Consumption-based Distributed Unification

Evelina Lamma[1], Paola Mello[2], Cesare Stefanelli[1], Pascal Van Hentenryck[3]

[1] DEIS, Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
{elamma,cstefanelli}@deis.unibo.it
[2] Istituto di Ingegneria, Università di Ferrara
Via Saragat, 44100 Ferrara, Italy
pmello@deis.unibo.it
[3] Department of Computer Science
Brown University, Box 1910, Providence RI 02912, USA
pvh@cs.brown.edu

Abstract. In distributed implementations of logic programming, data structures are spread among different nodes and unification involves sending and receiving messages to access them. Traditional implementations make remote data structures accessible to other processes by sending messages which carry either the overall data structure (*infinite-level copying*) or only remote references to these data structures (*0-level copying*). These fixed policies can be far from optimal on various classes of programs and may induce substantial overhead. The purpose of this paper is to present a new implementation scheme for distributed logic programming which consists of tayloring the copying level to each procedure. The scheme is based on a consumption specification which describes the way the procedure "consumes" its arguments locally. Both a high-level and a low-level implementations of the scheme are described. These implementations use the consumption specification to avoid uncessary copying and to request data structures globally. Moreover, the consumption specification (or an approximation of it) can be obtained through a static analysis inspired to traditional type analyses. Experimental results on the high-level implementation for a network of workstations show the potential of the approach.

1 Introduction

In distributed implementations of logic programming, data structures are spread among different nodes and unification involves sending and receiving messages to access them. Traditional implementations make remote data structures accessible to other processes by sending messages which carry either the overall data structure (*infinite-level copying*) or only remote references to these data structures (*0-level copying*). An intermediate approach is possible in some systems, where data structures are copied up to a certain level. For instance, an approach based on replication of data is used in [2], where each message carries the infinite-level copying of each argument of the goal to be solved. Ichiyoshi et al. [11] introduced the copying level concept in the implementation of the KL1 language on the MultiPSI machine. In the so-called

forward unification phase, atomic values are always entirely copied, unbound variables are never copied (i.e., only a remote reference to them is encoded into the exporting message), and any fixed level of copying can be "a-priori" determined for data structures.

It should be clear that any fixed policy can be far from optimal and may induce substantial overhead on various class of programs. A high copying level may cause overhead due to the transmission of unnecessary information; a low copying level might cause, instead, further messages to be exchanged for dereferencing and for accessing a remote structure in order to perform unification (remote dereferencing).

The purpose of this paper is to present a novel implementation scheme that, informally speaking, tailors the copying and dereferencing level to each argument of each procedure in the distributed program. The scheme is based on a consumption specification which describes how a procedure "consumes" its arguments locally. It uses this specification to improve both the copying (put) and the remote dereferencing (get) phases. The copying phase is improved by avoiding the copy of subterms that are not used by the predicate. The remote dereferencing phase is improved by requesting, from a remote node, entire subterms instead of accessing them piece by piece, thus reducing the amount of communication. Consumption specifications are expressed in terms of a simple generalization of tree-grammars [3] (or type graphs [7, 12]). They are simple enough to be provided by programmers. However, consumption specifications (or approximations of them) can be obtained through a static analysis inspired by traditional type analyses, making the approach fully transparent for programmers.

Two implementations of the scheme are described: a high-level implementation based on attributed variables following the lines suggested by [6], and a low-level implementation based on the Warren Abstract Machine (WAM) [1, 13]. The high-level implementation uses a blackboard for communication, while the low-level implementation uses explicit message passing. Experimental results using the high-level implementation show the potential of this approach.

The rest of the paper is structured as follows: Section 2 gives an overview of the distributed execution model. Section 3 presents our approach. Section 4 describes the high-level implementation, and Section 5 sketches the low-level implementation. Section 6 suggests how standard type analysis can be adapted to produce the consumption specification. Section 7 describes the experimental results. Section 8 concludes the paper.

2 Overview of the model for distributed unification

In our distributed model, a number of concurrent processes cooperate in solving the query. Each process is associated with a procedure of the program (i.e., a set of clauses). Processes communicate via message passing, since data structures can be spread over different nodes of the underlying architecture. In the following we focus our discussion on the implementation of the distributed unification. The main difference between distributed and sequential implementations of unification is that the first demands message management for data exchange.

In fact, the **put** phase in the traditional WAM loads the arguments into the registers but, in a distributed implementation, the **put** phase consists of preparing a message to be sent to the process responsible for executing the goal. For a given argument, this message preparation step works as follows. If the argument is an atomic value, the value is included in the message. If the argument is an unbound variable, a remote reference to this variable is created and inserted in the message. If the argument is a structure, different policies can be found in the literature [4, 9, 10, 11]:

- the argument is entirely copied in the message (*infinite-level copying*);
- a remote reference to the structure is included in the message (*zero-level copying*);
- the structure is copied in the message up to a certain level with remote references to the rest of the structure (*k-level copying*).

Following the adopted policy of copying, the arguments are packed in the message that is sent to the remote node responsible for the procedure execution. The traditional WAM **get** phase is also modified in the distributed case. In particular, before performing the head unification, the remote process executes a message reception phase to extract the arguments from the message. In addition, for efficiency reasons discussed later, the message may not contain all the necessary information needed for unification and therefore some remote references may be encountered when extracting the arguments from the message. In this case, it is necessary to ask the appropriate processes for the needed data structures. This task, i.e., requesting the value of a remote reference to a process, is known as *remote dereferencing*, and is performed by using message-passing. Also, when performing remote dereferencing, it is possible to follow several policies of copying. In particular, it is possible to request either all the referenced data structure or only a part of it. The two solutions (and any intermediate case) have the same advantages and inconvenients as the policies described for the message preparation phase.

3 Consumption-based distributed unification

None of the strategies described in Section 2 (zero-level copying, infinite-level copying, or k-level copying) is the most appropriate for all programs. Infinite-level copying has the problem of sending too much information, i.e., copying terms that are not used. Zero-level copying has the problem of sending many small messages, increasing the communication between processes. K-level copying combines the advantages and inconvenients of zero- and infinite-level copying. In addition, the choice of the copying policy not only depends on the structure of the application, but also depends on the type of architecture (in particular, the cost of communication), since it affects the cost of the various operations.

The main contribution of this paper is to present a novel implementation scheme that, informally speaking, tailors the copying and dereferencing level to each argument of each procedure in the distributed program. Consider the traditional **append/3** program.

```
append([],L,L).
append([F|T],S,[F|R]):- append(T,S,R).
```

and the top-level goal `top:-append(LongList,List,Res).` where `LongList` is a long list of complex data structures. In this program, `append/3` needs the spine of the list for the first argument but does not need to inspect the elements of the list. It also does not need to inspect its second and third arguments. As a consequence, the top-level should be compiled to prepare a message containing a copy of the spine of the list with a remote reference to its various elements for the first argument. Remote references to the two other arguments must be included in the message as well. This makes sure that the message preparation phase copies only what is really needed by `append/3` and that `append/3` is executed without requiring any remote dereferencing. An infinite-level policy would copy the whole list and thus potentially many irrelevant structures, while zero-level copying would require to dereference the remote list element by element, introducing notable communication cost.

3.1 The consumption specification

In order to determine how a procedure "consumes" its arguments, our implementation makes use of a consumption specification. Consumption specifications are simple enough to be provided by programmers but we also show in section 6 how to obtain them (or an approximation of them) through a static analysis inspired by traditional type analyses, making the approach transparent for programmers.

A consumption specification is expressed as a tree grammar [3] extended with an additional terminal `Remote`. This additional terminal simply specifies zero-level copying. The rest of the specification identifies what part of the term is consumed locally by the predicate. For instance, the consumption specification for `append/3` is

`append(`T_1`,`T_2`,`T_3`)` *where*

T_1 ::= [] | cons(Remote,T_1).
T_2 ::= Remote.
T_3 ::= Remote.

It specifies that `append/3` consumes the spine of the first argument and that the two other arguments are not consumed. Consumption specifications can be rather complex. The consumption specification for the program

```
process([]).
process([s(D)|R]):- process(R).
process([c(D)|R]):- process(R).
```

is given by `process(T)` *where*

T ::= [] | cons(T_1,T).
T_1 ::= s(Remote) | c(Remote).

It specifies that `process/1` consumes all the elements of the list but limited to the main functor of the element.

Consumption specifications are expressive enough to associate different levels of copying not only with each procedure argument but also with distinct subtrees of a given argument.

A formal definition of the meaning of the consumption specification is outside the scope of the paper, as it requires the definition of an instrumented operational semantics. Notice however that consumption specifications are guaranteed to be correct as soon as they represent a superset of the types of the procedures (with `Remote` denoting the set of all terms).

3.2 Exploiting the consumption specification

The consumption specification is exploited in a distributed implementation during message preparation, message reception, and remote dereferencing.

Message preparation. During message preparation, consumption specifications are used to include in the message only those parts of the data structures consumed by the procedure. For instance, a call to `append/3` in

`p:- ...,append(`t_1`,`t_2`,`t_3`), ...`

would be compiled into

```
....
copyT1(t1)
copyT2(t2)
copyT3(t3)
call append/3
....
```

where T_i denotes the type of argument i in the consumption specification defined previously. This code generates for each argument i a copy instruction which uses its type T_i. The instructions insert in the message the appropriate part of the term. In the case of the term t_1 with the consumption specification T_1 ::= [] | cons(Remote,T_1), the copy instruction `copyT1(`t_1`)` inserts inside the message the term `nil` if t_1 is nil, the term `<LIST,V1,V2>` if t_1 is a list with head `H` and tail `Ta`, `V1=copyRemote(H)`, and `V2=copyT1(Ta)`; it fails in any other case. Note that the instruction `copyRemote(t)` inserts inside the message the remote reference to the term `t` (zero-level copying).

Message reception. The consumption specification is used at message reception to request the remote data structures that are necessary for the procedure to execute locally. The main interest of consumption specifications in this context is the ability of requesting the needed data structures globally instead of element by element. Note also that these data structures are requested at the procedure level before executing any clause. To illustrate the benefits of the consumption specification, consider the following program

```
top :- reverse(LongList,Res).
reverse(L,Lr):- reverseAcc(L,[],Lr).
reverseAcc([],Acc,Acc).
reverseAcc([F|T],Acc,Res):- reverseAcc(T,[F|Acc],Res).
```

Assuming that a process is associated with each procedure, an infinite-level copying would copy the long list twice, from `top` to `reverse/2` and from `reverse/2` to `reverseAcc/3`. It would also copy irrelevant information (e.g., the elements of the list). A zero-level copying would access the list piece by piece, with high communication overhead. Using consumption information, our scheme would put a remote reference to the list when calling `reverse/2` and it would pass the remote reference to `reverseAcc/3`; procedure `reverseAcc/3` uses the consumption information to get only the spine of the list through remote dereferencing. The consumption specification for `reverse/2` is `reverse(Remote,Remote)` while the consumption specification for `reverseAcc/3` is `reverseAcc(`T_1`,Remote,Remote)` *where*

```
T1 ::= [] | cons(Remote,T1).
```

The code pattern generated for the procedure `reverseAcc/3` would be

```
reverseAcc/3:
     remote_dereferenceT1(A1);
     try C1;
     trust C2;
```

where `remote_dereferenceT1(A1)` gets the spine of the list in argument `A1` of the message.

It is important to note that the consumption specification is goal-independent, i.e., it is defined independently from the way the procedure is used. For instance, if the first argument of `reverseAcc/3` is a free variable, no remote dereferencing takes place. Remote dereferencing follows the specification to the extent possible.

4 The high level implementation

To verify experimentally our approach, we implemented a distributed logic language on top of SICStus Prolog [14] using the Linda library and attributed variables.

Attributed variables. Our high-level implementation was inspired by [6] and uses attributed variables to implement the communication variables, i.e., the variables occurring in a remote goal. Attributed variables introduced by Le Houitouze [8] are variables associated with an attribute (i.e., a term) and unification of these variables can be specified by programmers. Our high-level implementation attaches an attribute `rem(Process,Id,Bound)` where the pair `(Process,Id)` identifies uniquely a communication variable and `Bound` specifies if the variable is bound or unbound.

The blackboard structure. In the high-level implementation, message sending is achieved by writing the message onto a blackboard implemented via the Linda library. There is a server process which handles the blackboard. Prolog client processes can write (using `out/1`), read (using `rd/1`), and remove (using `in/1`) data (i.e., Prolog terms) to and from the blackboard. Partial bindings of the communication variables (determined by the consumption specification) are inserted in a message and posted on the Linda blackboard.

The consumption specification. The consumption specification is represented by Prolog facts of the kind `type(t,term)`. For instance, the copying specification:

```
T ::= [] | cons(T1,T).
T1 ::= s(Remote) | c(Remote).
```

is represented by the following Prolog facts:

```
type(t,[]).
type(t,cons(t1,t)).
type(t1,s(remote)).
type(t1,c(remote)).
```

The copy of terms in their blackboard representation is performed until a terminal `remote` is reached, thus avoiding unnecessary communication overhead. Notice that the consumption specification is also used for run-time type checking when constructing the message.

Message preparation. Consider the preparation of a message containing the communication variable `X`, associated with the consumption specification `T` described in the previous paragraph. Assume that `X` is bound to the list `[c([1,2,...,200])]`. In this case, the element of the list is copied but copying is limited to the main functor. A new communication variable `X1` is created and locally bound to `[1,2,...,200]`. Only the structure `[c(X1)]` is then posted into the blackboard. In fact, the message inserted in the blackboard contains the binding for the variable `X`:

`msg_binding(rem(1,2,bound),[c(rem(1,3,bound))])`

where the pair `(1,2)` identifies `X` and the pair `(1,3)` identifies `X1`.

Message reception. After receiving a message, the argument values are extracted in order to perform head unification and goal evaluation. This implies the building of a local structure starting from the blackboard representation of the arguments contained in the message. A new local structure `[c(X2)]` is created, where `X2` is a new attribute variable with the same identifier of `X1`. Notice that, during the head unification, some parts of the data structure may not be locally present. In this case, *remote dereferencing* is automatically raised by the unification of attribute variables. For instance, assume that `g([c([1|_])])` is the head of the clause. This implies the unification of the attribute variable `X2` with `[1|_]`. Therefore the unification handler (i.e.,`verify_attr/3`) is called and a request for a remote dereferencing of `X2` is sent to the appropriate process.

5 The consumption-driven abstract machine

Obviously the high-level implementation cannot achieve the performance of a specific abstract machine. This section sketches a low-level implementation based on the WAM [1, 13], suitably extended in order to perform the distributed unification driven by the consumption specification.

The distinguishing feature of this low-level implementation is the use at compile time of the consumption specification in both the message preparation and the message reception. In particular, both phases are compiled into instructions of the extended WAM by following the consumption specification, thus producing a more efficient code. In addition, the consumption specification allows for run-time type checking at message construction.

In our distributed model, a number of concurrent processes cooperate in solving the query. Each process runs on an independent abstract machine which is essentially a standard WAM with a complete set of registers and stacks. Contrary to the sequential case, the data structures in each local heap possibly contain references to the data areas of remote processes.

We extended the sequential WAM with respect to distributed unification by introducing the message heap `MSG`, i.e., a heap-like structure containing the goal arguments to be sent to remote processes. A global register `M` points to the next available address in the message heap. The basic memory element of both `Heap` and `MSG` structures is the standard cell, with an additional type (tag) `REM` denoting that the cell contains a reference to the heap of a process executing on a different node. The remote reference is expressed by a pair `pid/remote_offset`, where the `pid` is the identifier of the remote process owning the data structure and the `remote_offset` is the offset of the data structure in the heap of `P`.

The WAM instruction set has been extended by introducing new instructions dealing with the construction and the receiving of messages. We have introduced a new set of control instructions (e.g., `prepare_msg`, that prepares a message driven by the type `T` and then sends the message), as well as a set of message building operations (e.g., `build_list_in_msg`, that inserts a remote reference in the message).

The consumption specification drives also the reception of messages and produces a compiled code very close to the message preparation case. Let us focus on the message reception case when the message does not contain all the parts of the data structures requested by the consumption specification. In that case, a remote dereferencing is needed. To illustrate these ideas, consider the following code, which is part of the reception of a message driven by a consumption specification `TypeT`.

```
switch_on_list nil, list, REM, X
nil:  .....
list: .....
REM:  remote_dereference TypeT
      prepare_heap HeapTypeT
      return
default: ...
```

and assume that the message contains only a remote reference to the argument. In this case, the `switch_on_list` instruction transfers the control to the label `REM` which raises a remote dereferencing request to obtain the missing part of the data structure. Note that the `remote_dereference` instruction requests the part of the data structure according to the consumption specification `TypeT`.

6 Static analysis

In this section, we sketch how the consumption analysis can be obtained by a static analysis enhancing traditional type analyses. A complete description of the analysis is beyond the scope of the paper and we only try to convey the main ideas behind the approach. Recall that the consumption specification describes a superset of the type of the procedure. Consider, again, procedure `append/3`:

```
append([],X,X).
append([F|T],S,[F|R]) :- append(T,S,R).
```

A goal-independent type analysis produces the result

`append(T,Any,Any)` *where* `T ::= [] | cons(Any,T)`

which is essentially the consumption specification we showed in section 3 (replace `Any` by `Remote`). These type analyses have been investigated extensively in the literature (e.g., [5, 7, 12]). In general, to obtain effective consumption specifications, it is necessary to refine the result of the type analysis. Consider the program

```
p([]).                    q([]).
p([F|Ta]) :- q(Ta).       q([F|Ta]) :- q(Ta).
```

The standard analysis would produce the results `p(T)` and `q(T)` *where*

`T ::= [] | cons(Any,T).`

which are superset of the types of the procedures. Obviously a good implementation should not send the list to `p/1`, but only its first `cons` cell. In other words, the consumption specification for `p/1` should be $p(T_1)$ *where*

T_1 `::= [] | cons(Any,Any).`

which is in fact a larger superset of the type (e.g., what the analysis would return if the goal `q(Ta)` is omitted). To overcome this problem, traditional type analyses should be enhanced by annotating each functor with the predicate in which it occurs. The type analysis then produces the results $p(T_1)$ and $q(T_2)$ *where*

T_1 `::=` $[]_p$ | $cons_p$`(Any,`T_2`).`
T_2 `::=` $[]_q$ | $cons_q$`(Any,`T_2`).`

From these results, it is not difficult to obtain the consumption specification for `p/1`, since the type T_1 now indicates that `p` only accesses the first `cons` cell locally.

7 Experimental results

This section presents the experimental results obtained by running some benchmark programs on the system described in Section 4. These results give some indication of the practical usefulness of the consumption specification approach, although they are biased to reveal the communication overhead in the distributed execution.

To compare the consumption specification approach with traditional copying policies, we executed each program with different policies of copying. In particular, we compared both zero- and infinite-level copying policies with the policy driven by the consumption specification determined by the static analysis. All programs have been executed with the same granularity of execution, i.e., we allocated each procedure onto a different node, although in some cases this is not the best possible allocation strategy (we are not interested in obtaining the maximal performance, but only in the comparison of the different solutions for copying the arguments). Tables 7.1-7.4 show the results obtained by running the benchmarks on a network of SUN workstations (Sparc2) connected over an Ethernet network. The results are presented in terms of complexity of the data structures involved. All the presented programs work on lists and are executed with lists of different size.

As a first benchmark, consider the `append/3` program. We have considered for this program both zero- and infinite-level copying and the consumption specification requiring to send the spine of the list for the first argument and a remote reference for both the second and third arguments of the goal. Table 7.1 shows the results obtained with the first two arguments being lists of 10, 30, and 50 integer elements. Times are in milliseconds. As shown in Table 7.1, the gain achieved with the consumption specification approach increases with the length of the lists.

length	zero-level copy (0)	∞-level copy (∞)	consumption copy (c)	$0/c$	∞/c
10	122	56	24	5.08	2.33
30	365	80	32	11.40	2.50
50	585	112	40	14.62	2.80

Table 7.1: `append/3` with lists of integers

Table 7.2 shows the timings of the `reverse/2` program with zero-, infinite-level copying, and the consumption specification described in Section 3. Benchmarks have been executed with a list of 10 elements each one being, on its turn, a list of 10, 30, 50 integer values. Zero-level copying is very inefficient on this benchmark (i.e., it requires a remote dereference for each element of the list, i.e., 10 remote dereference requests for all cases). ∞-level induces to copy the list twice from the top-level goal to `reverse/2` and from `reverse/2` to `reverseAcc/3`.

length	zero-level copy (0)	∞-level copy (∞)	consumption copy (c)	$0/c$	∞/c
10	1020	240	208	4.90	1.15
30	1123	544	496	2.27	1.09
50	1210	800	568	2.13	1.40

Table 7.2: `reverse/2` with one list of lists of increasing size

Tables 7.3 and 7.4 show the results of the `keysort/2` and `quicksort/2` programs. `keysort/2` is a variant of `quicksort/2`, and sorts a list of strings on the basis of the first character of each string. An auxiliary procedure (`split/4`) used by `keysort/2` splits the list of strings on the basis of the first character of the first string. The results reported in Table 7.3 are obtained by sorting a list of 10 strings, of increasing length (i.e., 10, 30, 50 characters), and by adopting zero-, infinite-level copying and the copying policy driven by the consumption specification which requires to send to `split/4` only the first character of each string.

length	zero-level copy (0)	∞-level copy (∞)	consumption copy (c)	$0/c$	∞/c
10	4710	1640	816	5.77	2.01
30	4713	4184	832	5.66	5.02
50	4816	7360	848	5.67	8.68

Table 7.3: `keysort/2` with one list of strings of increasing size

Finally, Table 7.4 shows the results obtained by executing the `quicksort/2` program on a list of 10 characters. Differently from `keysort/2`, for this program the consumption specification requires to send to `split/4` the whole string.

length	zero-level copy (0)	∞-level copy (∞)	consumption copy (c)	$0/c$	∞/c
10	620	480	380	1.63	1.26

Table 7.4: `quicksort/2` with one list of 10 characters

8 Conclusions

Traditional distributed implementations of logic programming make remote data structures accessible to other processes by sending messages which carry the overall data structures or only remote references to these data structures. These fixed policies can be far from optimal on various classes of programs and may induce substantial overhead. This paper has presented a new implementation scheme for distributed logic programming which consists of tayloring the copying level for each argument of procedures. The scheme is based on a consumption specification which describes the way each procedure "consumes" its arguments locally. Both a high-level and a low-level implementations of the scheme are described. These implementations use the consumption specification to avoid uncessary copying and to request data structures globally. Moreover, the consumption specification (or an approximation of it) can be obtained through a static analysis inspired by traditional type analyses.

The high-level implementation has been built on top of SICStus Prolog by using both the attributed variable and the Linda libraries. The results obtained with the benchmark programs running on a distributed system composed of a network of workstations show the viability of the approach. The system is flexible with respect to the copying specification, and has been used to test different copying policies for the goal arguments. In particular, it showed an effective performance improvement when

adopting the consumption specification obtained by the static analysis of the program. This is a high-level implementation suitable for giving the idea of the usefulness of the technique. A significant gain in term of performances should be achieved by the low-level implementation sketched in the paper, by pushing the implementation of critical operations down to C, at WAM level.

Acknowledgments

We thank the Exchange Visitor Program between the university of Bologna and Brown University which made this work possible. This work has been partially supported by C.N.R. "Project on Program Analysis and Transformation ANATRA".

References

1. H. Ait-Kaci. *Warren's Abstract Machine*. The MIT Press, 1991.
2. G. Frosini P. Corsini and L. Rizzo. Implementing a parallel Prolog interpreter by using Occam and Transputers. In *Microprocessors and Microsystems, Vol. 13*, pages 271–279, 1989.
3. F. Gecseg and M. Steinby. *Tree Automata*. Akademiai Kiado, Budapest, 1984.
4. P. Kacsuk and M. Wise (eds.). *Implementations of Distributed Prolog*. J. Wiley and Sons, 1992.
5. N. Heintze and J. Jaffar. *A Finite Presentation Theorem for Approximating Logic Programs. Proc. 17th ACM Symp. on Principles of Programming Languages*. 197-209, 1990
6. M. Hermenegildo, D. Cabeza and M. Carro. *Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems. Proc. Int'l Conf. on Logic Programming ICLP95*, The MIT Press, 1995.
7. G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. *Journal of Logic Programming*, 13(2-3):205–258, 1992.
8. S. Le Huitouze. *A new data structure for implementing extensions to Prolog*. In P. Deransart and J. Maluszunski, editors, *Proc. Programming Language Implementation and Logic Programming*, pp. 136-150, Springer-Verlag, 1990.
9. S. Taylor. Parallel Logic Programming Techniques. Prentice-Hall International Editions, 1989.
10. E. Tick. Parallel Logic Programming. MIT Press, 1991
11. K. Nakajima, N.Ichiyoshi, K. Rokusawa and Y. Inamura. A new external reference management and distributed unification for KL1. In ICOT, editor, *Proc. Int'l Conf. FGCS-88*, pages 904–913. 1988.
12. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type Analysis of Prolog using Type Graphs. *Journal of Logic Programming*, 22(3), March 1995.
13. D.H.D. Warren. An abstract Prolog instruction set. Technical Report TR 309, SRI International, 1983.
14. Swedish Institute of Computer Science, Kista, S. *SICStus Prolog User's Guide*, 1990.

Towards Higher-Order Distribution in Functional Languages *

Víctor M. Gulías, Juan J. Quintela, José L. Freire
LFCIA, Departament of Computer Science
University of La Coruña, Spain
{gulias,quintela,freire}@dc.fi.udc.es

Abstract

A model is introduced to perform explicit distributed computations using a functional language, looking for improvements in the overall performance of programs. The distributed program is executed on a computer network using a master-slave architecture (distributed-memory multiprocessor). Great attention is paid to keep important properties of the functional paradigm such as referential transparency, high-order functions or abstraction. In fact, high-order functions and abstraction have been used to implement new distributed abstractions that become the building blocks of distributed programs with little effort. A prototype implementation has been developed using the functional language CAML SPECIAL LIGHT, generating code that runs on a network of Sparc workstations.

Keywords: Distributed Architectures, Functional Programming, ML, Compilers, Abstraction, High-order functions.

1 Introduction

It is well known that functional languages are well suited for elegant symbolic computation because of its properties: referential transparency, high-order functions, pattern-matching, user-defined {concrete, recursive, abstract} data types, focus on abstraction, and so on. However, all these nice tools are not for free: functional languages often demand more resources than non-declarative languages. On the other hand, functional languages have been considered as highly suited for parallel computation. In a program without side-effects, the evaluation of expressions can be performed concurrently without the risk of interference among different threads. With this hope in mind, a great deal of

*Supported by the *Xunta de Galicia* XUGA10502B94

effort has been done to implement functional languages that exploit automatically this fact and, in some cases, new software and hardware architectures have been defined [2]. Nevertheless, the results have been unsatisfactory. It seems to be necessary the introduction of hints or annotations which guide the compiler and runtime system to exploit properly system resources. In this *explicit functional programming* [8], the programmer is often not concerned with operational details, but the appropriate constructs for expressing parallism are availables to improve execution with no (or little) change of meaning.

In this paper, a model is introduced to perform explicit distributed computations using a functional language, looking for improvements in the overall performance of functional programs. Although space distribution is transparent, it is the programmer's work to suggest the runtime system a suitable decomposition of the computation and a suitable moment to initiate a new thread. The distributed program is executed on a computer network using a master-slave architecture (distributed-memory multiprocessor). Great attention is paid to keep the important properties of the functional paradigm that all of us are looking for. In fact, high-order functions and abstraction have been used to implement new distributed abstractions, such as a distributed map, distributed divide-and-conquer algorithms and so on, which can be the building blocks of distributed programs from sequential ones or from scratch with little effort. A prototype implementation has been developed using the strict functional languaje CAML SPECIAL LIGHT [9] (CSL, for short), a descending of CAML [11], member of the ML family. This prototype has been tested on a network of Sparc workstations.

This paper is organized as follows: the next section describes the architecture of the distributed system and details the way the work is dispatched to each agent. Section 3 shows how abstraction can be a powerful tool to implement distributed programs. Section 4 presents the prototype implementation and gives some examples and benchmarks. Finally, we conclude and present future work.

2 The Distribution Model

2.1 Previous Work

The present paper is an evolution of [4] in which an heterogeneous computation model is presented in order to integrate functional and imperative modules on a client-server architecture. Although the goal of that paper was to solve an integration problem among different paradigms, an improved version is presented in [3] to speed up program execution by spawning different tasks on a network of computers using computation servers. The distribution is carried out explicitly by indicating which remote services (functions) will be available in the remote servers. The function signature of each service is used to determine the communication protocol (encoding of arguments and results) and to generate automatically the stub code necessary to build the servers. A client process spawns remotely the computation of an expression using one of those services, breaking down the synchronism of the evaluation sequence of a strict language (evaluation of arguments, and then evaluation of the function body with those results). The main inconvenience was that functions were not first-class citizens, so many powerful facilities of functional languages (partial application, abstraction, high-order functions) should be abolished from programmer's mind.

2.2 Architecture

In figure 1, a master-slave model is showed. There is a unique *master* that manages the distribution of work among multiple *slave* processes. Each slave receives a stream of jobs, performing each of them sequentially and returning their results to the master process. A job is defined by a pair composed of a function `('a -> 'b)` and its argument of type `'a`. The slave process might be viewed as a loop that takes a job from the job queue and evaluates it to get a result of type `'b` that will be returned to the master agent.

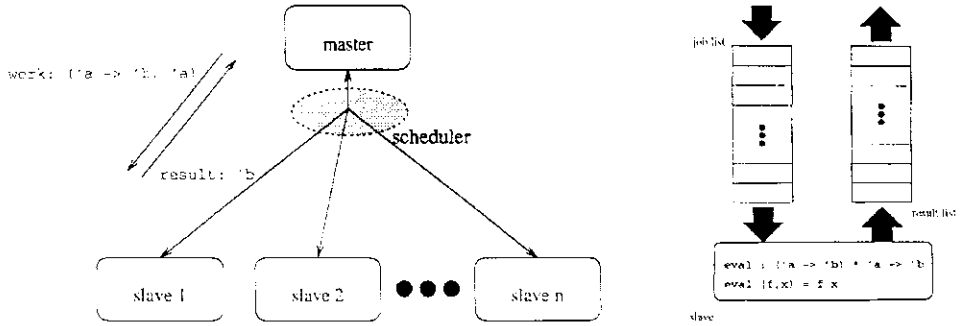


Figure 1: Master-Slave architecture

The key of this scheme is the master agent. This process carries out the main thread of execution. In the middle of this initial thread, the master agent spawns remote jobs on the slave processes to speed up the computation or to perform specialized operations such as vector arithmetic. As a result, the original task is divided into several subtasks that can be achieved concurrently with the main thread. Each task is submitted to a remote processor by the *scheduler*, a component of the master agent, using system parameters such as the current load of processors to decide the target slave agent. In the event of having subtasks of similar computational cost, this strategy degenerates into a round-robin policy. It is well known the importance of the scheduler to achieve the right tunning of the distributed system, but we have decided to use the number of pending tasks in the job queue as an indicator of the slave process workload for simplicity.

The reader should have noticed that if each task is performed sequentially in the slave process, a non terminating task might cause that other computations, needed to compute the final result, will not enter the evaluation loop because of the aforementioned non terminating task. To avoid this effect, the master agent must avoid the spawning of non strict tasks, i.e., tasks which are not 100% needed by the main thread, forbidding any speculative task. A speculative task might be spawned under the assumption of a programmer's proof of its termination. In [3], different scenarios are introduced taking into account the relation between the strictness of a remote computation and the temporal arrival of its resolution. In that paper, a failed speculative task, i.e., a speculative task which is not used in the future, is carried out until completion, and then its result is discarded by the garbage collector. To avoid the waste of computational resources, this task might be removed from the slave queue or the slave eval loop as soon as the garbage collector detects no further references to the failed speculative task.

In the master-slave scenario, there is no communication among the slave agents. The communication is only performed between the master and one slave. For that reason, each subtask sent to a slave should be finished without any additional spawning of new subtasks. With this restriction, the master's functions are: (a) to divide the problem in several sequential (atomic) subtasks and (b) to collect the results of these subtasks. Again, this restriction might be blurred, for example, by delivering to the master agent a join function of type `('a -> 'b -> 'c)` and two remote computations, `('a1 -> 'a, 'a1 )` and `('b1 -> 'b, 'b1)`. Then, the master agent will initiate these two computations and join their results as soon as the original value `'c` is required (figure 2).

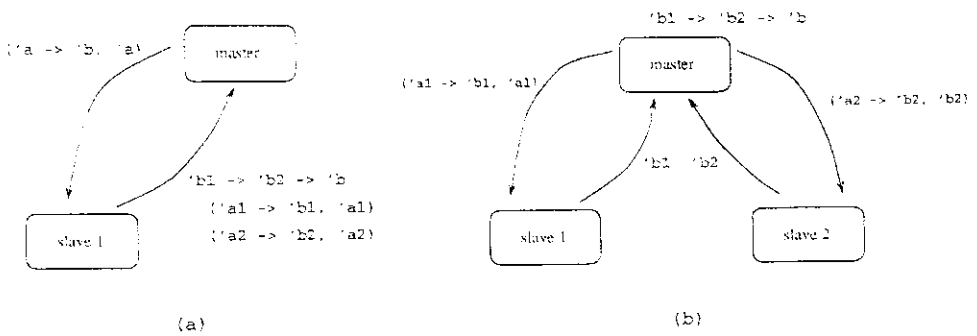


Figure 2: (a) the slave returns a decomposition of the initial task (b) the master spawns both subtasks and then joins the results

2.3 Communication Protocol

The distributed model uses a distributed memory architecture instead of a shared memory one, so encoding/decoding of data as well as its transmision along the communication network is needed. The encoding of data is a refinement of the one exposed in [3][4] in which a traversal of the internal representation of any value is carried out to build a flatten homomorphic representation using a stream of bytes. The type system is used as exposed in [1] to assure the correctness of data transmission among agents. The stream of bytes is sent using a network layer (in our case, the communication primitives offered by the parallel virtual machine, *pvm* [7]). The two main restrictions of the algorithm of data encoding exposed in [3][4] were:

- *No functional values allowed.* Functional values (*closures*) were not first-class citizens. As a consequence, remote services were implemented by using a "service table" in which all the remote services must be declared. The programmer did not have the possibility of creating functions, containing, perhaps, free variables inside the body of the function, and then evaluating those functions remotely. In this new prototype, this restriction has been removed and now it is possible to implement higher-order distributed functional programs, as shown in section 3. A closure might be viewed as a *(n+1)*-tuple, where the first component is a pointer to the code of the function (either bytecode or native code) and the other components are the free variable, which have been lifted from the body of the function.

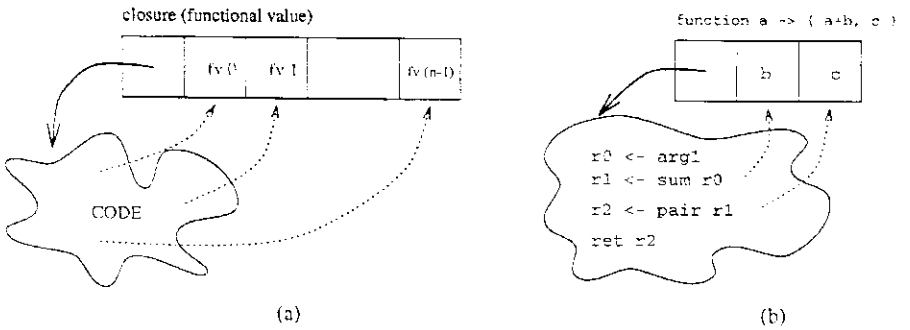


Figure 3: (a) Closure = Code Pointer + Free Variables (b) An Example Closure

There are different ways to encode the function code:

- To transmit the code associated to the function and link it dinamically with the remote process. In principle, that is only possible using the same machine code for each agent.

- If all the agents share the same code, it is enough to transmit a handle that identifies the intended block of code. The compiler must generate a table of pairs where each component associates a handle with the entry point of block of code. There are some special cases:
  * If the distributed system is using machine independent code for both master and slaves, it is sufficient to use the offset of the block of code. Each agent computes the actual address by adding this offset with its bytecode's base.
  * If all the agents share the same code and the same architecture, the actual address can be used. This choice is valid with native code.

- *No sharing considered.* In principle, not considering the sharing is not a big problem (we are loosing some space and we cannot use circular structures). Nevertheless, the target compiler used, Caml Special Light, encodes recursion by creating a circular data structure as shown in figure 4. Thus, the encoding algorithm has had to be redesigned to take into account sharing in order to allow distributed recursive functions.

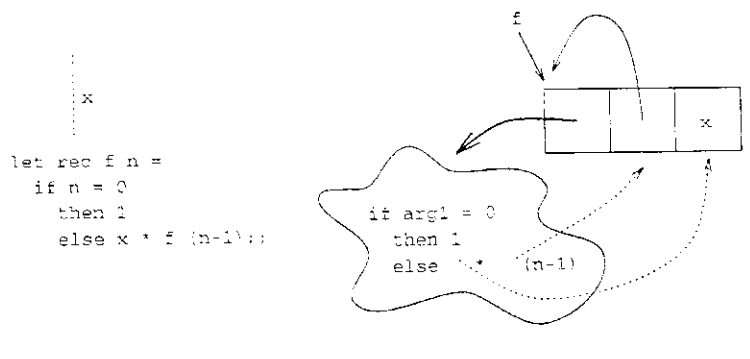


Figure 4: Example Representation of a Recursive Function

2.4 The Distribution Metaphor

Non-strict functional languages have been widely regarded as an attractive basis for parallel computing because they expose all the parallelism in a program. In a language with non-strict semantics, a function body can be entered without the evaluation of its arguments. Non-strict semantics is often combined with lazy evaluation. Lazy evaluation delays the evaluation of an expresion until it is absolutely known that its result must be computed to obtain the overall result. This strategy of evaluation decreases the parallelism in a program. An alternative to lazy evaluation is *lenient evaluation* [10], in which non-strictness is combined with eager evaluation. We can summarize the three main evaluation strategies to evaluate a function call $f\ e_1 \ldots e_n$:

- *Strict Evaluation.* First evaluates all the argument expressions e_1 to e_n, and then evaluates the function body.
- *Lazy Evaluation.* Starts the evaluation of f, delaying the evaluation of every e_i until they are needed.
- *Lenient Evaluation.* Starts the evaluation of f in parallel with the evaluation of all the arguments e_i.

The target language used in this experiment is `Caml Special Light` which has eager evaluation, as most of the dialects of ML. Thus, the synchronism of the evaluation sequence should be broken to allow the main program to spawn the computation of function arguments (and any expression, in general) asynchronously (lenient). To do so, the notion of Multilisp's *future* [6] must be introduced. A future is an abstract datatype that represents a promise of computing an expression for the rest of the program. This structure is combined with a non-strict construct to carry out lenient evaluation. In [5], a utility library is presented to implement lazy and lenient evaluation in the framework of a strict language as CAML LIGHT. A special datatype is defined to represent lazy values as follows:

```
type 'a lazy = Box of 'a
             | Delay of (unit -> 'a);;
```

Delayed computations are implemented using a closure to suspend the evaluation of the value of type `'a` until the lazy value is forced. An additional function, `force : 'a lazy -> 'a`, is implemented to demand the value:

```
let force = function Box x      -> x
                   | Delay susp -> susp ();;
```

The actual implementation encapsulates the `lazy` datatype as an ADT, which mutates itself to avoid repeating the evaluation of the suspension more than once. The previously presented distribution layer offers two primitives to deliver a request to the slaves (`request`) and to receive the result (`sync`). A future might be implemented as follows:

```
(* remote : ('a -> 'b) -> 'a -> 'b lazy *)
let remote f x = let req = request f x
                 in Delay (function () -> sync req);;
```

Remote and local computations can be merged in such a way that programmer can reason as done in sequential programming. It is even possible to choose at runtime whether the distribution is advisable or not, for example, by consulting the workload:

```
(* remote' : ('a -> 'b) -> 'a -> 'b lazy *)
let remote' f x = if slaves_too_busy ()
                  then Box (f x)
                  else remote f x;;
```

3 High-Order Distribution

The expresiveness of functional languages allows to implement and test abstract building blocks which can be instantiated in different scenarios. It is fundamental to use this abstraction mechanism to simplify the construction of distributed programs. Moreover, if typical programmer's patterns are detected, they might be replaced with an equivalent distributed version of that abstraction. A classic example is the `map: (a -> 'b) -> 'a list -> 'b list` function, which applies a function to each element in a list to build a new list with the results. Using the aforementioned primitives, a function `map': ('a -> 'b) -> 'a list -> ('b lazy) list` is implemented to evaluate all the elements concurrently and returning without waiting for any result:

```
let map' f = map (remote f);;
```

High-order function and partial application plays an important role in building this abstraction in an elegant way. An equivalent distributed version of `map` can be implemented as follows:

```
let dmap f = compose (map force) (map' f);;
```

where `compose` represents the function composition `compose f g = function x -> g (f x)`. Even though `dmap` carries out the evaluation in a distributed manner, its semantics is identical to the semantics of `map` (in absence of side-effects, of course) and the programmer can substitute freely one by the other without changing the meaning of his/her program.

Abstraction also provides an excellent framework to implement distributed algorithms. For example, *divide-and-conquer* (d&c) problems can be solved using the following abstraction:

```
let dDivideAndConquer is_triv solve_triv split join data =
  let rec aux d =
    if is_triv d
       then remote solve_triv d
       else let ( d', d'' ) = split d
            in let ( rd', rd'' )  = ( aux d', aux d'' )
               in Delay (function () -> join (force rd') (force rd''))
  in force (aux data);;
```

where `is_triv: 'a -> bool`, test if the problem is so simple that is not worth to spawn a new thread; `solve_triv: 'a -> 'b`, solves the problem sequentially; `split: 'a -> ('a * 'a)`, divides the input data in two partitions; `join: 'b -> 'b -> 'b`, combines two partial solutions to get a more complex solution; `data: 'a`, input data for the problem. The abstraction returns the solution of type `'b`. Pay attention to the fact that all the demands (`forces`) are delayed using suspensions in order to spawn as many requests as possible before forcing anything. Once the abstraction is implemented, it can be used to implement many examples like the fibonacci function:

```
let rec fib = function 0 -> 1
                     | 1 -> 1
                     | n -> fib (n-1) + fib (n-2);;

let dfib = dDivideAndConquer (function n -> n <= 20)
                             fib
                             (function n -> (n-1, n-2))
                             (+);;
```

There are several variations to the previous abstraction. For instance, the computation of a d&c algorithm is strict in the sense that no further work can be made until all the task tree has been evaluated. This can be avoided with an equivalent abstraction which does not force the whole result:

```
let dDivideAndConquer' is_triv solve_triv split join data =
  let ... (* same code here *)
  in aux data;;          (* force removed *)

let dfib' = dDivideAndConquer' ... ;; (* int -> int lazy *)
```

4 Benchmarks

The prototype has been built on top of CAML SPECIAL LIGHT, a functional language with eager evaluation. CSL generates bytecode (code for a virtual machine) and native code. Our prototype can create distributed programs in both formats, although it has some problems with closure encoding in the native code one, particularly in presence of partial evaluation due to optimizations carried out by CSL which change the representation of functional values. Thus, bytecode version has been chosen to perform these benchmarks. Moreover, the current implementation only works on an homogeneous cluster of computers. Both master and slaves share the same code, so closure marshalling is implemented by sending the actual address of the block code, which has a fixed position in memory (no garbage collector involved).

To measure the distributed programs, we have chosen the following scenario: a network of 16 SPARCstations 4, with 16Mb of memory running under Solaris and connected using 10 Mb Ethernet cards. In the experiments, T and T_1 represent the execution time and the sequential execution time in seconds, respectively; Sp is the speedup ($\frac{T}{T_1}$); Eff is the efficiency ($\frac{T}{p*T_1}$); p represents the number of processors (slave agents); dn is the criteria used to choose distributed or sequential execution. Measurements are taken by computing the average real time (unix command `time`) of three executions.

In table 1 we can observe comparative results for two implementations of *divide & conquer* algorithm (section 3) to compute the fibonacci function (1 master and p slaves). In the example, a list with 8 elements of value 36 (`[36;36;...;36]`) is evaluated. The sequential time for `map fib` xs is 702 sg. In the second implementation (using `dDivideAndConquer'`), results are better because tasks created for each element are merged with the others' and the slaves are idle less time. For $p = 16$, the efficiency decreases as long as the amount of overlaped work involved is not sufficient to keep busy all the slaves.

p	(1) T	(2) T	(1) Sp	(2) Sp	ΔSp	(1) Eff	(2) Eff	ΔEff
1	710	710	0.99	0.99	0.00	98.87%	98.87%	0.00%
2	380	359	1.85	1.96	0.11	92.37%	92.77%	5.40%
4	216	186	3.25	3.77	0.52	81.25%	94.35%	13.10%
8	127	105	5.53	6.69	1.16	69.09%	83.57%	14.48%
16	107	95	6.56	7.39	0.83	41.00%	46.18%	5.18%

Table 1: d&c example: (1) `map dfib xs` *vs.* (2) `map force (map dfib' xs)`, with $dn = (n \leq 30)$

As in many other distributed systems, the results depend on the amount of work done by each task. Spawning a new thread in a remote slave is an expensive operation that only can be worthwhile in the event of long tasks. Moreover, scheduling is vital to the right tunning of the system. The scheduler implemented only takes into account the current state of the system at the moment of dispatching a new task, and each slave carries out its chores by following a FIFO policy. Thus, systems with a uniform workload (most of the tasks performs, more or less, the same work) behaves fine, but not with non-uniform workload or networks with changing conditions. In the fibonacci example, the efficiency is quite good because the master always spawns tasks with a similar amount of work (controlled by the parameter dn).

Table 2 shows this behaviour. In this benchmark, the list of all possible solutions to the classical n-queens problem is obtained. The problem is divided into several subtasks by fixing a queen in the first row and then studying all the valid boards with this constrain. The whole solution is achieved by mapping the previous strategy for each position in the first row. The distribution is done by replacing sequential `map` with the abstraction `dmap`. In this case, given that the problem can be easily divided and distributed regularly, we have got high efficiency for $p \in \{1, 2, 4\}$. Nevertheless, the efficiency decreases as soon as the distribution of work disagrees with the current topology. For example, for $p = 8$, there are 12 tasks of the same computational cost that have to be distributed among 8 agents. Thus, 4 slaves carry out 2 tasks (idle $\simeq$ 0%), but the other four only one (idle $\simeq$ 50%). For $p = 16$ there are not enough tasks for all the slaves, so some of them remain idle during execution. As can be seen, results depend on the distribution of workload among slaves.

Processors	T	Speedup	Efficiency
1	347	0.99	99.14%
2	180	1.91	95.56%
4	92	3.74	93.48%
8	61	5.64	70.49%
16	36	9.56	59.72%

Table 2: `queens` 12 (sequential time $T_1 = 344$ sg)

5 Conclusions and Future Work

A prototype system to perform distributed computation using a functional language has been presented. The system uses a master-slave architecture in which the master splits the work while the slaves compute those subtasks. Although functional programming is quite suitable to exploit implicit distribution and/or parallelism, the explicit approach has been taken and it can be a support layer for future implicit implementations.

We focus on abstraction to simplify the development of distributed applications by implementing the distribution skeleton using high-order functions. The programmer can forget about the way his/her program is being executed. Classical techniques such as equational reasoning can be used to prove the correctness of functional code even though the distributed execution.

As many other distributed systems, results depend on the amount of work done by each task. Again, the scheduler has to be improved to manage better system information and, perhaps, we should include more information to guide the distribution process such as annotations or previous executions profiles.

The chosen architecture prevents the slaves to spawn new threads, which is a serious restriction. Although a solution was exposed in this paper, the general model should be modified to allow cooperation among slaves. The key idea is to implement the distribution layer on top of a local concurrent one, and to enable a mechanism to distribute dinamically the pending tasks such as work stealing. In addition, asymetric distribution seems to be an interesting possibility to integrate workers with different capabilities (for example, workers specialized in vector arithmetic, graphics, and so on). Those are our current lines of research.

References

[1] María-Virginia Aponte and Xavier Leroy. Llamado de procedimientos a distancia y abstracción de tipos. In *Proc. 20th CLEI PANEL latino-american computer science conference*, 1994.

[2] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-Structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[3] J.L. Freire, V.M. Gulías, and B.B. Fraguela. Extending Caml Light to perform distributed computations. In *Joint Conference on Declarative Programming (GULP-PRODE'95)*, Salerno, Italy, September 1995.

[4] V.M. Gulías and A. Valderruten. Une interface graphique distribuée pour le λ-calcul supportée par des serveurs fonctionnels. In *Journées Francophones des langages applicatifs (JFLA'95)*, pages 229–246, Bois d'Amont, France, January 1995. Published in INRIA Collection Didactique, 13.

[5] V.M. Gulías, A. Valderruten, and J.L. Freire. Evaluation Lazy avec Caml Light. In *Journées Francophones des langages applicatifs (JFLA'96)*, pages 77–94, Val Morin, Québec, January 1996. Published in INRIA Collection Didactique, 15.

[6] R. H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Prog. Lang. and Sys.*, 7(4):501–538, October 1985.

[7] Jack Dongarra and others. *A Users' Guide to PVM Parallel Virtual Machine*. Oak Ridge National Laboratory, July 1991.

[8] M.P. Jones and P. Hudak. Implicit and explicit parallel programming in haskell. Research Report YALEU/DCS/RR-982, Yale University Department of Computer Science, New Haven, CT, 1993.

[9] Xavier Leroy. Le système Caml Special Light: modules et compilation efficace en Caml. Rapport de recherche 2721, INRIA, November 1995.

[10] Klaus E. Schauser and Seth C. Goldstein. How much non-strictness do lenient programs Require? In *FPCA*, San Diego, CA, June 1995.

[11] P. Weis, M. V. Aponte, A. Laville, M. Mauny, and A. Suárez. The CAML reference manual. Technical report, Projet Formel, INRIA-ENS, 1989. Version 2.6.

Efficient Producer/Consumer Parallelism in Logic Programming

Ramiro Varela, Camino R. Vela, Jorge Puente
Artificial Intelligence Centre. University of Oviedo at Gijón
Campus de Viesques. E-33271 Gijón. Spain
Tel. +34-8-5182032. FAX +34-8-5182150
email:{ramiro,camino,puente}@aic.uniovi.es

Abstract

A model to exploit Producer/Consumer parallelism, as long as Independent AND- and full OR-parallelism in evaluation of logic programs is proposed. The efficiency of the model is due to the prevention of processes from duplication. This is achieved by means of a representation based on ordered structures. Additionally, a comparison with some classical models is established.

KEYWORDS: Producer/Consumer Parallelism, AND Parallelism, OR Parallelism, Ordered Structures

1 INTRODUCTION

It is well known that evaluation of logic programs offers many possibilities of parallelism. There are two main sources: *OR parallelism*, which stands for simultaneous exploitation of several clauses with the same conclusion, and *AND parallelism* which allows to evaluate at the same time two or more literals of a query. Moreover, there are some secondary sources: *unification, search, stream, producer/consumer,* etc. These types of parallelism can be exploited independently from each others, and of course each of them presents advantages and problems.

Many models have been proposed that use one or several of the above forms of parallelism. Most of them use, at least, one of the main sources: *AND* and *OR*. In the first case, it is usual a variant called *Restricted or Independent AND Parallelism (IAP)*. This consists of the ordered evaluation of predicates within a query, so that two or more predicates are evaluated in parallel only if they do not share any free variable. Therefore, if several literals have a common variable, one of them is chosen as producer of the variable and it is evaluated first. Then, the others, as consumers of the variable, can be evaluated in parallel with the shared variable instantiated to the term produced by the first one.

As it was pointed out [Con83, DeG84], even though *IAP* reduces the amount of parallelism with respect to *full AND parallelism* (simultaneous evaluation of all literals of the query), it is more efficient since the join of solutions from different predicates is simplified, and a lot of redundant work can be avoided. On the other hand, full *OR* parallelism is easy to implement. However, a combinatorially explosive number of partial solutions can be produced in highly non deterministic programs. If this happens, it could be better to explore fewer clauses at the same time, or even to revert to a depth first control strategy. Several models have been proposed to exploit both *IAP* and full *OR* parallelism:

[Con83, Kal85, 91, Li86, Gup92, 94, Esc93] among others.

Producer/consumer is a secondary source of parallelism that has to do with both *AND* and *OR* in presence of non determinism. Given two literals with common variables, it consists of starting evaluation of one instance of the second *(consumer)* literal to compute compatible solutions with one solution of the first *(producer)*, as soon as this solution appears. For example, if the query is $p(X,Y),q(Y,Z)$ and $p(X,Y)$ has multiple solutions, we can start exploring $q(y_i,Z)$ as soon as we have a new value (Y/y_i) from a solution to $p(X,Y)$. Producer/consumer parallelism is interesting in non deterministic programs. Furthermore, in presence of infinite relations, it holds the completeness of the system. As it is pointed out in [Kal85] exploitation of this parallelism was the point of departure for the *REDUCE-OR Process Model (RPM)*. Here it is called *Consumer/Instance Parallelism*. This was also one of our goals [Var95b].

In this paper we present our strategy to manage producer/consumer parallelism. This strategy is compared with the *RPM* one. As we will see along the paper, we improve the producer/consumer parallelism by avoiding generation of duplicated processes. For instance, if we have two solutions, $(X/x_1,Y/y_1)$ and $(X/x_2,Y/y_1)$, to the first predicate in the former example, only one process to solve $q(y_1,Z)$ is generated. This process can be created as soon as the first solution appears, and when the second one is produced, it can be combined with the solutions computed to $q(y_1,Z)$.

The rest of the paper is organised as follows: In section 2 we briefly outline the proposed model. Section 3 shows, via examples, problems that appear in producer/consumer parallelism management. Sections 4 and 5 present the algorithms to solve these problems. Finally, in section 6 we discuss the efficiency of the proposed schema and it is compared with the *RPM* strategy.

2 THE PROCESS MODEL

The proposed process model is thoroughly described in [Var95b]. Here we just outline it to introduce necessary concepts and terminology. The model exploits both independent *AND (IAP)*, and full *OR* parallelism. It is based on the usual *AND/OR* tree representation of processes. Hence, there are two types of processes: *AND processes* to solve queries and *OR processes* to solve single literals. An *AND* process generate an *OR* process to solve a predicate of its query. An *OR* process creates an *AND* process to solve the query obtained from the antecedent of a clause which consequent unifies with the literal of the node. The root of the tree is an *AND* node to solve the initial query. Leaves are either *AND* nodes created from facts, or *OR* nodes which predicate do not unify with the consequent of any clause.

As it is usual [Con83, Deg84, Kal85, 91], a partial order among literals within a query is established to manage *IAP*. In our model this order is represented by a directed graph named *Data Flow Lattice (DFL)*. For simplicity, here, we assume that this order is static and it can be known at compilation time from the variable names. Nodes in this graph are labelled with one literal (or with a null label), and arcs express precedence for evaluation. We denote by $\geq_{DFL}$ the ordering codified in the *DFL*. As it is shown in [Var94, 95a, 95b], we impose four properties to the *DFL*, in order to achieve an efficient strategy of *IAP*:

P1. *It has degree two:* every node has neither more than two incident nor emergent arcs.

P2. *Dependence of consumed variables:* every variable of a literal appears either in an immediate predecessor or in no ancestor.

P3. *It is a semilattice with respect to the infimum:* so it has minimum and every pair of nodes with common ancestors has supremum.

P4. *It is a structured graph:* given two nodes with one common child and with supremum, every path between an ancestor node of the supremum and one of those nodes passes through the supremum.

Figure 1 shows three examples of *DFLs*. As we can see, null nodes are introduced in order to guarantee the former properties. These nodes do not have any influence on the amount of parallelism expressed in the *DFL* because it is not necessary to create processes to solve them. They are like predicates that always are true.

AND processes generate *OR* processes and manage its answers, taking into account the order coded by the *DFL*, in the following way: first, an *OR* process to solve each maximal predicate of the *DFL* is started. When an answer arrives from a son *OR* process, it is joined with the answers to its ancestor predicates in the *DFL* and, if it is necessary, new *OR* processes are generated to search compatible solutions to its successor predicates. When a solution to the minimum predicate appears, it is also joined to solutions from the rest of predicates to get solutions to the query.

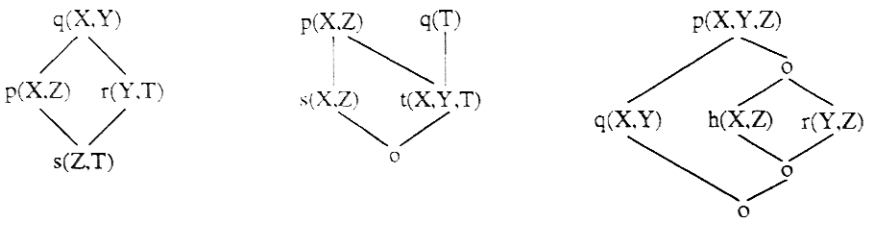
In order to proceed in that way, *AND* processes make a compact representation for partial solutions and *OR* process identifiers, by means of a structure named *Processes and Solutions Net (PSN)*. Figure 2a shows the *PSN* generated after evaluation of the query

$$q(X,Y),\ p(X,Z),\ r(Y,T),\ s(Z,T),$$

represented by the *DFL* of figure 1a, with respect to the logic program given by the set of facts

$$q(a,b),\ q(d,b),\ q(a,c),\ p(a,i),\ p(d,j),\ r(b,k),\ r(c,l),\ s(i,k),\ s(j,k),\ s(i,l).$$

A *PSN* has two kinds of nodes: nodes as $^{proc}q(X,Y)$ identifies *OR* processes generated to solve the predicates, and the others represent *partial solutions*, i.e. solutions to predicates (label *V* means *True*). The *PSN* is composed by a set of *inheritance nets*, one associated to each literal of the *DFL*. An inheritance net represents the *OR* processes that solve a predicate, as well as the set of computed solutions. These solutions are attached to the process identifiers that computed them as it is shown in figure 2c. Minimum nodes of an



a) b) c)

Figure 1

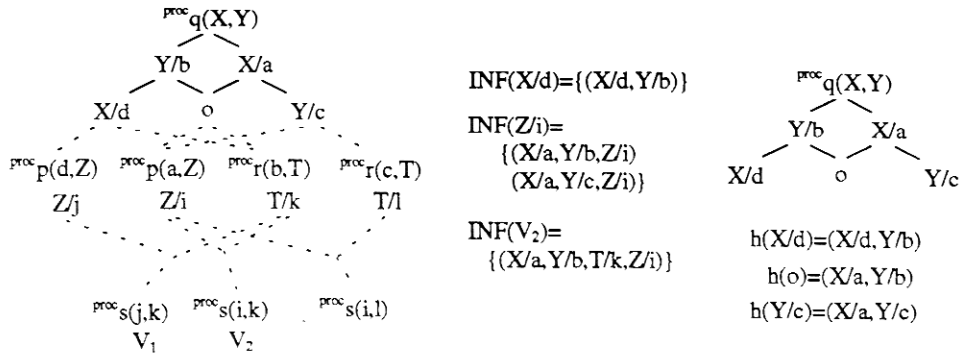


Figure 2

inheritance net are called *solution nodes*, because each of them represents one solution to the literal. These solutions are made explicit through the *inheritance function h*, as it is also indicated in figure 2c. Inheritance nets of predicates are linked to compose the *PSN* in such a way that a process is linked to a set of solution nodes as follows:

a) if the predicate has no predecessors in the *DFL*, there are no links from the process to any solution node,

b) if the predicate has only one predecessor, there are links to a set of solution nodes of the predecessor, and

c) if the predicate has two predecessors, there are links to a set of pairs of solution nodes, one from each predecessor.

The set of solution nodes linked to a process is called its *context*. Depending on the above cases, we said that the context is *null*, *single* or *double* respectively.

We denote as $\geq_{PSN}$ the partial order among solution nodes of the *PSN*. Hence, $s_p > _{PSN} s_q$ iff $p >_{DFL} q$, and s_p belongs to the context of the process that computed s_q.

Partial solutions are joined by the *inference function INF*. This function applied to a solution node returns the set of solutions represented by this node to the conjunction of the predicate and its ancestors in the *DFL*. Hence, if it is applied from solution nodes of the minimum, the function computes answers to the query. Figure 2b shows the application of the inference to various solution nodes. The function *INF* can be defined as follows, being *C* the context of the process that computed the solution node *s*.

$INF(s) =$

a) if *C* is null: $h(s)$,

b) if *C* is single: $h(s) * \bigcup_{s' \in C} INF(s')$,

c) if *C* is double: $h(s) * \bigcup_{(s_1, s_2) \in C} \{INF(s_1) * INF(s_2)\}$.

Here, "*" denotes the usual join of substitutions. It is defined for a pair of substitutions, S_1 and S_2, as

$S_1 * S_2 =$

a) if there is a variable with diferent instantiation in S_1 and in S_2, or one of the substitutions is *False*: *False*,

b) otherwise: $S_1 \cup S_2$.

Given two solution nodes s_p and s_q from predicates p and q respectively, we say that they are *compatible* iff one of the following conditions holds:

a) $h(s_q) \subset S$, for some $S \in INF(s_p)$, and q is ancestor of p in the *DFL*,

b) neither p is ancestor of q nor q is ancestor of p, and p and q have common ancestors and then supremum r, and there is a node s_r such that

$$h(s_r) \subset S_p \in INF(s_p) \text{ and } h(s_r) \subset S_q \in INF(s_q),$$

c) neither p is ancestor of q nor q is ancestor of p, and they have no common ancesors.

Roughly speaking, two solution nodes, from predicates p and q respectively, are compatible if they take part in a valid solution to the conjunction of p, q and every ancestor to p or q.

As it is pointed out in [Var95b], the above definition of *INF* does not suggest an efficient algorithm to join partial solutions, because of the repetitive search over the section of the *PSN* reachable both from s_1 and from s_2. An alternative definition that induces a more efficient algorithm is also proposed. Experimental results showed in [Var95a, b] prove that the inference can be efficiently computed. The new definition for *INF* is founded on the four properties imposed to the *DFL*. This definition requires a new format to represent solutions: a solution is not only represented by a *substitution* but also by a *set of marks* from some solution nodes (these marks play a similar role to the sync marks used in the *SYNC* model [Li86]). Therefore, a solution is denoted by a pair

$$(set\_of\_marks;\ substitution).$$

Now, the join operator among solutions is denoted by "**" and it is defined as

$(SM_1; S_1) ** (SM_2; S_2) =$

a) *if* $(SM_1 * SM_2) = NULL$: *False*,

b) *otherwise:* $(SM_1 * SM_2;\ S_1 \cup S_2)$.

Where in this case, "*" denotes the join among sets of marks, defined as

$SM_1 * SM_2 =$

a) if both SM_1 and SM_2 have one mark from the same literal but from different solution nodes: *NULL*,

b) otherwise: $(SM_1 \cup SM_2)$.

Here, it is worth noting that the number of marks within a solution is generally much lesser than the number of variables in the respective substitution; so the joining of marks is less costly than the joining of substitutions. In the next section we make clear these operations with some examples.

3 MANAGING PRODUCER/CONSUMER PARALLELISM

In this section we introduce our strategy to exploit producer/consumer parallelism via some examples that point out main problems to deal with this class of parallelism. In the next two sections we propose algorithms to solve them.

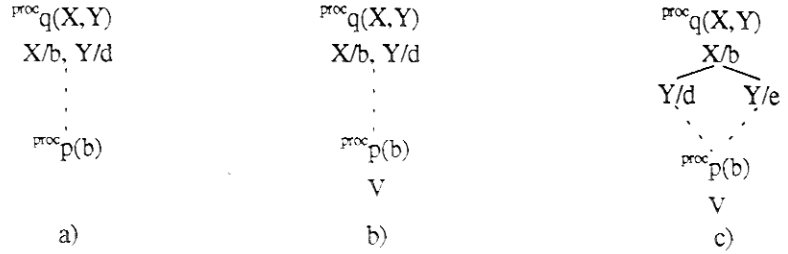


Figure 3

In the first example we have the situation of figure 3a. In this case producer/consumer parallelism allows to processes $^{proc}q(X,Y)$ and $^{proc}p(b)$ run in parallel, in such a way that anyone can produce a new answer. For example, if the second one produces the response *V*, we have the solution *(X/b,Y/d)* to the query as figure 3b shows. When the first process sends the response *(X/b,Y/e)* the *PSN* results in the one of figure 3c. As we can see, the new response does not give rise to a new process for *p(Y)* but that it is included in the context of a process what exists. In this situation we have a new solution to the query. The problem here is how to detect and extract this solution without re-computing solutions. At a first glance, one way to solve it is to detect solution nodes of the minimum compatible with the new one, and then apply the inference to these nodes but restricted to the new solution node; i.e. keeping every solution node of *q(X,Y)* but the new one *(Y/e)* masked.

The problem is a little bit more complex if a predicate has two successors in the *DFL*. Let us consider the example of figure 4a and the associated *PSN* of figure 4b. In this situation, if the process $^{proc}p(X,Y,Z)$ produces the response *(X/a,Y/b,Z/g)*, the *PSN* results in the one of figure 5a. In this *PSN* we can observe that there are not new processes, but an additional solution to the query is coded in the *PSN: (X/a,Y/b,Z/g,T/1,U/3)*. However, if the process $^{proc}p(X,Y,Z)$ sends the response *(X/a,Y/e,Z/h)*, neither processes are generated for the successors, *q(X,T)* and *r(Y,U)*, nor new solutions to the query arise. But solution nodes *(T/1)* and *(U/4)* become compatible as a consequence of the last solution to *p(X,Y,Z)*. Therefore, the process $^{proc}s(1,4)$ should be generated.

The former examples show the situations that can arise when a new solution to a predicate *p* does not generate a new process for the successor predicates: if *p* has only one successor, then additional solutions can appear to the query if there are solution nodes of the minimum compatible with the new solution. However, if *p* has two successors then either new solutions can appear to the query, or new processes might be necessary for the infimum literal of the successors of *p*. These situations should be carefully managed in order to control producer/consumer parallelism with no loss of completeness.

On the other hand, it is easy to see that if a new solution gives rise to a new process for a successor literal, no special action has to be taken to handle producer/consumer parallelism in order to guarantee completeness.

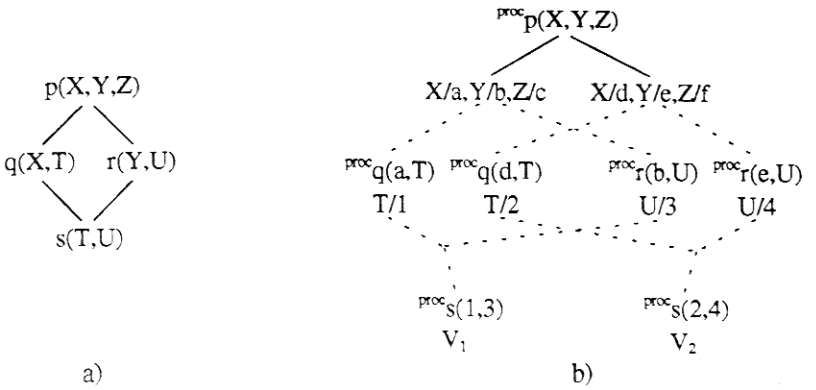


Figure 4

In the next two sections, we propose solutions to the former problems. First to extract new solutions to the query when a new solution node for a literal *p* arises that does not generate any process for the successor literals in the *DFL*. Then, to detect new processes that should be generated for the infimum of the successors of *p*.

4 EXTRACTING NEW SOLUTIONS

As we pointed out, to extract new solutions to the query, first we have to detect solution nodes of the minimum literal, *min*, compatible with the new solution node s_p. This is accomplished by means of the function *detect_solutions*(s_p,*min*). Then, we have to apply the inference from those nodes but restricted to the new solution node, s_p, that is to say keeping out the rest of solutions of the predicate *p*. This is achieved through the function *INFR*(s_p,s_r). This is a variant of the inference function defined in [Var95] that computes the inference from the solution node s_p, keeping out the solution nodes of *r* except s_r. These functions are defined as

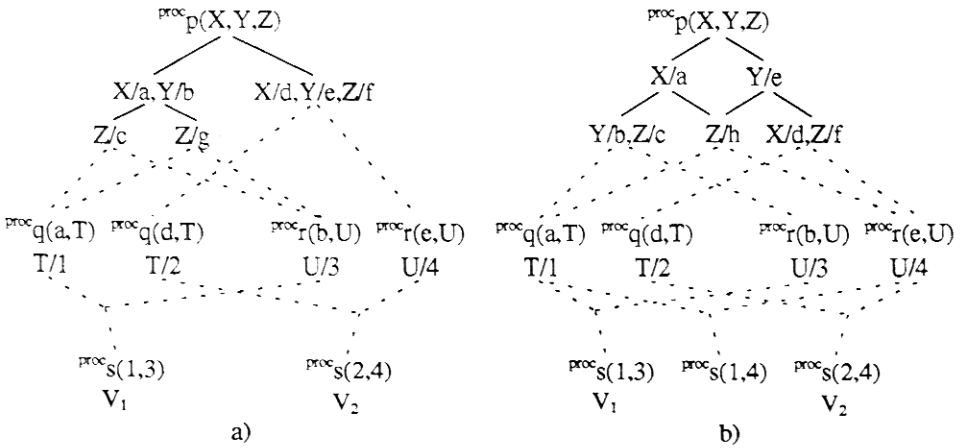


Figure 5

$detect\_solutions(s_p, q) =$

a) if $p=q$: $\{s_p\}$,

b) if p has one successor p_1: $\bigcup_{s_{p_1} <_{PSN} s_p} detect\_solutions(s_{p_1}, q)$,

c) if p has two successors p_1 and p_2 with infimum r (it can be proved that $r \geq_{DFL} q$)

$$\bigcup_{s_r \in CS} detect\_solutions(s_r, q),$$

where CS is the set of solution nodes of r compatibles with s_p, it is computed as

$$CS = \left(\bigcup_{s_{p1} <_{PSN} s_p} detect\_solutions(s_{p_1}, r) \right) \cap \left(\bigcup_{s_{p2} <_{PSN} s_p} detect\_solutions(s_{pq_2}, r) \right),$$

On the other hand, the *inference function* to apply from each solution node, s_q, computed by the function *detect_solutions*, restricted to the node s_p, is defined as

$$INFR(s_q, s_p) = INF\_R3(s_q, \varnothing, s_p),$$

where *INF_R3* is defined as follows, being CP a set of predicates

$INF\_R3(s_q, CP, s_p) =$

a) if $p=q$ and $s_p \neq s_q$: *False*,

b) if $q \in CP$: $\{(M_{s_q}; \varnothing)\}$,

c) if $q \notin CP$ and q has no predecessors in the *DFL*: $\{(\varnothing; h(s_q))\}$,

d) if $q \notin CP$ and q has only one predecessor p_1:

$$\{(\varnothing; h(s_q))\} ** \bigcup_{s_{p_1} >_{PSN} s_q} INF\_R3(s_{p_1}, CP, s_p),$$

e) if $q \notin CP$ and q has two predecessors p_1 and p_2 with no common ancestors:

$$\{(\varnothing; h(s_q))\} ** \bigcup_{s_{p_1}, s_{p_2} >_{PSN} s_q} \{INF\_R3(s_{p_1}, CP, s_p) ** INF\_R3(s_{p_2}, CP, s_p)\},$$

f) if $q \notin CP$ and q has two predecessors p_1 and p_2 with common ancestors, and then with supremum p_{12}:

$$\{(\varnothing; h(s_q))\} **$$
$$\bigcup_{s_{p1}, s_{p2} >_{PSN} s_q} \{INF\_R3(s_{p_1}, CP \cup \{p_{12}\}, s_p) ** INF\_R3(s_{p_2}, CP \cup \{p_{12}\}, s_p) **$$
$$\bigcup_{s_{p12}} \{\{(M_{s_{p12}}; \varnothing)\} ** INF\_R3(s_{p_{12}}, CP, s_p)\}\}.$$

Here, we have only to take into account nodes s_{p12} compatibles with both s_{p1} and s_{p2}, i.e. nodes s_{p12} which mark Ms_{p12} appears in some solution from the join

$$INF\_R3(s_{p_1}, CP \cup \{p_{12}\}, s_p) ** INF\_R3(s_{p_2}, CP \cup \{p_{12}\}, s_p)$$

Now, we make clear these functions through their application to the node *(Z/g)* of figure 5a. First, we detect solution nodes of the minimum *s(T,U)* compatible with node *(Z/g)*. As *p(X,Y,Z)* has two successors,

$$detect\_solutions(Z/g, s(T,U)) = \bigcup_{s_r \in CS} detect\_solutions(s_r, s(T,U)),$$

where the set of solutions CS is computed as

$$CS = detect\_solutions(T/1, s(T,U)) \cap detect\_solutions(U/3, s(T,U)) =$$
$$detect\_solutions(V_1, s(T,U)) \cap detect\_solutions(V_1, s(T,U)) = \{V_1\} \cap \{V_1\} = \{V_1\}$$

so

$$detect\_solutions(Z/g, s(T,U)) = detect\_solutions(V_1, s(T,U)) = \{V_1\}.$$

Therefore, we have to apply the inference from the node V_1 restricted to the new node Z/g

$$INFR(V_1, Z/g) = INF\_R3(V_1, \varnothing, Z/g) =$$
$$\{(\varnothing; h(V_1))\} ** INF\_R3(T/1, \{p(X,Y,Z)\}, Z/g) ** INF\_R3(U/3, \{p(X,Y,Z)\}, Z/g) **$$
$$\bigcup_{s_{p12}} \left(\{(M_{s_{p12}}; \varnothing)\} ** INF\_R3(s_{p_{12}}, \varnothing, Z/g) \right),$$

here, to know nodes s_{p12} we have to compute the previous joins within the expression, so

$$INF\_R3(T/1, \{p(X,Y,Z)\}, Z/g) = \{(\varnothing; h(T/1))\} **$$
$$\{INF\_R3(Z/c, \{p(X,Y,Z)\}, Z/g) \cup INF\_R3(Z/g, \{p(X,Y,Z)\}, Z/g)\} =$$
$$\{(\varnothing; T/1)\} ** \{False \cup \{(M_{Z/g}; \varnothing)\}\} = \{(M_{Z/g}; T/1)\},$$

and

$$INF\_R3(U/3, \{p(X,Y,Z)\}, Z/g) = \{(M_{Z/g}; U/3)\},$$

thus, taking into account that $M_{Z/g}$ is the only mark in the join of the above terms, we can compute the last term of the inference as

$$\bigcup_{s_{p12}} \left(\{(M_{s_{p12}}; \varnothing)\} ** INF\_R3(s_{p_{12}}, \varnothing, Z/g) \right) =$$
$$\{(M_{Z/g}; \varnothing)\} ** INF\_R3(Z/g, \varnothing, Z/g) = \{(M_{Z/g}; \varnothing) ** (\varnothing; h(Z/g))\} =$$
$$\{(M_{Z/g}; \varnothing) ** (\varnothing; X/a, Y/b, Z/g)\} = \{(M_{Z/g}; X/a, Y/b, Z/g)\}.$$

And finally the inference from V_1 restricted to Z/g is

$$INFR(V_1, Z/g) = \{(\varnothing; V_1) ** (M_{Z/g}; T/1) ** (M_{Z/g}; U/3) ** (M_{Z/g}; X/a, Y/b, Z/g)\} =$$

$$\{(M_{Z/g}; X/a, Y/b, Z/g, T/1, U/3)\}.$$

5 DETECTING NEW PROCESSES

The second problem to solve, in order to manage producer/consumer parallelism with no loss of completeness, is to detect new processes that should be created for the infimum q of the successors p_1 and p_2 of a predicate p, when a new solution s_p does not give rise to a new process either for p_1 and for p_2. Let q_1 and q_2 be the predecessors of q, then in order to detect new processes for q, we have to detect all pairs of solution nodes (s_{q1}, s_{q2}) such that s_{q1} and s_{q2} are compatible with s_p and a process associated to (s_{q1}, s_{q2}) does not exist yet. From every pair (s_{q1}, s_{q2}) in this situation, a new process for q must be generated. This is made by means of the *resolve_processes* function defined as

$resolve\_processes(s_p, \{q_1, q_2\}) =$

return the set of pairs of solution nodes (s_{q1}, s_{q2}), with s_{q1} from $detect\_solutions(s_p, q_1)$ and s_{q2} from $detect\_solutions(s_p, q_2)$, such that a process for q associated to (s_{q1}, s_{q2}) does not exist yet. As a side effect the respective processes to the returned pairs of solution nodes are generated.

In order to make clear this function, it is applied to the situation of figure 5b when the node (Z/h) has just appear. To detect and generate the necessary processes for $s(T,U)$, which is the infimum of the successors of $p(X,Y,Z)$, we call up the function as

$$resolve\_proccesses(Z/h, \{q(X,Y), r(Y,U)\}),$$

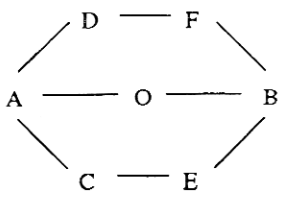
given that

$$detect\_solutions(Z/h, q(X,Y)) = \{(T/1)\} \text{ and } detect\_solutions(Z/h, r(Y,U)) = \{(U/4)\},$$

the set $\{(T/1, U/4)\}$ is returned and as a side effect the process $^{proc}s(1,4)$ is generated.

6 DISCUSSION

We have proposed an interpretation model for logic programs that allows to exploit producer/consumer parallelism, as long as *IAP* and full *OR* parallelism. An important feature of the model is the prevention of processes from duplication. This duplication might occurs in some *AND/OR* parallel models as *RPM* [Kal85, 91], the *extended And-Or tree model* proposed in [Gup92], and in the *Sync model* [Li87]. Moreover, in the last one producer/consumer parallelism is not allowed; as well as in the *Multisystem model* proposed in [Esc93]. Those models would generate two processes to solve $q(y_1, Z)$ in the example alluded to in the Introduction; whereas as we saw, with our model only one is required. A more significant example to compare our model with the *RPM* is proposed in [Var96]. There, we indicate the number of processes generated to solve the instance of the *Map-colouring* problem showed in figure 6a. This example is taken from [Kal91]. It is assumed that the node *O* is red coloured and the logic program is the one of figure 6b to compute all answers. The number of processes created is shown in figure 6c, where n is



a)

```
colour(red,A,B,C,D,E,F):-
    next(red,A),next(red,B),
    next(A,C),next(A,D),
    next(B,E),next(B,F),
    next(C,E),next(D,F).
next(red,green).
next(green,red).
.....
```

b)

Our Model	$2+4(n-1)+2n^2$
RPM	$2+4(n-1)+2(n-1)^4$

c)

Figure 6

the number of allowed colours $(n \geq 3)$.

On the other hand, our model introduces more complexity to compute the inference than the *RPM*, because we might compute several times the inference from some solution nodes, whereas in the *RPM* it is computed only once and its value is stored for further use, so that it wastes more space. Even so, we think that prevention of processes from duplication in non deterministic programs will overcome the different cost of the inference.

7 BIBLIOGRAPHY

[Con83] J. S. Conery. *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. Ph. D. Dissertation. Dpto. Information and Computer Science. University of California, Irvine. 1983.

[DeG84] D. DeGroot. *Restricted And-Parallelism*. Proceedings of the International Conference on Fifth Generation Computer Systems. North Holland. pp. 471-478. 1984.

[Esc93] G. Escalada-Imaz: *Exploiting Restricted AND-Parallelism and OR-Parallelism in Logic Programs with Multisystems*. Workshop on Parallelism and Implementation Technologies, 13-20. Madrid (1993).

[Gup92] G. Gupta. *Parallel Execution of Logic Programms on Shared Memory Multiprocessors*. Ph. D. Dissertation. Dpto. of Computer Science. University of North Carolina at Chapel Hill. 1992

[Gup94] G. Gupta and V. Santos Costa. *Optimal implementation of And-OR Parallel Prolog*. Future Generation Computer Systems, Vol. 10, No. 1, pp. 71-92. Apr. 1994.

[Kal85] L. V. Kale. *Parallel Architectures for Problem Solving*. Ph. D. Dissertation. Dpto. Computer Science. SUNY. Stony Brook. 1985.

[Kal91] L. V. Kalé. *The REDUCE-OR Process Model for Parallel Interpretation of Logic Programs*. The Journal of Logic Programming. Vol 11, pp. 55-84. 1991.

[Mut90] K. Muthukumar and M. Hermenegildo. *The CDG, UDG, and MEL Methods for Automatic Compile-Time Parallelization of Logic Programs for Independent And-Parallelism*. ICLP'90, pp. 221-237. MIT Press, Jun. 1990.

[Li86] P. P. Li and A. J. Martin. *The SYNC Model: A Parallel Execution Method for*

Logic Programming. Proceedings of the Symposium on Logic Programming, Salt Lake City, Sep. 1986.

[Var94] R. Varela. *El Modelo RPS para la Gestión del Paralelismo AND Independiente en Programas Lógicos*. Proccedings of the 1994 Joint Conference on Declarative Programming GULP_PRODE'94, pp. 251-265. 1994.

[Var95a] R. Varela, E. Sierra, L. Jimenez y C. R. Vela. *Combinación de Soluciones Parciales en Programación Lógica Paralela*. Proceeding of the VI Conference of the AEPIA. Alicante, November 1995.

[Var95b] R. Varela. *Un Modelo para el Cálculo Paralelo de Deducciones en Lógica de Predicados*. Ph. D. Dissertation. Dpt. Mathematics, University of Oviedo. December 1995.

[Var96] R. Varela and C. R. Vela. *AND/OR Trees for Parallel Deductions*. To appear in the International Conference on Intelligent Technologies in Human-Related Sciences, ITHURS'96. León, Spain. July 5-7.

This work has been supported by the University of Oviedo under the Project DF/95-213-9.