PONTELLI ENRICO

GULP '90 - Padova

Gruppo Ricercatori ed Utenti
di Logic Programming

GULP '90

Quinto Convegno sulla

Programmazione Logica

A cura di Annalisa Bossi

Padova, 6-8 Giugno 1990

Sala della Gran Guardia

Organizzato dal:

Dipartimento di Matematica Pura ed Applicata - Università di Padova

Con il Patrocinio di:

Università di Padova
Comune di Padova-Settore Spettacolo e Manifestazioni

Coordinatrice:

Annalisa Bossi (Università di Padova)

Comitato di Programma:

Pier Giorgio Bosco (CSELT, Torino)
Alfredo Ferro (Università di Catania)
Gilberto Filé (Università di Padova)
Giorgio Levi (Università di Pisa)
Alberto Martelli (Università di Torino)
Maurizio Martelli (Università di Genova)
Paola Mello (Università di Bologna)
Daniele Nardi (Università di Roma)
M. Vittoria Oneto (ELSAG, Genova)
Enrico Pagello (Università di Padova)
Alberto Pettorossi (Università di Roma)
Cristina Ruggieri (D.S. Logics, Bologna)
Maria Simi (Università di Udine)
Silvio Valentini (Università di Milano)

Comitato organizzatore:

Nicoletta Cocco (Università di Padova)
Susi Dulli (Università di Padova)
Luigi Marcolungo (Università di Padova)
Nino Trainito (LADSEB-CNR, Padova)

Con il Contributo di:

CNR
CERVED S.p.A.
ICON s.r.l.
ELSAG S.p.A.
Systems & Management S. p. A.
AIS-Artificial Intelligence Software S.p.A.
Cassa di Risparmio di Padova e Rovigo

Stampa: La Photograph, via del Santo, 48/6, Padova

INDICE

LINGUAGGI

OTTIMIZZAZIONI E TRASFORMAZIONI

SEMANTICA

RELAZIONI INVITATE

Prefazione

Questo volume raccoglie i lavori presentati al Quinto Convegno sulla Programmazione Logica: GULP '90. Il Convegno é una occasione di incontro e di discussione tra tutti coloro che nel mondo accademico ed industriale svolgono attività di ricerca o sviluppano applicazioni nel campo della Programmazione Logica. Il Convegno, che si tiene a Padova dal 6 all'8 Giugno 1990, prevede quattro relazioni invitate e la presentazione di 27 lavori riguardanti le seguenti aree:

- ambienti e strumenti;
- applicazioni;
- estensioni e variazioni;
- ottimizzazioni e trasformazioni;
- semantica.

Desidero ringraziare il GULP, ed in particolare il suo Presidente Giorgio Levi, per la fiducia dimostrata nell' affidarmi il coordinamento del Convegno. Vorrei inoltre ringraziare tutti coloro che hanno contribuito alla sua organizzazione:

- i membri del Comitato di Programma ed i revisori per il lavoro di selezione dei lavori;
- Giuliana Dettori e Paola Mello per gli indispensabili consigli sull' organizzazione;
- i membri del Comitato Organizzatore: Nicoletta Cocco, Susi Dulli, Nino Trainito e Luigi Marcolungo, per la preziosa collaborazione;
- il Dipartimento di Matematica Pura e Applicata dell' Università di Padova, il CNR, la Cerved S.p.A., la ICON s.r.l, la ELSAG S.p.A., la System & Management S.p.A., la AIS-Artificial Intelligence Software S.p.A. e la Cassa di Risparmio di Padova e Rovigo per i contributi dati per l'organizzazione del GULP '90;
- l'Università di Padova ed il Comune di Padova-Settore Spettacolo e Manifestazioni per il Patrocinio offerto;
- i relatori invitati, gli autori dei lavori e tutti i partecipanti.

Annalisa Bossi

Premessa

Il Gruppo ricercatori ed Utenti di Logic Programming (GULP) è affiliato alla Associazione internazionale di Logic Programming (ALP). Scopo del Gruppo è la divulgazione della Programmazione Logica e la creazione di un ambiente fertile per lo scambio di informazioni ed esperienze tra l'Università, il mondo della scuola e l'Industria. A tal fine il GULP promuove diverse attività, quali la pubblicazione di un bollettino, l'organizzazione di workshop, scuole avanzate e di un Convegno annuale.

Il Convegno si propone principalmente le seguenti finalità:

- Servire da punto d'incontro per tutti i ricercatori ed utilizzatori di Programmazione Logica e permettere a molti di essi di esporre i loro risultati e problemi sia teorici che applicativi.
- Delineare lo stato dell'arte della ricerca nel settore attraverso relazioni tenute da ricercatori invitati.
- Avvicinare studenti e ricercatori alle principali tematiche della Programmazione Logica attraverso brevi corsi introduttivi.

I precedenti Convegni, che si sono svolti a Genova (1986), Torino (1987), Roma (1988) e Bologna (1989), hanno riscosso un interesse sempre crescente, dimostrando, con la qualità dei programmi e la nutrita partecipazione di studenti, ricercatori ed utenti, la crescita della Programmazione Logica sia nel campo della ricerca che delle applicazioni industriali.

L'ottimo livello delle relazioni programmate per la quinta edizione del Convegno che si svolge a Padova conferma la crescita di qualità ed il successo dell'iniziativa.

A nome del GULP ringrazio Annalisa Bossi e gli altri colleghi padovani per l'eccellente organizzazione del Convegno.

Giorgio Levi

Presidente del GULP

AMBIENTI

E

STRUMENTI

Performance of Logic+Functional Programming on a distributed memory architecture

P.G. Bosco, C. Cecchi, C. Moiso, M. Porta and G. Sofi
CSELT - via Reiss Romoli 274 - 10148 Torino

1. Introduction

The languages K-LEAF [Bosco87] and IDEAL [Bosco86] are proper extensions of Prolog to include functional features. They have been proved to be efficiently implementable through resolution and, consequently, the WAM technology [Bosco89b].

The high potential for parallelism exploitation in logic languages has been already shown by several parallel implementations, most of which on shared memory (bus-based) architectures. Our approach to parallelism was to investigate highly scalable structures, based on a large amount of nodes, each with its own local memory, where a notion of logically shared address space can be obtained through a suitable efficient interconnection network. We focussed on OR-parallelism, but the class of application we were mainly interested in (real-time signal processing like image, speech, etc.), where, usually, blind OR-parallelism is not sufficient, requires some support for (multiple) best-first search. This implies (beyond some primitives for communication among OR-branches and for defining a notion of *score*) a dynamic process model, differently from the *worker* model of Aurora [Warren87] and PepSys [Baron88]. In accordance to these requirements we designed OP-Prolog [Giandonato88,Moiso89], an OR-parallel extension of Prolog conceived to be a powerful tool to program highly non-deterministic algorithms.

The problem of how to control the grainsize of such processes (as well as the tasks in the *worker* model) is more evident on distributed architectures than on shared-memory ones. We devised a simple set of annotations for Prolog (different from the one available in Aurora) to distinguish sequential procedures from parallel ones and to guarantee the fully sequential evaluation of a goal inside a parallel computation. The parallel version of K-LEAF is a proper extension of OP-Prolog, from which it inherited the annotation scheme [Bosco89a].

Regarding AND-parallelism (needed to support parallel reduction of strict functional terms) we restricted ourselves to the implementation of the mapping of AND to OR-parallelism [Bosco89a] like in [Carlsson88].

Along the paper we give an overview of the parallel abstract machine and its implementation on our distributed memory architecture PIPES (sections 2 and 3) while section 4 is mainly devoted to a detailed analysis of the performance of OP-Prolog and parallel K-LEAF, introduced by a short discussion on the sequential performance which is important for a correct interpretation of results in the parallel environment.

2. Parallel K-WAM Abstract Machine

The Warren Abstract Machine (WAM) has currently become the undiscussed state-of-the-art technique to efficiently implement sequential Prolog and parallel extensions of the language. Our K-WAM has been built on sequential WAM, preserving its basic mechanisms as much as possible and extending it along two almost disjoint directions [Bosco89a]: (a) the specific demands of OP-Prolog model [Giandonato88], namely support of process execution and control of parallelism; (b) the requirements of K-LEAF model, which entails lazy evaluation of functional terms. The latter component has already been illustrated deeply in [Bosco89b]. The former aspect is very shortly sketched in this section; due to lack of room, we prefer to focus on a few major points (organization of process run-time data structures, implementation of Cactus Stacks, Multiple Bindings, parallel call scheme), leaving out other nevertheless interesting issues (e.g. *parsetof* termination and deallocation of shared blocks). The detailed description of the abstract machine of OP-Prolog is reported in [Merlo88].

As well-known, the sequential WAM is a virtual machine provided with some dynamic areas: the Stack, the Heap, the Trail, the PDL and some Status Registers. Three additional areas have been introduced in parallel WAM: Binding List (BL) and Binding Array (BA), with which a solution to the OR-conflict problem has been given (described later); a Control Stack, recording Parallel Split Points, i.e. blocks of information related to the generation of new processes during a parallel call (the parallel counterpart of sequential choice points). Special attention has been paid to reduce the number of areas required. Therefore, for each WAM process the eight above-mentioned virtual areas have been compacted into four: (1) Control Stack, recording Status Registers, Split Points and the Trail; (2) Stack, including PDL; (3) Heap, including also BL elements; (4) BA.

A direct consequence of OP-Prolog eager process spawning is the need to support an environment of *Cactus Stacks*, into which the linear stacks of sequential implementation have to be transformed. To cope with the above requirement, the memory handling is based on a *demand fixed-length block* policy, namely the data memory has been partitioned into a collection of fixed-length blocks (typical satisfactory values ranging from 1K to 4Kbytes) which are delivered one at a time on demand as soon as a process needs to extend one of its areas (e.g. the Heap when allocating a new structure after a *put_struct* instruction). A serious problem arising from the block policy has been how to handle the so-called *Seniority Tests*, required in many situations during the dereference/binding algorithm (for instance, if *older(x,B)* is true, i.e. if address x is older than (the contents of) B register, the value bound to x location must be *trailed*). While in the sequential implementation these are very straightforward checks, consisting in a single address comparison operation, in the parallel case the strategy is a bit more complex but nevertheless satisfactorily efficient, based on the concept of *Block Sequence Number* (an integer qualifying the "age" of the block and written in its first cell, reserved to the purpose).

One of the main problems that a parallel implementation of Prolog must cope with is the OR-conflict, arising on variables which are still unbound when the computation forks at a split node. *Multiple environments* must be created so as to allow the alternative paths originated to assign different values to such variables.

A number of techniques have been proposed in literature, each one exhibiting its pros and cons. Anyway, the most interesting ones are centered around two basic concepts: *Binding Array* (BA) and *Binding List* (BL). Briefly, the former technique consists in transforming a reference to an unbound variable into an index of an array; multiple bindings can be simply created by assigning each process (or processor) its own BA copy. The BL is but a generalization of the Trail of the sequential model: an assignment to a variable in a shared part of the stack can be done only indirectly, by adding a pair <*var-address,value*> to BL (obviously, the dereference algorithm must be modified accordingly and implies, in some cases, a sequential scanning of the list).

Our solution has consisted in an original combination of these two methods, appropriately tailored to the specific nature of OP-Prolog execution model and distributed-memory physical implementation. Both eager process creation and distributed-memory requirement would suggest to provide each virtual process with its own BA copy, so minimizing process interference and remote accesses; but indiscriminate duplication of BAs would be impractical, since BAs are potentially ever-growing structures, unless appropriate measures are taken. As a consequence, we have distinguished a special class of variables, so-called *parallel local unbound*, recognized at compile time (i.e. those local variables in a parallel clause that will be unbound before a parallel split), for which the BA method is used. Multiple bindings for all the other variables are built through BL (but notice that in most cases, e.g. when referencing private areas of Stack or Heap, owing to the close separation between OP-Prolog sequential and parallel components, multiple bindings are not needed and assignments can be done directly, like in sequential Prolog).

For each of the basic constructs of the language (*parallel call*, *parsetof*, *all-solutions*, etc) a minimal number of new special WAM instruction have been provided along with an appropriate compilation scheme. For instance, the expansion of a (many-clause) parallel procedure

comprises: (a) one (or more) *parclause instruction sequences* ; (b) code for individual clauses.

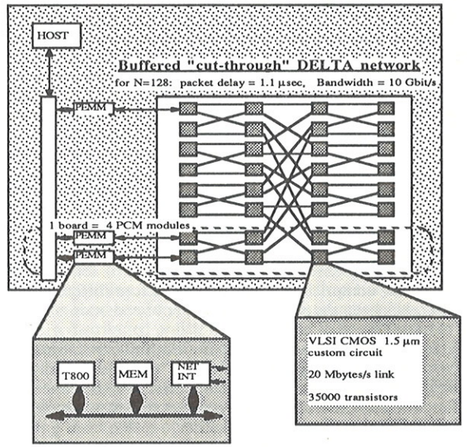
```
first_parclause   $cl1      /*  main parclause sequence  */
par_clause        $cl2
.........
par_clause        $cln
spawn
```

The above scheme has been inspired to the sequential indexing scheme (*try / retry / trust* instructions). Additional shorter parclause sequences, indexed by a *switch_on_term* instruction, may be expanded by the compiler. Code for each individual clause consists of a head (plus guard) part and a body part, separated by a *save_process* instruction.

The head execution phase, started by *first_parclause*, consists in the sequential execution of WAM instructions corresponding to all alternative clause heads (and guards) and is terminated by *spawn*. The *first_parclause* instruction allocates in the Control Stack an appropriate data structure, the Parallel Split Point, containing information required for memory deallocation upon process termination, most notably the number of new child processes. The execution of each head may fail or succeed (in the latter case *save_process* stores information to be used later by *spawn*), in both cases flow of control is routed, via *par_clause* instruction, to the execution of next clause head. At the end, new child processes are activated by *spawn* (each one provided with its own copy of Binding Array). The parent process overtakes execution of left-most branch, extending the parent stack blocks (namely, Stack, Heap and Control Stack), while new blocks are dynamically allocated for child processes, so originating the above-mentioned Cactus Stack structure.

3. Distributed implementation of K-LEAF on the PIPES architecture

As detailed in [Balboni87], the parallel architecture PIPES, developed at CSELT within ESPRIT Project n. 26, can be classified as a distributed memory MIMD (Multiple Instructions Multiple Data streams) processor, consisting in a 32-bit T800 Transputer Array, where two distinct communication networks have been provided: a cut-through routing packet-switched Delta communication network, allowing very fast one-to-any communications; a bidirectional ring, obtained by simply interconnecting Transputer links, for not urgent large transfers of data.

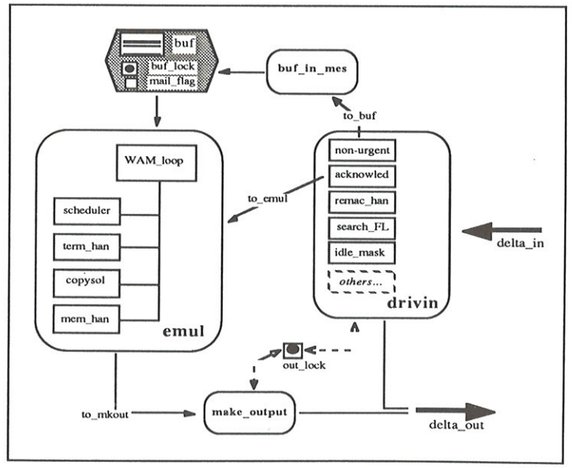


Though our architecture doesn't possess a physically centralized common memory, a concept of global distributed storage has been implemented (namely, a virtual shared memory is obtained by partitioning a single global address space into local blocks of memory).
The prototype currently operating at our labs is shown in the following figure, comprising a 16-Transputer Array (TR0,...,TR15), with 2Mbytes of RAM each, connected through a MSI implementation of the network, and a host processor, used for I/O purposes (loading programs, collecting solutions, monitoring significant parameters of parallel execution,...). The host processor is an ordinary PC IBM, supplied with a B004 Transputer board; this Root Transputer is connected via links 1 and 2 to TR0 and TR15 respectively.

Our distributed implementation has been based on a new parallel version of C language, specifically tailored for Transputer systems, namely "Parallel C", by 3L Ltd [3L88]. Differently from Occam, which only supports a CSP-like (Communicating Sequential Processes) communication model, based on synchronous message passing, Parallel C also allows a shared-memory communication paradigm. This is obtained by making use of the so-called "threads" (dynamically startable flows of execution, sharing code and static data). Of course, care must be taken to avoid undesired conflicts on shared locations; appropriate "semaphore" primitives are provided by the run-time library for the purpose.

Node organization

To reduce inefficiencies due to remote accesses, program code is replicated on each node of our architecture; precisely, on each element of the Transputer array a copy of the task depicted in the next figure is loaded.



Such an organization, comprising a total amount of four different threads, two semaphores and one shared buffer, has been the result of a number of constraints and choices, for instance:

(a) The low-priority *emul* thread, comprising not only the strict emulation of extended WAM instructions but also some system "modules" such as the scheduler, the termination handler, etc., has been implemented as a large monolithic block, including as many functions as possible (in our opinion this was the most rapid way of porting the pre-existent VAX implementation).

(b) It is a crucial point to ensure that the high-priority input *drivin* thread, receiving messages from other processors via the Delta network, must never be blocked; this implies that some form of non-blocking buffering for *non-urgent* messages (see later) must be provided. Therefore, communication between *drivin* and *emul* occurs through buffer *buf* and semaphore *buf_lock*.
Notice that the hardwired internal Transputer process scheduling is organised into two priority levels; while *low-priority* processes can be interrupted, either by re-activation of a high-priority process or at the end of a time-slice, an *high-priority* process must suspend itself.
(c) The absence of the *ALT* construct obliged us to introduce a semaphore *out_lock* to protect the shared *delta_out* channel.
(d) A restriction of Parallel C forbidding threads at different priorities from sharing semaphores obliged us to add two intermediate threads, *buf_in_mes* (low-priority) and *make_output* (high-priority).
(e) Special care was taken to avoid potential deadlocks.

In the following, besides some considerations about scheduling and messages, the basic components of the process structure will be discussed:
A number of different messages are discriminated by *drivin* , mainly: (1) *Non-urgent messages*, to be delivered to *emul* thread and not requiring immediate response (among these for instance, messages for the creation of a new process, for memory deallocation upon process termination, for sending a solution to the *parsetof parent process).* (2) *Acknowledges* in response to *emul* requests (e.g. when remote data are returned, or the result of a search on a remote Binding List). (3) requests from the outside for remote access (handled by the component *remac_han*). (4) requests from the outside for searching the local Binding List (handled by the component *search_BL*). (5) An *idle_mask* message (see later), enabling a distributed load-balancing function. Lastly, other messages have been provided which are responsible for loading a new object file, restarting the system with new parameters, extracting statistical information.
At the end of the execution of each K-WAM instruction, before fetching next one, *WAM_loop* tests (consulting *mail_flag*) if some non-urgent message has arrived in the meantime. If so, the message is delivered to the appropriate system module and immediately handled.
System modules (actually, collection of procedures) can be activated not only by external requests (i.e. when a non-urgent message is received from another processor), but also by internal requests (i.e. issued by the process currently running on that processor). For each module, different procedures are provided to handle both situations. (The only exception is *mem_han*, implementing the demand fixed-length block memory handling policy, which responds to internal requests only.)
Messages issued by *emul* can be *synchronized* or *asynchronous*. In the former case, *emul* is suspended, waiting for an acknowledge (a request for remote access is an example); in the latter case, *emul* continues execution, either going on with the same process (for instance, when a newly spawned process is distributed to a different processor), or suspending current process and restarting the first process of the scheduling queue (this occurs after notifying a solution to the parsetof parent process).

Distributed Scheduling

One of the main issues in a distributed multiprocessor is the implementation of an efficient parallel scheduling policy, able to achieve a fair load balancing of processes among processors. Two principles have been followed: a) in a physical architecture like ours, lacking a true common memory, a distributed policy is in any case to be preferred to a centralized one; b) in order not to compromise locality of reference, processes can be transferred only once; after a process has been inserted into the *Scheduling Queue* of a processing node, further migrations are forbidden (a process in the course of execution may have generated a lot of data structures in the local memory of a Transputer, which we don't want to move) .
Our load-balancing mechanism operates on a demand basis, so being able to distribute

processes only when really needed. It works as follows: (a) there is a *Scheduling Queue* for each processor; (b) when new processes are spawned in a processor, they are inserted into the Scheduling Queue of that node, ordered on the grounds of priority information; (c) all the time an *idle_mask* message circulates through a virtual ring network connecting all processors (realized via the Delta network); a bit set in a given position of the mask indicates that the corresponding processor is *idle* (i.e. the number of processes in that node is below a programmable threshold); (d) when a processor T_i receives the *idle_mask* message, if there are idle processors and if T_i has *sendable* processes (a process is sendable only if it hasn't been started yet; this measure is intended to reduce the cost of transmission and minimize non-local accesses), T_i sends one process to each idle processor, as long as there are sendable processes; at the end, *idle_mask* is propagated to T_{i+1} (note that if T_i is idle, before the propagation the corresponding bit in the mask is switched on). The above scheduling policy, which is able to favour locality of reference by preferably allocating a new process in the same node of the parent process, with minor adjustments and appropriate tunings has shown a very satisfactory load-balancing performance in a wide range of situations, and has been essential to achieve the speed-up figures reported in the next section.

Remote Access and Lazy Copying

The *remac_han* component of *drivin* is in charge of responding to three different requests coming from the outside: (1) read a memory location (it should be stressed that, due to the nature of Prolog and our implementation of multiple environments, remote write operations are not required); (2) read a Stack Frame (a child process residing in another processor will eventually need a remote ancestor stack frame; before going on with the continuation, a new local copy is created); (3) read the top level of a (sub)structure (this is the so-called *lazy copying* mechanism: a remote structure is incrementally copied, each level at a time; note that the copy operation is performed only when actually essential, i.e. when unifying a remote structure with a local one), including the special *prodvar* structure. Note that *remac_han* is programmed so that it can ensure a very fast response; for this reason, it doesn't do any dereferencing operation and just limits itself to read and pass one or a number of locations (therefore, the control of dereferencing is left to the processor requesting the remote access).

Searching Remote Binding Lists

During the dereferencing algorithm the need for a search on BL may arise. If a binding is not found in a local BL, a remote search is initiated, starting from the processor referenced by the bottom BL element. The *search_BL* component of *drivin* returns one of the following three results: (1) the desired binding, in case of success; (2) continuation processor (search must be continued in a different processor); (3) failure (if the end of BL is reached without finding a binding).

Solution Copying by *parsetof*

The computational model of the language prescribes that solutions produced by descendant processes (originated by a *parsetof*) must be wholly copied and collected in a list in the Heap of the parent process. A distributed implementation has to ensure some form of mutual exclusion in the copy operation, since different processes might compete in adding an element to the same shared list. In our system this is obtained by centralizing the function, in each processor, in the *copysol* module of *emul* thread, which handles with no interference both internal and external requests.

4. Performance of K-LEAF implementations

The performance of the sequential K-LEAF implementation on a variety of commercial machines and of the parallel implementation on PIPES is analysed throughout this section.

The (average) figures in milliseconds were obtained on slightly instrumented versions of K-LEAF executors running the following *all-solutions* benchmarks:

`queensp:` the *n-queens* problem formulated as a pure generate (permutations) and test problem. It is the most "abstract" version of the problem. It exploits K-LEAF lazy evaluation.

`queens1:` the standard *n-queens* Prolog benchmark, where the positions on the chessboard are generated by a predicate that non-deterministically outputs an integer from 1 to n.

`queens2:` a more efficient Prolog version of *n-queens* where the positions on the chessboard are selected from a previously constructed list of integers from 1 to n.

`fib:` the standard functional definition of Fibonacci numbers (AND-parallelism).

`s&m:` the standard *salt and mustard* Prolog benchmark with meta-calls.

`hamilton:` all the hamiltonian paths of length 7 in a fully connected graph of 10 nodes.

`matmult:` multiplication of a matrix 400x400 by a vector of 400 elements (AND-parallelism).

`primes:` a Prolog formulation of the Eratostenes's sieve for computing the prime numbers.

`logsim:` an event-driven logic simulator executed in "inverted" mode to act as a fault-finder in a two-bit adder. It exploits K-LEAF lazy evaluation [Bosco88].

`image:` the high level phase of an image understanding system described in [Moiso89].

According to our principle of a *user controllable parallelism* the parallel versions of some programs (`queens*`, `fib`, `image`, `logsim`, `hamilton`, `s&m`) have been quite straighforwardly derived from the sequential ones, by identifying the sources of useful parallelism and putting the right annotations on OR-parallel definitions and AND-parallel goals. In particular in `queens*` and `s&m` the test phase has been kept sequential. Moreover, to `queensi`, `queenss` and `fib` an extra parameter has been added in order to play with the granularity of parallel processes. This parameter represents the threshold below which the computation must proceed sequentially, executing the sequential (non-annotated) program. So for example $fib(22)^{10}$ means that recursion will generate parallel processes only down to $fib(10)^{10}$.

4.1 Performance of the sequential implementation

The emulated version (**C-emul** on Sun3/280) it is 3-4 times slower than a compiled (into machine language, as Quintus 2.2) Prolog.

Program	C-exp Sun3	Quintus 2.2 Sun3	C Sun3	C-exp T800	C-exp Dec3100	C-emul Sun3	C-emul T800
fib(29)	21300	47000	4800	106844	9900	190380	755949
queens1(9)	5150	12000	1000	17446	1810	34800	117944
primes(1000)	6100	21700	2600	21740	2510	76960	226256

The most interesting aspect is the viability of a portable compilation route from K-LEAF and Prolog directly to C (**C-exp**). The efficiency of this experimental compiled code is comparable with Quintus2.2. This is mainly due to the highly sophisticated C compilers available for RISC

machines which make less and less economic the compilation from high level languages into machine code. Better results come out when the compilation also includes some simple optimizations like choice-point elimination in case of *if-then-else* constructs and mapping of arithmetic expressions directly into the corresponding C expressions, after dereferencing variables.

Comparison of **C-exp** with the **C** column (with a rate from 1 to 5), where the execution times of efficiently implemented C imperative versions of the same algorithms are reported, suggests the applicability of well compiled and optimized logic/functional languages to wider fields of programming than the traditional *rapid prototyping* or *AI applications*.

A look to the Dec3100 column is in order. Its absolute figures (400KLIPS on a 10-11 MIPS machine; a 20MIPS version is already available) should be carefully considered before undertaking the adventure of designing a brand new sequential specialized processor.

4.2 Performance of the parallel implementation

Along this section we will concentrate on the parallel implementation of K-LEAF on the 16-nodes PIPES prototype. Most of the measurements were conducted on the system of distributed C-emulated K-WAM's, each being an OR-parallel extension of the sequential K-WAM.

Program	Tseq	T1	T16	T1/T16	T1/Tseq	Tseq/T16	Nproc
queensp(8)	32897	43322	3080	14.06	1.32	10.68	5509
queensp(9)	154889	208191	13937	14.94	1.34	11.11	24012
queens1(8)^{0}	23167	25071	1840	13.62	1.08	12.59	736
queens2(8)^{0}	9635	10982	960	11.43	1.13	10.03	736
fib(22)^{10}	26367	27701	1867	14.85	1.05	14.13	1218
logsim		106873	8106	13.18			2081
s&m	2265	2312	200	11.56	1.02	11.32	1024
image	24018	24040	2400	10.01	1.00	10.00	133
hamilton		17848	1370	13.02			720
matmult(400)	83916		6765			12.40	401

Good quasi-linear speedups are exhibited by programs with a great deal of potential parallelism (OR for `queens*`, AND for `fib`) and limited communication. Matrix multiplication (AND parallelism) in its original version on lists hardly achieved a speedup, due to the high rate of remote accesses to the node holding the matrix. The final version adopts a structured representation for matrix rows, which causes a complete row copying upon the first remote access.

The distributed binding scheme seem to cover quite well both strict K-LEAF (and Prolog) and lazy K-LEAF programs (`queensp`, `logsim`). The distributed search in binding list that was feared to be a serious bottleneck in highly lazy programs (where continuations and their result variables are built on the heap) is accceptably efficient when mixed with OR-parallelism. In many cases *lazyness* in expanding the search space means *locality* in binding lists and shorter searches.

In the last column the number of generated processes is included. This is an indication of the amount of parallelism present in the algorithm and can serve to understand the differences in speedup for the different programs. As a matter of fact `image` has a low inherent parallelism.
Comparison among sequential times **Tseq** (sequential programs by sequential executor on a T800) and parallel ones (**Tseq/T16**) give absolute speedups which, even for the benchmarks with stronger application flavour like `logsim` and `image` are beyond the critical efficiency threshold of 50% which is considered the threshold of interest for parallel machines. The column **T1/Tseq** gives the cost of parallelization. It reaches its maximum (35%) when most of the variables have to be searched in the binding list (like in `queensp`(8)) and granularity is not controlled (5509 generated processes). In the other cases, where suitable sequential pieces can be guaranteed (like

the safe test in `queens1`) the overhead can be kept less than 10%. The low overhead, 5%, imposed by the parallelization of `fib` is an indicator of an effective mapping of AND-parallelism into OR-parallelism by *parsetof*.

The following table reports the parallel performance of `queens1` and `fib` as functions of the granularity, which is controlled by an extra parameter. This is mirrored by the number of generated processes (**Nproc**) and the average size (**grain = Tseq/Nproc**). The sequential test in `queens1` guarantees large grainsize. The impact of granularity is more evident in `fib`, whose body is only made of three arithmetic operations and two calls. Parallelization efficiency around 50% (best speedup / 2) is obtained with grainsize of 0.9-1 milliseconds.

Program	Nproc	grain	T16	Tseq	Tseq/T16
queens1(8)^{0}	736	31	1840	23167	12.60
queens1(8)^{4}	356	65	1690		13.70
fib(20)^{2}	21890	0.46	2300	10071	4.37
fib(20)^{3}	13528	0.74	1750		5.75
fib(20)^{6}	3192	3.15	955		10.54
fib(20)^{10}	464	21.70	750		13.43

C-expansion on the PIPES architecture was tried in order to be able to show the possibility of getting a better absolute performance. However a quite limited effort has been devoted to it, only on sequential standard WAM instructions, which form the core of both parallel Prolog and K-LEAF systems, while the parallel structure is still generated by emulated instructions.

Program	TQuintus Sun3/280	Tseq C-exp	T1 C-exp	T16 C-exp	T1 / T16 emulated	T1 / T16 C-exp
queens1(9)^{5}	12000	17446	19482	1637	15.00	11.90
fib(29)^{15}	47000	106844	111372	7750	15.30	14.37

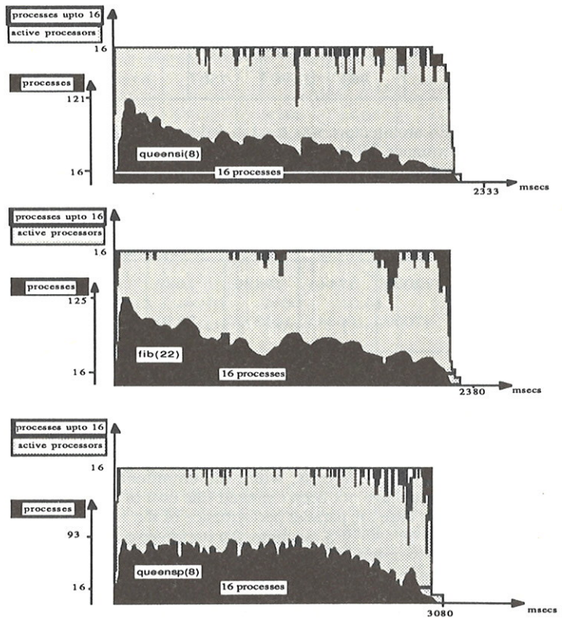
As regards the speedups reported in the last double column, as expected, *compiled* speedups are worse than *emulated* ones. One reason is the finer granularity of compiled sequential parts. This is reasonably compensated by "scaling up" the problems of a factor comparable with the compilation speedup, for example moving from queensi(8)^{4} to queensi(9)^{5}. Some inefficiency is still present, principally due to the increased rate of remote accesses over sequential execution. This is because we have faster sequential threads, while the communication run-time support (C procedures, high priority threads) have not been changed (the cost of a remote access remains the same, approx. 200µsecs). In any case the parallel performance of the 16-node machine is still more than 6 times that of Quintus2.2 on a 4MIPS machine. An improvement of this part is still possible by coding critical parts of the concurrent run-time support in assembly code. A final implementation should introduce *context-switching* to compensate network latency on remote accesses.

The number of accesses to nonlocal memories are reported below for some benchmarks. For sake of simplicity we have condensed into a single number the sum of the various kind of such remote accesses, namely: accesses to read the *content* of a variable, accesses to get the *top level of a structure*, accesses to remote *binding lists*, accesses to a remote *frame*. Their cost in the current prototype is quite high: it can be uniformly approximated to ~200µsecs (with ~40µsecs taken by physical communication of 40 bytes, and 160µsec taken by 4 switching's of Parallel-C threads).
Since the number of remote accesses is very much program dependent, reflecting the way structures are allocated throughout the program evolution, the induced overheads (**Tacc = rem-acc*200µsecs**), calculated w.r.t. **T1**, are significant only for the particular programs considered. They are rather small for the lazy and strict versions of `queens`, while, in the case of `logsim`, a lazy program largely dealing with streams communicating nested data structures, the

non-local accesses to closures and data make the overhead more significant (15%).

Program	Nproc	PE=8		PE=16		
		rec-proc	rem acc	rec-proc	rem acc	Tacc/ T1
queensp(8)	5509	48	4200	123	5500	.025
queens1(8) ^{0}	736	29	1230	201	6500	.052
fib(22) ^{6}	8360	27	80	45	135	0
logsim	2081	84	52150	230	79800	.149

The **rec-proc** columns report the number of processes transferred, according to the load balancing algorithm whose efficiency is shown by the following diagram. It reports on the lower scale the number of *ready processes* present in the architecture and in the upper scale the variation of such a number in the range 0-16, coupled with the number of *active processors* (grey area). The upper black area is a measure of inefficiency in trying to keep processors working on available tasks.



4.3 Lazy vs. eager computations

A quick comparative analysis of benefits and drawbacks of outermost (lazy) resolution of the functional component of K-LEAF is tried in the following.

Three kinds of comparisons are in order about **1)** the "programming" power offered by laziness, **2)** the capability of better controlling the exploration of a search space, **3)** the overheads induced by lazy computations.

Program	T1	T16
queensp(8) lazy	43322	3080
queensp(8) eager	559515	39270
queenss(8) ^{0}	13764	1140
logsim_2 lazy	68829	5410
logsim_2 eager	255192	19309

The two selected benchmarks are witnesses of the ability of a lazy L+F language to clearly and simply express the problem: a pure *generate and test* problem in the case of `queensp`, a (cyclic) network of stream-connected agents in the case of the logic simulator (inverted to work as a fault finder).

Regarding point **2)** comparison between eager and lazy computations shows the usefulness of laziness in performing a lazy expansion of the search space. Analogous results can be obtained, only in the sequential case, by Prolog systems with dynamic strategies, like Nu-Prolog [Naish85], but at the cost of an awkward annotation of the program.

However, coming to point **3),** this does not mean that to write a Prolog program with the same search behaviour is impossible; `queenss` is such a program where the generate and test phases have been intermixed in order to explore exactly the same search space. But this has been obtained at the cost of a clever program transformation and in general this may happen to be extremely complex. Anyway in this case, and in most of the cases, where a lazy strategy (and thus closure construction on the heap) is adopted, but not necessary, the overhead is in the range 200%-300%, both in the sequential and the parallel environment. Among such programs we also consider those compiled as fully lazy, in spite of the fact that they are inherently strict.

5. Related works

While a substantial experience is becoming available on the implementation of logic languages on shared memory parallel architectures, very few attempts on distributed structures have been tried and already evaluated.

As for the *shared memory* approach, an excellent work has been carried on by the PepSys Project [Baron88] and the GigaLips Project [Szeredi89] both based on SRI model.

The implementation of Aurora on shared-memory multiprocessors is in an advanced state [Szeredi89] showing linear speedups with high efficiency and limited cost of parallelization (<30%). The first rough comparison seems to indicate that (for 11 nodes of a Symmetry) they gain a 5-10% in relative speedup (perhaps due to the advantages of shared memory) while our absolute speedups are better (5-10%) due to a lower cost of parallelization, thanks to our annotations. It is worth to note that Prolog programs can be directly executed in parallel by Aurora (even if no annotations are introduced), while in our case they must be slightly modified (e.g. no cut in a parallel procedure is allowed) because of the stronger nature of our annotations and of the more rigorous interface among sequential and parallel computations, but, on the other hand, such annotations can guarantee a better control on the granularity of computations.

Regarding AND-parallelism, a *shared memory* implementation has been developed on the Symmetry at MCC [Hermenegildo89]. The only benchmark reported is matrix multiplication which has a linear speedup with 95% efficiency on a 10-node Symmetry. In our case, where pieces of the matrix have to be transmitted along the network, efficiency is around 80%.

W. r. t. shared memory architectures, the very similar nature of results tells us that the gain in scalability we get by moving from shared to distributed memory does not necessarily involve dramatic falls of performance. This is true if the programmer is conscious of working in a distributed parallel environment and able, through a suitable programming style [Butler88] and simple annotations [Hirschmann88], to control the evolution of parallel computations.

Among the approaches to a (almost-)distributed memory implementation of parallel Prolog, the BC-machine [Ali88] seems to have reached a good stage of development. It is based on extensive copying instead of a logically shared cactus stacks. Preliminar results [Ali89] on a Sun3/50 array show better (10%) absolute speedups on standard OR-parallel benchmarks, with a simpler (than SRI's) parallelization scheme. However, it is not yet clear how much this approach is applicable to a variety of programs, where the OR-branching is not always performed at the beginning.

A recent attempt to put Aurora on a switch-based multiprocessor is described in [Mudambi89] where the target machine is BBN Butterfly GP1000 (upto 40 processors) which is similar in spirit to our machine. Their cost of parallelization seems to be very high (more than 100% overhead w.r.t. SICStus) compared with ours (at most 30% on heap-intensive programs); moreover for queens(8), the only benchmark analyzed in detail, relative speedup (T1/T15) is 7.4 and absolute one is only 3.68. The speedup efficiency is thus still considerably low.

Our results on the implementation of Prolog and K-LEAF on a distributed architecture show that exploitation of OR and AND parallelism inherent in logic and functional languages is possible on *highly-scalable* architectures, with efficiencies (w.r.t sequential compiled code) comparable with those obtained on bus-based machines, provided that : 1) the network efficiently support both block transfer and non-local memory accesses in an any-to-any mode, e.g. by "cut-through" packet switching" (note that the current PIPES prototype does not have hardware support for processor-to-non-local-memory interface and context-switch at WAM level); 2) the programmer has a way to control, in an easy and disciplined way the grainsize of sequential computations; 3) in some cases, the programmer is conscious of the distributed nature of the machine, and knows how to accordingly tune the size of data structures transferred by a non-local access (like in `matmult` benchmark).

Acknowledgment

This work has been partially supported by ESPRIT Project n. 415.

6. References

[Ali88] **K. Ali,** OR-parallel execution of Prolog on BC-machine, Proc. 5th Int. Conference and Symposium on Logic Programming (MIT Press, 1988), 1531-1545.

[Ali89] **K. Ali,** Presentation at GigaLips Project meeting, April 1989 (1989).

[Balboni87] **G. P. Balboni, G. Giandonato and R. Melen,** A parallel architecture for AI-based real-time applications, in Proc. of 1987 AFCEA European Symposium (1987).

[Baron88] **U. Baron, J. Chassin de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J-C. Syre and H. Westphal,** The parallel ECRC Prolog system PEPSys: an overview and evaluation results, Proc. FGCS'88 (Nov. 1988), 841-850.

[Bosco86] **P.G. Bosco and E. Giovannetti,** IDEAL: An Ideal DEductive Applicative Language, Proc. 1986 Symp. on Logic Programming (IEEE Press, 1986), 89-94.

[Bosco87] **P.G. Bosco, E. Giovannetti, G. Levi, C. Moiso and C. Palamidessi,** A complete semantic characterization of K-LEAF, a logic language with partial functions, Proc. 1987 Symp. on Logic Programming (IEEE Press, 1987), 318-327.

[Bosco88] **P.G. Bosco, C. Cecchi and C. Moiso,** Exploiting the full power of logic plus functional programming, Proc. 5th Conf. on Logic Programming (MIT Press, 1988), 3-17.

[Bosco89a] **P.G. Bosco, C. Cecchi and C. Moiso,** IDEAL & K-LEAF implementation: a progress report, in Proc. PARLE '89, LNCS 365 (Springer-Verlag, 1989), 413-432.

[Bosco89b] **P.G. Bosco, C. Cecchi and C. Moiso,** An extension of WAM for K-LEAF: a WAM based compilation of conditional narrowing, in Proc. 6th Conf. on Logic Programming, (MIT Press, 1989), 318-333.

[Butler88] **R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek and R. Stevens,** Scheduling OR-parallelism: an Argonne perspective, Proc. 5th Conf. and Symp. on Logic Programming (MIT Press, 1988), 1590-1608.

[Carlsson88] **M. Carlsson, K. Danhof and R. Overbeek,** A simplified approach to the implementation of AND-parallelism in an OR-parallel environment, Proc. 5th Conf. and Symp. on Logic Programming (MIT Press, 1988), 1565-1577.

[Giandonato88] **G. Giandonato and G. Sofi,** Parallelizing logic programming based inference engines, Proc. of the International Conference on Supercomputing (1988), 282-287.

[Hermenegildo89] **M. Hermenegildo, K. Muthukumar, K. Greene, F. Rossi and R. Nasr,** An overview of the PAL project, MCC Technical Report ACT-ST-234-89 (1989).

[Hirschmann88] **L. Hirschmann, C. Hopkins and R. Smith,** OR-parallel speed-up in natural language processing: a case study, Proc. 5th Conference and Symposium on Logic Programming (MIT Press, 1988), 263-279.

[Merlo88] **C. Merlo, C. Moiso, M. Porta and G. Sofi,** Parallel Prolog for signal-understanding parallel machines: implementation and applications, ESPRIT Pilot Project n. 26, Deliverable 14b (Sept. 1988).

[Moiso89] **C. Moiso, M. V. Oneto, M.Porta and G. Sofi,** Applicazione di un linguaggio logico parallelo: riconoscimento di oggetti bidimensionali, in Proc. 4th National Congress on Logic Programming (Gulp '89), 405-419.

[Mudambi89] **S. Mudambi,** Performance of Aurora on a switch-based multiprocessor, Proc. 1st North-American Conf. on Logic Programming (MIT Press, 1989), 697-709.

[Naish85] **L. Naish,** Automating control for logic programs, Journal of Logic Programming 3 (1985), 167-183.

[Szeredi89] **P. Szeredi,** Performance analysis of the Aurora OR-parallel Prolog system, Proc. 1st North-American Conf. Logic Programming (MIT Press, 1989).

[Warren87] **D.H.D Warren**, The SRI Model for OR-Parallel Execution of Prolog. Abstract Design and Implementation, Proc. 1987 Symp. on Logic Programming (IEEE Press, 1987), 92-103.

[3L88] **3L Ltd,** Parallel C User Guide, Livingstone, Scotland (1988).

Verso un Supporto a Tempo di Esecuzione per Linguaggi Logici e Funzionali

Mauro Gaspari
Franco Saracco

DELPHI
Via Della Vetraia, 11,
Viareggio, Italia.

Net: mgaspari@delphi.uucp
fsaracco@delphi.uucp

Sommario

In questo articolo viene descritta un'implementazione del Prolog in un ambiente COMMON LISP. L'implementazione è basata su una estensione della macchina astratta Lisp con primitive della macchina astratta di Warren. Il concetto di variabile logica è stato introdotto in COMMON LISP. La nuova Macchina Astratta garantisce una piena interoperabilità tra i due linguaggi; il supporto a tempo di esecuzione è comune.

1. Introduzione

Gli studi nel campo dell'Intelligenza Artificiale hanno dato origine ad un insieme di nuove tecniche di programmazione che possono essere utilizzate per arricchire ed integrare le tradizionali tecniche e metodologie per lo sviluppo del software. In particolare, per quanto concerne i linguaggi di programmazione, questo tipo di studi ha contribuito alla nascita di linguaggi basati su nuovi paradigmi di programmazione, quali quello funzionale, logico, ad oggetti. Sostanzialmente le tecnologie di IA permettono di elevare il livello di astrazione a cui vengono scritti i programmi, colmando in parte l'abisso che esiste tra il cervello dell'uomo e la macchina di Von Neumann. Dal punto di vista implementativo questi linguaggi sono caratterizzati da diversi e specifici supporti a tempo di esecuzione la cui complessità è maggiore se maggiore è la complessità del linguaggio. Da quest'ultimo dipendono inoltre l'ambiente di sviluppo e l'insieme di strumenti di progammazione associati.

D'altro canto applicazioni complesse come la maggior parte di quelle che si suole definire di Intelligenza Artificiale, richiedono che parti diverse di codice vengano scritte con linguaggi diversi. Questo sostanzialmente per due motivi: il primo è che un linguaggio di programmazione può risultare più adeguato di un altro per risolvere un certo sottoproblema, il secondo è basato

riferimento alle problematiche di integrazione tra la Warren Abstract Machine (WAM, la macchina astratta su cui girano le più veloci implementazioni del Prolog [9,27]), e la Lisp Abstract Machine. Nel paragrafo successivo viene descritta un' implementazione del Prolog nell'ambiente basato sul COMMON LISP, utilizzando il STE comune. Infine nel paragrafo 4 vengono presentati alcuni *benchmarks* tramite i quali l'attuale implementazione viene paragonata alla versione 2.4.2 del Quintus Prolog.

2. Supporto Comune a tempo di esecuzione.

Nella progettazione del STE sono state individuate le funzionalità che devono essere comuni ai diversi linguaggi di programmazione che lo utilizzano:

- Threads (possibilità di avere flussi di esecuzione concorrenti);
- Input/Output a basso livello (compresi sockets di rete);
- Gestione della memoria (possibilità di allocare dinamicamente memoria in modo uniforme in tutti i linguaggi di programmazione supportati; garbage collection universale);
- Gestione della tabella dei simboli (possibilità di *linking e loading* statico e dinamico);

Tutte queste funzionalità sono state implementate in C. É stato implementato il concetto di thread (light weight process) su un ambiente funzionale, descritto con la semantica delle continuazioni [5]. La macchina astratta del Lisp è stata implementata in C. Il compilatore del COMMON LISP è un compilatore da Lisp a C: i sorgenti Lisp vengono tradotti in codice C; successivamente, utilizzando il compilatore C, in codice della macchina target. Inoltre, sono parte del STE e sono state implementate in C alcune funzionalità di CLOS [4]: le funzioni che riguardano il tipo *instance* (il tipo di dato utilizzato per implementare tutti gli oggetti di CLOS e, come vedremo, le strutture Prolog), e quelle che riguardano la gestione delle funzioni generiche.

La struttura completa dell'ambiente di programmazione risultante si può vedere in figura 1. La parte del STE che riguarda l'integrazione tra macchina astratta Lisp e la WAM è descritta ampiamente nei prossimi paragrafi.

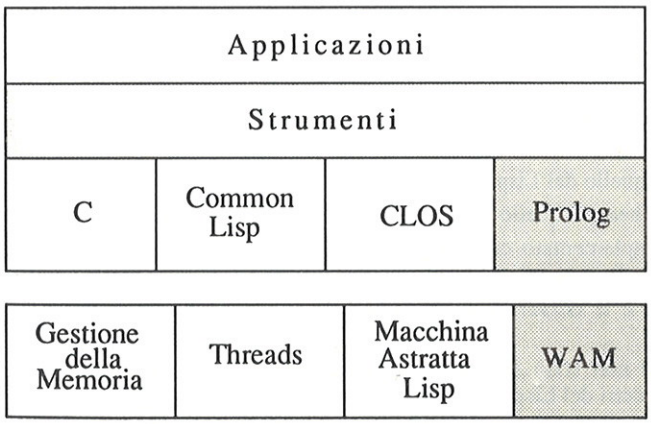


Figura 1

2.1 Integrazione con la WAM.

Una descrizione più dettagliata della macchina di Warren si può trovare in [9,27]. In questo paragrafo ci si limita a presentare brevemente le componenti principali della WAM: le aree dati e l'insieme delle istruzioni della macchina.

Le aree dati della WAM sono:

•L'*area codice* che contiene il codice da eseguire.
•L'*area di controllo* che contiene i registri della macchina astratta.
•Tre aree utilizzate a stack:
- Lo *stack locale* che contiene informazioni per gestire il *backtracking* e le chiamate ricorsive di procedura.
- Lo *stack globale* o (*heap*) che contiene le strutture dati create durante l'esecuzione.
- Il *trail* che contiene i riferimenti a variabili che possono essere legate e che devono essere annullati in caso di *backtracking*.

I programmi Prolog vengono compilati in sequenze di istruzioni, approssimativamente una istruzione per ogni simbolo Prolog.

L'insieme di istruzioni della WAM è diviso in:

•*Istruzioni_get*: corrispondono agli argomenti della testa di una procedura e servono per unificare tali argomenti con i parametri passati alla procedura.
• *Istruzioni_put:* corrispondono agli argomenti del corpo di una procedura e servono per caricare tali argomenti nei registri.
•*Istruzioni_unify:* corrispondono agli argomenti di una lista o di una struttura e possono operare in *modo_lettura* o in *modo_scrittura*. L'istruzione viene messa in *modo_lettura* o in *modo_scrittura* dall'istruzione *get* che la precede, a seconda che la lista o la struttura relativa esista già o meno.
• *Istruzioni procedurali*: corrispondono alla testa e al corpo di una clausola e servono a trattare l'ambiente e i trasferimenti di controllo.

Nel seguito si può vedere come le componenti della WAM sono integrate nella nostra implementazione:

• L'Area codice è quella del Lisp.
• L'Area controllo è quella del Lisp.
• Lo stack locale è quello del Lisp.
• Lo stack globale (heap) è quello del Lisp.
• Il *trail* è stato implementato come stack in C, e le funzioni che lo gestiscono fanno parte del STE e sono descritte in seguito.
• Le *istruzioni_get* e le *istruzioni_unify* sono state implementate in C nel STE, e sono descritte in seguito.
• Le *istruzioni_put* non sono state implementate perchè sono già implicite nel meccanismo del passaggio dei parametri del Lisp.
• Le *istruzioni procedurali* sono quelle del Lisp più quelle necessarie alla gestione del *trail*.

2.2 Il tipo di dato locative.

Le variabili logiche vengono legate durante il processo di unificazione. Tali legami possono poi dover essere annullati in un secondo tempo, quando un fallimento in una deduzione dà origine ad un *backtracking*, o quando si richiede una nuova soluzione. Nel processo deduttivo vengono generate un certo numero di variabili. Molte di esse sono temporanee o vengono legate a qualche altra variabile o ad uno slot di una particolare struttura; in questo caso la variabile conterrà una indirezione. É chiaro quindi che vi sarà un notevole risparmio di memoria se per tali variabili non verrà allocato alcuno spazio. Questo scopo si può raggiungere rappresentando le variabili con puntatori alle locazioni che contengono lo slot o il valore al quale tale variabile verrà legata.

Poichè il COMMON LISP non offre la possibilità di gestire puntatori ad uno slot di un altro oggetto, si è pensato di introdurre nello STE un nuovo tipo di dato, adatto allo scopo, chiamato *locative*, simile al tipo di dato *locative* usato nelle Lisp Machines [19]. Un *locative* è un oggetto Lisp usato come puntatore ad una singola cella di memoria, ed implementato come dato immediato, in modo da non occupare memoria nello heap.

Per permettere l'utilizzazione dei dati immediati nel Common Lisp è stato introdotto uno schema di *tagging* che utilizzando gli ultimi due bits di un indirizzo come *tag* identifica il tipo dell'oggetto rappresentato. Poichè tutti gli altri oggetti Lisp richiedono quantità di memoria multiple di 4 bytes, gli oggetti Lisp non immediati sono allineati in memoria su limiti di 4 bytes, e quindi, per macchine con indirizzamento al byte, gli ultimi due bits dell'indirizzo sono sempre zero.

In figura 2 viene riportata la tabella di *tagging* usata in COMMON LISP.

2.3 Istruzioni della WAM implementate.

Vengono di seguito elencate le istruzioni della WAM che sono state implementate nel STE.

Istruzioni_get:

get-variable v x [forma speciale]
funzionamento identico alla forma *setq*, viene cioè assegnato alla variabile *v* il valore *x*;

get-value v x [funzione]
i valori di *v* e di *x* vengono unificati. La funzione ritorna *t* se ha successo, altrimenti *nil*;

Tipo	Etichetta
fixnum	01
locative	10
character	11
pointer	00

Figura 2

get-constant c x [funzione]
il valore di *x* viene unificato con la costante *c*. La funzione ritorna *t* se ha successo, altrimenti *nil*;

get-nil x [funzione]
il valore di *x* viene unificato con la costante *nil*. La funzione ritorna *t* se ha successo, altrimenti *nil*;

get-cons x [funzione]
il valore di *x* viene unificato con una cella cons. La funzione ritorna *t* se ha successo, altrimenti *nil*;

get-instance x classe arietà [funzione]
il valore di *x* viene unificato con una istanza della struttura *classe* con *arietà* numero di argomenti. La funzione ritorna *t* se ha successo, altrimenti *nil*.

Istruzioni unify:

Le istruzioni di unificazione corrispondono agli argomenti di una lista o di una struttura e possono operare in *modo_lettura* o in *modo_scrittura*. Per esempio, se x contiene una cella cons, l'istruzione (get-cons x), metterà la WAM in *modo_lettura*. Dopo l'istruzione get, vi saranno due istruzioni di unificazione che legheranno gli slots della cella (prima il *cdr*, quindi il *car*, a causa dell'implementazione fisica delle celle *cons*). Se invece x contiene una variabile non istanziata, la *get* legherà questa variabile ad una nuova cella *cons* creata, e metterà la *WAM* in *modo_scrittura*. In *modo_scrittura* le istruzioni di unificazione assegneranno agli slots gli opportuni valori risultanti dall'unificazione. La funzione *get-instance*, che serve per trattare le strutture Prolog, ha un funzionamento analogo alla *get-cons*. Per tale funzione, si è adottata la dizione *get_instance* poichè le strutture dati con cui opera sono le cosiddette *"instances"*, utilizzate per implementare le strutture Prolog, le strutture Lisp e le istanze di CLOS.

unify-variable v [macro]
Modo_lettura: *v* viene assegnato al prossimo sottotermine;
Modo_scrittura: una nuova variabile viene memorizzata in *v* come prossimo sottotermine;

unify-value v [funzione]
Modo_lettura: il valore di *v* viene unificato col prossimo sottotermine;
Modo_scrittura: il valore di *v* viene memorizzato come prossimo sottotermine;

unify-constant c [funzione]
Modo_lettura: il prossimo sottotermine viene unificato con la costante *c*;
Modo_scrittura: la costante *c* viene memorizzata come prossimo sottotermine;

unify-nil [funzione]
Modo_lettura: il prossimo sottotermine viene unificato con la costante *nil*;
Modo_scrittura:la costante *nil* viene memorizzata come prossimo sottotermine.

Istruzioni procedurali:

Trail:

trail-mark [funzione]
fissa un punto di scelta mettendo una marca sul trail per il *backtracking*;

trail-restore [funzione]
scarica il trail fino all'ultimo punto di scelta;

trail-unmark [funzione]
fa un *trail-restore* e rimuove l'ultimo punto di scelta.

Locatives:

make-locative n [funzione]
Crea un locative di indirizzo *n*.

make-variable nome [funzione]
crea una nuova variabile logica con nome *nome* (usato a scopo di stampa);

dereference locative [funzione]
dato un locative ritorna l'oggetto al quale esso punta.

unboundp locative [function]
controlla se una variabile logica è legata o meno.

3. Un' implementazione del Prolog.

L'implementazione descritta è basata sulla tecnica delle *stack oriented downward success continuations*, che è una delle tecniche più frequentemente utilizzate per implementare il Prolog in ambienti procedurali ed è ampiamente descritta in [17,19,22,24], e sul concetto di *set abstraction* [12,15,23]. In ambiente COMMON LISP la tecnica delle *stack oriented downward success continuations* consiste nell'associare una funzione Lisp a ciascuna procedura Prolog. La funzione Lisp, che restituisce vero o falso, ha gli stessi argomenti della procedura Prolog più una continuazione da eseguire qualora la funzione abbia successo. La tecnica in base alla quale viene costruita la funzione è ispirata al concetto di *set abstraction*, utilizzato in altre proposte di integrazione tra linguaggi logici e funzionali [12].

Con tale tecnica alle clausole:

```
append([],X,X).
append([X|Y],Z,[X|W]) <- append(Y,Z,W).
```

è associata la seguente equazione che descrive l'insieme delle sue soluzioni:

```
append(X1,X2,X3) = {X1,X2,X3 | (X1 = [] and X2 = X and X3 = X) or
                               (X1 = [X|Y] and X2 = Z and
                               X3 = [X|W] and append(Y,Z,W)) }
```

Il corpo principale della funzione è dunque un *or* di forme che rappresentano le singole clausole associate alla procedura. Ognuna di tali forme è un'espressione *and* che contiene le istruzioni di unificazione della WAM e la chiamata della continuazione opportuna. Quest'ultima viene invocata solo quando l'*and* delle unificazioni ha successo.

Nella figura 3 è descritta un'implementazione della procedura Prolog *append* che concatena due

```
reverse(Xs,Ys)
        The list Ys is the result of reversing
        the list Xs.

naive_reverse([],[]).
naive_reverse([ X1| Xs], Zs) <-
            naive_reverse(Xs,Ys),
            append(Ys,[X1],Zs).
```

La funzione Lisp corrispondente è:

```
(defun naive_reverse (x y cont)
   (trail-mark)
          (or
            (and
                  (get-nil x)
                  (get-nil y)
                  (funcall cont))
          (trail-restore)
                   (let (X1 Xs (Ys (make-variable)))
                      (and
                           (get-cons x)
                           (unify-variable Xs)
                           (unify-variable X1)
                           (naive_reverse Xs Ys
                              (funarg (lambda()
                                  (append Ys
                                    (cons X1 nil)
                                     y cont)))))))
   (trail-unmark))
```

Figura 4.

è quindi necessario dichiarare, come nel caso delle funzioni a valore insieme (implementate da funzioni COMMON LISP che restituiscono più valori) il modo (ingresso o uscita) delle variabili di un predicato.

3.1 Compilatore

Si è visto che una procedura Prolog è realizzata attraverso una particolare funzione Lisp.

Chiamiamo compilazione il processo di traduzione da un insieme di fatti e/o regole che costituiscono una procedura Prolog nella funzione Lisp che la realizza. In particolare, nel processo compilativo, possono essere evidenziati tre passi fondamentali:

Per esempio, il seguente predicato:

```
cut(X,Y) :- diff(X,john ),!,bar(Y).
cut(john,paul).
```

viene tradotto in:

```
(defun cut (x y cont)
 (trail-mark)
 (catch 'cut-tag
  (or
    (diff x 'john
      (funarg (lambda ()
        (throw 'cut-tag (bar y cont)))))
    (trail-restore)
    (and
      (get-constant 'john x)
      (get-constant 'paul y)
      (funcall cont))))
    (trail-unmark))
```

Figura 6

L'effetto dell'operatore *cut* è stato raggiunto col meccanismo *catch/throw* del Common Lisp. Un esempio si può vedere in figura 6.

Alcuni predicati predefiniti del Prolog vengono implementati direttamente dai predicati Lisp, ovvero ad essi vengono associate delle funzioni Lisp che semplicemente invocano la continuazione che rappresenta il resto del programma se il corrispondente predicato Lisp ha successo.

Nell'esempio in figura 7, la funzione *prolog-=<* realizza il predicato Prolog =< semplicemente utilizzando la funzione Lisp <=:

```
(defun %prolog-=<% (left right cont)
 (if (<= left right)
       (funcall cont)
  nil))
```

Figura 7

Un altro esempio interessante è il modo in cui viene implementato l'usuale valutatore aritmetico fornito dal Prolog. Nell'esempio in figura 8 si può vedere la funzione Lisp *prolog-is* che implementa il predicato aritmetico Prolog *is* utilizzando le funzioni messe a disposizione dal STE, quali *locativep*, *unboundp* e *get-value*:

5. Conclusioni e sviluppi futuri.

La soluzione scelta per supportare variabili logiche ed unificazione è stata quella di estendere la macchina astratta Lisp con un nuovo tipo di dato (*locative*) e con le primitive necessarie per supportare la WAM. La nuova Macchina Astratta estesa è capace di supportare entrambi i linguaggi fornendo la interoperabilità tra di essi. Ciò significa che i programmi nei due linguaggi possono essere facilmente integrati: un programma scritto in uno di essi può chiamare una routine scritta nell'altro, senza bisogno di cambiare le rappresentazioni dei dati. Lo STE per entrambi i linguaggi è lo stesso.

L'implementazione chiamata DELPHI-Prolog (D-Prolog) è stata realizzata in DELPHI COMMON LISP (DCL), un' implementazione del COMMON LISP che comprende un'interfaccia con la grafica (CLX e/o OSF/Motif), la programmazione ad oggetti (CLOS) e primitive per gestire threads concorrenti di esecuzione. La sintassi, è quella *standard* del Prolog di Edinburgo. D-Prolog sarà disponibile come prodotto a partire dall' Aprile 1990.

Come abbiamo visto un predicato Prolog può chiamare funzioni Lisp, aprendo quindi l'ambiente Prolog a facilitazioni offerte dall'ambiente DCL, quali la grafica o la programmazione ad oggetti. É da notare inoltre che i termini Prolog hanno la stessa rappresentazione degli oggetti di CLOS. Nascono quindi diverse possibilità, come quella di usare uno strumento efficiente come l'unificazione per fare interrogazioni su una base di dati di oggetti.

Occorre tener comunque ben presente che l'integrazione descritta può essere raggiunta anche in modo tradizionale, come interpretazione del linguaggio, costruendo cioè un Prolog interprete/compilatore-incrociato in Lisp o viceversa. Tale soluzione presenta però grossi problemi di performance. Un linguaggio diventa per forza di cose privilegiato rispetto all'altro: quello nativo usa le strutture dati e l'implementazione più adeguate, l'altro è implementato rappresentando le proprie strutture dati con quelle del linguaggio nativo e, cosa molto importante, l'interpretazione o compilazione incrociata dà un qualcosa abbastanza lontano dalla performance ottimale.

I benchmarks fatti hanno dato prestazioni inferiori ma paragonabili a quelle del Quintus Prolog. Poiche alcune funzionalià, per esempio quelle legate alla compilazione delle chiusure, non sono state ancora implementate nell'attuale versione, si possono prevedere ulteriori ottimizzazioni. A nostro avviso, quindi, la tecnica si dimostra interessante, oltre che per la stretta integrazione, anche dal punto di vista delle prestazioni.

Tra i futuri sviluppi da menzionare quello dello studio dell'integrazione del Prolog con le primitive del Multithred, con CLOS e col concetto di persistenza (PCLOS) [6]. Interessante sarebbe inoltre utilizzare l'ambiente per implementare sistemi per la rappresentazione della conoscenza come Omega [7] o MULTILOG [20,21].

Ringraziamenti.

Ci preme soprattutto ringraziare il Prof. Giuseppe Attardi che è stato il principale inspiratore di questo lavoro e Federico Passaro per la collaborazione nell'implementazione.

[15] M. Buonanno, "SVF-LOG: a Lisp Implementation os Horn Clauses", Proc. Fouth Italian National Conference on Logic Programming, (GULP), Bologna, 1989.

[16] M. Cavalieri, E. Lamma, P. Mello, "An Abstract Prolog Machine for Dynamic Context Handling", Proc. ECAI 88, August 1988.

[17] J. Cohen, "Describing Prolog by its Interpretation and Compilation", Communications of the ACM, N.12, December 1985.

[18] Delphi SpA, "Delphi Common Lisp, User Reference Manual", Release 1.9, June 1989.

[19] K. M. Kahn, M. Carlsson, "How to Implement Prolog on a Lisp Machine", in Implementations of Prolog, 1986.

[20] H. Kauffmann, A. Grumbach, "Representing and Manipulating Knowledge within Worlds", Proc. First International Conference on Expert Database Systems, 1986.

[21] H. Kauffmann, A. Grumbach, "MULTILOG: MULTIple worlds in LOGic Programming", Proc. ECAI 1986.

[22] C. Mellish, S. Hardy, "Integrating Prolog in The POPLOG Environment", in Implementations of Prolog, 1986.

[23] J. A. Robinson, E. E. Sibert, "LOGLISP: Motivation, Design, Implementation", Int. Report, School of Computer and Information Science, Syracuse University (USA) 1982.

[24] A. Sloman, "Poplog, a Portable Interactive Software Development Environment", Internal Report, Sussex University, May, 1989.

[25] G. L. Jr. Steele, "COMMON LISP: The Language", Digital Press 1984.

[26] M. E. Stickel, "A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler", Proceeding of CADE '86, Springer Lecture Note for Computer Science, N. 230, 1986.

[27] D.S. Warren,"An Abstract Prolog Instruction Set", Technical Note 309, SRI International, Menlo Park, October 1983.

[28] M. Weiser, A. Demers, C.Hauser, "The Portable Common Runtime Approach to Interoperability", Internal Report, Xerox Corporation, March 1989.

THE EPSILON KBMS: ENVIRONMENT AND APPLICATIONS

Patrizia Coscia and Sandra Valeri

Systems & Management SpA
Vicolo San Pierino, 4
56100 Pisa

Abstract

The aim of this paper is to provide a general introduction to the Epsilon Knowledge Base Management System and illustrate through examples some of the most relevant features of the system. We show the power of Epsilon as an open environment, which can be easily customized and extended to face different kinds of problems, by defining new inference engines and tools through meta-programming. We also show how Epsilon structuring facilities (theories and links between them) and predefined inference engines can be used to develop real-life applications, such as expert systems.

1. Introduction

Prolog turned out to be a powerful and flexible language, suitable to develop Artificial Intelligence applications, including Expert Systems [StSh86, Ster85, SaSh86]. Expert Systems are nowadays the most known and widespread Knowledge Based Systems. It is common opinion that, in the development of a KBS, knowledge acquisition is the most costly and delicate phase; in general, in the development of a KBS, considerable advantages can be obtained also by making software production and handling more effective. Indeed, in developing large and complex real life applications Prolog showed some problems. The first need which is felt is the lack of structuring; moreover, standard Prolog resolution strategy not always is the most suitable one: depending on particular application fields, different reasoning modalities can be needed. Moreover, some applications require the ability of handling efficiently large amounts of data.

The Epsilon KBMS [CFLS88, VCST89, Epsi90] aims to answer to these requirements by providing a programming environment based on a commercial Prolog. The main concepts underlying Epsilon are:

- Extension of Prolog with a suitable knowledge structuring mechanism (theories and links between them);
- Provision of several reasoning modalities and related support tools;
- Ability of defining new reasoning modalities and tools;
- Integration with (relational) database technologies.

1990. The result is a system implemented on BIM_Prolog and Informix® under the UNIX™ O.S.; the user interface runs on a Macintosh™.

2. Epsilon: a Knowledge Base Management System

A knowledge base in Epsilon is composed of a set of heterogeneous entities, called **theories**, which may be related one another. Each theory contains a chunk of knowledge expressed in a logic language or stored in a relational database.
Each theory belongs to a specific **class** which defines the inference mechanism and the set of primitive operations to manipulate the theory.
Furthermore, a class can define a set of support **tools** which are common to all theories belonging to that class. In other words, a class defines all the mechanisms needed to interpret and manipulate the knowledge contained in each theory belonging to that class.

A class is defined by means of meta-programming: a class is a theory itself, containing a meta-interpreter. Therefore, a theory can be either an "object" theory, or an "engine" theory. An engine theory *E* defines a meta-interpreter for a class *E* of theories by means of the predicate *demo(Goal,Theory[,ExtraArg])*, where *Goal* is the goal to be solved, *Theory* is a theory, belonging to the class *E*, where *Goal* has to be solved, and *ExtraArg* is an optional meta-argument (e.g. prooftree, number of inference steps) which is needed in certain meta-interpreters. Epsilon provides some predefined classes; moreover, users can define new classes.

Theories can communicate with one another, i.e. they may exchange knowledge, either explicitly, by addressing other theories through their names by using the *demo* predicate, or implicitly, by exploiting relationships defined between theories.
A basic mechanism, called **link**, establishes and supports relationships between theories. There is a set of predefined links (related to some predefined classes); moreover, users can define new kinds of links by specifying their semantics within the involved theory class(es).

The concepts of **theory**, **class** and **link** provide logic programming with the well known advantages of software modularization and permit the coexistence in a simple unifying framework of different knowledge representation formalisms and different forms of reasoning. Furthermore, the set of predefined classes provides the user with a library of meta-interpreters (that can be used as they are or enhanced).

The basic language with partitioning in theories and the ability to explicitly address theories (*kernel* class) is a realization of the language-metalanguage amalgamation, without simulation of the object level syntactic structure and with partial evaluation playing the role of one reflection principle. The semantics of links and of the other classes is defined within the engines.

2.1 Predefined classes

The main predefined classes in Epsilon are **kernel**, **uncert**, **inheritance** and **database**.

The **kernel** class supports the Prolog standard functionalities and the structuring in theories; it provides, as support tools, tracer, explanator, partial evaluator and integrity constraints checker.

The **uncert** class extends the Prolog inference mechanism with uncertainty factors handling.

The **inheritance** class extends the Prolog inference mechanism with inheritance of knowledge between theories, by exploiting the link feature. This class provides four different inheritance modalities, which correspond to four different links and can be split into two groups:

- "is-a" modality: an *inheritance* theory T1, linked to a theory T2 by an "is-a" link, inherits from T2 clauses which are evaluated in the environment of T1. If the modality is "closed is-a", only clauses related to those predicates which are not already defined in T1 are taken into account; if the modality is "open is-a", all clauses are inherited and possible predicates of T2 already defined in T1 are considered as extending the definitions in T1;
- "consultance" modality: an *inheritance* theory T1, linked to a theory T2 by a "consultance" link, queries the metainterpreter of T2. Subgoals are evaluated in the environment of T2. If the modality is "closed consultance", a query is asked only for predicates not defined in T1, otherwise for each predicate which fails.

Multiple inheritance between theories is allowed: a theory T1 can inherit knowledge from several theories, each one linked to T1 by one of the above mentioned links.
This class provides a set of tools including tracer, explanator, query-the-user, a tool for forward reasoning, partial evaluator and integrity constraints checker.

The **database** class allows large amounts of data, contained in a relational database, to be handled as theories. This kind of integration, on the one hand, allows the logic representation of knowledge to be extended with the ability to access external databases, on the other hand it allows relational databases to be manipulated through the usual logic language features.
Databases can be created, deleted or queried exactly like other theories; moreover, already existing external databases can be integrated in an Epsilon knowledge base. The *database* class provides, as support tools, a set of operations to insert (retract) facts into (from) the database, to insert, retract and modify database relations, to retrieve information on a specified relation from the data dictionary of the physical database, to insert tuples extracted from a database into a logic theory, to check integrity constraints. A *database* theory can either be used directly, or queried by other theories, or inherited by an *inheritance* theory by means of a "consultance" link.

3. Defining new classes of theories

One of the most important Epsilon features is that it is explicitly open to the introduction of new inference mechanisms and new knowledge representation languages. In order to introduce a new inference mechanism, users can define a new class, i.e. create a new theory declared to be an engine. Introducing new inference mechanisms or language extensions implies redefining the mechanism for knowledge querying (i.e. the predicate *demo*); this can be achieved by defining its behaviour by a meta-program within the class. Analogously, the mechanisms for knowledge updating may need to be modified; this involves redefining the predicates *asserta_t, assertz_t* and *retract_t* within a class.
The definition of new inference and updating mechanisms could involve the introduction and definition of new kinds of links, in order to support new relationships among the theories belonging to the new class. The existence of a link between two theories simply means that they are able to share knowledge: how/when knowledge can be shared and how/when it can be used must be defined within the inference or updating mechanisms of the involved theories (i.e. within their class).
Besides defining the inference and updating mechanisms, classes can define tools which apply to theories belonging to the class.

The following sections illustrate these features through an example of definition of a new class which enables to model **states** of a world and **actions** which modify states.

3.1 An example: states and state transitions

In Artificial Intelligence the notion of state proved to be suitable to represent both real worlds and typical "problem solving" strategies. A state is a snapshot of a world at a given point in time; at different points in time, the world can be in different states.
This idea can be well illustrated in the context of a microcosm such as the Blocks World [GeNi87]. Consider a variation of this world in which there are just three blocks and let us suppose we are only interested in their vertical relationships and not in the lateral ones. In a generic state, therefore, each block can be either on the table or exactly on top of another block, and can have at most one other block exactly on top of it. Different states of this world correspond to different block configurations. States can be described by means of the predicates *on(X,Y)*, *free(X)*, *on_table(X)*. A world persists in one state until an action is performed that changes it to a new state. We assume that all actions are instances of the operators *stack(X,Y)* (which describes the action of picking up block X from the table and stacking it on top of block Y) and *unstack(X,Y)* (which describes the action of unstacking block X from block Y and placing it on the table). Each operator can be characterized by a list of prerequisites, a list of positive effects and a list of negative effects. The prerequisites of an action are the assertions that must hold before the action is executed; the positive effects are the assertions that become true after the action is executed; the negative effects are the

assertions that become false. E.g., in order to execute the action *unstack(c,a)*, block *c* must be on *a* and, after the action execution, a new state is generated where *c* is on the table and *a* is free, while *c* is no longer on *a*.
In the state generated by an action execution, a new action can be performed, and so on. Moreover, it is possible to get several descriptions of alternative scenarios simultaneously. In fact, multiple derived states can be generated from a state by means of alternative actions; these states form a tree rooted in the initial state (Fig.1).

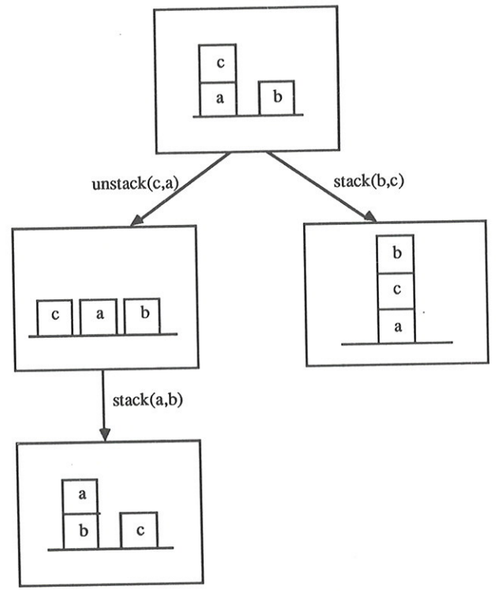


Fig. 1: Alternative scenarios

3.2 A solution in Epsilon

Epsilon theories provide a knowledge structuring mechanism suitable to represent states and the concept of class provides a convenient framework for defining the inference mechanism; relationships between chunks of knowledge can be represented by means of links whose semantics is defined within the class. In this framework relationships between chunks of knowledge are very peculiar and different from usual inheritance relationships, since they are relationships between states and describe the evolution of the world in time. The inference mechanism is also very peculiar and is not a simple enhancement of Prolog since, besides answering to queries on the world state, it has to model transforming actions.

The proposed solution in Epsilon is realized by defining a class (let's call it *planning)* and by representing states as theories belonging to this class, containing assertions which hold in the state. Operators which define actions are defined in the class meta-interpreter by means of the *action/1* predicate: e.g., *action(stack(X,Y))* asserts that *stack(X,Y)* is an operator. Prerequisites, positive effects and negative effects related to each action are also defined in the meta-interpreter, by means of the predicates *pre/2*, *del/2* and *add/2* respectively.
Fig.2 contains a portion of the *planning* meta-interpreter showing the definition of an action and part of the inference mechanism.

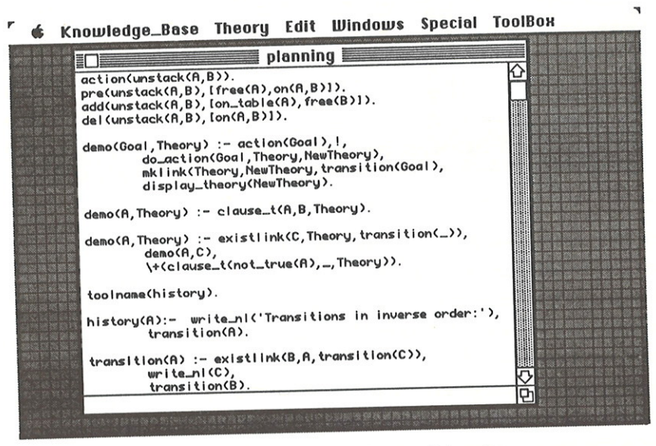


Fig. 2: A portion of the *planning* meta-interpreter

The *demo*/2 predicate describes the system behaviour following the activation of a goal. A goal in such a theory can be either an action request or a query on the world state.
When an action is requested (see the first definition of *demo*/2 in Fig.2), the system checks the prerequisites; if they hold, a new theory of class *planning* (whose name is generated by the system) is created. In this theory the new state is represented by difference with respect to the theory where the goal was asked, i.e. the assertions specifying the positive and negative effects are inserted. The new theory is linked to the previous one by a *transition* link labeled with the performed action.

Fig.3 shows the theory *world1*, created as a result of the activation of the goal *unstack(c,a)* in the theory *initial_state*, and a portion of the knowledge base dictionary (the *KB Info* window) which shows the link created between the two theories. The assertion *theory(TName,Flag,CName)* means that a theory *TName* of class *CName* is defined; *Flag* specifies whether *TName* is an *engine* or an *object* theory. The assertion *link(T1,T2,LName)* means that a link of kind *LName* is defined from theory *T1* to theory *T2*.

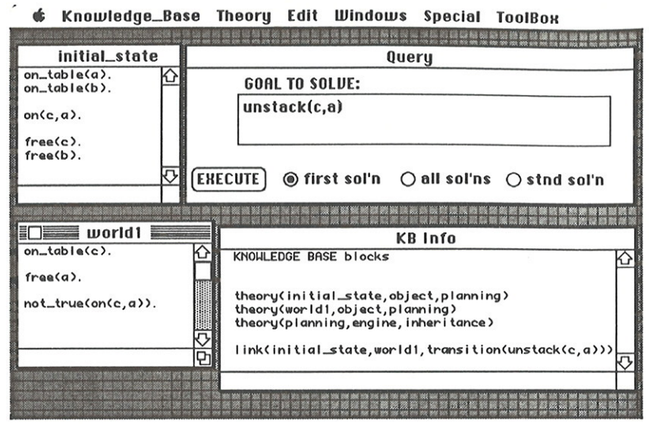


Fig. 3: The result of an action

Note that in a theory created after an action execution, the assertions which held before the execution and are still true are not included, since assertions not modified by an action can be verified in the previous (linked) theories. In the example in Fig.3, in the theory *initial_state* block *b* is on the table, and after the action *unstack(c,a)* it is still on the table, but the assertion *on_table(b)* is not included in the theory *world1*.

In fact, a goal about the world state is true in a *planning* theory either if it matches an assertion in the theory or if it is true in the previous (linked) theory and it does not match a negative effect of the last performed action (see second and third definitions of *demo/2* in Fig.2).

The *planning* theory also defines some ad-hoc tools: *history*, which displays the sequence of actions which led to the current state starting from the initial state, by exploiting the link labels; *show_state*, which displays explicitly the current state of the world, by exploiting the chain of linked theories which represent by difference the sequence of states starting from the initial state; *help*, which displays the list of available actions; *forward_plan*, which displays the sequence of actions needed to reach a state which satisfies the conditions specified by the user.

It is noteworthy that the proposed meta-interpreter is general enough to be used in other worlds. A world is defined by the set of assertions available to describe a state and by the set of actions available to describe a state transition, so that only these definitions need to be modified in order to model other worlds. Epsilon structuring features allow the description of the inference mechanism and support tools to be separated from the world description (definitions of actions and of assertions describing a state): the presented meta-interpreter can be made parametric with respect to various worlds, by writing it in a theory of class *inheritance* which inherits knowledge about the world from another theory. This allows the application domain to be changed without modifying the inference engine and its tools, by means of a simple link changing.

Finally, note that an engine theory can belong to any predefined or user-defined class (apart from *database*), and that therefore Epsilon permits multiple levels of meta-knowledge.

4. Enhancing meta-interpreters by inheritance

We will show how Epsilon theories can be used as building blocks to define enhanced meta-interpreters, by using predefined classes and links and exploiting Epsilon features for defining new classes and tools. Note that this is not the only way to define enhanced meta-interpreters in Epsilon, but it is an interesting case.

In order to enhance a meta-interpreter by some new features without losing the original one, a trivial solution is to duplicate the meta-interpreter to be enhanced and modify the copy. The main drawback of this option is, of course, redundancy; moreover, consistency maintaining problems arise if the original meta-interpreter has to be modified. Furthermore, if you want to enhance several meta-interpreters by the same features, you have to replicate also the definition of the enhancements for each meta-interpreter.

We will show how the Epsilon concept of link allows the above mentioned problems to be often solved in an elegant way.

4.1 An example: the query-the-user enhancement

The proposed solution is described through an example of definition of an enhanced meta-interpreter for the query-the-user functionality. This metainterpreter interactively queries the user for missing information, i.e. asks the user whether or not a given goal is true, if it can be proven in no other way. The implementation corresponds to adding, at the end of the meta-interpreter to be enhanced, clauses which define the desired behaviour and will be called when all the other clauses fail. Other meta-interpreters which define extensions or variations of Prolog can be enhanced this way.

The proposed solution is to write the additional clauses into a new theory (let's call it *supportqu*), and then define a second theory (let's call it *queryuser*), which inherits knowledge from the original meta-interpreter and from *supportqu*. Both *queryuser* and *supportqu* must be engine theories, because they have to extend the *demo* predicate.

Fig.4 shows the *supportqu* theory, the *queryuser* theory and a portion of the knowledge base dictionary which contains the link definitions in the case we want to enhance a meta-interpreter called *metaint* by the query-the-user facility. The theory *queryuser* is of class *inheritance*; two *clsisa* links are defined, the first one between *queryuser* and *metaint*, the second one between *queryuser* and *supportqu* (order is significant).

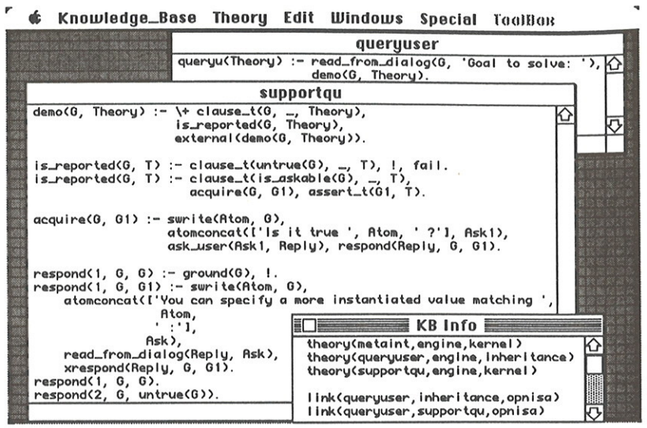


Fig. 4: The *supportqu* theory

Epsilon enables to add the query-the-user facility to the *metaint* class as a tool by including in the *metaint* theory a predicate *qutool/1* which calls *queryu/1* in the *queryuser* theory, and a declaration *toolname(qutool)*.

When *demo(Goal, Theory)* is called in *queryuser* (as a consequence of activating the query-the-user tool on *Theory*, belonging to the class *metaint*), no clause is found for *demo/2* in *queryuser*. Then the first *is_a* link is followed (according to the semantics of the link as defined by the *inheritance* class), the *metaint* theory is reached and the clauses which define *demo/2* in *metaint* can be used to solve *Goal*. If any subgoal fails, the second *is_a* link is followed, the *supportqu* theory is reached and the query-the-user mechanism is activated.

The methodology illustrated by the previous example allows the query-the-user facility to be provided to other classes as well, without rewriting code: it is sufficient to remove the link between *queryuser* and *metaint* and create a new link between *queryuser* and the class we want to enhance. In order to keep the query-the-user both for *metaint* and other classes, it is sufficient to duplicate only the small theory *queryuser*, while the theory *supportqu* (which contains the proper extension) can be inherited from any number of theories.

Note that it is possible to extend this way the *inheritance* class itself, since there is nothing to prevent you from linking *queryuser*, of class *inheritance*, to the theory *inheritance*.

4.2 Other examples

It is worth noting that this methodology can be used to obtain several kinds of enhancements.

One example is adding some inheritance rules to several meta-interpreters. Suppose that the *inheritance* class defines only one inheritance rule, e.g. *opnisa* (open is_a). You can embed, in another engine theory, rules defining other inheritance mechanisms: the example in Fig.5 shows a theory *inh_rules* which defines three inheritance mechanisms named *clsisa* (closed is_a), *clsconsult* (closed consultance) and *opnconsult* (open consultance). Then you can define engines of class *inheritance* which inherit from *inh_rules* via *clsisa*: meta-interpreters defined within these engines will be automatically enhanced by the *opnisa*, *clsconsult* and *opnconsult* inheritance rules.

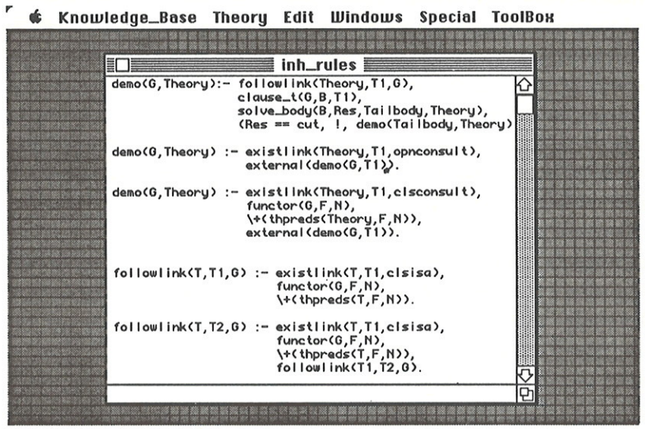


Fig. 5: An *engine* theory defining inheritance mechanisms

Another typical example is adding new "built-ins" to the language. For example, you can include in an engine theory the definitions of some commonly used predicates; this theory can then be used in order to extend the language of several classes, by exploiting the described methodology. Note that, in this case, the theory which defines the additional built-ins must be the first linked theory, because the additional clauses have to be added at the top of the meta-interpreter to be extended.

5. An Expert System

Thanks to its flexibility and knowledge representation features, Epsilon results to be a powerful development environment for expert systems and knowledge based applications. Epsilon theories permit to structure different kinds of knowledge about complex real-life problems, and the concept of class enables to choose, between several available reasoning modalities, the most suitable for the particular kind of knowledge and, when needed, to define ad-hoc inference engines.

One of the most interesting applications developed so far in Epsilon is an expert system which deals with the problem of company financial analysis. The purpose is to assess the company credit-worthiness, following the financial and credit policy established by Italian laws and normal bookkeeping principles. The expert system is compound of about 420 rules.

5.1 The financial analysis problem

A human expert who has to assess the company credit-worthiness can find most of the necessary data about the balance in two documents called **balance-sheet** and **income statement**, drawn up according to models deriving from laws in force. These models lack clearness and transparency with respect to the balance itself. Therefore, it is necessary to perform a **balance reclassification**, based on bookkeeping principles. Nevertheless, the absolute amounts which compound the reclassified balance are not enough to assess the company credit-worthiness. Usually, it is interesting to know the relative amounts, i.e to compare the amounts with respect to other elements of the company itself. E.g., realized profits are satisfactory or unsatisfactory according to stockholders' investment.

Ratios (traditionally subdivided into financial and economic ratios) are useful for analysing systematically the balance amounts. The ratio calculation permits to estimate the company **liquid assets**, **profitability** and **financial structure**. Finally, an analysis of this information enables to express a general judgement about the company credit-worthiness.

Note that, while the knowledge needed to the balance reclassification and ratio calculation is restricted to balance bookkeeping principles and calculation techniques, during the ratio analysis a deeper knowledge is needed which represents the human expert ability to use heuristics for determining required parameter, evaluate them and formulate judgements.

Summing up, the financial analysis is two-phased:

- **Balance-sheet analysis**: the balance-sheet and income statement data of the company are reclassified to highlight a new set of financial information such as the real state of assets, current and long-time liabilities, fixed capital, etc.; some financial ratios are also calculated.
- **Expert phase**: assessment of the profitability, liquid assets and financial structure of the company. The assessment is based on the results of the previous step and on data about the characteristics and the history of the company and about the general economic trend. Finally, a summary of the previous estimates is produced.

5.2 A solution in Epsilon

The expert system built to solve the above problem uses extensively the modularization features offered by Epsilon. The proposed architecture uses theories as knowledge modules which reflect the logic decomposition of the problem. Each theory included in the application knowledge base is related to a particular knowledge about different tasks involved in the company financial analysis.

As far as the balance-sheet analysis step is concerned, one theory (*reclassification*) contains knowledge about balance-sheet reclassification and another one (*ratios*) about financial ratio calculus. Data on the financial year balance are also contained in a theory (*balance_sheet*).

The real expert level is subdivided into three theories (*profitability*, *liquid_assets* and *financial_structure*), each one representing an expert of a particular domain, plus a main theory (*synthesis*) representing the expert which is in charge of deciding when and how to ask for single

expert opinions (if needed) and to make a synthesis among the experts' opinions.

Additional data on the characteristics and the history of the company and general economic trends are asked interactively when needed.

The various modules communicate both by means of explicit calls (*demo*) and by using the inheritance mechanisms provided by Epsilon.

The schema in Fig.6 shows the structure of the application knowledge base.

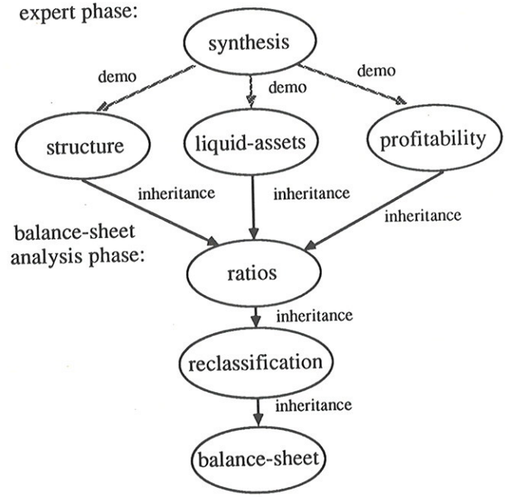


Fig.6: The expert system architecture

The application has been developed by using the predefined inference engines *kernel* and *inheritance*: namely, *synthesis* and *balance_sheet* are *kernel* theories and the other ones are *inheritance* theories.

The system generates (in a result theory) a detailed report, which explains the system reasoning and shows the values of some significant financial and economic ratios.

Though the system is specialized for commercial and industrial companies, it can be easily modified to take in companies operating in other fields such as banking, publishing, finance, etc. The theory structuring mechanism makes it possible to extend the system easily, by including new expertise.

Note that the theory *balance_sheet* is the only interface with data about the balance. Thanks to the modularity of the system, this is the only theory to be modified when these data are organized and/or stored in a different way, e.g. in an already existing relational database concerning the company accounting.

6. Conclusion

We have presented the Epsilon KBMS, whose aim is to provide a solution based on meta-programming to the problems of knowledge representation and manipulation. We have stressed mainly two of the most relevant characteristics of the system.

The first one is that Epsilon is explicitly open to the introduction of new inference mechanisms, new knowledge representation languages and new tools, by meta-programming. The presented examples (the definition of a class which enables to model states of a world and actions which modify states, and the description of a methodology for meta-interpreter composition by juxtaposition) also showed how links are powerful structuring facilities which emphasize the flexibility of meta-programming. These features make Epsilon a really open environment, which can be easily customized and extended to face different kinds of problems; theories and links can also be exploited to achieve software reusability and use meta-interpreters as building blocks for customizing the Epsilon environment.

The second addressed characteristic is the ability to develop real-life applications, such as an expert system for financial analysis, by exploiting Epsilon structuring facilities (theories and links between them) and predefined inference engines. Its flexibility and knowledge representation features make Epsilon a powerful development environment for expert systems and knowledge based applications. Epsilon theories permit to structure different kinds of knowledge about complex real-life problems and the concept of class enables to choose between several available reasoning modalities the most suitable for the particular kind of knowledge.

Summing up, although the most natural fields of application of Epsilon are the same as Prolog, its facilities make it possible to use the system to tackle a large variety of problems for which Prolog as it is would not be completely satisfactory.

Acknowledgments

We would like to thank all people who worked in the Epsilon project, in particular prof. Giorgio Levi, Mario Modesti, Giuseppe Sardu and Luigia Torre.

References

[BoKo82] K.A.Bowen and R.A.Kowalski, Amalgamating language and metalanguage in logic programming, in K.L.Klark and S.-A.Tarnlund, Eds., *Logic Programming*, Academic Press 1982, pp.153-172

[BoWe85] K.A.Bowen and T.Weinberg, A meta-level extension of Prolog, in *1985 IEEE Symposium on Logic Programming*, IEEE Computer Society Press 1985, pp.48-53

[CFLS87] P.Coscia, P.Franceschi, G.Levi, G.Sardu and L.Torre, Object level reflection of inference rules by partial evaluation, in D.Nardi and P.Maes, Eds., *Meta-level architectures and reflection*, North-Holland, 1987, pp.313-328

[CFLS88] P.Coscia, P.Franceschi, G.Levi, G.Sardu and L.Torre, Meta-Level Definition and Compilation of Inference Engines in the Epsilon Logic Programming Environment, Proc. Fifth International Conference and Symposium on Logic Programming, MIT Press, Kowalski and Bowen, Eds., 1988, pp. 359-373

[CoVa89] P.Coscia and S.Valeri, Implementation of the Database Interface: Basic Functionalities and Interpretive Approach, Esprit P530 Epsilon, Deliverable 5, May 1989

[Epsi90] Systems & Management, EPSILON User Manual, Esprit P530 Epsilon, Deliverable 7, Final Report, March 1990

[GeNi87] M.R.Genesereth and N.J.Nilsson, Logical Foundations of Artificial Intelligence, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987

[LeSa88] G.Levi and G.Sardu, Partial evaluation of metaprograms in a "multiple worlds" logic language, in D.Bjørner, A.P.Ershov and N.D.Jones, Eds., *Workshop on Partial Evaluation and Mixed Computation, Gl. Avernaes, Denmark*, October 1987, published in *New Generation Computing*, 6 (1988), pp. 227-247

[SaSh86] S.Safra and E.Shapiro, Meta-interpreters for real, *Information Processing 1986*, North-Holland, pp.271-278

[Shap83] E.Shapiro, Algorithmic Program Debugging, MIT Press, Cambridge, Massachussets, 1983

[Ster84] L.Sterling, Expert System = Knowledge + Meta-Interpreter, Tech. Report CS84-17, Weizmann Institute Of Science, Rehovot, Israel, 1984

[Ster85] L.Sterling, Meta-Interpreters for Expert Systems , *CAISR TR 134-85*, Case Western Reserve University, 1985

[Ster87] L.Sterling, A Meta-Level Architecture for Expert Systems , in D.Nardi and P.Maes, Eds., *Meta-level architectures and reflection*, North-Holland, 1987, pp. 301-312

[StBe89] L.Sterling and R.D.Beer, Metainterpreters for Expert System construction, *The Journal of Logic Programming* 1989, pp. 163-178

[StSh86] L.Sterling, E.Shapiro, The Art of Prolog, MIT Press, 1986

[Take86] A.Takeuchi, Affinity between meta interpreters and Partial Evaluation, *Information Processing 1986*, North-Holland, pp.279-282

[TaFu86] A.Takeuchi and K.Furukawa, Partial Evaluation of PROLOG Programs and its application to metaprogramming, *Information Processing 1986*, North-Holland, pp.415-420

[VCST89] S.Valeri, P.Coscia, G.Sardu and L.Torre, Epsilon General Description, Technical Report, Systems & Management, March 1989

Compilazione ed Esecuzione di un Linguaggio Logico Distribuito

T.Castagnetti, P.Ciancarini, M.Montanari

Dipartimento di Informatica - Università di Pisa[1]

SOMMARIO

Viene descritto il linguaggio logico parallelo Shared Prolog e le tecniche di compilazione e ottimizzazione che migliorano l'efficienza della sua implementazione distribuita.

Shared Prolog è un linguaggio logico parallelo che include Prolog come sua componente sequenziale. Un programma è un insieme di moduli Prolog, detti teorie, coordinati mediante regole di attivazione, chiamate pattern. Le teorie non comunicano tra di loro ma fanno riferimento ad una base di conoscenza comune, chiamata blackboard. La blackboard contiene solo fatti, i cui argomenti possono essere non completamente istanziati (non ground). In questo lavoro si mostra che la blackboard può essere distribuita tra più agenti. L'implementazione del linguaggio è stata resa più efficiente da alcune tecniche di compilazione e ottimizzazione basate sulla valutazione parziale.

1. INTRODUZIONE

La maggior parte delle proposte nel campo della programmazione logica concorrente - tra i linguaggi più noti, citiamo Flat Concurrent Prolog (e i suoi derivati Vulcan [Kahn 86] e Strand [Foster 89a]), Parlog (e i suoi derivati Parlog Plus [Foster 89b] e Polka [Davison 87]), e Flat Guarded Horn Clauses - si rifanno alla cosiddetta interpretazione a processi della programmazione logica. Tale interpretazione non è però l'unica possibile; in particolare questa interpretazione non si adatta per tutti quei linguaggi logici paralleli che non usano comunicazioni basate su canali, ad esempio DeltaProlog, CPU [Mello 86] e TS-Prolog [Futò 86].

Una interpretazione alternativa a quella a processi è quella a blackboard [Ciancarini 89]. La base di conoscenza viene vista come una lavagna, ovvero come una struttura dati condivisa contenente fatti e su cui possono leggere e scrivere in modo associativo (mediante assert e retract, oppure operazioni associative nello stile di Linda [Gelernter 85]) agenti governati da programmi logici. Questa interpretazione si adatta sia a linguaggi sequenziali che a linguaggi con operatori espliciti di parallelismo.

Shared Prolog è un linguaggio logico parallelo la cui semantica operazionale è stata data in termini dell'interpretazione a blackboard [Brogi 89]. Sul piano implementativo già a livello prototipale il linguaggio è stato realizzato su un'architettura distribuita [Ambriola 90], anche se il componente blackboard era centralizzato. In questo articolo si dimostra che il componente blackboard può essere distribuito; inoltre vengono esposte brevemente alcune tecniche di analisi statica e compilazione che migliorano l'efficienza dell'esecuzione dei programmi. La formalizzazione e discussione completa di tali tecniche è contenuta in [Castagnetti 90].

La struttura dell'articolo è la seguente: nel secondo paragrafo viene brevemente descritto il linguaggio e viene mostrato un esempio di programmazione in SP. Il terzo paragrafo discute una serie di tecniche di analisi della semantica statica dei programmi, allo scopo di introdurre una serie di ottimizzazioni di compilazione. Il quarto paragrafo descrive la macchina astratta del linguaggio e l'implementazione distribuita della blackboard. Il quinto paragrafo mostra alcune valutazioni e confronti di performance e infine, l'ultimo paragrafo elenca alcuni futuri sviluppi che verranno perseguiti dal progetto Shared Prolog.

---

[1] This paper has been partially supported by CNR-Progetto Finalizzato Sistemi Informatici-Calcolo Parallelo