Improving Distributed Unification through Abstract Interpretation

Anna Ciampolini, Evelina Lamma, Paola Mello, Cesare Stefanelli

DEIS, Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
{aciampolini,elamma,pmello,cstefanelli}@deis.unibo.it

Abstract

In this paper we apply static analysis techniques for improving the distributed implementation of logic programming languages, especially with regard to unification. In a distributed implementation, unification involves sending and receiving messages containing information about the goal arguments. Each message can contain either the entire copy of goal arguments or only a part of it. The aim of the analysis is to determine the *optimal* level of copying to be adopted for each goal argument of a given program. This level of copying is the minimal one which avoids the need for any *remote dereferencing* during the computation. In this respect, the analysis done is related to the *strictness analysis* for functional languages which helps in deciding whether a function argument can be evaluated directly, instead of evaluating it only when needed.
The analysis is grounded on abstract interpretation, and *size* relation analysis in particular. The optimal level of copying for each predicate argument is determined starting from the mode of the predicate, and by determining relations among the size and use of each term. The results of the analysis are exploited by a distributed implementation scheme, based on an abstract machine where the level of copying for each goal argument can be tuned at compile-time, thus reducing the overhead due to message exchange for remote dereferencing.

Keywords: Applications, Abstract Interpretation, Unification.

1 Introduction

In the distributed implementation of logic programming (see for instance [8]) one of the major problems is the efficient implementation of unification. In distributed architectures, in fact, data structures are spread among different nodes, and each interaction and information exchange is performed through message-passing. In particular, unification involves sending and receiving messages containing information about the goal arguments. An efficient implementation must reduce the amount of inter-node communication to obtain quick responses.

One of the key issues when implementing distributed unification is how and when to access remote data structures. Data structures can be made accessible to other processes by sending messages which carry either the overall values of bounded variables (*infinite-level copying*) or only remote references to these data structures (*0-level copying*). Also an intermediate approach can be followed, copying data structures till a certain level, and then sending them to other remote processes.
The trade-off between these choices (full-copying or remote referencing) heavily depends on the underlying architecture, and on the cost of communication in particular. 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 accessing a remote structurein order to perform unification (*remote dereferencing*). In distributed systems (e.g., a cluster of workstations), where the cost of remote accesses is relatively high, it should be better to copy data for later accesses in many cases, but this do not apply in general to tightly coupled parallel systems.

The above consideration has driven the approaches followed in the literature. An approach based on replication of data is used in [7] where each message carries the infinite-level copying of each argument of the goal to be solved. Ichiyoshi et al. [9] introduced the copying level concept in the implementation of the KL1 language on the MultiPSI machine. In the forward unification phase, atomic values are always entirely copied and unbound variables are never copied (i.e., only their address is encoded into the exporting message). Any intermediate, but fixed, level of copying can be "a-priori" determined for data structures.

However, fixing a specific policy of copying (and thus a specific level of copying for all data structures) might not be the optimal choice for all cases. In fact, which of the copying policies is most efficient depends not only on the characteristics of the target architecture, but also on the particular program and how the arguments have to be used. Sometimes it is convenient to entirely copy a goal argument and just send a remote reference for another one.

The key point is how to tune the copying level of each argument in program calls, in order to reduce the amount of messages exchanged.
In this paper, we face this problem and determine, by means of abstract interpretation, the level of copying to be adopted for each goal argument of a given program to be executed on a distributed architecture. In particular, we apply a bottom-up analysis in order to compute the *optimal* level of copying for each data structure occurring in predicate calls. By *optimal* we mean the (minimal) level of copying which avoids the need for any *remote dereferencing*. This level of copying is determined starting from the mode associated with the arguments of the invoked predicate and by applying *size relations*, the purpose being to determine how many levels of each argument are necessary (i.e., *used*) to avoid any remote dereferencing during the computation.

It is worth to notice that this kind of analysis is related to the *strictness analysis* for functional languages [14]. In lazy functional languages, in fact, abstract interpretation is employed to optimize the execution. In particular, it is possible to determine if a structure has to be evaluated (i.e., it is *strict*) during the computation. This helps in deciding whether a function argument could be evaluated directly, instead of evaluating it only when needed.

Many works have applied abstract interpretation techniques in order to derive so-called *size relations* [13, 12, 5]. Size relations express relationships among the arguments of atoms in the least Herbrand model. For instance, using the length of a list as its size, and considering list concatenation we have that $size(arg1) + size(arg2) = size(arg3)$ for the atoms in the least Herbrand model.

Size relations have been mainly used for proving the termination of programs [13, 12] but they are indeed of interest for compiler optimization (e.g., a list of known length can be implemented as an array). In this paper, we extend the framework for size relations analysis developed in [13, 12] and reported in [2]. Instead of simply determining the relations among the size of arguments of program predicates, we determine which part of each argument is really necessary during the computation. For the time being, we focus on programs working on lists. This notably simplifies the *size* of terms, being lists linear data structures. We are aware that, in a more general context, other *size* relations have to be considered for terms.

We implemented the distributed unification taking advantage from the information given by our analysis about the optimal level of copying. In [4], we presented an implementation scheme, based on the Warren Abstract Machine (WAM, [15]), where the level of copying can be tuned for each goal argument. In particular, instead of fixing one policy of copying like in [7, 9], we extend `put` and `get` instructions with a new argument representing the level of copying to be adopted. Now, thanks to the results of our analysis, the optimal level of copying can be specified for each predicate argument. First experimental results, obtained on top of a transputer-based architecture, show that knowledge of the optimal level of copying is useful in improving performances of distributed logic programs.

The paper is organized as follows. In section 2, we briefly recall the bottom-up framework for size relation analysis. In section 3, this framework is extended in order to relate the *use* of terms to their *size*. Section 4 discusses the case of non-deterministic logic programs. In section 5, we briefly describe the implementation, and discuss the impact of the analysis on it. Some conclusions and future work follows.

2 Size Analysis

In this section we briefly recall the application of abstract interpretation techniques in order to derive size relations (see [2] for a survey).

Let P be a logic program. The Herbrand base of P, $\mathcal{B}_P$, is defined as usual [10] as the set of all the ground atomic formulas built from the Herbrand universe of P. The Herbrand universe, $\mathcal{H}_P$ is the the standard one, composed of all the ground terms built from constant and function sysmbols of P. The least Herbrand model of a program P can be defined as the least fixpoint of a transformation function T_P defined as follows:

Definition 2.1 *Let P be a program. $T_P : Powerset(\mathcal{B}_P) \mapsto Powerset(\mathcal{B}_P)$ is defined as follows:*

$$T_P(I) = \{A \mid \exists\ A \leftarrow B_1, \ldots, B_n \in ground(P)\ \wedge\ B_i \in I\ \forall i = 1, \ldots, n\}$$

where ground(P) represents the ground version of P.

The purpose of the analysis is to derive so-called *size relations* [13, 12]. Size relations express relationships among the arguments of atoms in the least Herbrand model. For instance, using the length of a list as its size, and considering list concatenation (see example 2.5), for all the $append(t_1, t_2, t_3)$ atoms in the least Herbrand model we have that $size(t_1) + size(t_2) = size(t_3)$.

In this section and throughout the rest of the paper we focus on programs working on lists, assuming the size of interest is list length. Then, $size$ is defined as follows (see [2]):

Definition 2.2 *Let P be a program.* $size : \mathcal{H}_P \mapsto N$ *maps a term (list) of the Herbrand universe into a natural number as follows:*

$$size(t) = 1 + size(y) \ if \ t = [x|y];\ 0 \ otherwise$$

This function is then extended mapping atoms from the Herbrand base of a predicate p/m into m-tuples. This kind of size relation is also called a *norm* in the literature. These sets of m-dimensional points forms a complete lattice ordered by the usual subset relation.

Definition 2.3 *Let P be a program.* $size : \mathcal{B}_P \mapsto N^m$ *maps an atom with predicate* p/m *of the Herbrand base into a* m*-tuple as follows:*

$$size(p(t_1, \ldots, t_m)) = \langle size(p(t_1), \ldots, size(t_m)\rangle$$

In this kind of analysis, the focus is on a single predicate p/m at time. The concrete domain, represented by the atoms of the Herbrand base with predicate symbol p/m, is first mapped into a set of m-dimensional points. The abstract domain is represented by systems of linear equations (see [2]), i.e. equations of the form $a_1x_1 + \ldots + a_mx_m = b$, defining *linear varieties* of the m-dimensional space.

Thus, the abstraction function α maps a subset of the Herbrand base for predicate p/m into a system of linear equations $E^{p/m}$. In practice, the construction is a two layered abstraction, firstly mapping (set of) atoms into (set of) points, and then sets of points into systems of equations.

Given a system of linear equations $E^{p/m}$ for predicate p/m, the concretization function, γ, returns the set of ground atoms whose size tuple is a solution for $E^{p/m}$.

Definition 2.4 *Let* $E^{p/m}$ *be a system of linear equations for predicate* p/m*. Then:*

$$\gamma(E^{p/m}) = \{p(t_1, \ldots, t_m) \in \mathcal{B}_P \mid \langle size(p(t_1), \ldots, size(p_m)\rangle \ is \ a \ solution \ for \ E^{p/m}\}$$

This concretization function induces a preorder relation over systems of equations (see [2] for details).

Finally, the abstract function T_P^α is defined as $\alpha \circ T_P \circ \gamma$. The couple (α, γ) yields a Galois connection.

Example 2.5 *Let us consider the predicate for list concatenation:*

```
append([],X,X).
append([X|U],V,[X|W]) :- append(U,V,W).
```

The chosen size relation maps atoms for `append` *belonging to the Herbrand base of the program into the points* $\langle 0,0,0\rangle$, $\langle 0,1,1\rangle$, $\langle 0,2,2\rangle$, ..., $\langle 0,n,n\rangle$, ... $\langle 1,1,2\rangle$, $\langle 1,2,3\rangle$, ... $\langle m,n,m+n\rangle$. *The abstraction function* α *maps atoms of the Herbrand base into* 3*-dimensional points (through size), and then sets of* 3*-dimensional points into linear varieties. We have:*

$$T_P^\alpha \uparrow 0 = \perp^\alpha$$

$$T_P^\alpha \uparrow 1 = \alpha(\{append([],t,t)\})$$

Function size maps the atoms of kind $\{append([],t,t)\}$ *into the points* $\{\langle 0,n,n\rangle | n \geq 0\}$. *The minimal system of equations having these points as solutions describes a line through the points, i.e.:*

$$T_P^\alpha \uparrow 1 \quad = \quad \{x_1 = 0, x_2 = x_3\}$$

The next step of iteration produces:

$$\begin{aligned} T_P^\alpha \uparrow 2 \quad &= \quad \alpha(\{append([],t,t)\} \cup \\ &\qquad \{append([t_X|t_U],t_V,[t_X|t_W])|size(t_U)=0, size(t_V)=size(t_W)\}) \\ &= \quad lub(\alpha(\{append([],t,t)\}), \\ &\qquad \alpha(\{append([t_X|t_U],t_V,[t_X|t_W])|size(t_U)=0, size(t_V)=size(t_W)\})) \end{aligned}$$

Performing the least upper bound in the abstract domain produces the minimal set of equations satisfying both systems:

$$T_P^\alpha \uparrow 2 \quad = \quad \{x_2 + x_1 = x_3\}$$

The next iteration produces the least fixpoint of the T_P^α *function:*

$$T_P^\alpha \uparrow 3 = \{x_2 + x_1 = x_3\}$$

3 Extending the Analysis

3.1 Preliminaries

In this section, we discuss how size relation analysis can be extended in order to derive information not only about the sizes of the arguments of predicate calls, but also about the use of input data structures. The aim is to determine for each input argument of a given predicate p which is the *optimal* level of copying to be adopted in sending this argument when a call to p is raised. By *optimal* level of copying, we mean the minimal level which avoids the need for remote dereferencing during the computation (see section 5 for discussion and performances results). In the following, and throughout the rest of the paper, we will limit ourselves on linear data structures (lists). The extension of the analysis for more complex data structures is briefly discussed at the end of this section.

We start up with a rather intuitive example. Let us consider, again, the list concatenation program of example 2.5 implemented on a distributed architecture. Suppose to use this program with mode `append(+,+,-)`, and that any call corresponds to a message exchanged between two processes [1]. When the second clause is applied, we are interested in determining how many levels for the data structures bound to `U` and `V` must be copied in the message to be sent to the process handling the recursive call of `append`. In practice, we want the analysis to determine

[1] We are aware that this is not the optimal implementation. For recursive predicates a single process can be used.

that for each predicate call of `append(+,+,-)`, the first argument has to be passed to the invoked process with infinite level of copying (i.e., the whole data structure has to be sent) whereas for the second argument it suffices to pass a remote reference, and not the overall data structure. For list concatenation with mode `append(+,-,+)` we want, instead, the analysis to determine that the first argument has to be passed to the invoked process with infinite level of copying (i.e., the whole data structure has to be sent) whereas the third argument has to be passed till the length of the first one.

To this purpose, the size analysis framework described in section 2 has to be extended. This extension concerns not only the abstract domain and abstraction function, but also the concrete domain and T_P function. In particular, on the basis of *mode* information, we extend the concrete domain by annotating atoms for each predicate p/m with n-tuples if n is the number of input arguments for p/m[2].

In the following, we consider programs with no function symbols, and focus on linear data structures (*i.e.*, lists) in particular.

Definition 3.1 *Let P be a program, and let n be the number of input arguments for predicate p/m. Then, the extended Herbrand base for p/m is given by:*

$$\{p(t_1, \ldots, t_m)^{u_{i_1}, \ldots, u_{i_n}} \mid p(t_1, \ldots, t_m) \in \mathcal{B}_P\}$$

where $i_1, \ldots, i_n$ correspond to the indexes of the input arguments of p/m, and $\forall i = 1 \ldots n \; u_{i_j} \in [0, size(t_{i_j})]$.

Let $\overline{\mathcal{B}}_P$ denote the extended Herbrand base for a program P. In the following, we consider $\overline{\mathcal{B}}_P$ as the concrete domain.

The T_P function is extended accordingly, in order to explicitly take note of the depth of the terms used during the bottom-up computation. In practice, through clause applications and the T_P operator properly extended, we map annotated atoms of the extended Herbrand base into (possibly) new annotated atoms. To this purpose, given a clause C for predicate p/m we associate with each input argument t in the head of C a *used* index. According to the top-down approach, $used(t)$ represents the number of copying levels that are necessary for performing the head unification without any remote dereferencing. According to the bottom-up computation, the *used* index can be also interpreted as the number of levels added to the term at each step of computation.

Example 3.2 *Let us consider again the list concatenation program* `append(+,+,-)`*, and the interpretation $I_0 = \{\}$. Then, the application of the first clause to I produces the set of annotated atoms:*

$I_1 = \{append([], t, t)^{\{0,0\}} \mid \forall t \in \mathcal{H}_P\}$

meaning that each of the atoms $append([], t, t)$ is obtained (in one-step computation) by using nothing of either first and second argument. By applying to I_1 the second clause for append, we get the set:

$I_2 = \{append([t_1], t, [t_1|t])^{\{1,0\}} \mid \forall t, t_1 \in \mathcal{H}_P\}$

[2] A different notation could be adopted, by extending each atom for p/m in the Herbrand base with n additional arguments, and defining accordingly the extended Herbrand base of a program.

meaning that each of the atoms $append([t_1], t, [t_1|t])$ *is obtained starting from* I_1 *by using exactly one element of the first argument and nothing of the second one.*

Intuitively, the T_P function is extended in order to derive - at each application step - the *use* of input arguments. In practice, for a given (input) argument t_i of an atom produced through the application of a clause C, the use is determined as the size increasing obtained by applying C. Thus, through bottom-up clause application, in the (extended) least Herbrand model of a program we have each ground atom A annotated by information concerning how much of each input data structure is used in order to derive A.

The abstract domain is extended accordingly in order to explicitly consider not only relations between the size of arguments but also other (*use*) relationships between arguments.

Definition 3.3 *Let* P *be a program, and let* n *be the number of input arguments for a given predicate* p/m. $size_{use} : \overline{\mathcal{B}}_P \mapsto N^{m+n}$ *maps an atom with predicate* p/m *of the extended Herbrand base into a* $(m+n)$*-tuple as follows:*

$$size_{use}(p(t_1, \ldots, t_m)^{u_{i_1}, \ldots, u_{i_n}}) = \langle size(p(t_1), \ldots, size(t_m), u_{i_1}, \ldots, u_{i_n}\rangle$$

where $i_1, \ldots, i_n$ *correspond to the indexes of the input arguments of* p/m.

Then, the abstract domain is still represented by systems of linear equations (as in section 2), defining - this time - *linear varieties* of the $(m+n)$-dimensional space.

Given a system of linear equations $E^{p/(m+n)}$ for predicate p/m with n input arguments, the concretization function, γ, returns the set of ground (annotated) atoms whose $size_{use}$ tuple is a solution for $E^{p/(m+n)}$. Each atom is, in practice, annotated by the n-tuple of *use* value of its input arguments.

Definition 3.4 *Let* $E^{p/(m+n)}$ *be a system of linear equations for predicate* p/m *with* n *input arguments. Then:*

$$\begin{aligned}\gamma(E^{p/(m+n)}) \quad = \quad & \{p(t_1, \ldots, t_m)^{u_{i_1}, \ldots, u_{i_n}} \in \overline{\mathcal{B}}_P \mid \langle size(p(t_1), \ldots, size(p_m), u_{i_1}, \ldots, u_{i_n}\rangle \\ & is\ a\ solution\ for\ E^{p/(m+n)}\}\end{aligned}$$

where $i_1, \ldots, i_n$ *correspond to the indexes of the input arguments of* p/m.

The function T_P^α is still defined as $\alpha \circ T_P \circ \gamma$.

3.2 Examples

In the following, we consider again the list concatenation program with mode `append(+,+,-)`. The abstract function now maps (annotated) atoms of the extended Herbrand base of the program into 5-tuples $\langle x_1, x_2, x_3, x_4, x_5\rangle$ where

$$x_1 = size(t_1),\ x_2 = size(t_2),\ x_3 = size(t_3),\ x_4 = used(t_1),\ x_5 = used(t_2)$$

The abstract domain is therefore composed by systems of linear equations in the 5-dimensional space. The (iterated) application of the abstract function T_P^α works, for this program, as follows.

For the sake of simplicity, we omit annotations for atoms of the extended Herbrand base, and add explicitely the *used* information for input arguments:

$$T_P^\alpha \uparrow 0 \;=\; \perp^\alpha$$

$$\begin{aligned}
T_P^\alpha \uparrow 1 &= \alpha(T_P(\gamma(\perp^\alpha))) \\
&= \alpha(T_P(0)) \\
&= \alpha(\{append([], t, t) \mid used(t) = 0\}) \\
&= \langle 0, n, n, 0, 0\rangle \\
&= \{x_1 = 0, x_2 = x_3, x_4 = 0, x_5 = 0\}
\end{aligned}$$

The next step of iteration produces:

$$\begin{aligned}
T_P^\alpha \uparrow 2 &= \alpha(T_P(\gamma(\{x_1 = 0, x_2 = x_3, x_4 = 0, x_5 = 0\}))) \\
&= \alpha(T_P(\{append(t_1, t_2, t_3) \mid size(t_1) = 0, size(t_2) = size(t_3), used(t_1) = used(t_2) = 0\})) \\
&= \alpha(\{append([], t, t)\} \cup \\
&\quad \{append([t_X|t_U], t_V, [t_X|t_W]) \mid size(t_U) = 0, size(t_V) = size(t_W), \\
&\quad used(t_U) = 0, used(t_V) = 0\}) \\
&= lub(\langle 0, n, n, 0, 0\rangle, \langle 1, n, n+1, 1, 0\rangle) \\
&= lub(\{x_1 = 0, x_2 = x_3, x_4 = 0, x_5 = 0\}, \{x_1 = 1, x_3 = x_2 + 1, x_4 = 1, x_5 = 0\})
\end{aligned}$$

The minimal set of equations which satisfies both systems is:

$$T_P^\alpha \uparrow 2 \;=\; \{x_3 = x_1 + x_2, x_4 = x_1, x_5 = 0\}$$

Calculating $T_P^\alpha \uparrow 3$ we obtain:

$$\begin{aligned}
T_P^\alpha \uparrow 3 &= \alpha(T_P(\gamma(\{x_3 = x_2 + x_1, x_4 = x_1, x_5 = 0\}))) \\
&= \alpha(T_P(\{append(t_1, t_2, t_3) \mid size(t_3) = size(t_2) + size(t_1), \\
&\quad used(t_1) = size(t_1), used(t_2) = 0\})) \\
&= \alpha(\{append([], t, t) \mid used(t) = 0\} \cup \\
&\quad \{append([t_X|t_U], t_V, [t_X|t_W]) \mid size(t_W) = size(t_U) + size(t_V), \\
&\quad used(t_U) = size(t_U), used(t_V) = 0\}) \\
&= lub(\langle 0, n, n, 0, 0\rangle, \langle n, m, n+m, n, 0\rangle) \\
&= \{x_3 = x_1 + x_2, x_4 = x_1, x_5 = 0\} \\
&= T_P^\alpha \uparrow 2 \\
&= T_P^\alpha \uparrow \omega
\end{aligned}$$

The least fixpoint of the T_P^α, gives now an additional information about the copying level to be adopted for input arguments: in the list concatenation program with mode `append(+,+,-)`, the first argument have to be entirely copied (infinite level copying) since its use (x_4) coincides with its size (x_1), while no copy (0 level copying) is needed for the second argument since its use (x_5) is 0. In section 5, we will further discuss the impact of this kind of analysis on the resulting, distributed implementation.

If a different mode is adopted for the list concatenation program, say mode `append(+,-,+)` for instance, the analysis gives a different result, as expected. The abstract function now maps atoms belonging to the extended Herbrand base into 5-tuples $\langle x_1, x_2, x_3, x_4, x_5\rangle$ where

$$x_1 = size(t_1),\;\; x_2 = size(t_2),\;\; x_3 = size(t_3),\;\; x_4 = used(t_1),\;\; x_5 = used(t_3)$$

This time we have that the fixpoint of the abstract function is:

$$T_P^\alpha \uparrow \omega = \{x_3 = x_1 + x_2, x_4 = x_1, x_5 = x_1\}$$

This result informs us about the level of copying to be adopted for input arguments. As expected, the first argument has to be entirely copied (infinite level copying) since its use (x_4) is equal to its size (x_1) while the third argument has to be copied only until the size of the first one since its use (x_5) is equal to the size of the first one (x_1).

As a further example, let us consider the predicate for the extraction of the last element of a list:

```
last([X],[X]).
last([_|L],X) :- last(L, X).
```

with mode `last(+,-)`. We are interested in determining how many levels for the data structure bound to `L` must be copied in the message sent to the process handling the recursive call of `last`.

The abstract function maps atoms of the extended Herbrand base of the program into 3-tuples $\langle x_1, x_2, x_3 \rangle$, where

$$x_1 = size(t_1),\ x_2 = size(t_2),\ x_3 = used(t_1)$$

The least fixpoint of the abstract function turns to be:

$$T_P^\alpha \uparrow \omega = \{x_2 = 1, x_3 = x_1 - 1\}$$

This means that the number of levels to be passed for the input argument in order to avoid any remote dereferencing must be $size(t_1) - 1$. At implementation level, this means that all the elements except the last one of the first list has to be passed to the recursive call. If the last element of the input list is a complex term, this reduces the dimension of the message to be sent and the cost of communication as well.

We are aware that the analysis here discusses works well for linear data structures whose elements are atomic. In case of more complex data structures, two issues have to faced: the *size* relation has to be defined properly, and additional information about the use of terms has to be considered when applying clauses. As concerns the first issue, one possible *size* is, for instance, *term-size* which counts the number of functors (with strictly positive arity) occurring in a term. Yet another *size* relation is *term-depth* which gives the maximum depth of the tree representation of a term. For a discussion about the automatic inference of sizes (but in the context of program termination) see [5]. As concerns the second issue, in practice, when a clause is applied and an input term T_1 is unified with another one T_2, just a remote reference for T_1 suffices if T_2 is an output argument (0 used, and 0 level of copying for T_1). Nonetheless, the overall data structures (infinite level of copying) are needed for T_1 and T_2 if they both are input arguments.

4 Logic Programs with Alternative Clauses

The abstract analysis described in section 3 is effective if the program does not provide *alternative* clauses. In fact, in the case of two or more alternative clauses the method might be unable to return information about the copying level. For instance, let us consider the following simple example.

Example 4.1 *Let us consider the following program:*

```
p(X):-r(X).
r([a,b,c,d|X]).
r([h,i|X]).
r([f|X]).
```

Let us assume the mode `p(+)`*. We wish that the analysis of such a program returned the copying level value* 4 *for the argument of predicate* `p`*, that corresponds to the* worst *case when calling the predicate* `r` *(*i.e.*, calling* `r([a,b,c,d|X]).`*). Differently, if we apply the T_P^α discussed in section 3, the obtained result is:*

$$T_P^\alpha \uparrow \omega = \{x_1 = x_2 + n\}$$

where $x_1 = size(t)$ (being t the argument of predicate p), $x_2 = used(t)$ and n is an undefined integer. Therefore, no useful indication about the copying level to choose for the argument t is returned by the abstract analysis. This loss of information is due to the lub *operation at step 1 which generalises the results deriving by the alternative application of clauses* `r`.

In cases like this, it is desirable to do a finer analysis. For this purpose, we modify the abstract function α as follows:

$$\alpha = \mathcal{W} \circ \alpha'$$

where α' is the abstraction function introduced in section 3, and $\mathcal{W}$ is defined as follows:

Definition 4.2 *Let P be a program, and $\overline{\mathcal{B}}_P$ its extended Herbrand base, and $I \in powerset(\overline{\mathcal{B}}_P)$. Thus, $\mathcal{W} : powerset(\overline{\mathcal{B}}_P) \mapsto powerset(\overline{\mathcal{B}}_P)$ is defined as:*

$$\begin{aligned} \mathcal{W}(I) \;\; = \;\; & \{A \in \overline{\mathcal{B}}_P \mid A = p(t_1, \ldots, t_m)^{u_{i_1}, \ldots, u_{i_n}} \wedge \\ & \forall A' = p(t_1, \ldots, t_m)^{u'_{i_1}, \ldots, u'_{i_n}} \in I,\ \forall j = 1, \ldots n, u_{i_1} > u'_{i_1}\} \end{aligned}$$

In practice, $\mathcal{W}$ selects atoms annotated by higher values (*worst* case).

It can be proved that the resulting abstract function α is again monotonic, and therefore (α, γ), equipped with subset order relations is again a Galois connection.

With this further extension, in the case of a set of alternative clauses of the kind $p(t_1, \ldots, t_m) \leftarrow \ldots$, the abstract transformation function returns the worst (*i.e.*, the highest) $used(t_i)$ value for each input argument t_i. This allows us to avoid the loss of information outlined in example 4.1. In fact, if we now perform the abstract analysis on example 4.1 we obtain:

$$T_p^\alpha \uparrow \omega = \{x_1 = n + 4, x_2 = 4\}$$

meaning that the argument x of `p/1` has to be passed to the child process till the level 4 (*i.e.*, the first four elements of x).

5 The Abstract Machine for Distributed Unification

The abstract execution model we adopt for distributed unification is grounded on the Warren Abstract Machine (WAM [15, 1]), suitably extended. In the following, we mainly focus on the WAM instructions related to unification. We assume the reader is familiar with the WAM.

Mainly focusing on unification, our current implementation supports a limited parallel logic language with OR parallelism where programs are composed by binary definite clauses only. [3]

Since the distributed implementation deals only with binary definite programs, all operations related to WAM environments are removed, because environments are no longer needed for these programs. For the same reason, we do not need the local stack area.

The extended WAM run-time model is represented by a set of parallel processes distributed on a network of processors. More in detail, a binary logic program is mapped into a set of parallel processes (*OR* processes), each one representing a specific clause invocation. The top-level goal is mapped into an *OR* process as well. Thus, an initial goal of the form `:-p(X1, ..,XN)` is mapped into an *OR* process with as many child processes as the number of clauses with head `p/N`. Therefore, the *AND/OR* process tree of parallel logic programming [3] turns to be an *OR* tree. Since each process P is associated with a single clause of the program, variables (and data structures) occurring in the clause are allocated on an area (called *heap*) local to P.

The heap and the argument registers of each process are allocated onto a local memory area whose basic element is the *cell*. As in the standard WAM a type (tag) is associated with the information contained in the cell. We add the type `REM` for indicating that the cell contains a remote reference to the heap location of another remote process. Remote references are expressed by the identifier of the process P owning the data structure, associated with their offset in the remote heap of P.

In order to avoid conflicting bindings, each process keeps track of bindings between local terms and remote unbound output variables into a local table. During the unification phase, if an ancestor output variable has to be bound, a new copy for the variable is allocated in the table associated with the current process: only this local copy is bound. These bindings will be actualised in the backward propagation phase. To this purpose, each process also maintains the address of the parent process in order to propagate the results.

With regard to unification, in our distributed implementation, `put` instructions are used for preparing goal arguments and put them into a goal message (forward propagation phase). After the put phase, a `call` instruction takes care of sending the goal message to each child process. Then the parent process non-deterministically waits for an answer from one of its descendent, and finally propagates the result "via" a `proceed` instruction (backward propagation phase). When a child process receives a goal message, it reconstructs the goal arguments on its heap, and performs the head unification by means of `get` and `unify` instructions. In order to reduce the amount of the inter-node communication, instead of adopting a fixed policy of copying, we introduce in `put` and `get` (and `unify`) WAM instructions the level of copying as a parameter `L`. This parameter may be possibly specified during the compilation phase, (*e.g.*, on the basis of the abstract analysis results). It is worth to notice, however, that if we adopt the *used* values returned by the abstract analysis in the `put` phase, when calling the `get` instruction the

[3] This choice does not imply any limitation in the expressive power of the resulting language since each logic program can be transformed into a binary definite one [6]. However, it notably simplifies the underlying execution model.

specification of `L` is not needed, since the remote dereferencing is a-priori avoided.

As shown in section 3, during the computation a data structure can be used till a certain level depending on the size of another data structure. Therefore, in the general case, the level of copying to be specified in WAM instructions can be determined only when the size of some other argument is known. To this purpose, we add to the WAM a new instruction in order to compute the size of a data structure. The instruction `size A_i` determines the size of the data structure bound to the argument register `A_i`, and store this size into the new register `A_L`.

In the following, we show the compiled code of the example 2.5, with mode `append(+,+,-)`, assuming the top goal: `:- append([a,b],[c,d],X)`

```
top-goal:
         put_list  X1 3 /* copying level 3 */
         unify_constant  a
         unify_constant  b
         unify_nil
         put_list  X2 0 /* copying level 0 */
         unify_constant  c
          unify_constant  d
         unify_nil
         put_variable  X3, X3
         call  append/3

append/3(I):
        get_value  X2,X3
        get_nil  X1
        proceed

append/3(II):
        get_list  X1
        unify_variable  X4
        unify_variable  X1
        get_list  X3
        unify_value  X4
        unify_variable  X3
        call  append/3
```

We have tested this program with two different copying policies: the first one adopts the *optimal* copying levels returned by the abstract analysis; the second one, instead, uses an infinite level of copying (*i.e.*, all levels of input arguments are copied in goal messages). Both input arguments are lists of ten elements. The target architecture is a transputer-based machine.
From the analysis of the timing results of the two copying policies (195 ms for the optimal copying level case, 238 ms for the infinite copying level one), we can conclude that the copying level determined by the abstract analysis gives rise to a performance increase of about 20%. In the case of larger input data structures, this increasing rate becomes even higher.

6 Conclusions and Future Work

We have applied abstract interpretation techniques, and size analysis in particular, in order to improve the distributed implementation of unification. In the distributed implementation of logic programming, in fact, one of the major problems is the efficient implementation of unification, the tradeoff being sending to child processes the overall values for data structures (*infinite level copying*) or only remote references to these data structures (*0-level copying*).

We have extended the size analysis proposed in [13, 12] in order to derive relations between the use and size of input arguments for a given predicate. In particular, we determine the *optimal* level of copying to be adopted for each goal argument of a given program. This level of copying is the minimal one which avoids the need for any *remote dereferencing*. In this respect, this kind of analysis is related to the *strictness analysis* for functional languages [14], where the analysis determines whether a function argument could be evaluated directly, instead of evaluating it only when needed.

The results of the analysis are exploited by a distributed implementation scheme that we have defined as an extension of the Warren Abstract Machine (WAM). In this implementation, the level of copying for each goal argument can be tuned at compile-time. Tests done on sample programs have shown that by adopting the *optimal* level of copying instead of, for instance, the infinite one, we obtain an effective performance increasing (higher than 20% in some cases).

Up to now the framework discussed in the paper has been applied to linear data structures (i.e., lists) and programs with no function symbols. In case of more complex data structures, we have to define carefully the *size* of terms, and consider additional information in order to determine the use of (input) terms. In this respect, also unification operations taking place between (part of input) terms have to be considered explicitly. To this purpose, type analysis systems might be suitable [11]. This is a subject for future work.

References

[1] H. Aït-Kaci. *Warren's Abstract Machine.* The MIT Press, 1991.

[2] M. Bruynooghe and D. Boulanger. Abstract interpretation for (constraint) logic programming. Technical Report CW183, Dept. of Computing Science - K.U. Leuven, 1993.

[3] J.S. Conery, D.F. Kibler. AND parallelism and non-determinism in logic programs In *New Generation Computing* n.3, pages 43–70, 1985.

[4] A. Ciampolini and E. Lamma and P. Mello and C. Stefanelli Multilevel Copying for Unification in Parallel Architectures", In *Workshop Euromicro on Parallel and Distributed Architectures*, Malaga, January 1994.

[5] D. De Schreye S. Decorte and M. Fabris. Automatic inference of norms: a missing link in automatic termination analysis. In D. Miller, editor, *Proceedings of IEEE Symposium on Logic Programming ILPS93*, pages 420–436. The MIT Press, 1993.

[6] B. Demoen, A. Marien. Implementation of Prolog as Binary Definite Programs In *Proc. 2nd Russian Conf. on Logic Programming, LNAI 592*, pages 165–176, Springer-Verlag, 1992.

[7] 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.

[8] P. Kacsuk and M. Wise (eds.). *Implementations of Distributed Prolog.* J. Wiley and Sons, 1992.

[9] 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.

[10] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[11] P. Van Hentenryck. *Personal Communication.* January , 1995.

[12] K. Verschaetse and D. De Schreye. Derivation of linear size relations by abstract interpretation. In *Proc. 5th Symp. Programming Language Implementation and Logic Programming PLILP92, LNCS 631*, pages 296–310. Springer-Verlag, 1982.

[13] K. Verschaetse and D. De Schreye. Deriving termination proofs for logic programs. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 301–315. The MIT Press, 1991.

[14] P. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). *Abstract Interpretation of Declarative Languages*, Chapter 12, 1987.

[15] D.H.D. Warren. An abstract Prolog instruction set. Technical Report TR 309, SRI International, 1983.