* in the partitioned program. Figure 10 presents the modifications performed in the partitioned hanoi program (figure 8). The *granularity* annotation contains the granularity expression. This expression can be used to calculate the computational cost of a grain considering the size of its inputs. The two parameters of *granularity* are an expression identifier (*e1*, for example) and the expression itself. The *out_size* annotation contains the relation between the sizes of the input and output grains. This relation can be used to determine the output size of a grain regarding its input size. In this way, it is possible to compute the cost for achieving the grain results in distributed systems (message size to get output). For example, in figure 7 this was the unique information that was not available at compile-time.

The two parameters of *out_size* consist of a relation identifier (*r1*, for example)

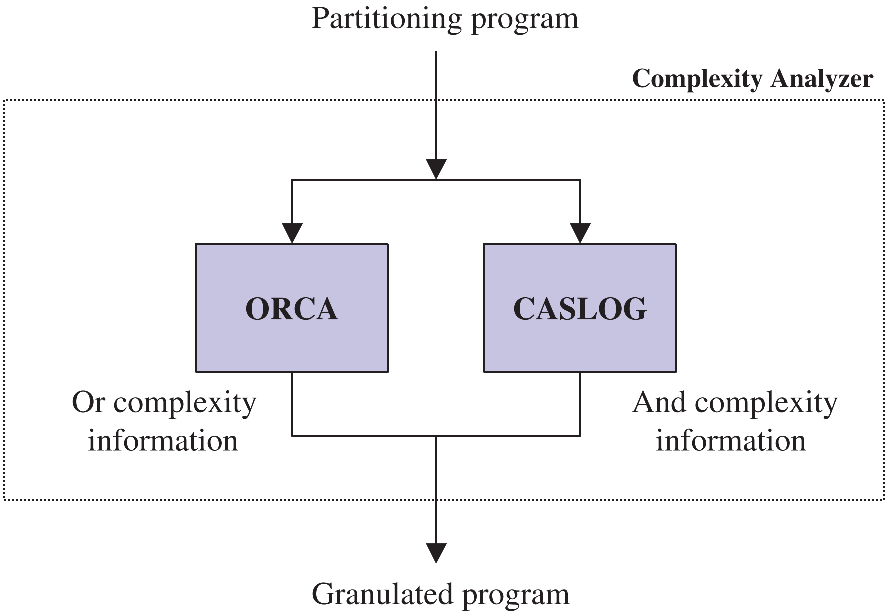


Fig. 9. Complexity Analyzer structure

and the relation itself. The notation used in the expressions is the same as those utilized in CASLOG [4][5]. In this notation, we use mathematical operators (*,/,+,-), mathematical functions (*fact*, *exp* and *log*, for example) and the *$* symbol. The latter indicates the size of a grain argument (*$1*, for example, represents the size of the first argument of the grain). Considering the *out_size* annotation, the relation between the argument sizes of a grain is described by the notation of the second parameter. For example, the notation *[$1,$2,$1+$2]* means that the grain has three arguments. The size of the third argument (output) is calculated using the first (*$1*) and the second (*$2*) argument sizes through the expression *$1+$2*.

The Complexity Analyzer introduces three new parameters in the grain annotations. The first one (for example, 39 in the second *grain_clause* annotation) contains a constant produced by ORCA. This constant is used to perform the OR complexity analysis. In the *grain_clause* annotation, this number represents the complexity of a clause. In the *grain_goal* and *grain_goals* annotations, the number represents the complexity of all next goals in the clause (pending resolvent). The second parameter (for example, *e1* in the second grain_clause annotation) consists of an identifier. It points to the expression that can be used to calculate the computational cost of the grain. This parameter binds the grain annotations with the granularity annotation. The third new parameter (last) consists of the identifier of the relation, which determines the grain output considering its inputs (*r1*, for example). This parameter binds the grain annotations with the *out_size* annotation. Some *granularity* and *out_size* annotations are used by more than one grain. This fact is very meaningful because they employ the same binding (for example, *g2*, *g2_3_1* and *g2_4* grains,

```
:- granularity(e1,$1*exp(2,$1)+exp(2,$1-1)-2).
:- granularity(e2,$1*exp(2,$1)+3*exp(2,$1-1)-2).
:- granularity(e3,$1+1).
:- out_size(r1,[0,0,0,0,1]).
:- out_size(r2,[$1,0,0,0,exp(2,$1)-1]).
:- out_size(r3,[$1,0,0,0,exp(2,$1)]).
:- out_size(r4,[$1,0,$1+1]).
:- out_size(r5,[$1,$2,$1+$2]).
:- out_size(r6,[0,$2,$2]).
:- grain_clause(hanoi/5,g1,[i,i,i,i,o],[int,void,void,void,length],
   1,1,r1).
:- grain_clause(hanoi/5,g2,[i,i,i,i,o],[int,void,void,void,length],
   39,e1,r2).
:- grain_goals(hanoi/5,g2_3,[N1,A,B,C,T],[i,i,i,i,o],[int,void,void,
   void,length],[int,atom(3),atom(3),atom(3),list(?,[struct(2,2,
   [atom(3)])])],19,e2,r3).
:- grain_goal(hanoi/5,g2_3_1,[i,i,i,i,o],[int,void,void,void,length],
   [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])],24,e1,r2).
:- grain_goal(hanoi/5,g2_3_2,[i,i,o],[length,void,length]),
   [list(?,[struct(2,2,[atom(3)])]),list(1,[struct(2,2,[atom(3)])]),
   list(?,[struct(2,2,[atom(3)])])],19,e3,r4).
:- grain_goal(hanoi/5,g2_4,[i,i,i,i,o],[int,void,void,void,length],
   [int,atom(3),atom(3),atom(3),list(?,[struct(2,2,[atom(3)])])],5,e1,r2).
:- grain_goal(hanoi/5,g2_5,[i,i,o],[length],
   [list(?,[struct(2,2,[atom(3)])])],0,e3,r5).
```

Fig. 10. Complexity annotations inserted by the Complexity Analyzer

which use the same granularity expression). Sometimes, the Complexity Analyzer can infer that a grain has a computational cost, which is constant. In this case, the granularity annotation will not be used. On the other hand, the constant will be annotated in the penultimate parameter of grain annotations (for example, *g1* grain).

7 GRANLOG Applications

GRANLOG was used to improve the exploitation of parallelism in three systems, namely: Andorra-I [6][7], PloSys [8] and DSLP [9].

Andorra-I is a parallel logic programming system that exploits both determinate AND parallelism and OR parallelism in shared-memory parallel computers. In Andorra-I, the compiler was adapted to transfer the granularity information from granulated program to abstract code (VRAM code [7]). The system uses the simplified information generated by ORCA module. Moreover, the engine was modified to access and update the runtime granularity information. This mainly consisted of adapting the loader and supporting the new VRAM instruction fields.

Regarding the scheduling strategies, modifications were performed to provide information to the reconfigurer only. The Granlog-Andorra integration presented stimulating preliminary results. We used 6 applications containing and-parallelism, or-parallelism and and-or parallelism and ran two versions of Andorra-I on a Sun SPARCstation-20 with 4 processors. Our results show that the GRANLOG information helped improving performance on all but one application. Our improvement ranged from 1% to 168%. These results were published in another paper [12].

PloSys explores OR parallelism using parallel machines with distributed memory. We needed to modify the abstract Plosys instructions to support granularity control. The instructions used to control the OR nodes now compute the complexity of choice-points using complexity of goals and pending resolutions. In addition, new algorithms were created to improve scheduling decisions using this information. In its original version, the scheduler strategy used the *number of choice-points* to guide its decisions. Now it uses dynamic computed complexity [13] and the implementation presented its first results that need to be improved.

DSLP (Distributed Scheduling for Logic Programming) is a distributed scheduler for exploiting OR parallelism and independent AND parallelism in logic programming. It was designed to work on shared memory and distributed memory systems. It uses two main informations to aid scheduling: system information (detected before and during the execution) and granularity information generated by GRANLOG.

The system was originally designed to support granularity control in both kinds of parallelism (AND/OR). Currently, DSLP has only a simulation version to control AND parallel execution. It uses the same information manipulated by Andorra-I. All scheduler decisions are guided by the *complexity of goals*. The first results are encouraging and show that the model can use the granularity information to increase the performance [9].

8 Conclusions

This work presented an integrated model for automatic granularity analysis of logic programs. Its main contributions are the integration of several concepts of granularity analysis in a single framework, the modeling, modularisation and implementation of these concepts, and the introduction of new annotation structures to encapsulate important information needed to improve the efficiency of scheduling strategies. GRANLOG integrates compile-time granularity analysis information with run-time information in order to provide more qualified data to support scheduling decisions.

We have been using GRANLOG information in systems such as Andorra-I [12], PloSys [13] and DSLP [9] with good results and minimum programming effort.

We plan to further investigate the impact of the GRANLOG information in these systems, and in the near future we intend to use GRANLOG to support scheduling decisions in other parallel logic programming systems such as YAPOR [16] and ACE [27]. One advantage of using GRANLOG is that we can improve modules separately depending on the systems needs.

We also intend to extend GRANLOG to perform better source code reorganization and program simulation.

Acknowledgments

We would like to thank Ana Paula Centeno, Cristiano André da Costa, Débora Nice Ferrari, Edvar B. Araújo, Luís Fernando Pias de Castro, Silvana C. de Azevedo and Thobias Salazar for the discussions and their helpful contributions. This research was partially supported by Protem-CC/CNPq during APPELO Project research grant and by PPGC/UFRGS and COPPE-Sistemas/UFRJ.

References

[1] S. C. Azevedo and J. L. V. Barbosa and C. F. R. Geyer, Automatização da Análise Global no modelo Granlog, *XXV Conferência Latino Americana de Informática (CLEI)* (Asuncion, Paraguay, August-September,1999) 601-612.

[2] S. Debray and R. Ramakrishnan, Abstract Interpretation of Logic Programs Using Magic Transformations, *Journal of Logic Programming* **18** (1994) 149-176.

[3] J. L. V. Barbosa and C. F. R Geyer, Análise de Complexidade na Programação em Lógica: Taxonomia, Modelo Granlog e Análise OU, *XXIII Conferência Latino Americana de Informática (CLEI)* (November 1997) 609-618.

[4] S. K. Debray and N. Lin, Cost Analysis of Logic Programs, *ACM Transactions on Programming Languages and Systems* **15** (1993) 826-875.

[5] N. Lin, *Automatic Complexity Analysis of Logic Programs* (PhD Thesis, University of Arizona, Tucson, 1993).

[6] R. Yang and T. Beaumont and I. C. Dutra and V. Santos Costa and D. H. D. Warren, Performance of the Compiler-Based Andorra-I System, *Proceedings of the Tenth International Conference on Logic Programming* (MIT Press, June 1993) 150-166.

[7] V. Santos Costa, *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I* (PhD Thesis, University of Bristol, 1993).

[8] E. Morel and J. C. Kergommeaux and C. F. R. Geyer, Side-effects in PloSys OR-parallel Prolog on Distributed Memory Machines, *Post-Conference Workshop on Parallelism and Implementation Technology for (Constrain) Logic Programming Languages (JICSLP'96)* (Bonn, Germany, September 1996).

[9] C. A. Costa and C. F. R. Geyer, Uma proposta de escalonamento distribuído para exploração de paralelismo na programação em lógica, *X Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho (SBA C-PAD)* (SBC, Búzios, Brazil, 1998) 61-64.

[10] L. F. Castro and C. F. R. Geyer, ParTy – A Parallel Types Analyzer, *Workshop on Parallelism and Implementation Technology for Logic Programming Languages* (October 1997).

[11] E.Tick, Compile Time Granularity Analysis for Parallel Logic Programming Systems, *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems* (ICOT, 1988).

[12] I. C. Dutra and V. Santos Costa and J. L. V. Barbosa and C. F. R. Geyer, Using Compile-Time Granularity Information to Support Dynamic Work Distribution in Parallel Logic Programming Systems, *XI Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho (SBAC-PAD)* (SBC, Natal, Brazil, 1999) 248-254.

[13] D. N. Ferrari and P. K. Vargas and C. F. R. Geyer and J. L. V. Barbosa, Modelo de Integração PloSys-Granlog: aplicação da análise de granulosidade na exploração do paralelismo OU, *XXV Conferência Latino Americana de Informática (CLEI)* (Asuncion, Paraguay, August-September 1999) 911-922.

[14] K. A. M. Ali and R. Karlsson, The Muse Or-parallel Prolog Model and its Performance, *Proceedings of the North American Conference on Logic Programming* (1990) 757-776.

[15] E. Lusk et al, The Aurora Or-parallel Prolog System, *New Generation Computing* **7** (1990) 243-271.

[16] R. Rocha and F. Silva and V. Santos Costa, *YapOr: an Or-Parallel Prolog System based on Environment Copying* (LIACC, Universidade do Porto, Technical Report DCC-97-14, December 1997).

[17] M. V. Hermenegildo and K. Greene, &-Prolog and its Performance: Exploiting Independent And-Parallelism, *Proceedings of the Seventh International Conference on Logic Programming* (June 1990) 253-268.

[18] K. Shen, Overview of DASWAM: Exploitation of Dependent And-parallelism, *The Journal of Logic Programming* **29** 1996.

[19] J. Montelius, *Penny - A Parallel Implementation of AKL* (Swedish Institute for Computer Science, May 1997).

[20] R. Moolenaar and B. Demoen, A Parallel Implementation for AKL, *Proceedings of the Programming Language Implementation and Logic Programming* (1993) 246-261.

[21] K. Ueda and M. Morita, A New Implementation Technique for Flat GHC, *Proceedings of the Seventh International Conference on Logic Programming* (June 1990) 3-17.

[22] P. L. Garcia and M. V. Hermenegildo and S. K. Debray, Towards Granularity Based Control of Parallelism in Logic Programming, *International Symposium on Parallel Computation* (September 1994).

[23] P. Cousot and R. Cousot, Abstract Interpretation and Application to Logic Programs *The Journal of Logic Programming* **13** (1992) 103-179.

[24] J. Bevemyr and T. Lindgren and H. Millroth, Reform Prolog: The Language and its implementation, *Proceedings of the Tenth International Conference on Logic Programming* (June 1993) 283-298.

[25] I. C. Dutra, *Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System* (University of Bristol, PhD Thesis, February 1995).

[26] J. S. Conery, *The AND/OR Process Model for Parallel Execution of Logic Programs* (University of California, Irvine, PhD Thesis, 1983).

[27] G. Gupta and E. Pontelli and M. V. Hermenegildo, &-ACE: A High Performance Parallel Prolog System, *Proceedings of the First International Symposium on Parallel Symbolic Computation (PASCO'94)* (1994).

[28] E. Pontelli and G. Gupta and F. Pulvirenti and A. Ferro, Automatic Compile-time Parallelization of Prolog Programs for Dependent And-Parallelism, *Proceedings of the Fourteenth International Conference in Logic Programming (ICLP)* (1997).

Towards the OAR Language and Computational Model

Salvador Abreu

Departmento de Informática
Universidade de Évora
PORTUGAL
`spa@di.uevora.pt`

Abstract

We present an initial specification for the language and computational model for OAR, an And-Or Tree Rewriting system implementation for a Logic Programming language along the lines of AKL and the Extended Andorra Model (EAM). OAR also supports namespace-structuring mechanism by implementing Contextual Logic Programming (CxLP). The design is intended to exploit implicit parallelism in a mixed shared- and distributed shared-memory (DSM) multiprocessor architecture.

1 Introduction

The work presented here presents a language and computational model inspired on AKL [7] and the EAM [11], amenable to being implemented on a SDSM (Software Distributed Shared Memory) platform such as TreadMarks [10] or DSM-Threads [13]. It can be argued that, in the long run, this kind of architecture is preferable to the use of shared-memory multiprocessors as these do not scale well beyond a certain point, cost considerations aside.

Moreover, and with today's increasing processor speed which is not matched by the memory systems, shared-memory multiprocessor systems will have increasing difficulty in dealing effectively with interdependent process threads which modify data in the same memory space.

By adopting a computational model which structures its writeable memory hierarchically, we aim to take advantage of the convenience provided by a very high-level language such as Prolog while being able to effectively use a resource for which it is difficult to extract good performance in the general case: a DSM system.

There have been proposals in the past for programming languages which rely on a computational model which defines a computation as a sequence of states called *configurations* which are essentially and-or trees, in which the transitions between the states are given by the application of one of a set of rewrite rules. One such system is AKL [7,15], for which at least three implementations were developed (a sequential one by Janson [9] and two parallel ones by Montelius [15] and Moolenaar [16].) Another such proposal, the EAM [20], initially proposed by David H. D. Warren has recently received effective work towards a full implementation [11] based on the YAP prolog engine.

OAR positions itself as a descendent of AKL implementations in that it purports to be a parallel and distributed logic programming system, relying on explicit AND-OR trees as the basis of its run-time model. Rather than taking the flexible "nuts-and-bolts at the programming language level" approach as provided by Oz [19], OAR targets a language which is essentially Prolog, extended with program structuring facilities geared towards more intensive code re-use and higher readability.

An issue which causes distress when attempting to use a Logic Programming language to implement large applications is the namespace structure: traditional Prolog provides no mechanism whatsoever to deal with the problem. More recently, the ISO Prolog Standard [1] settled on a *module system* which addresses the issue by grafting conventional imperative programming style namespace control directives onto the Prolog language, seeking compatibility with existing implementations such as Quintus or SICStus Prolog. It is our opinion that, while certainly useful and better than no modularity mechanism at all, this approach falls short of the expectations for a language with the declarativeness and flexibility of Prolog. There have been other proposals which deal with the issue of structuring Prolog programs in entities larger than clauses or predicates: Contextual Logic Programming (CxLP) [14] is one such proposal, which we chose to adopt in defining OAR. In this article we address the new issues which arise when combining the Contextual Logic Programming mechanisms with a computational model based on rewriting AND-OR trees.

The rest of this article is structured as follows: section 2 introduces the OAR language. In section 3 we introduce the notion of configuration in OAR while section 4 discusses the implications of having contexts in an And-Or Tree Rewriting System. In section 5 the high-level computational model is introduced. In sections 6 and 7 we discuss some architectural and implementation issues. We conclude with some remarks in section 8.

2 The Language

The language being targeted is Prolog augmented with (uniform) guard operators in clauses and contexts. Future versions of OAR may include features which extend Prolog without significantly affecting the computational model, in what concerns memory organization and parallel execution. An OAR program consists of:

- A *top-level goal*.
- A set of *unit definitions* which include *predicate definitions*.

Units are made up of a set of predicate definitions, themselves built from the usual set of clauses, which in OAR have the structure described in figure 1.

$$
\begin{aligned}
\mathsf{CLAUSE} &\rightarrow \mathsf{HEAD}\ \underline{\texttt{:-}}\ \mathsf{GUARD}\ \mathsf{BODY}\ \underline{\texttt{.}}\ \mid \mathsf{HEAD}\ \underline{\texttt{.}}\\
\mathsf{HEAD} &\rightarrow \mathsf{LIT}\\
\mathsf{GUARD} &\rightarrow \mathsf{GOALs}\ \mathsf{GUARDOP} \mid \varepsilon\\
\mathsf{BODY} &\rightarrow \mathsf{GOALs} \mid \varepsilon\\
\mathsf{GOALs} &\rightarrow \mathsf{GOALs}\ \underline{\texttt{,}}\ \mathsf{GOAL} \mid \mathsf{GOAL}\\
\mathsf{GOAL} &\rightarrow \mathsf{CONTEXT}\ \mathsf{LIT} \mid \mathsf{LIT}\\
\\
\mathsf{GUARDOP} &\rightarrow \underline{\texttt{!}} \mid \underline{\texttt{|}} \mid \underline{\texttt{:}}\\
\\
\mathsf{CONTEXT} &\rightarrow \mathsf{CONT}_{\mathsf{EXT}} \mid \mathsf{CONT}_{\mathsf{DOWN}} \mid \mathsf{CONT}_{\mathsf{SAVE}} \mid \mathsf{CONT}_{\mathsf{REST}} \mid \mathsf{CONT}_{\mathsf{SW}}\\
\mathsf{CONT}_{\mathsf{EXT}} &\rightarrow \mathsf{LIT}\ \underline{\texttt{>>}} \mid \mathsf{LIT}\ \underline{\texttt{>>}}\ \mathsf{CONT}_{\mathsf{EXT}}\\
\mathsf{CONT}_{\mathsf{SW}} &\rightarrow \mathsf{LIT}\ \underline{\texttt{><}}\\
\mathsf{CONT}_{\mathsf{DOWN}} &\rightarrow \underline{\texttt{\textbackslash/}}\\
\mathsf{CONT}_{\mathsf{SAVE}} &\rightarrow \underline{>}\\
\mathsf{CONT}_{\mathsf{REST}} &\rightarrow \underline{<}
\end{aligned}
$$

Fig. 1. Grammar for the OAR language

A few comments on the nonterminals occurring in the grammar of figure 1 are helpful in order to understand the meaning of an OAR program:

- GOAL is a literal (ie. a term), which may be prefixed with at most one *context operation*.
- $\mathsf{CONTEXT}$ is the optional context operation which may prefix a goal. It defines the calling context for the goal it applies to and takes on the forms:
  - · $\mathsf{CONT}_{\mathsf{EXT}}$ is the *context extension* operation which adds a qualified unit to the context.
  - · $\mathsf{CONT}_{\mathsf{DOWN}}$ is the *context contraction* operation which removes the topmost unit from the current context.
  - · $\mathsf{CONT}_{\mathsf{SAVE}}$ is the *context save* operation, in which the current context is assigned to the *auxiliary context* for the computation of the goal being considered. The current context remains unchanged.
  - · $\mathsf{CONT}_{\mathsf{REST}}$ is a *context revert* operation, which uses the current *auxiliary context* as the context for the computation of the goal being considered.
  - · $\mathsf{CONT}_{\mathsf{SW}}$ is the *context switch* operation, which swaps the specified topmost

unit from the current context

These are the exact same contextual goals as defined in [14], which provides an in-depth discussion of their use and semantics.

- A GUARDOP, or guard operator, may be one of $\underline{!}$ (cut), $\underline{|}$ (commit) or $\underline{:}$ (wait).
  - The first one (cut) intends to model the traditional Prolog behavior of pruning all subsequent alternatives for the predicate in which it occurs, obeying the textual order of the clauses that constitutes the predicate.
  - The second one (commit) is inspired in the commited-choice languages and prunes both forward (as the cut) and backward in the predicate's textual order. This guard operator endows the OAR language with *don't care nondeterminism* semantics, should it be used.
  - Finally, the "wait" guard operator behaves as in AKL and is used to allow for nondeterminate computation with OR-parallelism to be performed, by specifying a part of the clause (the guard) which is to be executed before any promotion may take place (see section 5, on rewrite rules).

Note that there may be at most one guard operator per clause. Whenever it is convenient to do so, the symbol $\underline{\%}$ may also be used to designate any (uniform) guard operator. Should there be no explicit guard operator in a clause, it behaves as if "$\underline{:}$ `true`" had been appended to the body.

The guard and body of clauses are made up of *goal literals* (or GOALs) in the grammar above, which can be further categorized as:

- Store operations: in a unification-based system these consist in the attempt to perform a test or a simple unification. This could easily be generalized to stand for constraint operations.
- Subgoal activations: in this case a set of *arguments* are created and a subgoal in invoked. These can be prefixed with a *context operation* which will define the context to be used for the subgoals in terms of the current context.

3 Configurations

The computational model for the OAR resembles the BEAM [11] in that it defines a structure similar to the EAM, i.e. the target language has the same restrictions on the legality of variable bindings. Contexts introduce another dimension to the problem as additional information must be attached to the representation of goals.

The state of the computation is, as in AKL [7] or in the BEAM [11], represented by a *configuration*. A configuration is essentially an and-or tree in which:

- The AND-nodes represent a conjunction of goals, such as the body of a clause. An AND-node is represented by an *AND-box* A which can be formally defined as follows:

$$\mathsf{A} = \mathsf{AND}^{\mathsf{K}}_{\mathsf{V}\sigma}(\mathsf{Os};\mathsf{Ca};\mathsf{Cs})$$

Where:

· V is a set of *variables*.

· σ is a set of *bindings* for V, ie. a collection of substitutions V_i/T_i which specify values for each variable $V_i \in \mathsf{V}$. In the present definition, we are only considering regular logical variables, even though we plan on allowing for a more general constraint setting in the future. The pair (V, σ) is called the *local store* for A.

· K is the AND-box's *background context*, which serves as reference for all goals which are to be evaluated within the AND-box.

· Os is a (possibly empty) set of OR-boxes (see below).

· Ca is a representation for the as-yet unexplored continuation for A, called the *AND-continuation*. This may simply be the address of the code that will continue the execution of A, in the context of some clause. Ca can be tought of as an ordered sequence of literals, possibly with one guard operator inserted. Ca can be represented as the address of a sequence of instructions that will execute within the AND-box.

· Cs is a set of *suspended continuations*, i.e. a set of pairs $(\mathsf{V}_i, \mathsf{C}_i)$ where V_i is an external variable [1] and C_i is a representation of the part of the computation that has been suspended because of an attempt to bind the external variable. In an actual implementation, this may simply be a code address. The C_i are assumed to be executing within the current AND-box and should have sufficient information to restore their Context.

AND-boxes have a variable number of child OR-boxes. OR-boxes get created whenever a subgoal enters a new clause and are removed when the goal fails or successfully terminates, when the box is promoted (see section 5 on rewrite rules).

An AND-box in which a failed store operation took place becomes *failed*, and takes on the syntax $\perp$.

- The OR-nodes stand for goal invocations (as seen from their parent AND-node) and are used to group various alternatives to the same call, as provided by the various clauses for the relevant predicate.

  We chose not to distinguish between two types of OR-boxes (ie. OR-boxes proper and CHOICE-boxes, as in the BEAM [11]) because we will assume that all clauses have a guard, even if simply that provided by a *wait* operator at the end of the body. This operator is intended to provide a mechanism for providing deep guards in a don't-know nondeterminism setting and thereby some measure of locally independent computation, amenable to parallelization. More formally, an OR-box O is defined as:

$$\mathsf{O} = \mathsf{OR}^{\mathsf{K}}(\mathsf{As}; \mathsf{Co})$$

[1] A variable V_i is said to be external for an AND-box A appearing in a configuration C if it belongs to one of the local stores in the path between C and A's parent.

Where:
· As is a set of AND-boxes, as per the previous definition.
· Co is a representation for the as-yet unexplored *OR-continuation* for O. Similarly to the previously defined AND-continuation, an OR-continuation may be represented simply by the address of the code that will generate the next unexplored alternative. An OR-continuation may be thought of as a sequence of instructions yet to try; some of these instructions will result in child AND-boxes being created.
· K is the OR-box's *background context*, which serves as reference for all goals which are to be evaluated within the OR-box. This is the context which resulted in the creation of the OR-box and will be inherited by descendent AND-boxes.

Similarly to what happens with AND-boxes, an OR-box with no alternatives (i.e. empty As and Co) is called a *failed OR-box* and is denoted $\perp$.

In both AND-boxes and OR-boxes there are several components named "continuations". These stand for as-yet unconstructed parts of the configuration, implicit in that they will only become an actual part of the configuration (ie. as OR-boxes or AND-boxes) when dealt with via the execution of appropriate abstract machine instructions.

4 Contexts

In CxLP, there is a *context* associated with each goal which is defined as a list of *unit terms* as per Monteiro and Porto's [14] approach. Executing a CxLP program in parallel and specifically with an And-Or Tree Rewriting System poses new specific problems:

- A goal's context must be preserved across promotions (see section 5.3).
- Before the actual goal code for a predicate starts being executed, an OR-box must have been created which must make provisions to represent the calling context, after any context modifiers which may apply.

These requirements point to an organization where both AND-boxes and OR-boxes have a *context* attribute. AND-continuations and suspended AND-continuations must also have a private context.

The context operations that may prefix a goal are considered part of the goal itself, and therefore inseparable from it. The rules which relocate continuations must carry the context along with the continuation.

5 Rewrite Rules

The rewrite rules are yet another variation on the usual scheme already found in other implementations of And-Or Tree Rewriting Systems such as AKL [7] or the BEAM [11].

The rules may be classified along two aspects: whether they're determinate or not and whether they perform forward or backward execution. Recall that "forward execution" comes as a result of exploring the code that makes up clauses while "backward execution" describes the transitions that take place as a result of factors other than code execution.

A few comments on the notation used to express rewrite rules in OAR:

- The letters A_i and O_i stand for an AND-box and an OR-box, respectively.
- We shall use α_i to denote a (possibly empty) sequence of AND-boxes and ω_i for a possibly empty sequence of OR-boxes.
- The symbol . denotes the list constructor and is used to specify the structure of sequences or sets. It can be used to single out an element from a sequence (eg. $\mathsf{A}.\alpha$ for a sequence of AND-boxes or $\mathsf{G_L.Ca}$ for a sequence of goals that make up an AND-continuation).
- The symbol $+$ is used to indicate a list append operation.
- The symbol ε is used to denote the empty sequence and applies to AND-boxes, OR-boxes and continuations.
- The symbol $\perp$ represents the FAIL box and applies to both kinds of boxes.
- Transitions are expressed as follows:

$$\left.\frac{\text{(present configuration)}}{\text{(next configuration)}}\right|_{\text{(optional conditions)}}$$

Should there be a precondition which must be satisfied for a rule to apply, it will be indicated on the right of the rule. Rules with no preconditions apply whenever the present configuration matches the rule.

Note also that access to non-local stores is read-only: should a worker try to bind a non-local variable (ie. one which resides in an and-box different from the one it's presently executing in) it will block.

5.1 Determinate AND-Rules

This set of rules focuses on the manipulation of AND-boxes.

AND-try This rule augments an AND-box by progressing in the evaluation of its AND-continuation: it removes an item from the AND-continuation and acts upon it, creating a child OR-box with an empty set of descendent AND-boxes and an appropriate value for its OR-continuation.

$$\frac{\mathsf{AND}^{\mathsf{K}}_{\mathsf{V}\sigma}(\mathsf{Os};\mathsf{G_L}.\mathsf{Ca};\mathsf{Cs})}{\mathsf{AND}^{\mathsf{K}}_{\mathsf{V}\sigma}(\mathsf{Os}'=\mathsf{OR}^{\mathsf{L}}(\varepsilon;\mathsf{Co}).\mathsf{Os};\mathsf{Ca};\mathsf{Cs})} \tag{1}$$

Where $\mathsf{G_L}$ is a goal with n matching clauses in which the ith clause has the structure H_i: $-\mathsf{G}_i\%\mathsf{B}_i$. Note that the newly created OR-box, the head of Os', starts with an empty set of child AND-boxes and Co is a reference to the first instruction in predicate H's indexing code. L is the new context derived from K and the context operations in G, which is passed to the new OR-box as its background context.

Binding (successful binding) The binding rule (or *constraint imposition rule*) sets the value for a previously unbound variable. It occurrs in a situation similar to that of the AND-try rule, being different from it in that the subgoal to be tried is a simple store operation.

$$\left.\frac{\mathsf{AND}^{\mathsf{K}}_{\mathsf{V}\sigma}(\mathsf{Os};\mathsf{G_{op}}.\mathsf{Ca};\mathsf{Cs})}{\mathsf{AND}^{\mathsf{K}}_{\mathsf{V}\sigma\theta}(\mathsf{Os};\mathsf{Ca};\mathsf{Cs})}\right|_{\mathsf{G_{op}}\equiv\theta} \tag{2}$$

Where $\mathsf{G_{op}}$ binds a set of variables $\overline{\mathsf{X}} \subset \mathsf{V}$ via a substitution θ. This transition is expected to represent the bulk of what is performed in a clause's execution.

Suspension (external binding) The suspension rule can be applied under circumstances similar to those in which the binding rule applies: it would set the value for a previously unbound variable. It differs from the *successful binding* rule in that the variable which is being bound is external: this rule causes the operation to suspend.

$$\left.\frac{\mathsf{AND}^{\mathsf{K}}_{\mathsf{V}\sigma}(\mathsf{Os};\mathsf{G_{op}}.\mathsf{Ca};\mathsf{Cs})}{\mathsf{AND}^{\mathsf{K}}_{\mathsf{V}\sigma}(\mathsf{Os};\mathsf{Ca};\mathsf{Cs}\cup\{(\overline{\mathsf{X}},\theta)\})}\right|_{\mathsf{G_{op}}\equiv\theta\wedge\overline{\mathsf{X}}\not\subset\mathsf{V}} \tag{3}$$

Where $\mathsf{G_{op}}$ attempts to bind a set of variables $\overline{\mathsf{X}} \not\subset \mathsf{V}$ via a substitution θ. θ then becomes suspended on $\overline{\mathsf{X}}$ and the pair $(\overline{\mathsf{X}},\theta)$ is added to the set Cs.

AND-collapse (failed binding) Analogous to the binding rule, there is a transition which is applicable when the attempted store operation fails:

$$\left.\frac{\mathsf{AND}^{\mathsf{K}}_{\mathsf{V}\sigma}(\mathsf{Os};\mathsf{G_{op}}.\mathsf{Ca};\mathsf{Cs})}{\perp}\right|_{\mathsf{V}\sigma\not\vdash\mathsf{G_{op}}} \tag{4}$$

$\perp$ represents a *failed AND-box*. This transition is taken whenever the attempted operation $\mathsf{G_{op}}$ fails in the store provided by V and σ.

AND-collapse (no choice) Another way in which an AND-box may fail is when it contains a failed OR-box:

$$\frac{\mathsf{AND}^{\mathsf{K}}_{\mathsf{V}\sigma}(\perp;\mathsf{Ca};\mathsf{Cs})}{\perp} \tag{5}$$

$\perp$ represents a *failed OR-box*. This transition is taken independently of there being any continuations or suspensions for the AND-box, which are dropped together with the AND-box itself.

5.2 Determinate OR-Rules

This set of rules deals with OR-boxes.

OR-try Similarly to AND-try, this rule augments an OR-box by progressing in the evaluation of its OR-continuation.

$$\frac{\mathsf{OR}^{\mathsf{K}}(\mathsf{As};\mathsf{G_o}.\mathsf{Co})}{\mathsf{OR}^{\mathsf{K}}(\mathsf{As}\cup\{\mathsf{A}\};\mathsf{Co})} \tag{6}$$

Where $\mathsf{G_o}$ is the first element of the OR-box's OR-continuation and A is the AND-box created by $\mathsf{G_o}$. Co represents the remainder of the OR-continuation. Initially we'll have $\mathsf{A} = \mathsf{AND}^{\mathsf{K}}_{\emptyset}(\varepsilon;\mathsf{Ca};\varepsilon)$, i.e. the OR-box's context is passed on to the new AND-box, which starts its existence with no variables or bindings, no OR-box children and no suspended continuations.

OR-reduction Whenever one of the alternatives in an OR-box fails, it simply gets removed from the OR-box's list of immediate successor nodes, i.e. a list of AND-boxes.

$$\frac{\mathsf{OR}^{\mathsf{K}}(\omega_1\bot\omega_2;\mathsf{Co})}{\mathsf{OR}^{\mathsf{K}}(\omega_1\omega_2;\mathsf{Co})} \tag{7}$$

OR-collapse (no alternatives) This rule applies when a choice-box becomes devoid of alternatives, possibly through successive applications if the OR-reduction rule.

$$\frac{\mathsf{OR}^{\mathsf{K}}(\varepsilon;\varepsilon)}{\bot} \tag{8}$$

The OR-box is removed from the configuration, being replaced by a FAIL node.

5.3 Promotion and Split Rules

These rules deal with the determinate promotion and all non-determinate operations. These transformations do not occur as a direct result of expanding any part of a predicate's definition: they stem from the inability to apply any of the other rules and their applicability is determined only by the current configuration, regardless of what is in the continuations.

Promotion This rule applies when an OR-box has a single successor node, which is then moved into the OR-box's parent node. More formally:

$$\frac{\mathsf{AND}^{\mathsf{K}}_{\mathsf{V}\sigma}(\mathsf{OR}(\mathsf{AND}^{\mathsf{K}'}_{\mathsf{V}'\sigma'}(\mathsf{Os}';\mathsf{Ca}';\mathsf{Cs}'));\mathsf{Ca};\mathsf{Cs})}{\mathsf{AND}^{\mathsf{K}}_{(\mathsf{V}\cup\mathsf{V}')\sigma\sigma'}(\mathsf{Os}';\mathsf{Ca}.\mathsf{Ca}'^{K'};\mathsf{Cs}\cup\mathsf{Cs}'^{K'})} \tag{9}$$

The promotion rule is contracting and it ensures that OAR configurations remain shallow. The context from the AND-box which is being promoted is transfered onto the corresponding AND-continuations and suspensions.

Split This rule is the basis for non-determinism in OAR and can be thought of as a rule for distribution of ANDs over ORs:

$$\frac{\mathsf{OR}^{\mathsf{K}}(\alpha_0\mathsf{AND}^{\mathsf{K}'}_{\mathsf{V}\sigma}(\omega_1\mathsf{OR}^{\mathsf{K}''}(\alpha_1\alpha_2)\omega_2)\alpha_3)}{\mathsf{OR}^{\mathsf{K}}(\alpha_0\mathsf{AND}^{\mathsf{K}'}_{\mathsf{V}\sigma}(\omega_1\mathsf{OR}^{\mathsf{K}''}(\alpha_1)\omega_2)\alpha_3\mathsf{AND}^{\mathsf{K}'}_{\mathsf{V}\sigma}(\omega_1\mathsf{OR}^{\mathsf{K}''}(\alpha_2)\omega_2)\alpha_3)} \tag{10}$$

This is the "classical" choice split as found in AKL and the BEAM, in which an OR-box has one of its successor AND-boxes singled out, thereby making it a candidate for promotion by rule 9. The present formulation for this rule allows for different strategies to be selected, depending on the lengths of α_1 and α_2: should one of these be of length 1 it will be selected for promotion. In order to preserve Prolog-like solution order, $|\alpha_1| \leq |\alpha_2|$ must hold.

The AND-box containing α_2 as is a copy of the original AND-box containing $\alpha_1\alpha_2$. The contexts are also copied to wherever it is appropriate.

Multi-split This is a variation on the *Split* rule, where instead of splitting the alternatives in two parts and copying the environment to the newly created AND-boxes, *several* such actions are performed concurrently, distributing the entire lower OR-box onto its ancestor. The rationale for doing this is that copying operations are expensive to perform and, considering the benefits of SWMR (single writer multiple reader) access disciplines to DSM memory areas, it may prove beneficial to chronologically concentrate all the copying work.

The split and multi-split rules are only applicable when none of the other ones are. The reason for this being that nondeterminate rules tend to be a lot more expensive than determinate ones.

6 Target Multiprocessor Architecture

The purpose of OAR being to efficiently and transparently execute on a mixed shared- and distributed-memory multiprocessor, it seems only fair to incorporate knowledge of the multiprocessor topology when allocating work among the various processors. When discussing distributed memory architectures, we are doing so from the SDSM (Software Distributed Shared Memory) point of view, as in TreadMarks [10] or DSM-threads [13], ie. the design for OAR does not use message-passing as its *visible* mechanism for sharing information, rather, we will assume that a device similar to shared-memory is available, albeit one with lower performance than the "real thing".

The memory organization used for OAR is the object of a separate effort, being developed [12] in cooperation with the people working on BEAM [11], and assumes a relaxed consistency protocol for the distributed shared memory such that found in the VIA-based HLRC implementation of Rangarajan and Iftode [18] which we also consider using.

W.R.T the multiprocessor model, we chose a most general model where processor pair connectivity is classified as being one of the following classes:

(i) **Physically Shared-memory.** A pair of processors having this connectivity will share the same physical memory, allowing for efficient (altough expensive) sharing of resources and fast synchronization. This represents a classical SMP organization.

(ii) **SDSM with low latency and high throughput.** Processors with this kind of connectivity are assumed to rely on a dedicated medium to high-speed communications channel, such as a back-to-back pair of fast Ethernet NICs using a reversing cable. Pairs of machines in this situation correspond to the concept of a "System Area Network" (SAN).

(iii) **SDSM with higher latency and lower throughput.** Processor pairs classified this way are all of those which do not fall any of the previous classes.

The distinction between classes 2 and 3 is not necessarily quantitatively determined: it can be given by knowledge of the network topology or be derived from dynamic measurements of the performance characteristics related to a pair of processors.

The actual hardware used to prototype OAR implements all three classes, as the system being built is initially made up of 2 clusters of 4 nodes each, each node being a dual processor SMP system with back-to-back fast Ethernet links to all other nodes in the same cluster, together with a single "external" Ethernet link, which is used to connect to the other cluster(s) and to the disk server and OAR supervisor node (the "server") via an ethernet switch. This architecture is depicted in figure 2 and we think it is representative of what can currently be implemented on a relatively tight budget.

7 Prototype Implementation

The prototype we have started to work on is based upon GNU Prolog [5]; this implementation was chosen because it is relatively easy to work on, and also because of of the textual nature of the various intermediate representations of a program, which provides a clear separation of the various steps in compiling a program and facilitates debugging.

From the multiprocessing standpoint, we started using TreadMarks [10] for the SDSM sharing of information and POSIX Threads as provided by GNU libc for the physically shared memory workers.

The hardware/software testbed platform is an array of Linux (kernel 2.3.99) diskless dual-processor machines (Intel Celeron 466's on Abit BP6 motherboards), connected with back-to-back fast ethernet interfaces and an ethernet switch.

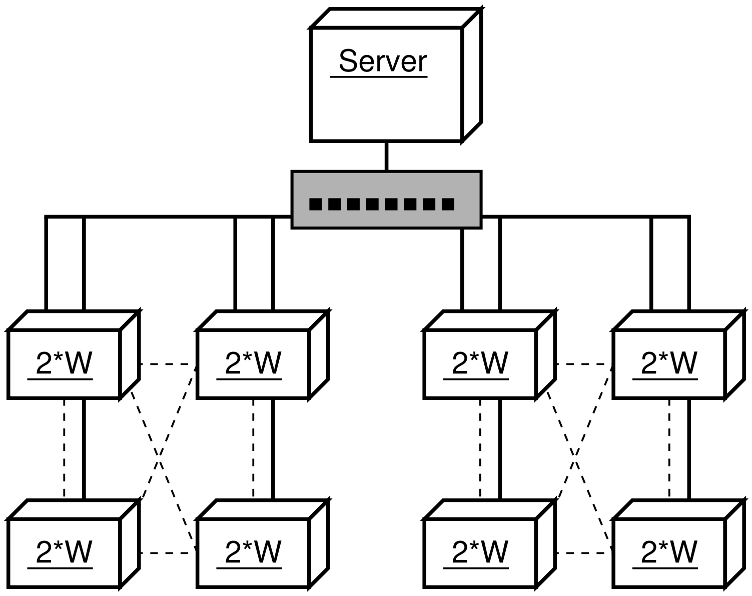


Fig. 2. OAR Prototype Hardware Architecture

8 Conclusions and Future Work

To answer the question "why yet another language?", we point out that what we are really doing here is to recover an existing language proposal, CxLP or Contextual Logic Programming, casting it in a parallel and distributed execution model. Our experience with actual programming of non-trivial systems using CxLP in the ALPES programming environment [4,2] indicates that this language very much compensates for the inadequateness of Prolog as an implementation language for large software engineering projects, without requiring excessive retraining on the part of the Prolog programmer.

As for the viability of the computational model proposed herein, it's still a bit early to present any conclusions as to the performance of the OAR system, as we are still in the first stages of its implementation, which is scheduled to have a working prototype in mid-2001.

Acknowledgements

The author would like to thank Vítor Santos-Costa, Fernando Silva and Ricardo Lopes for fruitful discussions on the architecture and memory organization of OAR. Discussions were also held with António Porto and Artur Dias regarding the integration of CxLP into the OAR model.

The FCT (Portuguese Government Agency) is acknowledged for supporting this work under research contract PRAXIS/P/EEI/10191/1998.

References

[1] Information technology – Programming languages – Prolog – Part 1: General Core. ISO/IEC 13211-1, 1995.

[2] Salvador Pinto Abreu. Prolog for User Interfaces and User Interfaces for Prolog. In António Porto, editor, *ICLP'91 Tutorial on Logic Programming Environments*, 1991.

[3] Maurice Bruynooghe and Jaan Penjam, editors. *Programming Language Implementation and Logic Programming, Proceedings*, volume 714 of *Lecture Notes in Computer Science*. Springer Verlag, 1993. ISBN 3-540-57186-8.

[4] Alpes Consortium. *ALPES Final Report*. Universidade Nova de Lisboa, 1989.

[5] Daniel Diaz and Philippe Codognet. GNU Prolog: Beyond Compiling to C. In Pontelli and Santos-Costa [17].

[6] Lars-Henrik Eriksson, Lars Hallnäs, and Peter Schroeder-Heister, editors. *Extensions of Logic Programming, Second International Workshop*, volume 596 of *Lecture Notes in Computer Science*. Springer Verlag, January 1991. ISBN 3-540-55498-X.

[7] Torkel Franzén, Seif Haridi, and Sverker Janson. An Overview of the Andorra Kernel Language. In Eriksson et al. [6], pages 163–179. ISBN 3-540-55498-X.

[8] Pascal Van Hentenryck, editor. *Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming*. MIT Press, June 1994. ISBN 0-262-72022-1.

[9] Sverker Janson. Implementation of andorra-based languages. In Hentenryck [8]. ISBN 0-262-72022-1.

[10] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX Winter 1994 Technical Conference*, pages 115–132.

[11] Ricardo Lopes and Vítor Santos Costa. The BEAM: Towards a first EAM Implementation. In *Proceedings of the Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, 1997.

[12] Ricardo Lopes, Fernando Silva, Vítor Santos Costa, and Salvador Abreu. The rainbow: Towards a parallel beam. In *Workshop on Parallism And Implementation Technology for (Constraint) Logic Languages*, July 2000. to appear.

[13] F. Müller. On the Design and Implementation of DSM-Threads. In *International Conference on Parallel and Distributed Processing*, pages 315–324, June 1997.

[14] Luís Monteiro and António Porto. A Language for Contextual Logic Programming. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic Programming Languages: Constraints, Functions and Objects*, pages 115–147. MIT Press, 1993.

[15] Johan Montelius. Penny, a parallel implementation of akl. In E. Pontelli and G. Gupta, editors, *Workshop on Design and Implementation of Parallel Logic Programming Systems*, pages 4–8, November 1994. ILPS'94 Post-Conference Workshop.

[16] Remco Moolenaar and Bart Demoen. A Parallel Implementation for AKL. In *Programming Language Implementation and Logic Programming, Proceedings*, pages 246–261, 1993.

[17] Enrico Pontelli and Vitor Santos-Costa, editors. *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000)*, number 1753 in Lecture Notes in Computer Science, Boston, MA, USA, January 2000. Springer-Verlag.

[18] M. Rangarajan and L. Iftode. Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance. Technical Report DCS-TR-413, Rutgers University, Department of Computer Science, April 2000. To appear in Proceedings of The Third Extreme Linux Workshop, October 10-12, Atlanta.

[19] Peter Van Roy and Seif Haridi. Mozart: A programming system for agent applications. In *International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*, November 1999. Part of International Conference on Logic Programming (ICLP 99).

[20] David H. D. Warren. The Extended Andorra Model with Implicit Control. ICLP90 Preconference Workshop, June 1990.

Compiling large disjunctions

Henk Vandecasteele, Bart Demoen, Gerda Janssens

Department of Computer Science Katholieke Universiteit Leuven B-3001 Leuven, Belgium
{henkv,bmd,gerda}@cs.kuleuven.ac.be

Abstract

In the context of data mining and inductive logic programming (ILP), large sets of queries must be evaluated against a set of examples. One particularly effective optimisation is to organise these sets of queries as a query pack, yielding speedup factors of up to 20. This was achieved through compiling a query pack for an enhanced WAM. The weak point in this story is that compilation of huge packs - they are represented as a nested disjunction with several thousands of goals and a nesting depth of more than 10 - with the usual techniques is too inefficient to give such learning systems also a high overall speedup when using the pack technology. Therefore, the compilation of large disjunctions with many variables had to be improved. In this paper, we investigate the problems current Prolog compilers have when dealing with such huge disjunctions. We propose some remedies and discuss the trade-offs. In particular the paper presents a new classification algorithm for variables – which is almost linear in the case of ILP – and a numbering scheme for variables that reduces the total number of variables. The paper also discusses an issue in the generated WAM code related to precise garbage collection. We have implemented a prototype compiler and present some preliminary figures. The paper also considers the possibility of incrementally compiling growing disjunctions/packs which is particularly interesting in the context of inductive learning.

1 Introduction

When inductive logic programming (ILP) is used for data mining purposes, one generates lots of hypotheses and checks whether they are valid with respect to a database (a set of examples) in order to discover the "best" hypothesis. In the case of the ILP systems XTILDE and WARMR developed by the K.U.Leuven group, a hypothesis is actually a Prolog query and it is executed with respect to a Prolog program, describing general knowledge about the problem domain and also containing the actual examples. Moreover, the ILP queries have some specific properties. First, a query is just a conjunction of atoms and different queries can have the same atoms at the start of the conjunction, i.e. they can have a common prefix. Next, during the data mining process valid queries are refined, i.e. some atoms are added at the end of the query. In both cases, it would be useful that common parts are only executed once. Current Prolog systems have no direct support for this, while tabling is no real solution.

ILPROLOG is a Prolog system that was recently developed purposefully to provide special support for optimisations and for requirements to ILP systems, in particular the validation of hypotheses. ILPROLOG is WAM-based and currently uses the XSB [14] compiler for the generation of the abstract machine code and a reader implemented by Koen De Bosschere [5]. The kernel is written in C using state of the art technology: its roots are in [8]. In order to improve the evaluation of a set of queries, we group them together in a so-called *query pack*, which is similar to a Prolog disjunction but its execution is tuned towards ILP as mentioned in Section 2. In [7] an extension to WAM supporting the execution of query packs is explained in detail. [3] reports how ILPROLOG has been used by the ILP systems XTILDE [2] and WARMR [6] and how they benefit from the query pack support. Experimental results show that the evaluation of a set of queries speeds up by a factor of 20 when query packs are used. However, when query packs are used, they must be generated and compiled and this reduces the efficiency gain. During experiments for [3] we observed that it sometimes takes more time to compile the query packs than to execute them. In some cases this meant a speed loss instead of a speedup. While, with the new compiler described here, we have a net efficiency gain of a factor 4, which is considerable.

Having identified the compilation of query packs as a bottleneck in our system, we reconsidered the compilation of (large) disjunctions as this covers also the issues for query packs. The relation between query packs and disjunctions is explained in more detail in Section 2. Section 3.1 discusses the general issues of compiling a disjunction, while section 3.2 describes the specific problems arising when compiling large disjunctions. Section 4 describes the new compiler which solves the problems. Section 5 revisits reachability, information needed during garbage collection, in the context of packs because our implementation does not respect the order of branches. Finally, Section 6 discusses the incremental compilation of packs.

2 Query Packs and Disjunctions

As mentioned above ILP systems usually generate a large amount of queries. These systems want to know for each of these queries whether it fails or succeeds. Almost always these queries have common prefixes. Executing these queries as query packs [3] is a technique that combines these queries, such that common prefixes are executed only once. Syntactically, query packs are similar to Prolog disjunctions. When grouping two queries with a common prefix, the query pack consists of the common prefix followed by a disjunction with two branches, where each branch corresponds to the part where the queries differ. Generalisation for grouping more queries is straightforward when disjunctions are also nested. In the following example the queries `q1`, `q2`, `q3` and `q4` are grouped as `qd`.

```
q1 :- a,b,c,d.

q2 :- a,b,e,f.

q3 :- a,b,e,g,h.

q4 :- a,b,i.


qd :- a,b, ( c,d ; e, ( f ; g, h) ; i).
```

Using disjunctions avoids recomputation of common prefixes such as `a,b` and `a,b,e` in the previous example. In the context of ILP we can go further: for each original query, we only want to know whether it succeeds or not [1]. Due to backtracking, normal Prolog execution detects all successes. Using cuts (or some of its special variants) does not help as it could prevent backtracking into branches that have not yet succeeded. Therefore, a dedicated execution for such disjunctions, then called *query packs*, is needed. The basic idea is that as soon as a branch succeeds, it will no longer be considered on backtracking. Suppose that in the above example the `c,d` branch (corresponding to the query `a,b,c,d`) succeeds, but the calls to `e` and `i` fail and that we backtrack to `b`. Now, it is as if the `c,d` branch is not there any longer and only the other alternatives have to be tried. The paper [7] describes possible implementations and shows that an extension of the WAM leads to better results than a meta-interpreter or a source-to-source transformation. This has been confirmed by the experiments for [3] where also the need of a good query pack compiler was observed. Note that the WAM implementation possibly changes the order in which the remaining branches are considered. For the example above, it might be the case that after backtracking to `b` the branch with `i` is considered before the originally second one.

In order to get a better idea of for example typical XTILDE query packs, we report the following properties for a set of query packs taken from the experiment on the Mutagenesis data set [12,3]:

- *Nbq* is the number of queries in a pack.
- *Nbv* is the number of variables in a pack (as computed by numbervars).
- *MaxL(F)* with MaxL the maximum length of the queries (i.e. the maximum number of atoms in the queries of the pack) and F the fraction of the queries that do have the maximum length.
- *MinL(F)* with MinL the minimum length of the queries in the pack and F the

[1] we are not interested in answer substitutions, neither in multiple successes.

Pack	Nbq	Nbv	MaxL(F)	MinL(F)	Nbd	MaxBr(F)	MinBr(F)	Br(F)
pack1	46	141	3(100)	3(100)	1	46(100)	46(100)	46(100)
pack2	432	818	6(64)	4(11)	8	57(12)	48(12)	56(50)
pack4	3616	6306	9(80)	7(2)	64	59 (5)	57(56)	57(56)
pack6	12616	21546	12(86)	10(0.6)	216	76(0.5)	59(69)	59(69)
pack7	1498	2667	7(74)	5(3)	27	58(7)	52(4)	57(44)
pack8	16744	28905	10(78)	7(0.3)	295	60(1)	57(39)	58(46)
pack9	39706	68024	11(82)	8(0.1)	689	66(0.1)	58(49)	58(48)

Table 1
Some properties of typical query packs.

fraction of the queries that do have the minimum length.

- *Nbd* is the number of disjunctions in a query pack (where e.g. (a;b;c;d) is counted as one disjunction with a branching factor of four or having four branches).
- *MaxBr(F)* is the maximum number of branches MaxBr in the disjunctions, with F the fraction of the queries that have the maximum branching factor.
- *MinBr(F)* is the minimum number of branches in the disjunctions with F the corresponding fraction.
- *Br(F)* is the branching factor Br that corresponds to the largest fraction (F) of disjunctions.

3 Compiling disjunctions

3.1 General issues in compiling a disjunction

Standard WAM does not prescribe how to compile clauses containing an explicit disjunction - the ; control construct. E.g. [1] does not mention such disjunctions at all. As a consequence, Prolog implementations based on WAM have designed their own particular way of dealing with the disjunction. One approach is to introduce a **new predicate** for the disjunction and source-transform the disjunction away into a sequence of clauses, as exemplified in the following:

```
a :- b, (c ; d), e.
```

is transformed to:

```
a :- b, newpred, e.

newpred :- c.

newpred :- d.
```

This approach simplifies in many ways the rest of the implementation- see later, but also has its pitfalls: when the branches of the disjunction contain a cut (!/0), the new predicate must contain an ancestor cut, which means that the transformed program is no longer ISO-compliant. Also, if the goals `a,b,c,d` and `e` contain variables, one would like to give the newpred as few arguments as possible: this poses a complexity issue and is one of the reasons why in our setting, this is not an optimal choice. Next problem is that the choice point created for newpred, contains most often more slots than the choice point created for an inlined compilation of the disjunction (see later). Finally, the transformation in the above example leads to the creation (at runtime) of 2 environment frames instead of one, which is a time and space loss.

The other solution to compiling a disjunction is to **inline** it: this is done for instance in systems like XSB [14] and Yap [15]. Since ILPROLOG is currently based on the XSB compiler, we have inherited the approach. For a clause like

Example 3.1

`a :- b, (c ; d), e.`

The following code is generated:

```
allocate 2

call b/0

trymeorelse 3

call c/0

jump 4

label 3

trustmeorelsefail

call d/0

label 4

deallex e/0
```

This approach doesn't suffer from problems with larger choice points, more environments or with cuts, but is tricky: optimal classification of variables in temporary and permanent is not obvious, the notion of the first occurrence of a (permanent) variable needs to be re-investigated (shown in an example later) and even the permanent variable numbering (assignment to an environment slot) is no longer obvious.

- **Example 3.2**
  ```
  a :- (b(X) ; c(X)). % X can be void
  a :- (b(X) ; c(X,X)). % X can be temporary
  ```
- **Example 3.3**
  ```
  a :- (b(X), c(X) ; d(Y), e(Y)).
  ```
  can be rewritten to:
  ```
  a :- (b(X), c(X) ; d(X), e(X)).
  ```
  showing that X and Y (of the original clause) share the same slot.
- **Example 3.4**
  ```
  a :- (b(X) ; c), d(X).
  ```
  The occurrence of `X` in `d(X)` is not a first occurrence, even though following the second branch of the disjunction, it wasn't initialised - meaning the compiler must emit code that initialises `X` in the second branch.

Implementations have not always gotten this right: SICStus Prolog [13] does not give the singleton warning for Example 3.2 and XSB used to choke on Example 3.4. Neither Yap nor XSB make X and Y share the same slot in Example 3.3. Some of the problems compiling disjunction inlined disappear if the branches don't meet at the right. That's probably why M-Prolog used to restrict the usage of disjunction to non-meeting branches.

Another problem with inlined disjunction is related to precise garbage collection: systems like Yap and ILPROLOG put at each continuation point in the code the information which permanent variables are live at that point. This is similar to having pointer maps as described in [4]. So, the actual code for Example 3.1 is

```
allocate 2
call a/0
active_yvar 0
trymeorelse 3
active_yvar 0
call b/0
active_yvar 0
jump 4
label 3
trustmeorelsefail
call c/0
active_yvar 0
label 4
deallex d/0
```

where the pseudo-instruction *active_yvar* has as arguments some description [2] of which permanent variables are active at that (continuation) point. If the *(re)trymeorelse* instruction makes sure that the continuation pointer saved in its choicepoint points to the *active_yvar* instruction following the *trymeorelse*, one can have a uniform view - during marking - of reachability.

All these potential pitfalls related to the inlined compilation of disjunction have at least one implementor made regret this choice [10].

3.2 Issues in compiling a large disjunction

The problems current compilers have with large disjunctions, are mainly that

(i) the complexity of the overall compilation is non-linear in the size of the disjunction

(ii) large disjunctions contain too many permanent variables. This is a problem because Prolog implementations typically have a fixed maximum number of permanent variables, because it affects all kinds of design decisions. Just think about get/put/unify instructions that refer to permanent variables by their number and which have a fixed format for that number (8/16 bits).

[2] might be through a table.

	Yap-2.2.1 time	Yap-2.2.1 permvars	XSB time	XSB permvars	SICStus-3.8.2 time
pack1	0.01	2	0.57	2	0.12
pack2	0.26	13	24.39	13	2.31
pack4	*	*	1641.9	83	60.25
pack6	*	*	-	-	597.44
pack7	2.37	38	282.29	38	14.24
pack8	*	*	-	-	1194.6
pack9	*	*	-	-	M

Table 2
Current Prolog systems and large disjunctions

Both problems will be addressed in section 4. Here we give some empirical data for our claims in the form of a table which contains for a number of systems, some compilation characteristics for a set of packs taken from the Mutagenesis data set [12,3].

The measurements were performed on a Pentium II, 266MHz, 128Mb. Timings are in seconds.

In the table, a * means that execution was aborted because of a segmentation violation, M means that the process got out of memory, and - means it took just too long to wait for. The times for Yap and SICStus include read, compile and load. For XSB they include read, compile and produce a file with the abstract machine code. The timings of XSB were obtained by executing a slightly adapted XSB compiler under ILPROLOG . This doesn't affect the number of permvars. Using this XSB compiler under ILPROLOG is 2 to 4 times faster than using the original within the XSB-system.

The table clearly shows that ordinary compiler technology is inadequate for compiling large disjunctions. In Section 4.5, Table 3, we'll show the figures for the new compiler.

4 The new compiler

It turns out that current compilers do not cope adequately with such large clauses with many disjunctions and many variables. A major issue is the time-complexity of the algorithm for classifying the variables. In our case it is essential that such an algorithm is linear in the number of variables occurring in the system. Due to the huge amount of variables (quickly approaching hundred thousand variables, see Table 1) it is essential to determine the exact scope of the variables, so that the space for the variables in the environment can be reduced as shown in Example 3.3. It is obvious that correctness of the generated WAM-code should be preserved, in particular special care should be taken for the initialisation problem as shown in Example 3.4. A new ILPROLOG compiler is being developed that tackles the above problems, as is explained in the next subsections. This compiler is derived from the compiler used in [9].

4.1 Classifying variables for disjunctions that do not meet

In the context of the ILP applications, there definitely was a need for a better algorithm for classification of variables: variables are either permanent, temporary or void. Moreover, we can focus on clauses with disjunctions that do not meet, i.e. the nested disjunctions in a clause form a tree and no disjunction is followed by another call. The *classification algorithm* should recognise that the same variables that occur in different branches are actually different variables. In addition, for register allocation and other optimisations, the number of occurrences of variables should be counted, not only for temporary variables, but also for permanent variables. The algorithm should be blazingly fast, to be able to handle query packs with loads of variables and many branches.

The ILPROLOG compiler is currently written in Prolog. Its classification algorithm first gathers the relevant information and then the classification can be done. The classification algorithm is based on some abstract syntax of the clause, generated in some earlier phase of the compiler: each occurrence of a variable is replaced by a term `var(Orig, New)` where `Orig` is the original variable in the clause and `New` is a fresh variable which is different for each occurrence of the variable in the original clause. During the classification algorithm the `Orig` variable will be used to store temporary information. The `New` variable will contain the final result, namely the type of the variable – void, temporary or permanent – and the number of occurrences of the variable.

As an example we will use:

Example 4.1

```
q:- a(X), (a(X, Y);a(X, Y, Y)).
```

The outcome of the algorithm should classify X as a permanent variable. The first occurrence of Y as a void variable, the last two occurrences of Y as a temporary variable.

The input for the algorithm described here for example 4.1 is:

Example 4.2

$$\texttt{q:- a(var(X, New}_1\texttt{), (a(var(X, New}_2\texttt{), var(Y, New}_3\texttt{));}$$
$$\texttt{a(var(X, New}_4\texttt{), var(Y, New}_5\texttt{), var(Y, New}_6\texttt{))).}$$

The classification algorithm will parse the tree of disjunctions depth first, from left to right. During this traversal each branch in a disjunction gets a unique number. This number always grows during the parsing. Since a branch in a disjunction may contain a disjunction as well, we get a tree structure. Given the numbering scheme above each occurrence of a variable in the clause will have a *branch number*, namely the number of the current branch. We also define a *root set*, this is the set of all numbers on the branches from the root of the clause to the branch the variable occurs in, thus including its *branch number*.

For tracking information on permanent and temporary variables the classification algorithm divides clauses into *chunks* which are also numbered. The idea is that variables occurring in only one chunk are not permanent. When defining chunks, we make a distinction between Prolog builtins that are inlined by the underlying Prolog system and builtins that are defined by Prolog clauses. For chunks, the latter are considered as ordinary (Prolog) calls, and the former are simply called builtins. When scanning a clause from left to right, the first chunk is the part of the clause beginning with the head of the clause and ending with the first call in the body (if any). Note that there can be inline builtins between the head and the first call. The next chunks are parts of the clause starting just after a previous chunk and ending with a call (if any).

The chunks are also numbered depth first, from left to right in the tree of disjunctions. Actually one can use the chunk number of the first chunk of a branch as the *branch number* for that branch.

During the parsing of the clause the variables `Orig` are instantiated to a list of terms of the format `branch(Occurrences, branchNr)`. The same variable occurring in two different branches can be viewed as different variables whenever there is no occurrence of the same variable in a common branch on the path from the root of the clause. For each such *different* variable a term `branch( Occurrences, branchNr)` will be added to the list.

On the first occurrence of a variable the `Orig` variable is instantiated as `[branch(Occurrences, BranchNr)]` with `BranchNr` its *branch number*, and we store the `New` variable in a data structure contained in `Occurrences`. On subsequent occurrences of this variable we have to check whether the `BranchNr` is a member of the *root set* of that occurrence. If this is the case we have found a next occurrence of the same variable and we add the `New` variable to the data structure in the variable `Occurrences`. In case `BranchNr` is not a member of the *root set* we have found an occurrence of the variable which is syntactically identical, but is actually another variable. In this case we add a new term `branch(Occurrences, BranchNr)` to the list in variable `Orig`. Moreover, since we are numbering depth first, from left to right we will never find another occurrence of a variable which belongs to a term `branch(Occurrences, BranchNr)` which was is not the last created one. Therefore we adapt this list in variable `Orig` using destructive assignment (setarg/3): we put the new term in front of the list. Then we always have to add subsequent occurrences of a variable in the first term of the list, or add a new term to the list.

Inside the variable `Occurrences` we have again a data structure to keep information necessary to make the difference between void, temporary an permanent variables. This `Occurrences` variable is again a list of terms `chunk(NewVars, ChunkNr)`. Whenever we find an occurrence of a variable we check if the number of the Chunk where the variable is found is equal to the `ChunkNr` of the first term of the list `Occurrences`. If this is the case, we add the `New` variable to the list `NewVars`, again in front of the list using setarg/3. If this chunkNr is not equal, we create a new term `chunk(NewVars, ChunkNr)`, which is put in front of the list `Occurrences` again using setarg/3. Afterwards we can safely check only the first element of the list `Occurrences` since chunk numbers are increasing.

After parsing the first Branch of the code of Example 4.1 we have the following instantiation of variables:

Example 4.3

X = [branch([chunk([New_2],2),chunk([New_1],1)],1)],
Y = [branch([chunk([New_2],2)],2)]

The final instantiation if the variables after complete parsing:

Example 4.4

X = [branch([chunk([New_4],3),chunk([New_2],2),chunk([New_1],1)],1)],
Y = [branch([chunk([New_5,New_6],3)],3),branch([chunk([New_2],2)],2)]

Given the data structures build during the traversal of the complete clause with its disjunction tree, the next step is the actual classification of the variables. The result of the traversal of the clause returns a list of the variables in the clause. For each of these variables we consider the following cases:

- If the variable has several `branch/2` terms then each of these terms corresponds to a different variable.
- If a `branch/2` term contains only one occurrence of the variable, we classify the variable as `void` and `New` is instantiated to the term `void`.
- If the list contained in the first argument of the branch/2 term has only one element, then the variable occurred only within one chunk of the clause implying that it is a temporary variable. All `New` variables of all occurrences within `Occurrences` of this branch/2 term are unified and instantiated to the term `temp(_, Count, _)`, with `Count` the number of occurrences of this variable. The other two arguments will be used later in the compiler, mainly for register allocation.
- If the list contained in the first argument of the branch/2 term has more than one list, we have a permanent variable. Similarly, all `New` variables are unified and instantiated to the term `perm(_, Count, _)`.

To allow the garbage collector to collect the data structures generated during the classification we overwrite the `Orig` variable with the atom `[]`. The output of the algorithm for Example 4.1 is then:

Example 4.5
```
q:- a(var([], perm(1)),
             (a(var([], perm(1)), var([], void));
              a(var([], perm(1)), var([], temp(1)), var([], temp(1)))).
```

This concludes the classification algorithm for variables for disjunctions that do not meet. Remains to say a word on time complexity. All operations done at each occurrence of a variable are constant time except one. Namely the test whether the current *root set* contains the *branch number* of the first occurrence. The size of the root set is bounded by the depth of the tree of the disjunctive clause. Then if *d* is the depth of the tree, *n* the number of occurrences of variables and *m* a measure for the size of the clause we have that the time complexity is of the order $n*d + m$. Since d will always be rather small compared to n and m, the algorithm behaves almost linear.

4.2 Classifying variables for disjunctions that meet

The algorithm described above does not work in case the branches in a disjunction meet again. The structure in terms of disjunctions is then no more a tree. Luckily the large clauses we had to deal with the disjunctive branches did not meet, but it would be nice to have an algorithm for general clauses that behaves almost linear as well.

4.2.1 Motivation

An example of a program problematic for the previous algorithm is:

Example 4.6
```
p:- (a(X); b(X)), c(X).
```

The variable X would be regarded as two different variables in the two branches of the disjunction, but unfortunately the occurrence of X in the call to `c/1` makes them the same variable. This does not look too difficult to fix. But image a situation like:

Example 4.7
```
p:- (a(X); (g, (b(X); c(X)), (d(X);e(X)) ) ).
```

For this program the analysis should decide that the variable X actually represents two variables. One variable that occurs in the call to `a/1`, and all other occurrences of X is the other variable. One of the difficulties is to recognise at the time of seeing the term `d(X)` that the occurrence of X in `b/1` and `c/1` are now the same variable. While the occurrence of X in `a/1` is another variable.

4.2.2 The algorithm

The solution to the problem is to use a different numbering scheme for the branches. Situations like in Example 4.7 can be overcome by using branch numbers in the disjunctions with calls to `d/1` and `e/1` which are smaller than the branch numbers in the disjunctions with calls to `b/1` and `c/1`, but larger than the number of the branch starting at the call to `a/1`. For this purpose we need a numbering scheme in which we can always find a number between two given numbers: Rational numbers would be perfect, but we have used reals! Then we have at each point in the algorithm in addition to the number of the current branch a *delta*. All numbers in descendants of this branch should take numbers between the current branch number and the current branch number plus *delta*. Whenever we encounter a disjunction that meets we use numbers within that disjunction between the current branch number plus *delta/2* and the current branch number plus *delta*. All branches after the disjunction will get numbers between the current branch number and the current branch number plus *delta/2*. While in the previous algorithm branch numbers always grow during the algorithm, here these numbers can get smaller during the process.

Whenever we encounter an occurrence of a variable where the current branch number is smaller that the branch number of the variable in the first term in the `Orig` variable, we detected a variable created in a disjunction that appears in the code after the branches of the disjunction have met. Possibly we have to merge different terms in the `Orig`, because the algorithm decided at an earlier stage that it has found different variables. Not all of the terms in `Orig` variable will have to be merged, only the ones where its corresponding branch number is larger than the current.

The problem that arises now is the new branch number that should go with these merged branches. In case of Example 4.7 it should be the branch number of the branch starting at call `g/0`. Given the current representation this number is quite hard to find! Instead of keeping the branch number of the first occurrence of the variable we keep the complete *root set* including the branch number of the first occurrence of the variable. Then finding the new *root set* of branches that must be merged is quite easy. We just have to remove all branch numbers from the old *root set* that are larger than the branch number of the new occurrence of the variable.

Computing these branch numbers for Example 4.7 is quite technical but we worked it out for the technical readers among you. We annotated the clause with the tuples **b(BranchNr, Delta)** at each start of a branch:

Example 4.8

b(0,256) `p:- (`**b(128,64)** `a(X); (`**b(192,64)** `g,(`**b(224,16)** `b(X);` **b(240,16)** `c(X)),`
`(`**b(208,8)** `d(X);` **b(216,8)** `e(X)) ) ).`

It is important to note that the branch numbers in the disjunction containing `d` *and* `e` *are smaller than those in the disjunction containing b and c, but larger that the number of the branch startin at* `a`. *Meaning that X in* `a(X)` *is a different variable.*

When working with this new numbering scheme, keeping *root sets* instead of a branch number in each term in list `Orig` and merging terms when detecting shrinking branch numbers we can solve the problem. The remainder of the algorithm, parsing the clause and final classification, remains the same.

With respect to time-complexity this algorithm is not as good as the previous one. Merging different terms in the list `Orig` cannot be done in constant time. Currently this is linear in the number of occurrences of the variables involved in the operation. Considering that this part of the compiler is only used for handwritten code, meaning relatively small clauses, we do not really care about this potential quadratic behaviour.

4.3 Reusing slots for permanent variables in the environment

The usual option for assigning slots in the environment is to assign a slot in the environment whenever a permanent variable is detected. This can be done during analysis of the variables and results in a one on one mapping between permanent variables and environment slots. For a disjunctive clause one could use the same slot for different variables of which the lifetimes do not overlap. This can be seen in Example 3.3. For handwritten code such reuse will only result in minor advantage, but in our case such reuse is essential since the computer-generated clauses we are dealing with will exceed 65536 (2^{16}) permanent variables. When reusing slots in the environment, this can easily be reduced to less than 100 simultaneous permanent variables. Without complicated data structures or another pass through the clause it is quite difficult to assign slot numbers in the classification phase of the program. Therefore we choose to assign slots to the permanent variables dynamically during the generation of code.

In case of disjunctive clauses where branches never meet the scheme is quite simple. During the generation of code the compiler remembers the next available slot in the environment. Whenever the first occurrence of a permanent variable is encountered this number is used and the next number to be used is incremented by one. If the compiler finds a disjunction, the next number to be used can be the same for all branches. Indeed when the first occurrence of a permanent variable is found within one branch, this permanent variable will be surely dead whenever another branch in the disjunction is executed and can be safely reused.

In case of disjunctions that do meet, we can never be sure that permanent variables that occurred the first time in a branch will not be used in another branch. An occurrence of this variable in the code after the branches have met, could link another branch. For this purpose we count the occurrences of all permanent variables in a branch. If this count is not equal to the total number of occurrences – we collect this information in the classification phase, then the slot allocated for this permanent variable cannot be safely used by other branches in the disjunction. Still we use the same method as above and we start with the same slot in the environment to be used first for the different branches of disjunctions. Next to that we have a list of forbidden slots in the environment: The slots that were used for permanent variables in previous branches and not all occurrences of that permanent variables were seen in that branch.

4.4 The initialisation problem for disjunctions that meet

Exactly those permanent variables from the previous sections where:

- the first occurrence was seen by the compiler in some branch of a disjunction, and
- not all of the occurrences were seen by the compiler in that very same branch

cause problems with initialisation. Since not all occurrences were seen in the same branch of the disjunction, there must be an occurrence in some code after the disjunction, when the different branches of the disjunction have met. Of course during execution one never knows which branch of the disjunction has been executed before the code with these extra occurrences. In these extra occurrence one must assume that the variable is initialised, if the wrong branch was taken it is not.

For this reason we need some bookkeeping for all such variables that:

- occur in a some branch of a disjunction,
- do appear in some code after the disjunction,
- do not appear in the code before the disjunction,
- and do not appear in another branch of the disjunction.

WAM code should be generated to initialise this permanent variable in this other branch where it does not appear. Of course at the end of the WAM-code for this branch, when it is sure that the branch will not fail.

4.5 Evaluation

In Table 3 we can see a comparison between the XSB compiler [14] we used before in our ILPROLOG system and our new compiler. We included the SICStus [13] timings for comparison as well. For both systems reading the highly disjunctive clause from file, compiling and write a file with the WAM code.

For the new ILPROLOG compiler the time between brackets is the compilation time only: the reader is responsible for the difference between the two numbers. It is relevant for us to have this number, as eventually, the connection between the new compiler and the system will be without files. And yes, we can improve the reader within the ILPROLOG system a lot!

From the table we can see that the compiler has a very reasonable behaviour concerning execution time. The main reason for this is the difference in time-complexity for the classification of variables when disjunctions are involved.

Another striking aspect is the number of slots needed in the environment. When using a classic scheme the number of slots needed goes up very rapidly (last column). Note that theses are the same as in the XSB permvars column. When reusing

	XSB time	XSB permvars	SICStus time	ILPROLOG time	ILPROLOG new #permvars	ILPROLOG classic #permvars
pack1	0.57	2	0.12	0.05 (0.05)	2	2
pack2	24.39	13	2.31	0.66 (0.56)	7	13
pack4	1641.9	83	60.25	10.62 (5.77)	9	83
pack6	-	-	597.44	117.21 (20.69)	11	257
pack7	282.29	38	14.24	2.8 (1.84)	8	38
pack8	-	-	1194.6	234.25 (28.25)	11	365
pack9	-	-	M	1445.47 (57.06)	14	822

Table 3
Our new ILPROLOG compiler compared to the XSB compiler we used before

slots this number of slots can be reduced to a very small amount (column labeled new #permvars). Remark that many Prolog-systems get into trouble when having more than 256 permanent variables. This would be the case for pack number 6, 8 and 9.

5 Precise reachability revisited for packs

During garbage collection of the heap pointers from within the environment to the heap should be used in the marking phase. Unfortunately there could be permanent variables within the environment that are already dead! Even worse permanent variables that have not been initialised, ILPROLOG does not clean the environment on creation, could contain invalid information. For this purpose we use the technique of "active yvars" introduced in Section 3.1. After each call we have an extra opcode which has as a parameter all permanent variables alive after the call. During garbage collection the algorithm can find this information using the continuation pointer in the environments. In the case of disjunctions we need also information on the permanent variables alive at the time of creation of the choicepoint. Thus a *try_me_else* instruction needs "active yvars" as well. Moreover within normal disjunction we can improve the precision of the garbage collection by adding an "active yvar" to *retry_me_else* instructions as well, because some permanent variables alive at in the first alternatives of the disjunctions could be dead in later branches of the disjunctions, as is the case in the second branch of Example 5.1 with variable X.

Example 5.1

```
p:- p(X), (q(X); d; e).
```

Unfortunately during execution of packs the order of execution of the different branches is not known in advance, as discussed in Section 2.

Example 5.2
`p:- p(X), (q(X); r(X); s).`

for instance in Example 5.2 branch `s` could be executed before branch `r(X)`. This means we cannot shrink the set of permanent variables, when we move from one branch to another in the disjunction. So we have to stick to the set of permanent variables alive at the time of starting the disjunction with possible imprecise garbage collection.

6 Incremental compilation of query packs

Even though the current query pack compilation times are acceptable and scale well, it remains worthwhile looking at speeding up the compilation process. Apart from the obvious *rewrite it in C*, we want to explore the possibility of incrementally compiling query packs: During a particular learning session usually a sequence of packs of queries needs to be evaluated, one at a time. The packs in this sequence are not independent. The n-th pack is basically the (n-1)-th pack from which some branches have been pruned and to which the remaining branches are extended. E.g. pack 1 could be[3]:

```
a, (b, (c ; d) ; e, (f ; g) ; h)
```

and pack 2 could be:

```
a, (b, d, (x ; y) ; e , f, (u ; v) ; h, (i ; j))
```

The idea of incrementally compiling the pack is to transform the code for pack i into the code for pack (i+1). We just point out some of the issues one has to take care of:

- One of the things that can happen by extending a branch, is that a last occurrence of a permanent variable, is no longer last: in ILPROLOG this poses no particular problem as variables are always initialised on the heap, so ILPROLOG doesn't have a *PUT_UNSAFE_VALUE* instruction. If it would have had one, then it would be a matter of being able to find out which *PUT_UNSAFE_VALUE* instruction need to be changed into a *PUT_YVAL*.
- In the inductive learning setting of XTILDE, each branch of the pack ends with a call to a predicate that reports which branch succeeded. This call has as argument an indication of the branch which might need to change for the new pack.
- Cutting a branch away (from a certain point) might delete a set of permanent variables from the pack: it would be best if the incremental compiler knows

[3] we keep using the notation for disjunction for convenience

about it so that these slots can be reused. While doing such reuse, the idea is not to touch the assignments of surviving permanent variables. This means that an incrementally compiled pack could potentially use more than the minimal environment. However, the allocation policy of the compiler described earlier, is such that slots of removed variables are either used by other variables (in parallel branches) or these slots have a highest number that can be freed without extra cost of reallocating other variables. This is true because, within one branch, first occurrences of permanent variables are numbered sequentially. Then permanent variables initialised in deleted branches, must have the highest number.

- Typically when a branch is extended, the extension will contain variables that occurred already. This means that the classification of a variable might need to be changed from *void* or *temporary* to *permanent*. It seems better to avoid such change of classification and let the pack compiler classify every variable as permanent, even if WAM would classify it differently.
- Reachability information about the permanent variables must also be adapted.

In order to make this incremental compilation easier, it is important that there is cooperation with the pack generation phase: currently, the generation of the (extensions to a) pack is totally separated from the compilation process itself. However, the generation phase *knows* that it builds an extension with variable that existed already, because it makes such decisions on the basis of the language bias. The idea is clear: the generation phase can convey such information to the compiler, so that true incrementality can be achieved.

7 Conclusion and Future Work

The speedups obtained by packs are impressive, and especially their scalability - now that we have the better compiler - is most comforting. However, other techniques must be incorporated in ILPROLOG to make other parts of the system scale well: support for very large datasets, fast switching between examples and better indexing than Prolog systems usually provide. Especially the latter is of the kind that could *counteract* the speedups of packs. However, we anticipate that there will always be a net gain of packs.

Acknowledgement

We thank the people from the learning group, in particular Hendrik Blockeel and Jan Ramon for teasing us with such nice problems. The third author was for this research partly funded by the project FWO-Vl. G.0246.99 and partly by the Esprit project Aladin 28623.

References

[1] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, Massachusetts, 1991. See also: http://www.isg.sfu.ca/~hak/documents/wam.html.

[2] H. Blockeel and L. De Raedt. *Top-down Induction of First Order Logical Decision Trees* Journal of Artificial Intelligence, Vol. 101, 1-2, pp. 285-297, June 1998.

[3] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. *Executing Query Packs in ILP* Accepted for the 10th International Conference on Inductive Logic Programming (ILP 2000), July 2000.

[4] P. Branquart and J. Lewi. A scheme of storage allocation and garbage collection for ALGOL 68. Proceedings of the IFIP Working Conference on ALGOL 68 Implementation, July 1970.

[5] Koen De Bosschere. http://www.elis.rug.ac.be/~kdb/

[6] L. Dehaspe and H. Toivonen. *Discovery of frequent Datalog patterns* Data Mining and Knowledge Discovery 3(1): 7-36, 1999.

[7] B. Demoen, G. Janssens, H. Vandecasteele. *Executing Query Flocks for ILP* Proceedings of BENELOG'99, Maastricht, 5 November 1999. Also available on the web:
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=18794

[8] B. Demoen, P-L. Nguyen. *So many WAM variations, so little time.* To appear in the Proceedings of CL2000, London, July, 2000

[9] H. Vandecasteele. Compiling Prolog to ANSI C. In Y. Deville, editor, *Proceedings of the Eight Benelux Workshop on Logic Programming*, 1996.

[10] Vitor S. Costa - private communication about Yap - beginning 2000.

[11] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.

[12] A. Srinivasan, S.H. Muggleton, and R.D. King. Comparing the use of background knowledge by inductive logic programming systems. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, 1995.

[13] http://www.sics.se/sicstus/

[14] http://xsb.sourceforge.net/

[15] http://www.ncc.up.pt/~vsc/Yap/

Assumption Grammars for Generating Dynamic VRML Pages

Anima Gupta

Department of Computer Science
University of North Texas
P.O. Box 311366
Denton, Texas 76203, USA
E-mail: agupta@cs.unt.edu
WWW: http://www.cs.unt.edu/˜ agupta

Paul Tarau

Department of Computer Science
University of North Texas
P.O. Box 311366
Denton, Texas 76203, USA
E-mail: tarau@cs.unt.edu
WWW: http://www.cs.unt.edu/˜ tarau

Abstract

We describe a Prolog to VRML mapping allowing generation of dynamic VRML pages through CGI and server side Prolog scripts. BinProlog's Assumption Grammars (a form of multi-stream DCGs with implicit arguments and temporary assertions, scoping over the current AND-continuation) are used to mimic VRML syntax and semantics directly, without using a preprocessor. The resulting scripting language allows quick development of integrated knowledge processing and high quality data visualization applications. Using BinProlog's multi-threading, we describe a design integrating in a self-contained Prolog application a Web Server, a Data Extraction module and an Assumption Grammar based VRML generator.

Keywords: *Automatic Software Generation, Internet Programming, Logic Programming, Prolog, Definite Clause Grammars, Assumption Grammars, Virtual Worlds*

1 Introduction

VRML (Virtual Reality Modeling Language) [13,10,2,11] is a de facto Internet standard for 3D visualization. Its applications range from modeling unique objects like the International Space Station, to shared virtual reality and 3D visualization of complex data.

VRML does not provide extensive programming language constructs, and using its embedded JavaScript-like scripting language is cumbersome. Although an External Authoring Interface [22] allows controlling VRML worlds from Java, the combination of multiple layers running within a browser makes EAI applications extremely fragile.

Prolog is a flexible and powerful AI language, particularly well suited for knowledge processing, Internet data extraction, planning, machine learning, hypothetical reasoning etc.

This has motivated us to combine the two technologies to allow the integration of our Prolog based Internet data extraction tools and VRML's 3D visualization and animation capabilities.

In this paper we describe a Prolog to VRML mapping, which allows generation of dynamic VRML pages through CGI and server side Prolog scripts, following the technique introduced in [15].

Assumption Grammars [20,5] are multi-stream DCGs with implicit arguments and temporary assertions, scoping over the current AND-continuation [1]. We will use them to mimic VRML syntax and semantics with an Assumption Grammar based BinProlog code generator. The resulting scripting language will allow quick development of integrated knowledge processing and high quality data visualization applications. Using BinProlog's multi-threading, we will integrate in a self-contained Prolog application a Web Server, a Data Extraction module, and an Assumption Grammar based VRML generator. From the user's point of view our integrated set of tools looks like a dynamic VRML page which reacts to user input through a set of HTML forms and VRML anchors [2]. This allows the user to provide some parameters or click on sensor enabled VRML areas and see as a result the output of our server side VRML generator, displayed by the browser as a 3D animation.

2 Overview of Assumption Grammars

2.1 *Assumed code, intuitionistic and linear implication*

Intuitionistic `*/1` adds temporarily a clause usable in later proofs, through calls to `-/1`. Such a clause can be used an indefinite number of times, mostly like asserted clauses, except that it vanishes on backtracking. Its scoped version `=>` (intuitionistic implication [8]) as used in

```
Clause=>Goal or [File]=>Goal
```

[1] This allows producing and consuming backtrackable prolog assertions as a means to exchange information between arbitrary points of the parsing process.

[2] Hyperlinks attached to VRML nodes.

makes `Clause` or the set of clauses found in `File` available only during the proof of Goal. Both vanish on backtracking. Clauses are usable an indefinite number of times in the proof, i.e. for instance

```
?- *a(13),-a(X),-a(Y)
```

will succeed, as `a(13)` is usable multiple times, and therefore matches both `a(X)` and `a(Y)`.

The linear assumption operator `+/1` adds temporarily a clause usable *at most once* in later proofs, through use of `-/1`. This assumption also vanishes on backtracking. Its scoped version `-:` (linear implication [9,7]) as used in

```
Clause-:Goal or [File]-:Goal
```

makes `Clause` or the set of clauses found in `File` available only during the proof of Goal. Assumptions vanish on backtracking and are accessible only inside their stated scope. Each clause is usable at most once in the proof, i.e. for instance

```
?- +a(13),-a(X),-a(Y).
```

fails as `a(13)` is usable only once and therefore consumed after matching `a(X)`.

One can freely mix linear and intuitionistic clauses and implications for the same predicate, for instance as in:

```
?-a(10)-:a(11)=>a(12)-:a(13)=>(-a(X),-a(X)).
X=11 ;
X=13 ;
no
```

At the time a(X) is called with `-a(X)` we have the assumed facts a(10), a(11), a(12) and a(13), as implications are embedded left to right. Given that X is consumed twice, only the values 11 and 13 assumed with intuitionistic implications `=>` will be returned as solutions. The (two) attempts to consume twice the values `a(10)` and `a(12)`, assumed with linear implication, will produce no solutions.

2.2 The Assumption Grammar API

Assumption Grammars [20,5] are logic programs augmented with linear and intuitionistic implications scoped over the current continuation, and implicit multiple DCG accumulators [3].

The complete Assumption Grammar API consists of 3 assumption operators:

`*A: adds intuitionistic assumption A`

[3] An accumulator is a sequence of chained logic variables, for instance S1,S2,S3 as used in the DCG rule `a-->b,c` which expands to `a(S1,S3):-b(S1,S2),c(S2,S3)`. Assumption Grammars provide the equivalent of an arbitrary number of such variable chains - through and internal implementation using backtrackable destructive assignment, instead of the program transformation approach used to implement single stream DCGs.

```
+A: adds linear assumption A
-A: matches an assumption in current scope or fails
```

and 3 DCG-stream operators.

Implicit DCG-streams can be seen as providing implicit input and output sequence arguments. While operationally equivalent to program transformation based DCGs, no preprocessing is required, as terminals interact with the implicit DCG stream directly, through a set of BinProlog built-ins:

```
#<Tokens : initializes/unifies the DCG stream with a list of Tokens
#Token: matches or inserts a Token in the implicit DCG stream
#>State: returns the current State of the implicit DCG stream
```

The following pattern explains the use of implicit accumulators in matching/recognition mode:

```
?- #<[a,b,c,d],#A,#B,#>LeftOver.
A=a,
B=b,
LeftOver=[c,d]
```

For generation mode we start with an unbound value `Xs` for the implicit DCG stream, then we bind it to list elements `a,b,c` by stating that `a,b,c` are terminals.

```
?- #<Xs,#a,#b,#c,#>[].
Xs=[a,b,c]
```

The combination of DCG stream operations and assumptions is particularly well suited for generating hierarchical data structures where properties hold for selected subobjects.

3 An Overview of VRML

VRML'97 is a hierarchical scene description language which is human readable [13,2]. The VRML scene is composed of **nodes** and **routes**. Each node can have a unique name for routing or re-use, a type, a number of associated events (both incoming and outgoing), a number of fields which hold the data for the node. In addition, group nodes can have many other nodes as children. One special node (script) can hold a Javascript program to control behavior in the scene. Group nodes can hold many other nodes, associating them in some fashion. Positioning of objects in the world is achieved through the **transform** node. The transform node applies **scale** operations, **rotations** and **translations** to its children. To achieve complex repeated transforms, transform nodes may be nested. Sensors can be thought of in two categories, those associated with geometry and those for sensing time. The sensors can respond to various conditions such as proximity to an avatar, mouse clicks, mouse movement or whether an object is in the field of view of the user. Each interpolator node has a set of **keys** and a set of **keyValues**. When a **set_fraction** eventIn is received, the values are interpolated to get the correct **value_changed** eventOut. Prototypes are the mechanism used for expanding the VRML language. A prototype has a new type name followed by a list of fields and events in brackets. Interacting with VRML worlds or animating them will cause events to be generated. These are sent from one node to another. Here are some typical event sequence patterns: usually a Sensor of some kind generates an event to start the animation. Sometimes processing of events is required which can be done through a simple script node. For non-reactive subcomponents (for instance to run an animation or to supply a stream of events for a color interpolation) Time Sensors can be used.

VRML's hierarchical space representation suggests a natural mapping to Prolog terms. Interestingly enough, both VRML and Prolog describe *atemporal* structures - space in the case of VRML and logical axioms in the case of Prolog. To some extent, VRML's declarative **space** representation and Prolog's declarative **truth** representation share the same difficulties when faced with the problem of representing time and change. Not surprisingly, VRML's event propagation mechanism (needed for animation) has a strong declarative flavor. In fact, it is quite similar to Prolog's DCG argument chaining: events are propagated with ROUTE statements between chained nodes.

3.1 Syntactical similarities between Prolog and VRML

VRML's concrete syntax is closer to languages like Life [1] than to Prolog, i.e. all arguments are *named*, not positional. However, by using compound terms tagged with VRML's attribute names and some user defined operators (like `@`) we can easily mimic VRML syntax in Prolog.

3.1.1 Syntax mapping of some key idioms

Our mapping is so simple that it does not even need a preprocessor. Instead, we use Prolog's reader with some prefix and infix operator definitions such that a standard Prolog reader accepts constructs like `@[...]` and `@{...}` as terms. As the same applies to Assumption Grammars, this pseudo-VRML notation turns out to be just plain Prolog. The details of the mapping are as follows:

```
Prolog         VRML
------------------------
  @{  } ===>   {  }
  @[  ] ===>   [  ]
  f(a,b,c) ==> f a b c
  a is b   ==> a 'IS' b
  a=b      ==> a b
------------------------
```

The key idea behind this mapping is to allow writing unrestricted VRML code in Prolog, while being able to interleave it with “compile time” Prolog computations. The following example illustrates how to build pseudo VRML in Prolog, in the (interesting) case when a VRML PROTO construct is generated, which in turn can be used to model multiple instances of VRML objects.

```
proto anAppearance @[
   exposedField('SFColor')=color(1,0,0),
   exposedField('MFString')=texture@[]
]
@{
   'Appearance' @{
       material='Material' @{
         diffuseColor is color
       },
       texture='ImageTexture' @{
         url is texture
       }
    }
}.
```

The reader will notice that the resulting VRML code is in fact a close syntactic variant of its Prolog template.

```
#VRML V2.0 utf8

PROTO anAppearance
[
  exposedField SFColor color 1 0 0 ,
  exposedField MFString texture []
] {
  Appearance { material
    Material { diffuseColor IS color }
    texture ImageTexture {
      url IS texture
    }
  }
}
```

3.2 Building Prolog Macros

While VRML’s PROTO constructs allow significant code reuse, it is sometimes preferable to build simpler VRML code through Prolog ”macro”-like constructs, which, depending on their parameters, will expand to multiple VRML node instances.

```
shape(Geometry):-
  % based on scoped assumptions (=>)
  default_color(Color),
  toVrml(
     aShape @{
       geometry(Geometry@{}),
       Color
     }
  ).

sphere:- shape('Sphere').
cone:- shape('Cone').
cylinder:- shape('Cylinder').
box:- shape('Box').
```

3.3 Using Prolog macros

We have added a ‘ (backquote) notation to force evaluation of Prolog macros by our generator.

```
group @{
  children @[
    `transform(
       translation(0,10,4),scale(2,2,2),rotation(0,0,0,0),
       [`sphere, `cone]
     ),
    `transform(
       translation(5,0,0),scale(1,3,6),rotation(0,1,0,1.5),
       [`box]
     )
  ]
}.
```

The generated code looks as follows:

```
Group {
  children [
    Transform {
      translation 0 10 4 scale 2 2 2 rotation 0 0 0 0
      children [ # aShape is a VRML 2.0 PROTO!
        aShape { geometry Sphere{} color 0.7 0.8 0.8
        } ,
        aShape { geometry Cone{} color 0.6 0.3 0.9
        }
```

```
      ]
    } , .....
  ]
}
```

Using assumptions in the VRML generator macros

The following example illustrates the use of a scoped implication to color *blue* a complete sub-object (cone) expressed itself as a macro.

```
def saucer = `transform(
   translation(0,1,1),scale(0.6,0.2,0.6),rotation(0.5,0.5,0,1.5),
   [
     `sphere,
     % this can used to propagate a color over a region
     `(color(blue)=>>cone)
   ]
  )
```

3.4 The Assumption Grammar based VRML Generator

Our generator is implemented as a predicate **toVrml/1** which takes a Pseudo-VRML tree and expands it into an implicit argument DCG stream. The (simplified) code looks as follows:

```
toVrml(X):-number(X),!,#X.
toVrml(X):-atomic(X),!,#X.
toVrml(A@B):-!,#indent(=),toVrml(A),toVrml(B).
toVrml(A=B):-!,toVrml(A),toVrml(B).
toVrml(A is B):-!,#indent(=),toVrml(A),#'IS',toVrml(B).
toVrml({X}):-!,#'{',#indent(+),toVrml(X),#indent(-),#'}'.
toVrml(`X):-!,X.
toVrml(X):-is_list(X),!,
  #'[',#indent(+),vrml_list(X),#indent(-),#']'.
toVrml(X):-is_conj(X),!,vrml_conj(X).
toVrml(T):-compound(T),#indent(=),vrml_compound(T).
```

A final step is performed by a simple post-processor which applies indentation hints and formats the actual VRML output. VRML'97 ROUTES are structurally similar to DCGs: they are used to *thread* together streams of change expressed as event routes. While a DCG-like notation could have been used to generate such ROUTE statements automatically, we have preferred to use the simpler syntax mapping mechanism as it is closer to what VRML programmers expect.

4 Web Application architectures

4.1 CGI script based VRML generation

We have integrated our VRML generator with a BinProlog based CGI scripting toolkit [17]. This allows generation of dynamic VRML content returned to the browser as a result of a POST-method call (Fig 1).

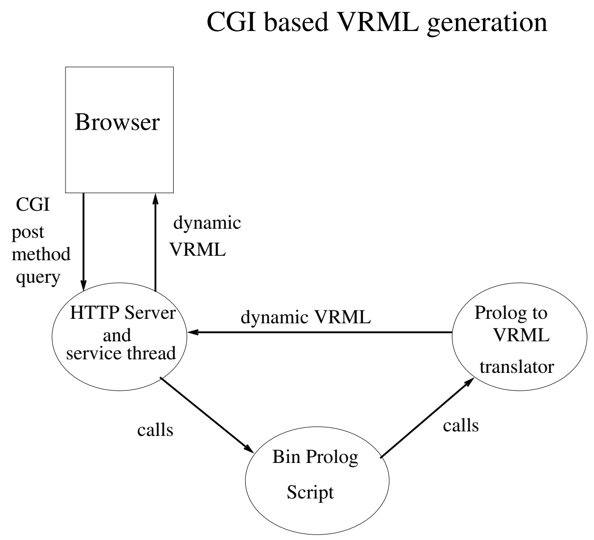


Fig. 1. A CGI based Architecture

The process of generating a VRML page with a CGI script is as follows:

- *the user clicks on an HTML form or VRML anchor*
- *a CGI script is invoked by the HTTP server*
- *the BinProlog based VRML generator is activated*
- *the dynamic VRML page is sent back to the client*
- *the browser's VRML plugin displays the results*

A CGI based demo of our generator implementing this architecture is available at `http://www.binnetcorp.com/BinProlog`.

Persistent server state can be maintained using BinProlog's remote predicate calls to a persistent BinProlog server or through some higher level networking abstractions like mobile threads [19] or remote blackboards [6].

4.2 Server Side Prolog based VRML generation

By using the BinNet Internet Toolkit [17] we can further simplify this CGI-based architecture as shown in Fig. 2). Instead of calling a CGI script each time, a form is sent to the server with a SUBMIT request (POST or GET) to provide a dynamic

VRML page. We can then simply embed the VRML generator in the server. As in most modern Web server architectures (Apache, Microsoft, Sun etc.), a new BinProlog *thread* is spawned inside the server process for each request. Server state can be shared between multiple users and made persistent by having a backup thread periodically writing the server's state to a Prolog dynamic clause file. This feature is particularly important for shared virtual world applications like LogiMOO [21].

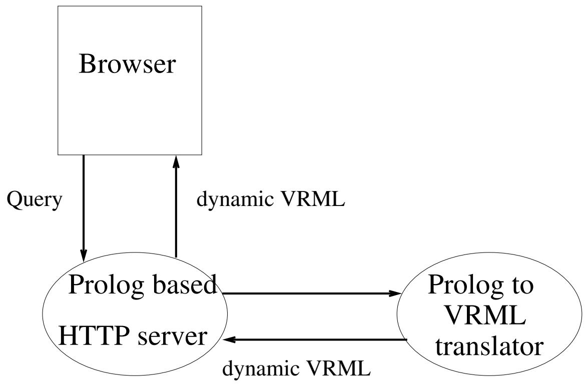


Fig. 2. Server Side Prolog based VRML Generation Architecture

To summarize, the process of generating a VRML page with a generator embedded in the Web server is as follows:

- *the user clicks on an HTML form or VRML anchor*
- *the VRML generator working as a server side Prolog thread is activated*
- *the dynamic VRML page is sent back to the client*
- *the browser's VRML plugin displays the results*

The multi-stream nature of Assumption Grammars is instrumental in embedding the VRML generator in ordinary CGI script processing - which uses its own DCG stream to parse input. The same thing happens in the case of the server side use - other DCG streams are used to parse elements of the server side HTTP protocol as well as to generate output. Assumption Grammars based I/O shares some similarities with Monadic [23] I/O - as present in functional languages like Haskell, [12]. For instance, by using Assumption Grammars we are able to separate the logic of sequencing the output from formatting and writing to the output stream.

4.3 *Combining the VRML Generator with Web Data Extraction and Persistent Server State*

A number of ongoing projects in our research group involve Web data extraction. A typical application architecture involves *blackboard* based agent coordination (Fig.

3).

4.3.1 Linda Based Agent Coordination

Our agent coordination mechanism is built on top of the popular Linda [4] coordination framework, enhanced with unification based pattern matching, remote execution and a set of simple client-server components merged together into a scalable peer-to-peer layer, forming a network of interconnected virtual places. The key Linda operations are the following:

```
out(X): puts X on the server
in(X):  waits until it can take an object
        matching X from the server
all(X,Xs): reads the list Xs matching X
        currently on the server
```

The presence of the `all/2` collector avoids the need for backtracking over multiple remote answers. Note that the only blocking operation is `in/1`. Typically, distributed programming with Linda coordination follows consumer-producer patterns with added flexibility over message-passing communication through associative search.

4.3.2 Agent Coordination with Blackboard Constraints

A natural extension to Linda is to enable agent threads with *constraint solving* for the selection of matching terms on the *blackboard*, instead of plain unification. This is implemented in Jinni [18] through the use of 2 builtins:

`Wait_for(Term,Constraint)`: waits for a Term on the blackboard, such that Constraint is true, and when this happens, it removes the result of the match from the blackboard with an `in/1` operation. Constraint is either a single goal or a list of goals `[G1,G2,..,Gn]` to be executed on the server.

`Notify_about(Term)`: notifies about this term one of the blocked clients which has performed a `wait_for(Term,Constraint)` i.e.

```
notify_about(stock_offer(aol,89))
```

would trigger execution of a client having issued

```
wait_for(stock_offer(aol,Price),less(Price,90)).
```

In a *client/server* Linda interaction, triggering an atomic transaction when data, for which a constraint holds, becomes available, would be expensive. It would require repeatedly taking terms out of the blackboard, through expensive network transfers, and put them back unless the client can verify that a constraint holds. Our *server side* implementation checks a blackboard constraint only after a match occurs between new incoming data and the head of a suspended thread's constraint checking clause, i.e. an indexing mechanism is used to avoid useless computations. On the other hand, a mobile client thread can perform all the operations atomically on the server side, using local operations on the server, and come back with the results.

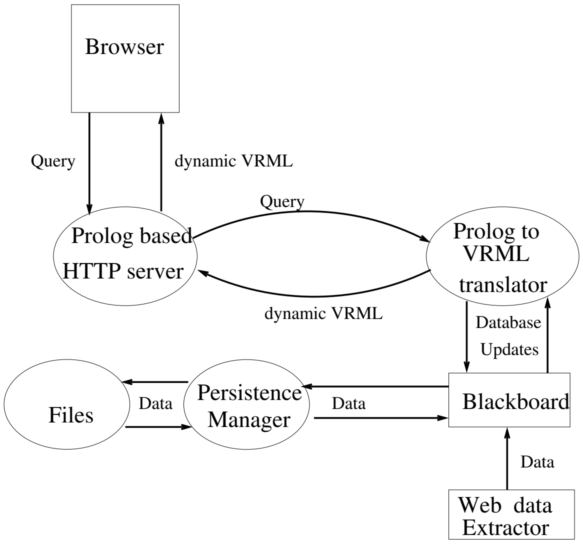


Fig. 3. Prolog HTTP Server With VRML Generator, Web Data Extraction and Persistent State

In a typical application (see Fig 3), a Web based data extractor fetches stock quotes and related historical information, which is visualized using our VRML generator. In particular, before committing to a stock market transaction, a visual animation of the projected stock marked dynamics can help human users to quickly validate the proposed buying or selling action. Trader agents, watching for triggers expressed as blackboard constraints run in the server process on separate threads. They allow expressing complex conditions for buy or sell transactions, far beyond the *stop* and *limit* transactions conventional online brokerages can offer. The state of the server (Web data and user agents) is made persistent through a separate thread which periodically saves terms on the blackboard and thread suspension records on the server, to Prolog dynamic clause files.

5 Related Work

Dynamic generation of HTML Web pages with Prolog CGI scripts has been pioneered by the Madrid group's Pillow library [3]. Currently most free and commercial Prologs offer basic support for CGI programming. Among them the BinNet Internet Toolkit [17] offers extensive server and client side tools, ranging from a Prolog based Web server supporting SSI (Server Side Includes) written in Prolog, to a programmable Internet Search Engine (spider). The first description of a Prolog term to VRML mapping (of which the one used in this paper is an extension) has been presented in [15]. This seems to be the model also followed (using different implementation techniques) by [14], which also pioneers the use of VRML visualization in constraint logic programming. Dynamic VRML manipulation using a Java + EAI + VRML plugin + Prolog interpreter in Java architecture is part if the BinNet Jinni demos - available online at `http://www.binnetcorp.com/Jinni`). This prototypes a shared virtual world - allowing synchronized manipulation of multiple VRML objects by human users or Jinni based Prolog agents. Global state kept on a server ensures that all users see the same VRML landscape.

An important difference between the approach described in this paper and [14] is the ability to use assumptions to parameterize *deep* components of VRML trees. For instance, in [14] a code transformer needs to be written to replace cylinders by cones in the Prolog sources of a VRML page. We can achieve the same effect (see subsection 3.3) by expanding a generic macro `shape` and then pass the actual value (`cone` or `cylinder`) as an intuitionistic assumption. At the time the macro is called, the effect of the assumption will generate either a cylinder or a sphere, depending on the assumption.

The approach described in this paper also differs from [14] as we use Assumption Grammars - as outlined in [15]. Among the advantages of Assumption Grammars over conventional DCGs:

- no preprocessor is needed, therefore VRML syntax is emulated simply by adding a few Prolog operator definitions
- defining a reentrant macro language can be achieved more easily than with repeated calls to the DCG term expander
- the use of assumptions and (scoped) implications allows injecting modifications of colors and various other attributes in the VRML code

On the downside, as only BinProlog [16] and Jinni [18] support Assumption Grammars at this time, portability of our generator is quite limited.

6 Conclusion

We have described a novel use of Prolog as a program generator for empowering special purpose languages without processing abilities like VRML. We have described two Web based architectures for generating dynamic VRML, supporting persistent server side and multi-user synchronization. Not only Prolog can be used as a macro language for building compact template files but typical AI components like Prolog planners can be used for realistic avatar movement in VRML worlds.

Our syntax-mapping based generator mechanism allows reuse of VRML programming skills, while the presence of Prolog as a powerful macro language provides an interesting synergy between Prolog's planning, scheduling, Internet data extraction ability and VRML's Internet ready animated 3D visualization.

Acknowledgment

We thank the anonymous referees for their valuable comments and suggestions.

References

[1] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3/4):195, 1993.

[2] D. R. J. L. Ames, Andrea L./Nadeau. *VRML 2.0 Sourcebook.* Wiley, Dec. 1996.

[3] D. Cabeza, M. Hermenegildo, and S. Varma. The PiLLoW/CIAO Library for INTERNET/WWW Programming using Computational Logic Systems, May 1999. See http://www.clip.dia.fi.upm.es/Software/pillow/pillow.html.

[4] N. Carriero and D. Gelernter. Linda in Context. *CACM*, 32(4):444–458, 1989.

[5] V. Dahl, P. Tarau, and R. Li. Assumption Grammars for Processing Natural Language. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 256–270, MIT press, 1997.

[6] K. De Bosschere and P. Tarau. Blackboard-based Extensions in Prolog. *Software — Practice and Experience*, 26(1):49–69, Jan. 1996.

[7] J. S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, May 1994. Available as University of Pennsylvania Technical Reports MS-CIS-92-28 or LINC LAB 269.

[8] J. S. Hodas and D. Miller. Representing objects in a logic programming language with scoping constructs. In D. D. H. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 511 – 526. MIT Press, June 1990.

[9] J. S. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear logic. *Journal of Information and Computation*, 110(2):327–365, May 1994.

[10] K. Jamsa Kris A./Schmauder;Phil/Yee, Nelson/Jamsa. *VRML Programmer's Library*. May 1997.

[11] A. D. Martin McCarthy. *Reality Architecture: Building 3D Worlds in Java & VRML.* Feb. 1998.

[12] S. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language. Technical report, Feb. 1999. available from: http://www.haskell.org/onlinereport/.

[13] G. B. Rikk Carey. *The Annotated Vrml 2.0 Reference Manual.* June.

[14] G. Smedbäck, M. Carro, and M. Hermenegildo. Interfacing Prolog and VRML and its Application to Constraint Visualization. In *The Practical Application of Constraint Technologies and Logic programming*, pages 453–471. The Practical Application Company, April 1999.

[15] P. Tarau. Logic Programming Tools for Advanced Internet Programming. In J. Maluszynski, editor, *Logic Programming, Proceedings of the 1997 International Symposium*, pages 33–34, MIT press, 1997. http://www.cs.unt.edu/ tarau/research/PapersHTML/ptut/art.ps.

[16] P. Tarau. BinProlog 7.0 Professional Edition: Advanced BinProlog Programming and Extensions Guide. Technical report, BinNet Corp., 1998. Available from http://www.binnetcorp.com/BinProlog.

[17] P. Tarau. BinProlog 7.0 Professional Edition: Internet Programming Guide. Technical report, BinNet Corp., 1998. Available from http://www.binnetcorp.com/BinProlog.

[18] P. Tarau. Inference and Computation Mobility with Jinni. In K. Apt, V. Marek, and M. Truszczynski, editors, *The Logic Programming Paradigm: a 25 Year Perspective*, pages 33–48. Springer, 1999. ISBN 3-540-65463-1.

[19] P. Tarau and V. Dahl. High-Level Networking with Mobile Code and First Order AND-Continuations. *Theory and Practice of Logic Programming*, 1(1), Jan. 2001. to appear at Cambridge University Press.

[20] P. Tarau, V. Dahl, and A. Fall. Backtrackable State with Linear Affine Implication and Assumption Grammars. In J. Jaffar and R. H. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security*, Lecture Notes in Computer Science 1179, pages 53–64, Singapore, Dec. 1996. Springer.

[21] P. Tarau, K. De Boschere, V. Dahl, and S. Rochefort. LogiMOO: an Extensible Multi-User Virtual World with Natural Language Control. *Journal of Logic Programming*, 38(3):331–353, Mar. 1999.

[22] The External Authoring Interface Workgroup. VRML-EAI Specification. Technical report, 1999. available from http://www.vrml.org/WorkingGroups/vrml-eai/index.html.

[23] P. Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, pages 1–17, 1993.

Multicast Protocols for Jinni Agents

Satyam Tyagi and Paul Tarau

Department of Computer Science
University of North Texas
P.O. Box 311366
Denton, Texas 76203
E-mail: tarau@cs.unt.edu

Keywords: *Mobile Computations, Intelligent Mobile Agents, Logic Programming, Blackboard based Coordination, Networking Software Infrastructure, Multicast, Java3D based Shared Virtual Reality, IP Transparent Communication*

Abstract

We extend the Jinni Agent Programming Infrastructure with a multicast network transport layer. We show that the resulting protocol emulates client/server exchanges while providing high performance multicasting of remote calls to multiple sites. Jinni's orthogonal design, separating blackboards from networking and multi-threading, turns out to be easy to adapt to support a generalization of Jinni's Linda based coordination model. The resulting system is particularly well suited for building large scale, agent based, IP transparent, fault tolerant, tele-teaching and shared virtual reality applications.

1 Introduction

The advent of networked, mobile, ubiquitous computing has brought a number of challenges, which require new ways to deal with increasingly complex patterns of interaction: autonomous, reactive and mobile computational entities are needed to take care of unforeseen problems, to optimize the flow of communication, to offer a simplified, and personalized view to end users. These requirements naturally lead towards the need for *agent programs* with increasingly sophisticated inference capabilities, as well as autonomy and self-reliance. To host them with minimal interference to their main mission, we need a software infrastructure providing a minimal basic ontology - ideally as simple and self contained as the IP packet universe on top of which the Internet is built.

Conveniently encapsulated as *agents*, software artifacts enabled with autonomy, dynamic knowledge exchange and network transparent mobility (as key features) have emerged.

An important number of early software agent applications are described in [3] and, in the context of new generation networking software, in [31,14].

Mobile code/mobile computation technologies are pioneered by General Magic's Telescript (see [17] for their Java based *mobile agent* product) and IBM's Java based Aglets [20]. Other mobile agent and mobile object related work illustrate the rapid growth of the field: [25,18,19,21,32,33,26]

Implementation technologies for mobile code are studied in [1]. Early work on the Linda coordination framework [7,8,4] has shown its potential for coordination of multi-agent systems. The logical modeling and planning aspects of computational Multi-Agent systems have been pioneered by [13,11,22,34,10,12,23,9].

Jinni 2000 is a multi-threaded Java based Prolog system with modular, plugin networking layer, designed as an agent programming library, through a combination of Java and Prolog components. It supports mobile agents either directly through a mobile thread abstraction or as a combination of remote predicate calls, client/server components, multiple networking layers and blackboards.

1.1 Overview of the paper

The paper is divided into 8 sections. Section 2 describes the Jinni's Mobile Agent Architecture briefly and some of Jinni's functionalities and advantages. Section 3 introduces us to the recently developed Multicast layer for Jinni and some of its advantages. Section 4 outlines the Synchronization of Multicast Agents with Blackboards. Section 5 explores some important properties of multicast networks, which are significant for Jinni in particular, and mobile agents and mobile computing in general. In section 6 we describe two applications for Multicast Agents using the Jinni Agent Programming Infrastructure. Section 7 discusses some of the problems and related areas to be explored in future. Finally we conclude with section 8 stating current achievements, ongoing applications and new possibilities.

2 The Jinni Architecture

2.1 Ontology

Jinni is based on simple **Things, Places, Agents** ontology.

Things are Prolog terms (trees containing constants and variables, which can be unified and other compound sub-terms).

Places are processes with at least one server and a blackboard allowing synchronized multi-user Linda and Remote Predicate Call transactions. The blackboard stores Prolog terms, which can be retrieved by agents.

Agents are collections of threads executing various goals at various places. Each thread is mobile, may visit multiple places and may bring back results.

2.2 Basic Features

Jinni is a Prolog interpreter written in Java, which provides an infrastructure for mobile logic programming (Prolog) based agents. It spawns interpreters as threads over various network sites and each interpreter has its own state. Computation mobility is mapped to data mobility (through use of meta-interpreters, data can be treated as code). Mobile threads can capture first order "AND"-continuations (as "OR" continuations would cause backtracking, which is not a good idea over the network) and resume execution at remote site by fetching code as needed.

Shared blackboards are used for communication and coordination of agents. Jinni has an orthogonal design and separates high level networking operations (supporting remote predicate calls and code mobility), from Linda coordination code. It has various plugins for GUI, different Network layers (in particular the multicast layer described in this paper) and a Java3d interface, which can be plugged in as an extension. Jinni 2000 embeds a fast incremental compiler, which provides Prolog processing within a factor of 5-10 from the fastest C-based implementations around.

For more details on Jinni see [27,28].

2.3 Data, Code and Computation Mobility

While *data* and *code* mobility present no challenge in a Java environment, migrating the state of the computation from one machine or process to another still requires a separate set of tools. Java's remote method invocations (RMI) add transparent control mobility and a (partially) automated form of *object mobility* i.e. integrated code (class) and data (state) mobility.

Mobility of live code is called *computation mobility* [5]. It requires interrupting execution, moving the state of a runtime system (stacks, for instance) from one site to another and then resuming execution. Clearly, for some languages, this can be hard or completely impossible to achieve (C/C++) while in other languages like Java it still requires class specific serialization methods (providing writing and reading of objects to/from byte streams).

Conventional mobile code systems like IBM's Aglets [20] require serialization hints from the programmer and do not implement a fully generic reflective computation mobility infrastructure. Aglets do not provide code mobility as they assume that code is already available at the destination site. In practice this means that the mobile code/mobile computation layer is not really transparent to the programmer.

In contrast, our architecture is based on building an autonomous layer consisting of a reflective interpreter, which provides the equivalent of implicit serialization and supports orthogonal transport mechanisms for data, code and computation state. The key idea is simply that by introducing interpreters spawned as threads by a server at each networked site, *computation mobility* at object-level is mapped to *data mobility* at meta-level in a very straightforward way. A nice consequence is transport independence coming from the uniform representation of data, code and computation state (in practice this means that Corba, RMI, HLA or plain/multicast sockets can be used interchangeably as a transport mechanism).

2.4 Advantages of Computation Mobility

Current agent programming infrastructures use message passing as the key communication mechanism. Existing Concurrent Constraint Logic Programming languages (and multi-paradigm languages like Distributed Oz) support distributed programming and coordination through monotonic stores and shared logical variables.

By contrast, we want to explore the impact of mobile live computations, lightweight multi-engine/multi-threaded script interpreters, blackboard constraints and multicast based coordination on building mobile multi-agent systems. Our distributed Linda blackboards generalize concurrent constraint programming stores by allowing non-monotonic updates of assertional constraints [28]. With our multicast layer we implement transparently replication of shared blackboards distributed over different places.

We refer to [27,15] for more detailed discussion on the advantages of computation mobility.

3 A Multicast Layer for Jinni

3.1 Multicast

Definition: 1. In a network, a technique that allows data, including packet form, to be simultaneously transmitted to a selected set of destinations. Some networks, such as Ethernet, support multicast by allowing a network interface to belong to one or more multicast groups.

2. To transmit identical data simultaneously to a selected set of destinations in a network, usually without obtaining acknowledgment of receipt of the transmission. [30].

The key concepts of Multicast Networks are described in [16]. On an Ethernet (Most popular LAN architecture) each message is broadcasted and the machine seeing its own address grabs the message. The multicast packets are also sent in a similar way except that more than one interface picks them up.

The multicast packets received at the interface but not subscribed to are rejected at interface level, if the interface has hardware and software to support it. Interfaces are able to handle often 64 but up-to 512 groups. On overloading interfaces go into “multicast promiscuous” mode and send all packets to the TCP/IP stack. Here after searching through the list of groups a host is subscribed to packets are accepted or rejected. This requires some CPU cycles [24].

We consider 512 is a pretty significant number presently and even if it does go into the "multicast promiscuous" mode the penalty is insignificant. Thus the spreading and cloning of agent threads/agents themselves on the whole network or a subset multicast group is now a single step operation. This leads to important speed up especially when multiple copies of same code need to be executed at different places (like for parallel searches or for simultaneous display in shared virtual reality applications).

3.2 Multicast in Jinni

Multicasting has various interesting properties, which make it well suited for an agent platform like Jinni. An important advantage of multicasting agents is that, the same code can be run in parallel at the same time in one single operation at different remote sites, retrieving different data available at different sites.

3.2.1 The API

A minimal API consists of two predicates one to run multicast servers, which service requests for multicast clients and second to send multicast requests to these servers:

run_mul_server This joins a multicast group with an address and port. The server now is listening on this port and can receive remote requests for local execution. (When we join a group we are telling the kernel, "I am interested in this multicast group. So, deliver (to any process interested in them, not only to me) any datagram that you see in this network interface with this multicast group in its destination field.")

run_server, which was there for unicast servers is extended to
run_mul_server defaults to **Host = 224.1.1.1, Port = 7001**
run_mul_server(Port) defaults to **Host = 224.1.1.1**
run_mul_server(Host, Port)

Notice that **run_mul_server** has a Host field as well, because it does not run on the local IP but on a multicast address i.e. 224.x.x.x 239.x.x.x

remote_mul_run(G) This is kind of a goal (G), which is multi-casted to the group to be run remotely on all the multicast servers accepting requests for this group.

remote_run(G), which was there for unicast requests is extended to
remote_mul_run(G) defaults to **Host = 224.1.1.1, Port = 7001**
remote_mul_run(Host, Port, X, G, W, R) X:Answer Pattern, G:Goal, W:Password, R:Returned Result

A set of multicast servers is run, which listen for packets on their group and respond back on the group multicast address. All the clients listening on this group receive these results. An important issue here is that the server should be able to distinguish between a request and a reply. Otherwise it would keep responding back to its own replies. This is solved by introducing a simple header distinguishing the two types of messages, which are stripped of when appropriate.

4 Synchronizing Multicast Agents with Blackboard Constraints

The synergy between mobile code and Linda coordination allows an elegant, component wise implementation. *Blackboard operations are implemented only between local threads and their (shared) local blackboard.* If interaction with remote blackboard is needed, the thread simply moves to the place where it is located and proceeds through local interaction. The interesting thing with multicast is that the thread can be multi-casted to a set of places and can interact at all these places locally. This gives an appearance that all these blackboards are one to the members of this multicast groups. For example the **remote_mul_run(mul_all(a(X),Xs))** operation is multi-casted to all servers in the group. It collects lists of matching results at these remote servers and the output is unicast-ed from all these remote sites to the local blackboard.

This can be achieved as follows:

mul_all(X,Xs):-mul_all('localhost',7001,X,Xs).

mul_all(Port,X,Xs):-mul_all('localhost',Port,X,Xs). - (the defaults where our server, which receives the results is running)

mul_all(Host,Port,X,Xs):-all(X,Xs), - executes on remote servers.
remote_run(Host,Port,forall(member(Y,Xs),out(Y))) - executes back on our server.

Host and Port determine the address we want the answers to come back on and the answers are written on the local blackboard from where they can be collected.

Note: in/1 and out/1 are basic Linda blackboard predicates[7]

Places are melted into peer-to-peer network layer forming a 'web of interconnected worlds'. Now different worlds can be grouped into multicast groups. The member of multicast groups viewed as these groups as one world. The concept ex-

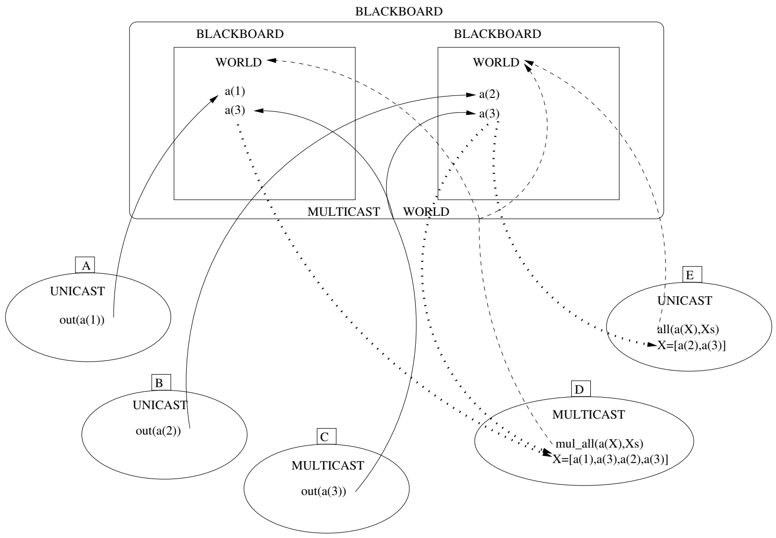


Fig. 1. Basic Linda operations with Multicast

tends further as groups can intersect and be subsets or supersets and basically have combination of unicast and multicast worlds. The point is that each agents views the world it is interacting with depends on how it is moving its threads to different places.

In the Fig. 1 A, B unicast outputs to two different worlds while C multicasts output to both. The 'unicast all' in E is able to collect from only one blackboard while 'multicast all' in D collects from both. The **all** operations collect multiple copies of the term **a(3)**, but a sort operation could remove such duplication.

5 Some Properties and Consequences

There are various interesting properties of multicast networks, which open up many possibilities for future work especially regarding mobile agents and new architectures for their implementation.

As previously discussed there are three types of mobility in a network software environment:*data mobility, code mobility and computation or thread mobility*. An important shortcoming of computation mobility was that if the thread was providing a service or listening on a particular **(host, port)** it could no longer do so once it moved. In other words, ports are not mobile.

Some properties of multicast addresses and ports overcome exactly this shortcoming. These properties are:

- multicast address and port are same for a multicast group and are independent of host or IP address of the host(*IP Transparent*)
- it is possible to have two servers with same multicast address and port running on the same machine. (*In other words we do not need to investigate if a server with same port and IP is already running.*) Both servers can respond to the request, a

client can chose if it wants one or all answers. Also a header can be put, which helps servers discard requests not intended for them.

This means that when live code or a thread migrates it can just does a *joingroup* [1] on the same group it belonged to and start listening or providing service on the same multicast address and port.

5.1 Impact on mobile Computers and transient IP-address systems

A mobile computer like a laptop, palmtop etc. does not have a permanent IP-address because one may connect to one's office, home, in an airplane etc. The transient IP address can also come from one connecting through a dialup connection to an ISP. Such systems can launch mobile agents and receive results when connected and hence can be clients [15].

An important impact of the proposed multicast agent architecture on such transient IP-address systems is that they can also *provide a service* and *listen* on a known multicast address and port whenever they are connected to the Internet. This is possible because to listen on a multicast port one's IP address is not needed. One can have any IP address and still listen on the same Multicast address and port.

Another concept in the Jinni architecture is that of **mobile Linda servants**[29]. A **servant** is an agent, which is launched and on reaching the destination can pull commands from the launching site or other clients, and run them locally.

```
servant:-
  in(todo(Task)),
  call(Task),
  servant.
```

Note that the servant is a background thread and blocks when none of the clients in the group have a task to be done i.e. no 'busy wait' is involved [29].

We will try to expand on these two concepts of multicast and servants to generalize the client/server architecture especially for mobile and transient IP-address systems.

Case 1 Mobile Servers

Even when the mobile server is disconnected it can have **servant agents** running on it, doing the processing for its clients and posting results or queries on the local blackboard. In the mean time, the clients keep making lists of the tasks they want to get done on the server. When the server comes up, the servant can pull the list of tasks to be done by clients and run them. Also the server can have a new IP address but the same multicast address, when the server reconnects. The clients having knowledge of this can collect the required responses from the servers' blackboard.

[1] When we join a group we are telling the kernel, "I am interested in this multicast group. So, deliver (to any process interested in them, not only to me) any datagram that you see in this network interface with this multicast group in its destination field."

Case 2 Mobile Clients
Even when disconnected, the mobile client can launch an agent on a different machine, which can do the listening for it. Whenever the client reconnects it can pull list of tasks and results (*the agents do processing*) from the agent and destroy the agent. Whenever the client is disconnecting it can launch the agent again.

This concept can also be extended, as both clients and servers can be made mobile or with unknown or transient IP-addresses with multicasting. As we discussed before, to communicate on a multicast channel we do not need to know the IP. We explore this concept of IP transparent communication further in the next subsection. Some ideas of this mixed mobility of computers and agents are discussed in [6].

Applets as servers
Another possible use can be in the case of applets, which can not listen on ports but can connect to the server they are coming from. This means they can connect to their server pull tasks from it and execute them locally on their current machine (*hence pretend to be servers*).

This architecture could possibly be three-tier. The applets connect to their server, which offers services to other clients. The server now posts the requests on the local blackboard to be pulled by the applets and executed locally on their machine.

5.2 IP Transparent Architecture for a Platform of Mobile Agents.

Consider a group of agents moving freely in the network. Let's assume each agent is a member of two multicast groups: a common shared group address between all agents and a unique personal multicast address. Whenever they propagate they do a *joingroup* on both these multicast groups.

The analogy for private address is that of a cell phone. Each agent can communicate with the others on its private multicast address **being completely unaware about the location of one it is sending messages to.**

The best analogy for the shared common address is that of a broadcast radio channel. Whenever an existing agent spawns a new agent it gives the new agent the set of addresses known to it and the new agent chooses a new private multicast address and communicates to the rest of the agents (via the shared common group address) its private address. Metaphorically, this is like (**broadcasting on the radio channel its cell phone number to communicate with it**)

The main advantage of this architecture is that it carries out communication amongst different agents without any knowledge of each others current location, i.e. no agent requires the knowledge of other's IP address to communicate whether they want the communication to be public within a group or private.

Among the application of a such an architecture could be found in situations where the agents need to communicate with each other but do not have a fixed itinerary or the itinerary changes with the decision the agent makes. The address need not be communicated on each hop but only when a new agent is spawned it needs to make its address known.

For now, the important question of lost messages during transportation of agents remains unanswered. One must not forget multicast is based on UDP and messages can be lost. However, real-time applications like video games, which consider late messages as lost messages could be target applications. Also one of the reliable multicast protocols [2] may be used.

5.3 Fault tolerant computing

Here we suggest a protocol, which tries to make sure that even when some agents crash the number of agents is preserved.

Consider the above architecture with each agent having **k** (**k** is a small number greater than 1 say 2 or 3) copies and its own number. Each agent issues a heart beat message on its private channel with its number with a fixed beat time interval (T_{beat}). The agents will have an expiry timeout in case the beat is not received for a particular amount of time from a particular agent (T_{exp}). The agent with the next number in cyclic [**(n+1) mod k**] order generates a new agent with **id** number **n** on expiry of timer (T_{exp}).

Although, each agent is listening for only the previous agent in the cyclic order but even if one agent is left the whole agent group will grow back again. Consider a group of three agents **1**, **2** and **3**. If any two die let's say **2** and **3** then **1** will spawn **3**, which will in turn eventually spawn **2** and hence the group will grow back again.

This makes sure with a good probability that we always have approximately **k** agents on each private channel.

6 Applications

We will now overview a number of applications likely to benefit from a multicast agent infrastructure.

6.1 Tele-teaching

A set of intelligent agents on student machines, join the multicast group of a teaching server (**run_mul_server**).

The agents can always be given required information or 'todo' tasks from the server as needed on the multicast channel (**remote_mul_run(out(a(X)))**).

The server can collect responses posted on the local blackboards by the agents with the extended blackboard concept (**remote_mul_run(mul_all(a(X)))**) .

The application is more suited to present circumstances as most routers are incapable of multicast. It is however easy to ensure that classrooms are on a single

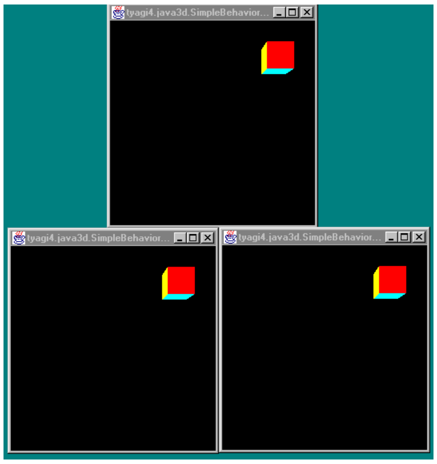


Fig. 2. *Synchronized Java3d worlds using multicast.*

LAN capable of multicast. The main advantage here is that even though the system is interactive the model is not *message based - query/response*. The agents are reactive and intelligent and the responses/queries are posted on the local blackboard from which the server can collect periodically or be informed to collect after a certain time. The model is flexible and can be extended and made more flexible by adding unicast channels and subset multicast groups for teamwork in students.

6.2 Java3D based Shared Virtual Reality

The three concepts of *intelligent logic programmable agents*, *multicast single step synchronization* and *Java3D visualization* provide an interesting synergy for game programming. We will now explore an implementation architecture we have prototyped on Jinni 2000's multicast layer.

6.2.1 The User's interface

The user interface is based on shared Java3D virtual worlds.
Each user can join at anytime by joining a given multicast group.
Each user can create and own objects, which he/she/it can manipulate.
The user is opaque to the knowledge if he/she/it is playing against/with another user or an intelligent agent.

6.2.2 The implementation

The main advantage we have in our implementation is that there is no centralized server. The application is completely distributed. If one user's machine goes down only the objects controlled by him/her go down. This is achieved by having the state being multi-casted to all users and stored only on the local blackboards from where it is to be collected when a new user logs in. The next subsection describes a basic Java3D API on which the virtual world is built and the interface is provided.

6.2.3 The basic API

java3d_init initializes and opens Java3d window and joins the multicast group and collects (**remote_mul_run(mul_all(a(X)))**) current state from the blackboards of other users.
new_obj creates a new object and puts the state on local blackboard (**out(a(X))**) and multicasts the predicate call to currently subscribed users.
move_obj moves the object if owned and modifies (**in(a(X)),out(a(Y))**) the state on local blackboard and multicasts the predicate call to currently subscribed users.
remove_obj removes the current object clears entry (**in(a(X))**) from local blackboard and multicasts the change to currently subscribed users.

The blackboards preserve the current state. The multicasting makes sure all updates are single step. The agent scripts are written in Prolog and the visualization is based on Java3D. The logic of agents can be changed and different agents can have different personalities as per the learning logic, algorithm and experience of the agent. The agents generate keyboard and mouse events to play with humans (making them more similar to human interaction).

The Fig. 2 shows three multicast synchronized Java3D worlds, running under our prototype, in three process windows. In a real game they are distributed over the network.

7 Some Problems and Future work

There are some inherent problems with multicast. The protocol is UDP based (probably because it is not a great idea for each receiver to send an acknowledgment to each sender and flood the network). Also one can never know how many users are currently subscribed to a group. This makes blocking reads (**in(a(X))**) impossible, as we do not know how many responses to loop for. Currently we have implemented multicast **outs**, which do not require responses and non blocking multicast reads (**mul_all**), which collect responses from remote sites and respond on a unicast channel. Some possible scenarios for experimentation would be first matching response or first (k) matching response. Also currently multicast application remains untested on the Internet, as we are confined to the Ethernet LAN. With the new generation routers capable of multicast, it would be interesting to test the applications and protocols over larger domains. The unreliability in the protocol makes it unsuitable for some applications. Our multicast agents work well in real time applications for which a delayed packet is equivalent to a lost packet. Some future work would depend on implementation of reliable multicast protocols and its impact assuming that Internet routers will become more and more multicast aware.

8 Conclusion

We have outlined here an extension of Jinni with a transport layer using multicast sockets. We have also shown some interesting properties of multicast, which have opened various new possibilities for mobile agents and mobile computers. We are currently working on certain applications, which we have shown can be greatly simplified, speeded up and improved with multicast extended version of Jinni. We suppose that the possibilities and applications we have shown here is only a starting point for an unusual niche for Logic Programming based software tools. The spread of multicast technology from simple LANs to the complete Internet and the development of reliable protocols for multicast [2] will necessitate further exploration, to achieve greater insights on mobile agent technology and realize its full potential and possible impact.

9 Acknowledgments

We thank the anonymous referees for their valuable comments and suggestions.

References

[1] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and Language-independent Mobile Programs. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 127–136, Philadelphia, Pa., May 1996.

[2] K. P. Birman. Reliable Multicast Goes Mainstream. Technical report, Dept. of Computer Science, Cornell University, 1999. Available from http://www.cs.odu.edu/ mukka/tcos/e_bulletin/vol9no2/birman.html.

[3] J. Bradshaw, editor. *Software Agents*. AAAI Press/MIT Press, Menlo Park, Cal., 1996.

[4] A. Brogi and P. Ciancarini. The Concurrent Language, Shared Prolog. *ACM Trans. Prog. Lang. Syst.*, 13(1):99–123, 1991.

[5] L. Cardelli. Mobile Computation. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems - Towards the Programmable Internet*, pages 3–6. Springer-Verlag, LNCS 1228, 1997.

[6] L. Cardelli. Abstractions for Mobile Computation. Technical report, Microsoft Research, Apr. 1999.

[7] N. Carriero and D. Gelernter. Linda in Context. *CACM*, 32(4):444–458, 1989.

[8] S. Castellani and P. Ciancarini. Enhancing Coordination and Modularity Mechanisms for a Language with Objects-as-Multisets. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061 of *LNCS*, pages 89–106, Cesena, Italy, April 1996. Springer.

[9] B. Chaib-draa and P. Levesque. Hierarchical Models and Communication in Multi-Agent Environments. In *Proceedings of the Sixth European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW-94)*, pages 119–134, Odense, Denmark, Aug. 1994.

[10] P. R. Cohen and A. Cheyer. An Open Agent Architecture. In O. Etzioni, editor, *Software Agents — Papers from the 1994 Spring Symposium (Technical Report SS–94–03)*, pages 1–8. AAAIP, Mar. 1994.

[11] P. R. Cohen, M. L. Greenberg, D. M. Hart, and A. E. Howe. Trial by Fire: Understanding the Design Requirements for Agents in Complex Environments. *AI Magazine*, 10(3):32–48, 1989.

[12] P. R. Cohen and H. J. Levesque. Communicative Actions for Artificial Agents. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 65–72, San Francisco, CA, June 1995.

[13] P. R. Cohen and C. R. Perrault. Elements of a Plan Based Theory of Speech Acts. *Cognitive Science*, 3:177–212, 1979.

[14] G. Cugola, C. Ghezzi, G. P. Picco, and G. Vigna. A characterization of mobility and state distribution in mobile code languages. In *2nd ECOOP Workshop on Mobile Object Systems*, pages 10–19, Linz, Austria, July 1996.

[15] A. K. David Chess, Colin Harrison. Mobile agents: Are they a good idea? Technical report, IBM Research Division, T. J. Watson Research Center, 1995.

[16] S. Deering. *Multicast routing in a datagram internetwork*. PhD thesis, Stanford University, Dec. 1991.

[17] GeneralMagicInc. Odissey. Technical report, 1997. available at http://www.genmagic.com/agents.

[18] R. S. Gray. Agent tcl: A flexible and secure mobile agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop*, pages 9–23, July 1996. http://www.cs.dartmouth.edu/ agent/papers/tcl96.ps.Z.

[19] R. S. Gray. Agent tcl: A flexible and secure mobile-agent system. Technical Report PCS-TR98-327, Dartmouth College, Computer Science, Hanover, NH, Jan. 1998. Ph.D. Thesis, June 1997.

[20] IBM. Aglets. Technical report, 1999. http://www.trl.ibm.co.jp/aglets.

[21] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. Agent TCL: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, July/August 1997.

[22] R. Kowalski and J.-S. Kim. A Metalogic Programming Approach to Multi-Agent Knowledge and Belief. In V. Lifschitz, editor, *AI and Mathematical Theory of Computation: Papers in Honour of John McCarthy*. Academic Press, 1991.

[23] Y. Lésperance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. Foundations of a Logical Approach to Agent Programming. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II (LNAI 1037)*, pages 331–346. Springer-Verlag: Heidelberg, Germany, 1996.

[24] D. Mcgregor. CPU Performance and Multicast. Technical report, Naval Postgraduate School Computer Science Department. Available from http://www.stl.nps.navy.mil/ mcgredo/projectNotebook/mcast/ethernet.html.

[25] B. Steensbaard and E. Jul. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 68–78, Copper Moutain, Co., Dec. 1995.

[26] M. Straßer, J. Baumann, and F. Hohl. Mole – a Java based mobile agent system. In *2nd ECOOP Workshop on Mobile Object Systems*, pages 28–35, Linz, Austria, July 1996.

[27] P. Tarau. A Logic Programming Based Software Architecture for Reactive Intelligent Mobile Agents. In P. Van Roy and P. Tarau, editors, *Proceedings of DIPLCL'99*, Las Cruces, NM, Nov. 1999. http://www.binnetcorp.com/wshops/ICLP99DistInetWshop.html.

[28] P. Tarau. Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents*, pages 109–123, London, U.K., 1999.

[29] P. Tarau and V. Dahl. High-Level Networking with Mobile Code and First Order AND-Continuations. *Theory and Practice of Logic Programming*, 1(1), Jan. 2001. to appear at Cambridge University Press.

[30] N. C. S. Technology and S. Division. Telecommunications:Glossary of TeleCommunication terms:Federal Standard 1037C. Technical report, General Service Administration Information Technology Sevice, Aug. 1996. Available from http://www.its.bldrdoc.gov/fs-1037/dir-023/_3404.htm.

[31] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), Apr. 1996.

[32] D. E. White. A comparison of mobile agent migration mechanisms. Senior Honors Thesis, Dartmouth College, June 1998.

[33] J. E. White. Telescript technology: Mobile agents. In Bradshaw [3]. Also available as General Magic White Paper.

[34] M. Wooldridge. *The Logical Modelling of Computational Multi-Agent Systems*. PhD thesis, Department of Computation, UMIST, Manchester, UK, Oct. 1992. (Also available as Technical Report MMU–DOC–94–01, Department of Computing, Manchester Metropolitan University, Chester St., Manchester, UK).

Implementing cTI: a constraint-based left-termination inference tool for LP

S. Hoarau

IREMIA
Université de La Réunion, FRANCE
Email: `seb@univ-reunion.fr`

F. Mesnard

IREMIA
Université de La Réunion, FRANCE
Email: `fred@univ-reunion.fr`

U. Neumerkel

Institut für Computersprachen
Technische Universität Wien, AUSTRIA
Email: `ulrich@complang.tuwien.ac.at`

Abstract

We present the implementation of cTI, a system for bottom-up termination inference. Termination inference is a generalization of termination analysis/checking. The architecture of cTI is presented and some optimizations and performance results for medium-sized programs are discussed.

1 Introduction

Termination is a crucial aspect of program verification. For logic programs, the problem is of particular importance because there is *a priori* no syntactic restriction on queries. Termination has been the subject of many works in the last fifteen years in the logic programming community [31,2,26]. The research efforts can be divided in two groups (a survey is given in [17]). The first group characterizes termination (e.g. [1]) with undecidable criteria. The second weakens those criteria to obtain computable sufficient conditions (e.g. [32]). Our approach belongs to the latter stream. Our main innovation compared to other works of termination analysis (e.g.

[32,21,14,7,13]) is to provide a framework where we can *infer* sufficient universal termination conditions from the text of any Prolog program. Inference means that we adopt a bottom-up approach to termination. *There is no need to define a class of queries of interest.* All other works we are aware of require to add such a class. Moreover, such classes can be easily simulated within our framework.

Currently the only requirement we impose on programs is that they must not create infinite rational terms. Hence we only consider NSTO programs [16,15] that can be safely executed with any standard complying system or an execution with occurs check. Our system, called cTI, has been realized with SICStus Prolog and can be used at http://www.complang.tuwien.ac.at/ctihttp://www.complang.tuwien.ac.at/cti. cTI is also integrated in the LP environment GUPU [25]. A formal justification of the algorithms underlying cTI can be found in [23] while [5] is a more conceptual paper, describing applications of termination inference and works in progress. In this paper, we present some insights, optimizations, and run times for three main algorithms that lie at the heart of cTI.

2 An Overview of cTI

Syntactic informations are often too weak to reason about non-trivial programs. Some semantics informations is required. For this reason our analyzer uses three main constraint structures [20]: Herbrand terms (CLP($\mathcal{H}$)) for the initial program P, non-negative integers (CLP($\mathbb{N}$)) and booleans (CLP($\mathcal{B}$)) for approximating P. The correspondence between these structures relies on *approximations* [23], which are a simple form [18] of abstract interpretation [9,10]. We present our method to infer termination conditions by using the predicates app/3, app3/4, and nrev/3.

app([], Xs, Xs).
app([X|Xs], Ys, [X|Zs]) ←
    app(Xs, Ys, Zs).
	nrev([], []).
	nrev([X|Xs], Ys) ←
	    nrev(Xs, Zs),
	    app(Zs, [X], Ys).
		app3(Xs, Ys, Zs, Us) ←
		    app(Xs, Ys, Vs),
		    app(Vs, Zs, Us).

(i) The initial Prolog program P is mapped to $P^{\mathbb{N}}$, a program in CLP($\mathbb{N}$) using an approximation based on a symbolic norm. In our example, we use the term-size norm:

$$\|t\|_{\mathrm{Term-Size}} = \begin{cases} 1 + \sum_{i=1}^{n} \|t_i\|_{\mathrm{Term-Size}} & \text{if } t = f(t_1, \ldots, t_n), n > 0 \\ 0 & \text{if } t \text{ is a constant} \\ t & \text{if } t \text{ is a variable} \end{cases}$$

E.g.$\|f(0,0)\|_{\mathrm{Term-Size}} = 1$. All non-monotonic elements of the program are approximated by monotone constructs. E.g., Prolog's unsound negation \+X is approximated by X;true. It is maintained that if a goal in $P^{\mathbb{N}}$ is terminating, then also the corresponding goals in P terminate.

$\mathrm{app}_{\mathbb{N}}(0, \mathrm{Xs}, \mathrm{Xs}).$ $\mathrm{app}_{\mathbb{N}}(1{+}\mathrm{X}{+}\mathrm{Xs},\mathrm{Ys},1{+}\mathrm{X}{+}\mathrm{Zs}) \leftarrow$ $\quad\mathrm{app}_{\mathbb{N}}(\mathrm{Xs}, \mathrm{Ys}, \mathrm{Zs}).$	$\mathrm{nrev}_{\mathbb{N}}(0, 0).$ $\mathrm{nrev}_{\mathbb{N}}(1{+}\mathrm{X}{+}\mathrm{Xs},\mathrm{Ys}) \leftarrow$ $\quad\mathrm{nrev}_{\mathbb{N}}(\mathrm{Xs}, \mathrm{Zs}),$ $\quad\mathrm{app}_{\mathbb{N}}(\mathrm{Zs}, 1{+}\mathrm{X}, \mathrm{Ys}).$	$\mathrm{app3}_{\mathbb{N}}/4$ same as app3/4

(ii) In $\mathbb{N}$ we compute a model of all predicates. The model describes with a finite conjunction of linear equalities and inequalities the relations between the arguments of a goal (inter-argument relations IR) that hold for every solution. The actual computation is performed with CLP($\mathbb{Q}$). In our example we are able to determine the least model. In general, however, only a less precise model is determined. For recursive predicates, the actual source of potential non-termination, we compute linear strictly decreasing level mappings (see section 4) called μ's. For instance, the meaning of $\mu^{\mathbb{N}}_{\mathrm{app}}$ is: for each recursive call to app/3, the first and the third argument decrease.

(least) models	level mappings
$\mathrm{IR}^{\mathbb{N}}_{\mathrm{app}}(x,y,z) \equiv z = x+y$	${\mu^{\mathbb{N}}_{\mathrm{app}}}^1(x,y,z) \equiv x$
	${\mu^{\mathbb{N}}_{\mathrm{app}}}^2(x,y,z) \equiv z$
$\mathrm{IR}^{\mathbb{N}}_{\mathrm{nrev}}(x,y) \equiv x = y$	$\mu^{\mathbb{N}}_{\mathrm{nrev}}(x,y) \equiv x$
$\mathrm{IR}^{\mathbb{N}}_{\mathrm{app3}}(x,y,z,u) \equiv u = x+y+z$	—

(iii) $P^{\mathbb{N}}$ is mapped to $P^{\mathcal{B}}$, a program in CLP($\mathcal{B}$). Here 1 means that an argument is bound w.r.t. the considered norm. Note that the obtained program no longer maintains the same termination property. Its sole purpose is to determine the actual dependencies of boundedness within the program. The simplified structure permits us to compute always the least model. In a single boolean term all previously computed linear level mappings are represented.

$\mathrm{app}_{\mathcal{B}}(1, \mathrm{Xs}, \mathrm{Xs}).$ $\mathrm{app}_{\mathcal{B}}(1{\wedge}\mathrm{X}{\wedge}\mathrm{Xs},\mathrm{Ys},1{\wedge}\mathrm{X}{\wedge}\mathrm{Zs}) \leftarrow$ $\quad\mathrm{app}_{\mathcal{B}}(\mathrm{Xs}, \mathrm{Ys}, \mathrm{Zs}).$	$\mathrm{nrev}_{\mathcal{B}}(1, 1).$ $\mathrm{nrev}_{\mathcal{B}}(1{\wedge}\mathrm{X}{\wedge}\mathrm{Xs},\mathrm{Ys}) \leftarrow$ $\quad\mathrm{nrev}_{\mathcal{B}}(\mathrm{Xs}, \mathrm{Zs}),$ $\quad\mathrm{app}_{\mathcal{B}}(\mathrm{Zs}, 1{\wedge}\mathrm{X}, \mathrm{Ys}).$	$\mathrm{app3}_{\mathcal{B}}/4$ same as app3/4

least models	level mappings
$\mathrm{IR}^{\mathcal{B}}_{\mathrm{app}}(x,y,z) \equiv (x \wedge y) \Leftrightarrow z$	$\mu^{\mathcal{B}}_{\mathrm{app}}(x,y,z) \equiv x \vee z$
$\mathrm{IR}^{\mathcal{B}}_{\mathrm{nrev}}(x,y) \equiv x \Leftrightarrow y$	$\mu^{\mathcal{B}}_{\mathrm{nrev}}(x,y) \equiv x$
$\mathrm{IR}^{\mathcal{B}}_{\mathrm{app3}}(x,y,z,u) \equiv (x \wedge y \wedge z) \Leftrightarrow u$	—

The meaning of the boolean model is: when any call app(X,Y,Z) terminates sucessfully, then Z is finite iff X and Y are finite and, for nrev/2, when any proof of a call nrev(X,Y) terminates, X is finite iff Y is finite w.r.t. the used norm. Now, if we know (i.e. using level-mappings) that a call to app/3 terminates, then there is a finite number of successes and we may ensure the third argument is *bound* iff the two first arguments are *bound* w.r.t. the norm.

(iv) Finally, using all previously determined informations, $P^{\mathcal{B}}$ is translated into a system of boolean formulæ(see theorem 2.1) that ensures the propagation of the decreasing conditions of the level mappings through the call graph. The resolution of this system (computation of its greatest fixpoint by means of a boolean μ-solver [8]) gives, for each predicate symbol, a boolean term called a *termination condition*.

$$\begin{aligned} Pre_{\text{app}}(x,y,z) &\equiv x \vee z \\ Pre_{\text{nrev}}(x,y) &\equiv x \\ Pre_{\text{app3}}(x,y,z,u) &\equiv (x \wedge y) \vee (x \wedge u) \end{aligned}$$

So any call ← C, app(X,Y,Z), where C is a CLP($\mathcal{H}$) constraint, left-terminates if X or Z are *ground* in C etc. The main result on which cTI is based on is the following Theorem 2.1, which ensures correctness of the left-termination condition.

Theorem 2.1 ([22]) *Let P be a program, p and q be two predicate symbols of P. Assume that p is defined by m_p rules r_k: $p(\tilde{x}) \leftarrow c_k,\ p_{k,1}(\tilde{x}_{k,1}), \ldots\ p_{k,n_k}(\tilde{x}_{k,n_k})$ and for each $q \notin \bar{p}$ and appearing in the rules defining $\bar{p}$, a left-termination condition Pre_q has been computed. If the set of boolean terms $\{Pre_p\}_{p\in\bar{p}}$ verifies:*

$$\forall p \in \bar{p} \begin{cases} Pre_p(\tilde{x}) \rightarrow_{\mathcal{B}} \mu_p^{\mathcal{B}}(\tilde{x}), \\ \big[\forall 1 \le k \le m_p,\ \forall 1 \le j \le n_k, \\ \qquad \Big(Pre_p(\tilde{x}) \wedge c_k^{\mathcal{B}} \wedge \bigwedge_{i=1}^{j-1} Post_{p_{k,i}}(\tilde{x}_{k,i})\Big) \rightarrow_{\mathcal{B}} Pre_{p_{k,j}}(\tilde{x}_{k,j})\big] \end{cases}$$

then $\{Pre_p\}_{p\in\bar{p}}$ is a left-termination condition for $\bar{p}$.

Running cTI

Table 2 (page 156) presents timings of cTI using some standard benchmarks [1].

We have chosen twelve medium-sized well-known logic programs. Most of them are taken from [4] except CREDIT, PLAN and MINISSAEXP. Table 1 gives the following measures:

- *#facts* and *#rules*: the number of facts (unit clauses) and rules (non-unit clauses) in the program;
- *#cycles*: the number of strongly connected components (sccs, i.e. cycles of mutually recursive predicate symbols) in the call graph;
- *length*: the number of predicate symbols in the longest cycle in the call graph;
- *#vars*: the average arity of the predicate symbols of the longest cycle in the call graph.

[1] collected by Naomi Lindenstrauss, http://www.cs.huji.ac.il/ naomilhttp://www.cs.huji.ac.il/ naomil and also available at http://www.complang.tuwien.ac.at/cti/benchhttp://www.complang.tuwien.ac.at/cti/bench.

Table 1
Informations about analyzed programs.

Program	#facts	#rules	#cycles	length	#vars
ANN	101	99	44	2	4
BID	24	26	20	1	3
BOYER	63	78	25	2	5
BROWSE	4	29	15	1	5
CREDIT	33	24	24	1	2
MINISSAEXP	37	223	100	5	4
PEEPHOLE	72	80	11	2	4
PLAN	12	17	16	1	3
QPLAN	63	87	38	3	8
RDTOK	7	57	12	4	5
READ	15	75	17	7	7
WARPLAN	43	68	33	3	6

The columns of Table 2 indicate the time for TNA analysis (section 5), the ratio of TNA arguments found in the program, the time for computing a model $\mathrm{IR}_{\mathbb{N}}$ (section 3), determining of the level mappings μ (section 4), computing the least model $\mathrm{IR}_{\mathcal{B}}$ (section 3), inferring the left termination condition TC_{l} (see [8]) and the total runtime. The timings given are average execution times over ten iterations.

Table 2
cTI 0.22 on a Pentium II, 300 MHz, 64Mo, SICStus 3.8.3, 3.3 Mlips.

times in [ms] program	TNA	res_TNA	$\mathrm{IR}_{\mathbb{N}}$	μ	$\mathrm{IR}_{\mathcal{B}}$	TC	Total
ANN	113	8%	8790	14183	4936	1270	29292
BID	30	31%	1476	1190	333	206	3235
BOYER	83	14%	9683	1606	820	290	12482
BROWSE	33	2%	1673	3340	450	190	5686
CREDIT	30	41%	1113	700	323	146	2312
MINISSAEXP	230	19%	12043	19570	3383	1253	36479
PEEPHOLE	76	13%	5723	24593	1920	680	32992
PLAN	20	24%	2043	1090	336	233	3722
QPLAN	116	21%	8026	16953	4566	1733	31394
RDTOK	80	19%	3430	3730	903	490	8633
READ	120	10%	5886	20826	1773	25106	53711
WARPLAN	63	14%	4736	6796	1153	293	13041
mean % of the total	0%	-	28%	49%	9%	14%	

3 Fixpoint Computations

As explained in section 2, cTI determines models of two versions of the initial program: $P^{\mathbb{N}}$, the CLP($\mathbb{N}$) version, and $P^{\mathcal{B}}$, the CLP($\mathcal{B}$) version. For this purpose, we developed an abstract immediate consequence operator U_P. This operator is quite similar to the well-known T_P. Proofs can be found in [18] (in french).

3.1 The algorithm

The key of our abstract computation is the notion of *rational interpretation* for a predicate symbol p:

Definition 3.1 *Let P be a program and p be a predicate symbol of P. We call a rational interpretation of p an equivalence of the form: $p(\tilde{x}) \leftrightarrow c$ where c, a disjunction of conjunctions of atomic constraints, is a finite formula s.t. $vars(c) \subseteq \tilde{x}$. We extend this notion to P: a rational interpretation of P is a set I containing exactly one interpretation for each predicate symbol p of P.*

We write $\mathcal{I}$ the set of all rational interpretations. We want to compute a rational interpretation which is a model of P.

Definition 3.2 *U_P is a function on $\mathcal{I}$ defined for any rational interpretation I of a program P by:*

$$\begin{aligned} U_P(I) = \{\mathrm{p}(\tilde{x}) \leftrightarrow c \mid\ & c \equiv \textstyle\bigvee_{cl \in P} \left(\exists_{-\tilde{x}}(c_0 \wedge \bigwedge_{1 \leq i \leq n} c_i)\right), \\ & cl \equiv \mathrm{p}(\tilde{x}) \leftarrow c_0 \diamond \mathrm{p}_1(\tilde{x}_1), \ldots, \mathrm{p}_n(\tilde{x}_n) \textit{ and} \\ & \forall i \in [\![1;n]\!], \mathrm{p}_i(\tilde{x}_i) \leftrightarrow c_i \in I\} \end{aligned}$$

We define the successive powers of U_P as usual.

Proposition 3.1 *U_P is monotone and continuous.*

Now let us establish a link between the meaning of a program P and the U_P operator. First, we give a ground semantics of a rational interpretation:

Definition 3.3 *Let I be a rational interpretation, we define the semantics of I by: $[I] = \{p(\tilde{d}) \mid p(\tilde{x}) \leftrightarrow c \in I, \tilde{d} \in \tilde{D}_\chi, \models_\chi c(\tilde{d})\}$ where D_χ is the domain of computation.*

Proposition 3.2 *For all $n \in \mathbb{N}$, we have $T_P \uparrow n \subseteq [U_P \uparrow n]$.*

Figure 1 presents an algorithm for the U_P operator.

A corollary of propositions 3.1 and 3.2 states that the least fixpoint of T_P is included the least fixpoint of U_P.

3.2 Widenings

In general, lfp(U_P) is not reachable in finite time. With a *widening* operator (∇) [10] convergence of the U_P operator can be enforced. In cTI, we use wideing for the

---

function $\mathtt{Up}(P,I)$: J

Require: P is a program, I is a rational interpretation of P.
Ensure: J is a rational interpretation of P.

```
 1: J ← ∅ ;
 2: for all clause(p(x̃) ← c ◇ p_1(x̃_1),...,p_n(x̃_n)) ∈ P do
 3:    for i ← 1 to n do
 4:       let p_i(x̃_i) ↔ c_i ∈ I ;
 5:    end for
 6:    let p(x̃) ↔ c' ∈ J ;
 7:    c ← ⋁(c', Π_x̃(c_1 ∧ ... ∧ c_n)) ;
 8:    J ← update(J, p(x̃) ↔ c)) ;
 9: end for
10: return J ;
```

Fig. 1. An abstract T_P-like operator.

computations in $\mathbb{N}$, whereas the boolean model is finitely reachable. Widening has a major impact on precision and speed of the computation. (We realized a *generic* fixpoint calculator for both CLP($\mathbb{Q}$) and CLP($\mathcal{B}$) [6,19,24]). A simple widening (∇_1) on a system of linear inequalities is due to Cousot and Cousot [10]. Here is an equivalent definition [27]:

Definition 3.4 *Let S_1 and S_2 be two sets of linear inequalities defining two polyhedra in $\mathbb{R}^n$. Then:*

$$S_1 \nabla_1 S_2 = \{\beta \in S_1 \mid S_2 \Rightarrow \beta\}$$

Cousot and Cousot in [11] propose an improved version of ∇_1, namely ∇_2 which has been simplified for the sake of efficiency in [28]:

Definition 3.5 *Let S_1 and S_2 be two sets of linear inequalities defining two polyhedra in $\mathbb{R}^n$. Then:*

$$\begin{aligned} S_1 \nabla_2 S_2 = \; & \{\beta \in S_1 \mid S_2 \Rightarrow \beta\} \bigcup \\ & \{\gamma \in S_2 \mid S_1 \Rightarrow \gamma \bigwedge \exists \beta \in S_1((S_1 - \{\beta\}) \cup \{\gamma\}) \Rightarrow \beta\} \end{aligned}$$

Comparisons of the impact of ∇_1 and ∇_2 on the accuracy of cTI are currently investigated. Figure 2 presents an algorithm for successive iterations of U_P until it reaches a fixpoint.

3.3 Optimization

The generic fixpoint computation engine is used for both $\mathbb{N}$ and $\mathcal{B}$. Making it as efficient as possible is therefore quite important. Currently all unit clauses defining the predicate symbols of the analyzed scc are taken into account in a first pass. Non-unit clauses are processed thereafter. Table 3 shows the impact of this optimization

function `itere_Up`$(P, max) : I_n$

Require: P is a CLP program, max is a non-negative integer.

Ensure: I_n is a model of P, s.t. $\text{lfp}(\texttt{Up}) \subseteq I_n$ and $\texttt{Up}(P, I_n) = I_n$.

```
n ← 0 ;
I_n ← ∅ ;
repeat
    I ← Up(P, I_n) ;
    n ← n + 1 ;
    if n ≤ max then
        I_n ← I ;
    else
        I_n ← I_{n-1} ∇ I ;
    end if
until I_n ⊆ I_{n-1}
return I_n ;
```

Fig. 2. An algorithm to reach a super set of $\text{lfp}(\texttt{Up})$.

comparing cTI 0.19 and the optimized version 0.22. Note that we also replace the union operator of line 7 of the algorithm presented in Fig. 1 by a convex hull (in both versions), which can be easily encoded in CLP($\mathbb{Q}$) using a technique proposed in [12] (see also [3]).

Table 3
Comparison between cTI 0.19 and cTI 0.22.

times in [ms]	$IR_{\mathbb{N}}$			$IR_{\mathcal{B}}$		
Programs	0.19	0.22	gain	0.19	0.22	gain
ANN	8850	6340	28%	4900	4900	0%
BID	1470	950	35%	330	330	0%
BOYER	9640	8140	16%	820	820	0%
BROWSE	1700	1170	31%	450	460	-2%
CREDIT	1110	800	28%	320	330	-3%
MINISSAEXP	12040	8210	32%	3330	3390	-2%
PEEPHOLE	5710	3360	41%	1910	1910	0%
PLAN	2030	980	52%	340	340	0%
QPLAN	8000	5880	27%	4670	4550	3%
RDTOK	3420	1990	42%	890	900	-1%
READ	6360	4690	26%	2270	1790	21%
WARPLAN	4750	3270	31%	1160	1150	1%
average	-	-	32%	-	-	1%
max.	-	-	52%	-	-	21%

4 Computing level-mappings

Level mappings are a key concept of many approaches to termination. They map ground atoms to natural numbers. K. Sohn and A. Van Gelder developed in 1991 an algorithm (SVG in short, see [30]) based on linear programming which ensures the existence of linear level mappings. We present an extension of this algorithm for the automatic generation of level mappings. Since this method, despite its power, does not seem to be very well-known in the context of termination analysis, we recall it after some preliminaries. Our extension to SVG is presented thereafter.

4.1 Preliminaries

We consider pure CLP($\mathbb{N}$) programs, with three predefined symbols for constraints: $=$, $\geq$, and $\leq$ with their usual meaning. Those programs are abstractions of (constraint) logic programs using (fixed or inferred) norms. We assume that clauses are written in flat form: $p_0(\tilde{x_0}) \leftarrow c_0, p_1(\tilde{x_1}), c_1, \dots, c_{l-1}, p_l(\tilde{x_l}), c_l$, with $i \neq j \rightarrow \tilde{x_i} \cap \tilde{x_j} = \phi$. In order to simplify this presentation, *we disallow mutually recursive predicates*. Within cTI itself, there is no such restriction. Note that we frequently switch to CLP($\mathbb{Q}^+$) as some problems in this structure are computationally cheaper (e.g. satisfiability), trading precision for speed. From now on, we write CLP for CLP($\mathbb{N}$) or CLP($\mathbb{Q}^+$). Section 3 shows how we can compute a model M for a CLP program P, where each predicate $p(\tilde{x})$ is defined as a (finite) conjunction of CLP constraints. We use this model to simplify the program P.

Definition 4.1 *Let M_P be a model of the CLP program P. The definition of a predicate p is* simplified wrt M *when, for the clauses defining p/n, we add to the right of each predicate $q(\tilde{x})$ its meaning $c_q(\tilde{x})$ relative to M_P. Moreover, those predicates $q/m \neq p/n$ which appear in the bodies are replaced by* true *(e.g. the dummy constraint* $0 = 0$*). Hence we end with a finite set of CLP clauses of the form: $p(\tilde{x_0}) \leftarrow c_0, p(\tilde{x_1}), c_1, \dots, c_{l-1}, p(\tilde{x_l}), c_l$. The simplified program is denoted P_M^{simpl}.*

We are interested in the automatic discovery of linear level mappings.

Definition 4.2 *Let p/n be a recursive predicate symbol of a CLP program P. A* linear level mapping *μ for $p(x_1, \dots, x_n)$ is a linear relation $\sum_{i=1}^{n} \mu_i x_i$, where the coefficients μ_i are non-negative integers.*

Such linear level mappings should satisfy a property ensuring their usefulness for left-termination:

Definition 4.3 *A linear level mapping μ for p is* valid wrt $P_{M_P}^{simpl}$ *if for each recursive clause defining p in P_M^{simpl}, say $p(\tilde{x_0}) \leftarrow c_0, p(\tilde{x_1}), c_1, \dots, c_{l-1}, p(\tilde{x_l}), c_l$, for $k = 0$ to $l-1$, $\wedge_{i=0}^{k} c_i \rightarrow \mu^T \tilde{x_0} \geq 1 + \mu^T \tilde{x_k}$, where μ^T denotes the transposed vector of μ.*

4.2 The algorithm SVG

Let us first quickly review the algorithm of Sohn and Van Gelder. It aims at checking the existence of one valid linear level mapping. SVG starts with a pure CLP program P and a constrained goal. A top-down boundedness analysis (see [23,24]) reveals the calling modes of each predicate. Arguments are detected as either bounded (denoted b) or unbounded (u). A CLP model M is computed and P is simplified to P_M^{simpl}. Then SVG examines each recursive procedure p/n in turn (the precise order does not matter). Let us symbolically define the level mapping for $p(x_1,\ldots,x_n)$ as $\mu^T\tilde{x} = \sum_{1\leq i\leq n}\mu_i^{u\ or\ b}x_i$ where $\mu_i^u = 0$ is x_i is labelled as unbounded wrt the calling mode of p/n and $\mu_i^b \geq 0$ if x_i is labelled as bounded. Each clause r_i is processed. For one such clause, l simplified rules (for $k = 0$ to $k = l-1$) are constructed: $p(\tilde{x_0}) \leftarrow \wedge_{0\leq j\leq k}c_j, p(\tilde{x_k})$. One can assume that the constraint $C_{ij} = \wedge_{0\leq j\leq k}c_j$ is satisfiable, already projected onto $\tilde{x_0}\cup\tilde{x_k}$, only contains inequalities of the form $\leq$, and implies $\tilde{x_0} \geq 0$ and $\tilde{x_k} \geq 0$. Such a simplified rule gives rise to the following (pseudo-)linear programming problem

$$minimize\ \theta = \mu^T(\tilde{x_0} - \tilde{x_k})\ subject\ to\ C_{ij} \tag{11}$$

A valid linear level mapping μ exists (at least for this recursive call of this clause) if $\theta^* \geq 1$ where θ^* denotes the minimum of the objective function. Unfortunately, because of the symbolic constants μ, (11) is *not* a linear programming problem.

The *clever* idea of the authors is to consider its dual form:

$$maximize\ \eta = \tilde{y}\beta\ subject\ to\ \tilde{y} \geq 0 \wedge \tilde{y}A \geq (\mu, -\mu) \tag{12}$$

By duality theory (see [29] for instance), we have $\theta^* = \eta^*$. Now, the authors observe that μ appears linearly in the dual problem (it is not true for (11)) because no μ_i appears in A. Hence (12) can be rewritten, by adding $\eta \geq 1$ and $\tilde{\mu^b} \geq 0 \wedge \tilde{\mu^u} = 0$, as S_{ij}, a set of linear inequations. If the conjunction $S_p = \wedge_{i,j}S_{ij}$ for each recursive call and for each clause defining p/n is satisfiable, then there exists a valid linear level mapping for p/n.

4.3 An extension of SVG

Instead of checking satisfiability of S_p, we can project it onto μ (we do not need the top-down boundeness analysis explained subsection 4.2, all arguments are assumed bounded). Hence we get in one constraint *all* the valid linear level mappings. It remains to compute the maximal elements of $\Pi_\mu(S_p)$, given the partial order: $\mu^1 \succeq \mu^2$ if $\forall i \in [\![1;n]\!]\mu_i^1 \neq 0 \rightarrow \mu_i^2 \neq 0$.

Example 4.1 *For app/3, let $\mu(x,y,z) = ax+by+cz$. We have $\Pi_\mu(S_p) = \{a+c \geq 1\}$. There are two maximal elements: $\mu^1(x,y,z) = x$ and $\mu^2(x,y,z) = z$.*

In some sense, given a model for a program, this extension is complete. But a more precise model can lead to more maximal elements. Hence the precision of the inferred CLP model is important. From an implementation point of view, this algorithm heavily relies on the costly projection operator. We found that a good strategy is to project constraints as soon as possible.

5 Inferring termination neutral arguments

Predicates frequently contain arguments that have no influence on universal left termination. Differences are a frequent pattern with this property. In most uses of differences (e.g. list differences), one argument of the difference is termination neutral.

Definition 5.1 (Termination neutral argument (TNA)) *Let G be a goal* $\leftarrow p(\ldots, x_i, \ldots)$, *where the i-th argument is a variable* x_i *that occurs only at this position and let* θ *be a substitution s.t. DOM(*θ*) =* x_i. *The argument i is a left-termination neutral argument (TNA), if*

$$\forall \theta : G\theta \, \mathrm{left-terminates} \Leftrightarrow G \, \mathrm{left-terminates}$$

Since the TNA property is independent of a particular norm, the analysis can be performed prior to any other analysis. The information is currently used to accelerate the search for level mappings: if the *i*-th argument of *p* is detected as TNA, we set its corresponding coefficient μ_i to 0 before computing μ. Table 4 summarizes the gain we obtain when applying the TNA analysis on the six most demanding programs.

Example 5.1 *The second argument of app/3, the third of app3/4, and the second of nrev/2 are termination neutral. For app/3, we can safely state* $\mu(x,y,z) = ax + cz$.

To determine some termination neutral arguments we represent each argument of a predicate with a boolean 0-1 variable where 1 means termination neutral and 0 means that the argument may influence termination. Constraints are imposed that tell which arguments may influence termination. The actual result is computed by labeling for a maximal satisfying assignment.

We consider a clause (with nontrivial head unifications explicit) to consist of the following possibly empty sequences of goals $S_1 \circ S_2 \circ S_3 \circ S_4$. S_1 contains any goals, S_2 is a possibly nonterminating goal, S_3 are always terminating goals, S_4 is an always failing goal followed by any other goals. S_3 and S_4 are ignored in the analysis. Head variables that occur in S_1 may influence termination. All these variables are set to 0. A head variable H that first occurs in S_2 may influence termination only if the argument B where it occurs in is not termination neutral. The constraint $\neg B \Rightarrow \neg H$ ensures that if B is false (may influence termination), then H is false, too. The following predicate shows all constraints imposed for our three predicates. A labeling process finds the desired answer.

$\text{tnaargs}([\text{app}(A_1,A_2,A_3), \text{app3}(B_1,B_2,B_3,B_4), \text{nrev}(C_1,C_2)]) \leftarrow$
% app/3 clause 1: $[] \circ [] \circ [A_1 = [], A_2 = A_3] \circ []$
% app/3 clause 2:
% $[A_1 = [X|Xs], A_3 = [X|Zs]] \circ [\text{app}(Xs, A_2, Zs)] \circ [] \circ []$
$A_1 = 0, \neg A_2 \Rightarrow \neg A_2, A_3 = 0,$
% app3/4:
% $[\text{app}(B_1, B_2, Vs)] \circ [\text{app}(Vs, B_3, B_4)] \circ [] \circ []$
$B_1 = 0, B_2 = 0, \neg A_2 \Rightarrow \neg B_3, \neg A_3 \Rightarrow \neg A_4,$
% nrev/2 clause 1: $[] \circ [] \circ [C_1 = [], C_2 = []] \circ []$
% nrev/2 clause 2:
% $[C_1 = [X|Xs], \text{nrev}(Xs, Zs)] \circ [\text{app}(Zs, [X], C_2)] \circ [] \circ []$ % not optimal!
% $[C_1 = [X|Xs]] \circ [\text{nrev}(Xs, Zs)] \circ [\text{app}(Zs, [X], C_2)] \circ []$ % ideal
$C_1 = 0, \neg A_3 \Rightarrow \neg C_2$

For app/3 and app3/4 the results are optimal. However, in nrev/2 we are not able to infer that the second argument is termination neutral. While the second goal app/3 is in fact always terminating —cTI is also able to prove this— we do not have this information right at hand at this point in time prior to the actual termination inference. In fact, the primary purpose of TNA analysis within cTI is to speed up the subsequent inference process, sacrificing precision for speed. The analysis performed in this manner is typically faster than parsing the corresponding program text with read/1 using SICStus' tokenizer written in C. It has never been observed [2] to take longer than parsing the program twice.

Table 4
Impact of TNA on level mappings computation.

Program	with TNA	no TNA	gain
ANN	14120	14442	2 %
MINISSAEXP	19578	21222	8 %
PEEPHOLE	25938	25602	-1 %
QPLAN	16590	16888	2 %
RDTOK	3772	4034	7 %
READ	20862	21292	2 %
arith. mean	16810	17247	3 %
max. gain	-	-	8 %

6 Conclusion

We have presented the main algorithms of cTI, our bottom-up left-termination inference tool for logic programs and given some running for medium-sized logic programs. The analysis requires three fixpoint computations and the inference of

[2] cTI's WWW interface provides an option "TNA only" to perform this comparison

well-founded orders. We have described some optimizations and measured their impacts.

Still, more work is required to cope with certain kinds of programs like CHAT, as suggested by P. Tarau. A more detailed look to this program shows that it contains a predicate symbol with 14 arguments and one scc of 30 mutually recursive predicate symbols with 8 arguments per predicate symbol on the average. Our current implementation, using the constraint solvers of SICStus Prolog, is neither able to compute a numeric model nor the boolean model in reasonable time. Table 2 indicates that the bottleneck is situated in the numeric computations. We therefore consider to replace the generic solvers by more specialized tools (polyhedra manipulations, a a simplex solver optimized for projections, and a more efficient BDD package).

References

[1] K.R. APT. *From Logic Programming to Prolog*. Prentice Hall, 1997.

[2] K.R. APT and D. PEDRESCHI. Studies in pure prolog: Termination. In J.W. LLOYD, editor, *Proc. of the Symp. in Computational Logic*, pages 150–176. Springer, 1990.

[3] F. BENOY and A. KING. Inferring argument size relationships with CLP($\mathbb{R}$). In *Logic Program Synthesis and Transformation*. Springer-Verlag, 1997.

[4] F. BUENO, M. GARCIA DE LA BANDA, and M. HERMENEGILDO. Effectiveness of global analysis in strict independence-based automatic program parallelization. In *Proceedings of International Symposium on Logic Programming*, pages 320–336, 1994.

[5] S. BURCKEL, S. HOARAU, F. MESNARD, and U. NEUMERKEL. cti: Bottom-up termination inference for logic programs. submitted for publication, 2000.

[6] M. CARLSSON. Boolean constraints in sicstus prolog. Technical Report T91:09, Swedish Institute of Computer Science, 1994.

[7] M. CODISH and C. TABOCH. A semantics basis for termination analysis of logic programs. *Journal of Logic Programming*, 41:103–123, 1999.

[8] S. COLIN, F. MESNARD, and A. RAUZY. Constraint logic programming and mu-calculus. *ERCIM/COMPULOG Workshop on Constraints*, 1997.

[9] P. COUSOT and R. COUSOT. Abstract interpretation: a unifed lattice model for static analysis of programs by construction or approximation of fixpoints. *Conference Record of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[10] P. COUSOT and R. COUSOT. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.

[11] P. COUSOT and R. COUSOT. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Lecture Notes of Computer Science*, volume 631. Springer-Verlag, 1992. Proceedings of PLILP'92.

[12] B. DE BACKER and H. BERINGER. A CLP language handling disjunctions of linear constraints. In *Proc. of ICLP'93*, pages 550–563. MIT Press, 1993.

[13] S.K. Debray and N.W. Lin. Cost analysis of logic programs. ACM Transactions on Programming Languages and Systems, 15(5):826–875, November 1993.

[14] S. DECORTE. *Enhancing the power of termination analysis of logic programs through types and constraints*. PhD thesis, Katholieke Universiteit Leuven, 1997.

[15] P. DERANSART, A. ED-DBALI, and L. CERVONI. *Prolog: The standard, reference manuel*. Springer-Verlag, 1996.

[16] P. DERANSART, G. FERRAND, and M. TÉGUIA. NSTO programs (not subject to occur-check). *Proc. of the Int. Logic Programming Symp.*, pages 533–547, 1991.

[17] D. DE SCHREYE and S. DECORTE. Termination of logic programs : the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.

[18] S. HOARAU. *Inférer et compiler la terminaison des programmes logiques avec contraintes*. PhD thesis, Université de La Réunion, 1999.

[19] C. HOLZBAUR. Ofai clp(q,r) manual, edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute, 1995.

[20] J. JAFFAR and M. J. MAHER. Constraint Logic Programming: a survey. *Journal of Logic Programming*, 19:503–581, 1994.

[21] N. LINDENSTRAUSS and Y. SAGIV. Automatic termination analysis of logic programs. In *Proceedings of the 14th ICLP*, pages 63–77, 1997.

[22] F. MESNARD. *Etude de la terminaison des programmes logiques avec contraintes, au moyen d'approximations*. PhD thesis, Université Paris VI, 1993.

[23] F. MESNARD. Inferring left-terminating classes of queries for constraint logic programs by means of approximations. In *Proceedings of JICSLP'96*, pages 7–21. The MIT Press, 1996.

[24] F. MESNARD. Entailment and projection for CLP($\mathcal{B}$) and CLP($\mathbb{Q}$) in SICStus Prolog. *1st International Workshop on Constraint Reasoning for Constraint Programming*, 1997.

[25] U. NEUMERKEL. GUPU: A prolog course environment and its programming methodology. In M. Maher, editor, *Proc. of JICSLP'96*, page 549. MIT Press, 1996. http://www.complang.tuwien.ac.at/ulrich/gupu/.

[26] S. RUGGIERI. *Verification and validation of logic programs*. PhD thesis, Università di Pisa, 1999.

[27] H. SAĞAM. *A Toolkit for Static Analysis of Constraint Logic Programs*. PhD thesis, University of Bristol, 1997.

[28] H. SAĞLAM and J. GALLAGHER. Static analysis of logic programs using CLP as a meta-language. Technical Report CSTR-96-003, University of Bristol, Department of Computer Science, 1996.

[29] A. SCHRIJVER. *Theory of linear and integer programming*. John Wiley and Sons, 1986.

[30] K. SOHN and A. VAN GELDER. Termination detection in logic programs using argument sizes. In *Proceedings of PODS'91*, pages 216–226, 1991.

[31] T. VASAK and J. POTTER. Characterization of terminating logic programs. In *Proceedings of the 1986 IEEE Symposium on Logic Programming*, 1986.

[32] K. VERSCHAETSE and D. DE SCHREYE. Deriving termination proofs for logic programs, using abstract procedures. In *Proceedings of the 8th ICLP*, pages 301–315, 1991.