Improving Distributed Unification through Type Analysis

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@ing.unife.it
[3] Dept. 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 (*zero-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 an implementation scheme for distributed logic programming which consists of tailoring the copying level to each procedure. The scheme is based on a consumption specification which describes the way the procedure "consumes" its arguments locally. The consumption specification (or an approximation of it) can be automatically obtained through a static analysis inspired by traditional type analyses. The paper also describes a high-level distributed implementation that uses the consumption specification to avoid unnecessary copying and to request data structures globally. Experimental results 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 (*zero-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 [4], where each message carries the infinite-level copying of each argument of the goal to be solved. Ichiyoshi et al. [12] introduced the copying level concept in the implementation of the KL1 language on the MultiPSI machine where a fixed level of copying can be "a-priori" used 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 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 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. The implementation uses this specification to improve both the sending of arguments to goal processes and the remote dereferencing phase possibly occurring during clause head unification. The copying phase is improved by avoiding the copy of subterms that are not used by the called procedure. 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 [5] (or type graphs [8, 17]). Consumption specifications (or approximations of them) can be obtained automatically through a static analysis inspired by traditional type analyses, making the approach fully transparent for programmers.

The paper also describes a high-level implementation of the scheme based on attributed variables following the lines suggested by [7]. The implementation uses a blackboard for communication. Experimental results show the potential of this approach.

2 Consumption-based Distributed Unification

In our distributed model, a number of concurrent processes cooperate in solving the query. 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 distributed unification. The main difference between distributed and sequential implementations of unification is that the first demands message management for data exchange.

The `put` phase in the standard Warren Abstract Machine (WAM) [1, 18] loads the arguments in 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* phase 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 [9, 15, 16, 12]:

- the argument is copied in its entirety in the message; this is called *infinite-level copying*;
- a remote reference to the structure is included in the message; this is called *zero-level copying*;

- the structure is copied in the message up to a certain level with remote references to the rest of the structure; this is called *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 standard 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 might be necessary to request to the appropriate remote processes 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 of the policies described for the message preparation phase.

None of the previously described strategies (infinite-level copying, zero-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 depends not only on the structure of the application, but also on the architecture type (in particular, the communication cost), since it affects the cost of the various operations. An analysis of the architectural factor in the choice of the copying policies may be found in [2].

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 Prolog program for list concatenation [13]:

```
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. `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 may not need to inspect its second and third arguments. As a consequence, the top-level goal 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. We are interested in driving unification by the minimal level needed for term inspection, which we call consumption.

2.1 The Consumption Specification

In order to specify how a procedure minimally "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 4 how to automatically obtain them (or an approximation of them) through a static analysis inspired by traditional type analyses, making the approach transparent to programmers. In practice, a consumption specification describes a superset of the type of the procedure, as it will be clear in Section 4.

A consumption specification is expressed as a tree grammar [5] 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*

```
T1 ::= [] | cons(Remote,T1).
T2 ::= Remote.
T3 ::= 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(T1,T).
T1 ::= s(Remote) | c(Remote).
```

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

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

2.2 Exploiting the Consumption Specification

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

During message preparation, consumption specifications are used to include in the message only those parts of the data structures consumed by the called procedure. For instance, a call to `append/3` requires for each argument a copy instruction which inserts in the message the appropriate part of the argument. In case of a term `t` with consumption specification `T ::= [] | cons(Remote,T)`, the copy instruction inserts inside the message only the spine of the list, i.e., only the reference to each element of the list.

When receiving a message the consumption specification is used 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.

3 A High-level Implementation

To experimentally verify 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 [7] and uses attributed variables to implement the communication variables, i.e., the variables occurring in a remote goal. Attributed variables introduced by Le Houitouze [11] 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,Type)` where the pair `(Process,Id)` uniquely identifies a communication variable, `Bound` specifies if the variable is bound or unbound and `Type` is the type associated with the variable.

The blackboard structure. The 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).`. 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 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,t),[c(rem(1,3,bound,list))])
```

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.

4 Static Analysis

In this section, we sketch how the consumption specification can be obtained by a static analysis enhancing traditional type analyses. We convey the main ideas behind the approach. Recall that the consumption specification describes a superset of the type of the procedure. Consider the list concatenation code. 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 previously (replace `Any` by `Remote`). These type analyses have been investigated extensively in the literature (e.g., [6, 8, 17]). Type analysis of logic languages, and Prolog in particular, is of primary importance for high performance compilers. In the sequential case, type analysis has been applied in order to improve indexing, to specialize unification and to produce more efficient code for built-in predicates.

We exploit type analysis to automatically obtain the consumption specification. To this purpose, we can profitably use a system like `GAIA` [17], where type analysis is based on abstract interpretation [3] and type graphs [8]. The appealing feature of `GAIA` is its good trade-off between accuracy and efficiency and the ease with which type analysis can be combined with other analyses that can be useful in our case, such as, for instance, mode analysis.

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, and GAIA in particular, would produce the results p(T) and q(T) where T ::= [] | cons(Any,T)., which are superset of the types of the procedures.

Notice that an efficient implementation of distributed unification should not send the whole list to p/1, but only its first cons cell with arguments which are remote references to the head and tail of the list. In this respect, the consumption specification for p/1 is $p(T_1)$ *where* T_1 ::=[] | cons(Remote,Remote).. This consumption specification is a larger superset of the type (i.e., what the analysis would return if the goal q(Ta) is omitted in the clause for p/1). To determine the consumption specification from the type, traditional type analyses should be enhanced by annotating each functor with the predicate in which it occurs. Then, the widening operator for type graphs [17] is applied only for types related to the same predicate.

When enhanced in this way, the type analysis for the above program now determines 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. To this purpose, it suffices to substitute each occurrence of Any with Remote in type T_1 and any reference to other types (e.g., T_2) with Remote as well, thus obtaining the consumption specification for p and q:

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

5 Experimental Results

This section presents the experimental results obtained by running some example programs on the system described in Section 3. These results give some indication of the practical usefulness of the consumption specification approach. Obviously, the high-level implementation cannot achieve the performance of a specific abstract machine. For this reason, in [10] we sketched a low-level implementation based on the WAM, suitably extended in order to perform the distributed unification driven by the consumption specification.

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 "optimal" policy driven by the consumption specification determined by the static analysis sketched in Section 4. 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 6.1-6.4 show the results obtained by running the examples on a network of SUN workstations (Sparcstation2) 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 length.

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 6.1 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 6.1, 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 6.1: `append/3` with lists of integers

Table 6.2 shows the timings of the usual `reverse/2` [13] program with zero-, infinite-level copying, and the consumption specification described in Section 2. 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, because 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 6.2: `reverse/2` with one list of lists of increasing size

Tables 6.3 and 6.4 show the results of the `keysort/2` and `quicksort/2` programs [13]. `keysort/2` is a variant of `quicksort/2`, and sorts a list of strings on the basis of the first character of each string. The results reported in Table 6.3 are obtained by sorting a list of 10 strings, each one 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, that for `keysort/2` is `keysort(`T_1`,`T_1`)` *where*

```
T1 ::= [] | cons(T2,T1).
T2 ::= cons(char,Remote).
```

It specifies that `keysort/2` consumes the first character of each string. The `split/4` procedure in `keysort/2` splits the list of strings on the basis of the first character of first string and its consumption specification is `split(T2,T1,T1,T1)` which requires to send to `split/4` only the first character of the first argument and the first character of each (nested) string for the other arguments.

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

Finally, Table 6.4 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 6.4: `quicksort/2` with one list of 10 characters

6 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 an implementation scheme for distributed logic programming which consists of tailoring 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. The implementation scheme uses the consumption specification to avoid unnecessary copying and to request data structures globally. Moreover, the consumption specification (or an approximation of it) can be automatically obtained through a static analysis inspired by type analysis, as shown in Section 4.

The paper has also described a high-level implementation of the scheme, built on top of SICStus Prolog by using both the attributed variable and the Linda libraries. The results obtained with some example programs running on 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 could be achieved by a low-level implementation, by pushing the implementation of critical operations down to C, at WAM level.

References

1. H. Aït-Kaci: Warren's Abstract Machine. The MIT Press (1991).
2. A. Ciampolini, E. Lamma, P. Mello and C. Stefanelli: Multilevel Copying for Unification in Parallel Architectures. Proc. Second Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Society Press (1994) 518-525.
3. P. Cousot and R. Cousot: Abstract Interpretation and Application to Logic Programs. Journal of Logic Programming, **13** (1992) 103-180.
4. G. Frosini, P. Corsini and L. Rizzo: Implementing a parallel Prolog interpreter by using Occam and Transputers. Microprocessors and Microsystems, **13** (1989) 271-279.
5. F. Gecseg and M. Steinby: Tree Automata. Akademiai Kiado, Budapest (1984).
6. N. Heintze and J. Jaffar: A Finite Presentation Theorem for Approximating Logic Programs. Proc. 17th ACM Symp. on Principles of Programming Languages (1990) 197-209.
7. 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).
8. 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)(1992) 205-258.
9. P. Kacsuk and M. Wise (eds.): Implementations of Distributed Prolog. J. Wiley and Sons (1992).
10. E. Lamma, P. Mello, C. Stefanelli and P. Van Hentenryck: Consumption-based Distributed Unification in parallel architectures. Proc. APPIA-GULP-PRODE96, M. Martelli and M. Navarro eds., San Sebastian (S) (1996).
11. S. Le Huitouze: A New Data Structure for Implementing Extensions to Prolog. P. Deransart and J. Maluszunski eds, Proc. Programming Language Implementation and Logic Programming, Springer-Verlag (1990) 136-150.
12. K. Nakajima, N. Ichiyoshi, K. Rokusawa and Y. Inamura: A new external reference management and distributed unification for KL1. ICOT editor, Proc. Int'l Conf. FGCS-88 (1988) 904-913.
13. L. Sterling, E. Shapiro: The Art of Prolog, MIT Press (1986).
14. Swedish Institute of Computer Science, SICStus Prolog User's Guide, S. Kista (1990).
15. S. Taylor: Parallel Logic Programming Techniques. Prentice-Hall International Editions (1989).
16. E. Tick: Parallel Logic Programming. MIT Press (1991).
17. P. Van Hentenryck, A. Cortesi, and B. Le Charlier: Type Analysis of Prolog using Type Graphs. Journal of Logic Programming, 22 (3) (1995).
18. D.H.D. Warren: An abstract Prolog instruction set. Technical Report TR 309, SRI International (1983).