Deriving a Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic programs

K. Muthukumar
MCC and Department of Computer Science
The University of Texas at Austin
Austin, TX 78712 - USA
muthu@cs.utexas.edu

M.V. Hermenegildo *
Universidad Politécnica de Madrid (UPM)
Facultad de Informática
28660-Boadilla del Monte, Madrid - Spain
herme@fi.upm.es *or* herme@cs.utexas.edu

Abstract

Bruynooghe described a framework for the top-down abstract interpretation of logic programs. In this framework, abstract interpretation is carried out by constructing an *abstract and-or tree* in a top-down fashion for a given query and program. Such an abstract interpreter requires fixpoint computation for programs which contain recursive predicates. This paper presents in detail a fixpoint algorithm that has been developed for this purpose and the motivation behind it. We start off by describing a simple-minded algorithm. After pointing out its shortcomings, we present a series of refinements to this algorithm, until we reach the final version. The aim is to give an intuitive grasp and provide justification for the relative complexity of the final algorithm. We also present an informal proof of correctness of the algorithm and some results obtained from an implementation.

*This author is supported in part by ESPRIT project 2471 "PEPMA."

1 Introduction

Abstract interpretation is a useful technique for performing a global analysis of a program in order to compute, at compile-time, characteristics of the terms to which the variables in that program will be bound at run-time for a given class of queries. The technique of abstract interpretation for flow analysis of programs in imperative languages was first presented in a sound mathematical setting by Cousot and Cousot [2] in their landmark paper. Later, it was shown by Bruynooghe [1], Jones and Sondergaard [7], Debray [3], and Mellish [10] that this technique can be extended to flow analysis of programs in logic programming languages, and several frameworks or particular analyses have evolved ([8], [13], [14], [15], ...).

In [1], Bruynooghe described a framework for the top-down abstract interpretation of logic programs. In this framework, abstract interpretation is carried out by constructing an *abstract and-or tree* in a top-down fashion for a given query and program. Essentially, starting with the abstract call substitution for the query, abstract substitutions at all points of the abstract and-or tree are computed and finally, the success substitution for the query is computed. If the given program has recursive predicates, then fixpoint computation is necessary.

In this paper, we present an efficient fixpoint algorithm for a top-down abstract interpreter which follows the top-down framework mentioned. This algorithm is derived gradually, evolving from a simple and intuitive fixpoint algorithm (although inefficient), through a series of refinements, to the final one. Other intermediate algorithms which look intuitively correct but which have minor errors are also considered. These errors are pointed out and suitable modifications are made leading to further refinement of the algorithm. The final version of the algorithm is described also in [11] as part of a particular abstract interpreter, although somewhat particularized for that particular application and lacking motivation justifying its complexity. The purpose of this paper is to present the algorithm in its general form and the motivations behind it.

The rest of the paper proceeds as follows: section 2 presents a brief review of the framework used for abstract interpretation and describes the need for fixpoint computation in this context. Section 3 then describes a simple first cut at a fixpoint computation algorithm. This naive algorithm constructs a new abstract and-or tree for every iteration in the fixpoint computation. We observe that this is not efficient since fixpoint computation can be "localized" to recursive predicates. We then proceed to construct a less simple-minded algorithm based on such an idea in section 4. However, we then show that this algorithm has some errors. We present a corrected and refined version of it in section 5. Upon analysis, this refined version is still found to be inefficient. We discuss how to overcome the inefficiency and present the final refinement of our fixpoint computation algorithm in section 6. Finally, we give an informal proof for the correctness of this algorithm in section 7. It is important to note that this fixpoint algorithm is *independent* of the *abstract domain* used in the abstract interpreter. This enhances its usability in different abstract interpreters. Using an abstract domain introduced in [6] and algorithms for computation of abstract entry and success substitutions presented in [12] which use such an abstract domain, we have implemented an abstract interpreter based on the fixpoint algorithm presented in this paper. Results from this implementation on various benchmarks are presented in

appendix A.

2 Framework for Top-Down Abstract Interpretation of Logic Programs

In this section, we present a brief review of the framework for top-down abstract interpretation and describe the need for fixpoint computation in the context of abstract interpretation. A detailed discussion of this framework can be found in [1] and [11]. For a given program and query, an abstract interpreter interprets this program using *abstract* substitutions instead of *concrete* substitutions. The output of the abstract interpreter is a list of abstract substitutions together with the locations of the clauses where they occurred. This information can be used to speed-up the *sequential* as well as *parallel* execution of programs as described in ([9],[3],[15],[11],[6], ...).

An abstract substitution is a finite representation of a possibly infinite set of concrete substitutions. The former and latter are related to each other via a pair of functions referred to as the *abstraction* (α) and *concretization* (γ) functions. The details of these functions as well as the abstract domain (in which abstract substitutions are represented) depend on the type of information to be obtained from the analysis. In this paper, we assume that the set of all possible abstract substitutions for a given clause is finite.[1] This is to guarantee the termination of fixpoint computation.

The input to the abstract interpreter is a set of clauses (the program) and set of "query forms." In its minimal form (least burden on the programmer) such query forms can be simply the names of the predicates which can appear in user queries (i.e., the program's "entry points"). In order to increase the precision of the analysis, query forms can also include a description of the set of abstract (or concrete) substitutions allowable for each entry point. The goal of the abstract interpreter is then to compute in abstract form the set of substitutions which can occur at all points of all the clauses that would be used while answering all possible queries which are concretizations of the given query forms. It is convenient to give different names to abstract substitutions depending on the point in a clause to which they correspond. Consider, for example, the clause h :- $p_1, \ldots, p_n$. Let λ_i and λ_{i+1} be the abstract substitutions to the left and right of the subgoal $p_i, 1 \leq i \leq n$ in this clause. See figure 1(b).

Definition 1 *λ_i and λ_{i+1} are, respectively, the* abstract call substitution *and the* abstract success substitution *for the subgoal p_i. For this same clause, λ_1 is the* abstract entry substitution *(also represented as β_{entry}) and λ_{n+1} is the* abstract exit substitution *(also represented as β_{exit}).*

Control of the interpretation process can itself proceed in several ways, a particularly useful and efficient one being to essentially follow a top-down strategy starting from the query forms. One framework that uses such a strategy is described in detail in [1]. In a similar way to the concrete top-down execution, the abstract interpretation process can then be represented as an abstract AND-OR tree, in which AND-nodes and OR-nodes alternate. A clause head h is an AND-node whose

[1] In fact, in general it is sufficient that the abstract substitutions form a *cpo* of finite height.

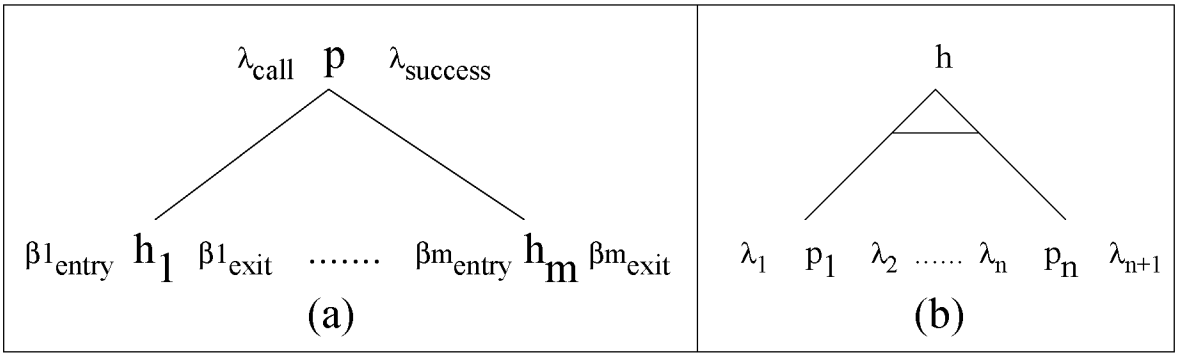

Figure 1: Illustration of the abstract interpretation process

children are the literals in its body $p_1, \ldots, p_n$ (figure 1(b)). Similarly, if one of these literals p can be unified with clauses whose heads are $h_1, \ldots, h_m$, p is an OR-node whose children are the AND-nodes $h_1, \ldots, h_m$ (figure 1(a)). During construction of the tree, computation of the abstract substitutions at each point is done as follows:

- *Computing success substitution from call substitution*: Given a call substitution λ_{call} for a subgoal p, let $h_1, \ldots, h_m$ be the heads of clauses which unify with p (see figure 1(a)). Compute the entry substitutions $\beta 1_{entry}, \ldots, \beta m_{entry}$ for these clauses. Compute their exit substitutions $\beta 1_{exit}, \ldots, \beta m_{exit}$ as explained below. Compute the success substitutions $\lambda 1_{success}, \ldots, \lambda m_{success}$ corresponding to these clauses. The success substitution $\lambda_{success}$ is then the *least upper bound* (LUB) of $\lambda 1_{success}, \ldots, \lambda m_{success}$. Of course the LUB computation is dependent on the abstract domain and the definition of the $\sqsubseteq$ relation.

- *Computing exit substitution from entry substitution*: Given a clause h :- $p_1, \ldots, p_n$ and an entry substitution λ_1, λ_1 is the call substitution for p_1. Its success substitution λ_2 is computed as above. Similarly, $\lambda_3, \ldots, \lambda_{n+1}$ are computed. Finally, λ_{n+1} is obtained, which is the exit substitution for this clause. See figure 1(b).

Given this basic framework, it is clear that a particular analysis strategy needs to:

- Define an abstract domain and substitution framework, and the $\sqsubseteq$ relation,

- Describe how to compute the *entry substitution* for a clause C given a subgoal p (which unifies with the head of C) and its call substitution,

- Describe how to compute the *success substitution* for a subgoal p given its call substitution and the exit substitution for a clause C whose head unifies with p.

Such information represents the "core" of a particular analysis strategy. In this paper, we assume that the abstract domain has already been defined for the application in hand and that algorithms for computing entry and success substitutions as well as the LUB of these abstract substitutions have been provided.

In addition to the three points above, there is, however, one more issue that needs to be addressed. The overall abstract interpretation scheme described works in a relatively straightforward way if the program has no recursion. Consider, on the other hand, a recursive predicate p. If there are two OR-nodes for p in the abstract AND-OR tree such that

- they are identical (i.e., they have the same atoms),
- one is an ancestor of the other, and
- the call substitutions are the same for both,

then the abstract AND-OR tree is *infinite* and an abstract interpreter using the simple control strategy described above will not terminate. In order to ensure termination, some sort of *fixpoint computation* is required.

The algorithms that we describe in this paper for fixpoint computation use *memo tables* [4]. A memo table contains the results of computation already performed. Frequently it is used to avoid needless recomputing. However, in the context of fixpoint computation described in this paper, its main use is to store - possibly incomplete - results obtained from an earlier round of iteration. An entry in the memo table has at least fields: subgoal, its projected call substitution (λ) and its projected success substitution (λ'). Additional fields can be used to characterize the information in this entry (Sections 5 and 6).

3 A Naive Approach to Top-Down Fixpoint Computation

Informally, this approach can be described as follows. Start with an empty memo table with three fields: subgoal, λ and λ'. Using the call substitution of the query, the construction of the abstract and-or tree is started. For a general subgoal p, given its call substitution λ_{call}, λ is computed by projecting λ_{call} onto p. If there is an entry in the memo table for p (modulo renaming of variables) and the same λ, then the value of λ' is obtained from this entry. Else, a subtree for this node is started by computing the *entry* substitutions for all clauses whose heads unify with p. The *exit* substitutions for these clauses are then computed and from these, λ' is computed. This is the broad picture. However, there are some finer points that need to be explained in detail.

Firstly, the procedure described above needs to be repeated until fixpoint is reached for the abstract and-or tree, i.e. until it remains the same *before* and *after* one round of iteration. In order to do this, we could keep two memo tables, one from the past iteration (*old* memo table) and one for the current iteration (*current* memo table). Secondly, we need to use a *flag* to signal the termination of fixpoint computation.

The old memo table is empty when we start the first iteration. Between the end of one iteration and the beginning of another, the old memo table is emptied, the contents of the current memo table are transferred to the old memo table and thus the current memo table is "emptied". For a subgoal p with a projected call substitution λ, if there is no entry in the current memo table, then it is checked if there is an entry in the old memo table. If there is such an entry, then it is copied on to the current

memo table. Else, a new entry is added (p,λ, $\perp$) to the current memo table. After this step, the subtree computation for p is started. When this computation is over, the old value of the projected success substitution λ' in the current memo table replaced with the new value. If there is a change in this value, then a flag which signals the end of fixpoint computation is set to *false*. Else, its value is not changed. Before the beginning of every iteration, this flag is set to *true*.

This procedure can be optimized further. Classify each predicate as follows: If it is recursive, call it a *changing* predicate. If it is non-recursive, call it *non-changing* iff it does not have any changing predicates as one of its subgoals. Otherwise, call it a changing predicate. This can be illustrated with the following example:

```
p(0).
p(X)   :- q(X,Y),p(X).
q(X,Y) :- Y is X-1.
r(X)   :- p(X),write(X).
```

Here, `p(X)` is classified as changing since it is a recursive predicate. Also, `r(X)` is classified as changing even though it is a non-recursive predicate, since one of its clauses has a changing predicate i.e. `p(X)` in its body. The predicate `q(X,Y)` is classified as non-changing since it is non-recursive and it does not have a clause whose body contains a changing predicate.

The idea behind this classification is as follows: After the first iteration, there is no need to recompute the subtree for a non-changing subgoal s with a projected call substitution λ if it already has an entry in the old memo table. This is because it is a non-recursive predicate and all the descendents in its sub-tree are also non-recursive, Hence the value of λ' for this subgoal, once computed, does not change with each subsequent iteration. Consequently, the value of λ' from the old memo table can be copied on to the current memo table and the subtree computation for this subgoal need not be started. Thus this optimization helps to avoid needless recomputation.

Following is a formal version of this algorithm in pseudo-pascal format. It is assumed that, for the given abstract domain procedures for the following procedures are given:[2]

- `call_to_entry(Lambda,Subgoal,Clause,Beta_entry)`: i.e. given the projected call substitution for `Subgoal` and a clause whose head unifies with `Subgoal`, this procedure computes the entry substitution `Beta_entry` for this clause.
- `exit_to_success(Beta_exit,Subgoal,Clause,Lambda,Lambda_prime)`: i.e. given the exit substitution `Beta_exit` for `Clause`, compute the projected success substitution of `Subgoal` from this clause.
- `lub(List_of_abstract_substitutions,Their_lub)`: This procedure computes the `lub` of a given list of abstract substitutions.

```
procedure fixpoint_compute(program,query,call_subst)
```

[2]The algorithms presented in sections 4, 5 and 6 also make use of these procedures.

```
begin
  classify each predicate of the program as ``changing'' or ``non-changing'';
  current_memo_table := nil;
  repeat
    flag := true;
    old_memo_table := current_memo_table;
    current_memo_table := nil;
    call_to_success(program,query,call_subst,success_subst,old_memo_table,
                    current_memo_table,flag);
  until flag = false;
end.

procedure call_to_success(program,subgoal,call_subst,success_subst,
                          old_memo_table,current_memo_table,flag)
begin
  project call_subst on subgoal to obtain lambda;
  if there is an entry in the current_memo_table corresponding to
  (subgoal,lambda), then
    get the value of lambda_prime from this entry;
  else if subgoal is ``non-changing'' and there is an entry in the
  old_memo_table corresponding to (subgoal,lambda), then
    copy that entry i.e. (subgoal,lambda,lambda_prime) to the
    current_memo_table;
  else if subgoal is ``changing'' and there is an entry in the
  old_memo_table corresponding to (subgoal,lambda), then
    copy that entry i.e. (subgoal,lambda,old_lambda_prime) to the
    current_memo_table;
    lambda_to_lambda_prime(program,subgoal,lambda,lambda_prime,
                           old_memo_table,current_memo_table,flag);
    if (lambda_prime <> old_lambda_prime) then
      flag := false;
    endif;
  else /* there is no entry for (subgoal,lambda) either in the old
          or in the current_memo_table */
    create a new entry (subgoal,lambda,bottom) in the current_memo_table;
    lambda_to_lambda_prime(program,subgoal,lambda,lambda_prime,
                           old_memo_table,current_memo_table,flag);
    if (lambda_prime <> bottom) then
      flag := false;
    endif;
```

```
  endif;
  compute success_subst from  lambda_prime and call_subst;
end;

procedure lambda_to_lambda_prime(program,subgoal,lambda,lambda_prime,
                                 old_memo_table,current_memo_table,flag)
begin
  lambda_prime := bottom;
  for each clause whose head unifies with subgoal do
    call_to_entry(lambda,subgoal,clause,beta_entry);
    if clause has a nil body then
      beta_exit := beta_entry;
    else
      entry_to_exit(beta_entry,clause_body,beta_exit,
                    old_memo_table,current_memo_table,flag);
    endif;
    exit_to_success(beta_exit,subgoal,clause,lambda,clause_lambda_prime);
    lambda_prime := LUB of lambda_prime and clause_lambda_prime;
    replace the entry corresponding to (subgoal,lambda) with the
    entry (subgoal,lambda,lambda_prime) in the current_memo_table;
  od;
end;

procedure entry_to_exit(beta_entry,clause_body,beta_exit,
                        old_memo_table,current_memo_table,flag)
begin
  call_subst := beta_entry;
  for each subgoal in clause_body (from left to right) do
    call_to_success(program,subgoal,call_subst,success_subst,
                    old_memo_table,current_memo_table,flag);
    call_subst := success_subst;
  od;
  beta_exit := success_subst;
end;
```

3.1 Drawbacks of this approach

Though the algorithm presented in this section is simple and it avoids some recomputation, it is still quite inefficient. For example, consider a program which has two changing predicates, p and q. Assume further that p reaches its fixpoint in the first iteration itself but q reaches its fixpoint only after

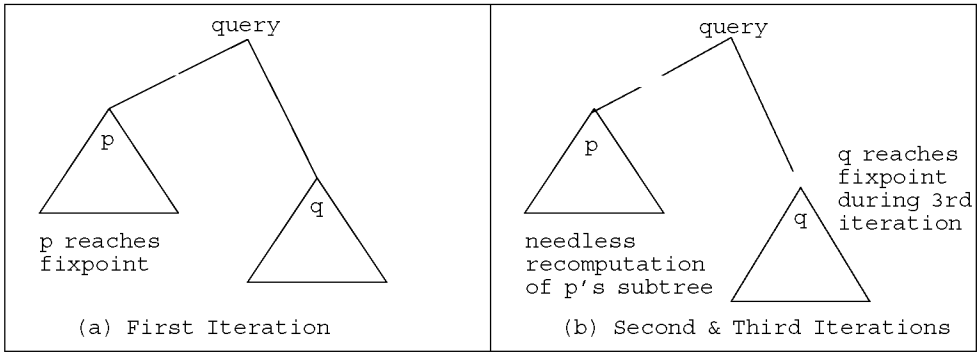

Figure 2: Abstract and-or trees for the three iterations

three iterations. Hence, three iterations are required before fixpoint is reached for the whole program. Even though fixpoint has been reached for p during the first iteration (figure 2(a)), its subtree will be explored during the second and the third iterations (figure 2(b)). This is clearly unnecessary leading to wasted work.

This problem arises because fixpoint computation is done for the whole program rather than just for the *recursive* predicates. The solution to this problem seems to be to *localize* the fixpoint computation for a predicate. We investigate such an approach in the next section.

4 An Algorithm Which Localizes Fixpoint Computation

In this section we develop an algorithm which seeks to *localize* the fixpoint computation for a recursive predicate. Naturally, before the construction of the abstract and-or tree is begun, the predicates of the program are classified as *recursive* or *non-recursive*. In this algorithm, there is only one memo table.

Briefly, the central idea of this algorithm is as follows. Start with the query and its call substitution. For a given subgoal p and its projected call substitution λ, if it has an entry in the memo table, the value of λ' can be used from this entry and there is no need to start the computation of a subtree for this subgoal. If there is no such entry in the memo table, the construction of its sub-tree is started by computing the entry substitutions of all clauses whose heads unify with p. If p is a non-recursive subgoal, then the exit substitutions from these clauses are "lubbed" and p's projected success substitution λ' is computed using this value.

If p is a recursive subgoal, then fixpoint computation is started for p's subtree. A *flag* which signals the completion of this fixpoint computation is initialized to *true*. A new entry is created in the memo table with the values of p and λ. The value of λ' in this entry is initialized to $\perp$. For each clause whose head unifies with p, computation of a subtree is started. After the computation of the exit substitution for this clause, the projected success substitution due to this clause is computed and is lubbed with the old value of λ' to give its new value. If this is different from the old value, then the value of *flag* is changed to *false*. Else, its value is not changed. After the computation of subtrees for all clauses is

completed, then the value of *flag* is examined. If its value is *true*, then fixpoint computation for p is over. Otherwise, it is restarted by changing the value of *flag* to *true* and recomputing the subtrees for all clauses for p.

Following is a formal version of this algorithm in pseudo-pascal format:

```
procedure compute_abstract_and_or_tree(program,query,call_subst)
begin
  classify each predicate of the program as ``recursive'' or ``non-recursive'';
  memo_table := nil;
  call_to_success(program,query,call_subst,success_subst,memo_table);
end.

procedure call_to_success(program,subgoal,call_subst,success_subst,memo_table)
begin
  project call_subst on subgoal to obtain lambda;
  if there is an entry in the memo table corresponding to (subgoal,lambda), then
    get the value of lambda_prime from this entry;
  else if subgoal is ``non-recursive'', then
    lambda_to_lambda_prime(program,subgoal,lambda,lambda_prime,memo_table);
    record the entry (subgoal,lambda,lambda_prime) in the memo table;
  else /* need to do fixpoint computation since subgoal is recursive */
    create a new entry (subgoal,lambda,bottom) in the memo table;
    fixpoint_compute(program,subgoal,lambda,lambda_prime,memo_table);
  endif;
  compute success_subst from lambda_prime and call_subst;
end;

procedure lambda_to_lambda_prime(program,subgoal,lambda,lambda_prime,memo_table)
begin
  lambda_prime := bottom;
  for each clause whose head unifies with subgoal do
    call_to_entry(lambda,subgoal,clause,beta_entry);
    if clause has a nil body then
      beta_exit := beta_entry;
    else
      entry_to_exit(beta_entry,clause_body,beta_exit,memo_table);
    endif;
    exit_to_success(beta_exit,subgoal,clause,lambda,clause_lambda_prime);
    lambda_prime := LUB of lambda_prime and clause_lambda_prime;
  od;
end;
```

```
procedure fixpoint_compute(program,subgoal,lambda,lambda_prime,memo_table)
begin
  lambda_prime := bottom;
  repeat
    flag := true;
    for each clause whose head unifies with subgoal do
      call_to_entry(lambda,subgoal,clause,beta_entry);
      if clause has a nil body then
        beta_exit := beta_entry;
      else
        entry_to_exit(beta_entry,clause_body,beta_exit,memo_table);
      endif;
      exit_to_success(beta_exit,subgoal,clause,lambda,clause_lambda_prime);
      old_lambda_prime := lambda_prime;
      lambda_prime := LUB of old_lambda_prime and clause_lambda_prime;
      if (lambda_prime <> old_lambda_prime) then
        flag := false;
        replace the entry corresponding to (subgoal,lambda) with
        the entry (subgoal,lambda,lambda_prime) in the memo table;
      endif;
    od;
  until flag = true;
end;

procedure entry_to_exit(beta_entry,clause_body,beta_exit,memo_table)
begin
  call_subst := beta_entry;
  for each subgoal in clause_body (from left to right) do
    call_to_success(program,subgoal,call_subst,success_subst,memo_table);
    call_subst := success_subst;
  end;
  beta_exit := success_subst;
end;
```

4.1 Error in this algorithm

The algorithm just described is incorrect. Consider a program which has two mutually recursive predicates p and q. Without loss of generality, assume that fixpoint computation for p was started first. The subtree for p contains a node for q since they are mutually recursive. Now, it is possible that the subtree for q contains a node for p with the same λ as the ancestor node for p (See figure

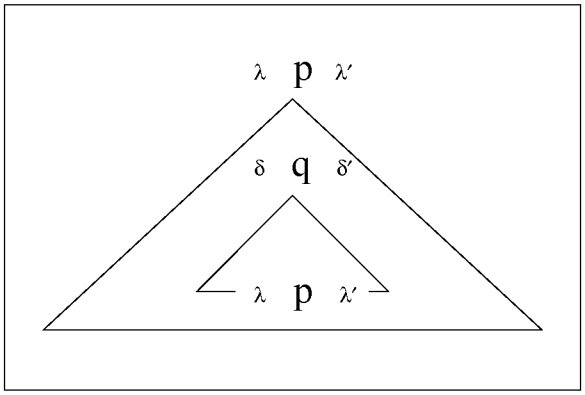

Figure 3: Mutual recursion

3). Consider the following scenario. During the first round of fixpoint computation for p, fixpoint computation for q is also started. This fixpoint computation for q involves using an approximate value of λ' for p from the memo table. After fixpoint computation for q is over, the memo table is appropriately updated. But this entry is not *complete* since it has used an approximate value of λ' for p. Assume that it takes at least two rounds of iteration to reach fixpoint for p. During the second round of the subtree exploration for p, the subtree for q is not explored, since it has an entry in the memo table already. This results in an *approximate* value for the value of λ' for q, and consequently for p, in the memo table. Hence this algorithm is incorrect.

The solution to this problem seems to be to characterize each entry in the memo table as *approximate* or *complete*. In the next section, we investigate such a solution.

5 Refinement #1

The algorithm in this section uses a memo table which, in addition to the three fields: subgoal, λ and λ', uses a fourth field to characterize the value of λ' and a fifth field for the node ID of subgoal. The three values used to characterize λ' and their meanings are as follows:

- *fixpoint*: fixpoint has not been reached for subgoal in this entry and hence this value of λ' is not complete.
- *approximate*: fixpoint has been reached for the subgoal (q) in this entry but, in doing so, it has used a possibly *incomplete* value of the projected success substitution of some other subgoal (p) from the memo table. p occurs in the subtree for q. Hence this value of λ' is incomplete.
- *complete*: fixpoint has been reached for this subgoal and the value of λ' is complete.

In order to detect occurrences of mutual recursion (i.e. as in figure 3), we need to introduce node IDs for recursive subgoals. The IDs are used as follows: When fixpoint computation is started for a predicate q, it is assigned a unique ID. A set variable called subtree_ids[q] is initialized to $\emptyset$. While fixpoint computation is in progress for this predicate, suppose a value from the memo table is used for

λ' of a subgoal p and this value is characterized as *fixpoint*. Then the node ID for p is added to the set subtree_ids[q]. After the fixpoint computation for q is over, this set variable is examined. If it contains any other ID in addition to the ID for q, then the same situation as in figure 3 has occurred, i.e. q is mutually recursive with another predicate. Hence the value of λ' computed for q is characterized as *approximate*. On the other hand, if the variable subtree_ids[q] contains only the node ID for q, then this entry in the memo table is characterized as *complete*.

Consider the following scenario. We are in the middle of fixpoint computation for a subgoal p. In its subtree, we encounter a subgoal q which has an entry in the memo table for its λ. If this entry is characterized as *complete*, then the value of λ' from this entry can be used for q. If it is labeled as *fixpoint*,again, the value of λ' from this entry can be used for q. But this time, the node ID for q is added to subtree_ids[p]. On the other hand, if it is characterized as *approximate*, then the same situation as in figure 3 has occurred and fixpoint computation for q has to be started again.

Following is a formal version of the algorithm in pseudo-pascal format. The following procedures are essentially the same as in section 4 and are therefore not repeated: `compute_abstract_and_or_tree`, `lambda_to_lambda_prime`.

```
procedure call_to_success(program,subgoal,call_subst,success_subst,
                          memo_table,in_set,out_set)
begin
  project call_subst on subgoal to obtain lambda;
  if there is an entry in the memo table corresponding to (subgoal,lambda), then
    if this entry is characterized as ''complete'', then
      get the value of lambda_prime from this entry;
      out_set := in_set;
    else if this entry is characterized as ''fixpoint'',
      get the value of lambda_prime from this entry;
      out_set := in_set + ID for subgoal;
    else /* this entry is characterized as ''approximate'' */
      replace this entry from the memo table with the same values of
      (subgoal,lambda,lambda_prime) but with the label ''fixpoint'';
      fixpoint_compute(program,subgoal,lambda,lambda_prime,
                       memo table,in_set,out_set);
    endif;
  else if subgoal is ''non-recursive'', then
    lambda_to_lambda_prime(program,subgoal,lambda,lambda_prime,memo_table);
    out_set := in_set;
    record the entry (subgoal,lambda,lambda_prime,complete,ID)
    in the memo table;
  else /* need to do fixpoint computation since subgoal is recursive */
    create a new entry (subgoal,lambda,bottom,fixpoint,ID) in the memo table;
    fixpoint_compute(program,subgoal,lambda,lambda_prime,memo_table,in_set,out_set);
```

```
  endif;
  compute success_subst from lambda_prime and call_subst;
end;

procedure fixpoint_compute(program,subgoal,lambda,lambda_prime,memo_table,
                           in_set,out_set)
begin
  get the value of lambda_prime from the entry in memo table for this subgoal;
  subgoal_outset := {};
  repeat
    flag := true;
    for each clause whose head unifies with subgoal do
      call_to_entry(lambda,subgoal,clause,beta_entry);
      if clause has a nil body then
        beta_exit := beta_entry;
      else
        entry_to_exit(beta_entry,clause_body,beta_exit,memo_table,{},clause_outset);
      endif;
      exit_to_success(beta_exit,subgoal,clause,lambda,clause_lambda_prime);
      old_lambda_prime := lambda_prime;
      lambda_prime := LUB of old_lambda_prime and clause_lambda_prime;
      if (lambda_prime <> old_lambda_prime) then
        flag := false;
        replace the entry corresponding to (subgoal,lambda) with the
        entry (subgoal,lambda,lambda_prime,fixpoint,ID) in the memo table;
      endif;
      subgoal_outset := subgoal_outset + clause_outset;
    od;
  until flag = true;
  if (subgoal_outset - {ID for subgoal}) = {}, then
   label := complete;
  else
   label := approximate;
  endif;
  replace the entry for (subgoal,lambda) with the new
  entry (subgoal,lambda,lambda_prime,label,ID);
  out_set := in_set + (subgoal_outset - {ID for subgoal});
end;

procedure entry_to_exit(beta_entry,clause_body,beta_exit,memo_table,in_set,out_set)
```

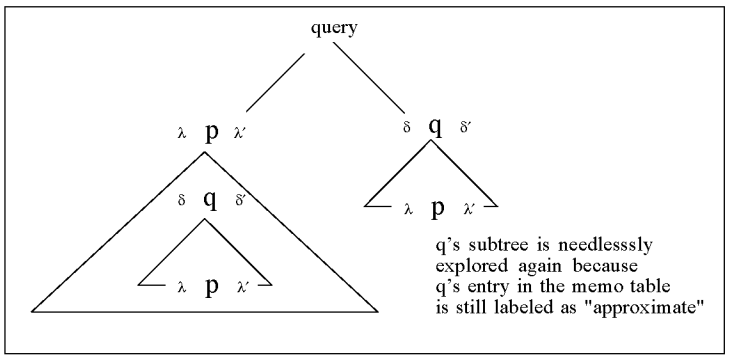

Figure 4: Inefficiency in Refinement #1

```
begin
  call_subst := beta_entry;
  i_set := in_set;
  for each subgoal in clause_body (from left to right) do
    call_to_success(program,subgoal,call_subst,success_subst,memo_table,
                    i_set,o_set);
    call_subst := success_subst;
    i_set := o_set;
  od;
  beta_exit := success_subst;
  out_set := o_set;
end;
```

5.1 Drawbacks of this approach

This algorithm can be improved further. Consider the same scenario as in figure 3 i.e. there are two mutually recursive predicates p and q. Fixpoint computation is started for p first. While doing this, fixpoint computation is started for q too. However, since this fixpoint computation uses an *incomplete* value of λ' for p, the entry for q in the memo table is labeled *approximate*. The consequence of this is that, during subsequent iterations for p, the subtree for q is explored every time. After each one of these fixpoint iterations is completed for q, its entry in the memo table is labeled *approximate*. After the last round of iteration for p is over, its entry in the memo table is labeled *complete* but the entry for q is still labeled *approximate* even though it has used a *complete* value of λ' for p. Now suppose that further computation of the abstract and-or tree entails computation of δ' for q again (See figure 4). Since the entry for q is still labeled *approximate*, its subtree will be explored again. This is clearly unnecessary and makes this algorithm *inefficient*. In the next section, we develop a modification of this algorithm which overcomes this drawback.

6 Final Refinement

The algorithm presented in this section overcomes the above drawback by doing some book-keeping. Consider figure 4 again. Every time after fixpoint is reached for q, the variable `subgoal_outset` is examined. In this case, it contains the IDs for p and q. Every ID other than that of q is added to a new variable `depend_list` i.e. `depend_list[`q`]` := p. When fixpoint computation for p is finally over, the `depend_lists` of all nodes are examined to check if the ID for p occurs in them. Since `depend_list[`q`]` satisfies this criterion, the label for entry for q in the memo table is changed to *complete*, if its value is *approximate*.

Following is a formal version of this algorithm in pseudo-pascal format. Except the procedure for `fixpoint_compute`, the other procedures are the same as in section 5 and are therefore not repeated.

```
procedure fixpoint_compute(program,subgoal,lambda,lambda_prime,memo_table,
                           in_set,out_set)
begin
  get the value of lambda_prime from the entry in memo table for this subgoal;
  subgoal_outset := {};
  repeat
    /* same as the body of the repeat-until loop for
       fixpoint_compute in the Refinement #1 algorithm */
  until flag = true;
  if (subgoal_outset - {ID for subgoal}) = {}, then
   label := complete;
   for each Pid such that (ID for subgoal) occurs in depend_list[Pid] do
     remove (ID for subgoal) from depend_list[Pid];
     if depend_list[Pid] = {}, then
       change the label for the entry for Pid from ``approximate'' to ``complete'';
   end;
  else
   label := approximate;
   depend_list[ID for subgoal] := subgoal_outset - {ID for subgoal};
  endif;
  replace the entry for (subgoal,lambda) with the new
  entry (subgoal,lambda,lambda_prime,label,ID);
  out_set := in_set + (subgoal_outset - {ID for subgoal});
end;
```

We have implemented this algorithm as part of the abstract interpreter for the &-prolog system [5] at MCC. Appendix A contains the results of running this interpreter on an example program.

7 Outline of the proof of correctness of the final version of the fixpoint algorithm

In [11], we had given an outline of proof of correctness of this algorithm. We repeat it here for the sake of keeping this paper self-contained. We are working on a formal proof of correctness and plan to report it in future.

Proposition 1 *Given the following:*

- *an abstract domain that satisfies the conditions:*
  - *that the number of distinct (modulo renaming of variables) abstract substitutions for a clause is finite,*
  - *that they form a lattice with respect to a given partial order*
- *correct, terminating procedures to compute the following:*
  - *abstract entry substitution β_{entry} for a clause C given the abstract call substitution λ_{call} of a subgoal* sg *which unifies with the head of C*
  - *abstract success substitution for a subgoal* sg *given its abstract call substitution and the abstract exit substitution of a clause C whose head unifies with* sg
  - *LUB of two abstract substitutions (of the same clause)*

the fixpoint computation algorithm described above correctly computes the abstract AND-OR tree (i.e., the abstract substitutions at all points) for a given program and goal. Also, it terminates for all inputs.

Proof (Sketch): The *correctness* of this algorithm follows from:

- the fact that it computes the abstract projected success substitution λ' of a subgoal *sg* as the LUB of the abstract projected success substitutions λ'_i computed from the clauses C_i, where $C_i, i = 1, \ldots, n$ are all clauses whose heads unify with *sg*.
- the fact that if an atom *sg* with the same projected call substitution (λ) (modulo renaming of variables) appears in different nodes of the tree, it has the same value for the projected success substitution (λ') at these nodes

Termination: When the given program has no recursive predicates, it is clear that this algorithm terminates since it builds the abstract AND-OR tree in a top-down fashion and that tree cannot have two nodes with the same atom and projected call substitution (modulo renaming of variables), with one node being the descendent of the other.

When the given program has recursive predicates, the termination of this algorithm follows from:

- the fact that the subtree of a node with a recursive predicate p is finite. Since p can only have a finite number of *distinct* call substitutions, the subtree can only have a finite number of occurrences of nodes who have a variant of p and which themselves have subtrees. All other nodes

with p as their predicates use the approximate value of the projected success substitution from the memo table (since they have an ancestor with the same atom and projected call substitution (modulo renaming of variables)) and hence do not have any descendent nodes.

- given that the subtree of a node with a recursive predicate p is finite, it is easy to see that the complete construction of this subtree takes only a finite number of steps. Broadly speaking, the construction of this tree proceeds as follows: First the approximate value of the projected success substitution is computed as the LUB of the projected success substitutions computed from p's non-recursive clauses. Then the sub-tree is dynamically traversed in a depth-first manner and we return to the root of the subtree. At this time, the value of the projected success substitution is updated as the LUB of the old value and the value computed from p's recursive clauses.

  If there is a change in this value, then the dynamic depth-first traversal is continued again. Note that this "looping" through the depth-first traversal can take place only a finite number of times, since the LUB operation is obviously monotonic and the abstract substitutions for a clause form a *finite* lattice and so the fixpoint will be reached in a finite number of steps.

  If there is no change in the value of the projected success substitution for this node, then its subtree is *complete* and so we have reached the end of fixpoint computation for this node.

□

8 Conclusions and Future Work

We started with a naive algorithm for fixpoint computation for top-down abstract interpretation of logic programs and presented a series of refinements to this algorithm. We have provided an informal proof of correctness of the final algorithm and shown the results of running the implementation of this algorithm on an example. The results of running this program on several examples are encouraging in that the abstract interpreter is fast enough to be cost-effective. We are working on a formal proof of correctness of this algorithm and we plan to report on its performance on various benchmark programs in the future.

References

[1] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.

[2] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[3] S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, 5(3):207–229, September 1988.

[4] S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Fourth IEEE Symposium on Logic Programming*, pages 264–272, September 1987.

[5] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[6] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[7] N. Jones and H. Sondergaard. A semantics-based framework for the abstract interpretation of prolog. In *Abstract Interpretation of Declarative Languages*, chapter 6, pages 124–142. Ellis-Horwood, 1987.

[8] H. Mannila and E. Ukkonen. Flow Analysis of Prolog Programs. In *Fourth IEEE Symposium on Logic Programming*, pages 205–214, San Francisco, California, September 1987. IEEE Computer Society.

[9] A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The Impact of Abstract Interpretation: an Experiment in Code Generation. In *Sixth International Conference on Logic Programming*, pages 33–47. MIT Press, June 1989.

[10] C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 463–475. Springer-Verlag, July 1986.

[11] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. Technical Report ACA-ST-232-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, March 1989.

[12] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[13] T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.

[14] A. Waern. An Implementation Technique for the Abstract Interpretation of Prolog. In *Fifth International Conference and Symposium on Logic Programming*, pages 700–710, Seattle,Washington, August 1988.

[15] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699, Seattle, Washington, August 1988. MIT Press.

A Implementation Results

In this section, we present the results of running an implementation of an abstract interpreter which uses the final version of the fixpoint algorithm discussed in section 6. The goal of this abstract interpreter is to infer the groundness and independence of program variables so that run-time groundness and independence checks can be eliminated for an Independent And-Parallel execution of a given logic program.

Details of the abstract domain used for this abstract interpreter can be found in [12]. However, we give a brief description of this abstract domain here to make this paper self-contained. The motivation behind this abstract domain is that abstract substitutions based on it should provide the *sharing* information between the sets of terms to which program variables are bound.

The abstract substitution for a clause is defined to be a *set of sets of program variables* in that clause. Informally, a set of program variables appears in the abstract substitution if the terms to which these variables are bound share a variable. For example, if a clause has two program variables X and Y, the value of an abstract substitution for this clause may be $\{\{X\},\{X,Y\}\}$. This abstract substitution corresponds to a set of substitutions in which X and Y are bound to terms t_X and t_Y such that (1) at least one variable occurs in both t_X and t_Y (represented by the element $\{X,Y\}$) and (2) at least one variable occurs only in t_X (represented by the element $\{X\}$).

Following is an example program (quicksort using difference lists):

```
:- qmode(qsort(Xs,Ys,[]),[[Ys]]). %% query and its call substitution

qsort([],A,A).
qsort([C|D],A,B) :-
        partition(D,C,E,F),
        qsort(F,G,B),
        qsort(E,A,[C|H]),
        G=H.

partition([],_,[],[]).
partition([C|D],A,B,[C|E]) :-
        C > A,!,
        partition(D,A,B,E).
partition([C|D],A,[C|E],B) :-
        C =< A,
        partition(D,A,E,B).
```

The results of running the abstract interpreter on this program are presented in table 1. Basically, the output contains the abstract substitutions at all points of the clauses which have been used for building the abstract and-or tree for the given query. Lists are used in the place of sets for abstract substitutions. The first column gives the position of a subgoal within a clause i.e. `partition/4/2/1`

Position within clause	Subgoal	Abstract call substitution
partition/4/2/1	C>A	[[B],[E]]
partition/4/2/2	partition(D,A,B,E)	[[B],[E]]
partition/4/3/1	C=<A	[[B],[E]]
partition/4/3/2	partition(D,A,E,B)	[[B],[E]]
qsort/3/2/1	partition(D,C,E,F)	[[A],[B],[E],[F],[G],[H]]
qsort/3/2/2	qsort(F,G,B)	[[A],[B],[G],[H]]
qsort/3/2/3	qsort(E,A,[C|H])	[[A],[B,G],[H]]
qsort/3/2/4	G=H	[[A,H],[B,G]]

Table 1: Results of abstract interpretation for the quicksort program

refers to the *first* subgoal for the *second* clause of `partition/4`. The second column gives the subgoal itself and the third column its abstract call substitution.