A Technique for Recursive Invariance Detection and Selective Program Specialization

F. Giannotti
CNUCE-CNR
Via Santa Maria 36,
56100 Pisa, Italy
fosca@gmsun.cnuce.cnr.it

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

Abstract

This paper presents a technique for achieving a class of optimizations related to the reduction of checks within cycles. The technique uses both Program Transformation and Abstract Interpretation. After a first pass of an abstract interpreter which detects simple invariants, program transformation is used to build a hypothetical situation that simplifies some predicates that should be executed within the cycle. This transformation implements the heuristic hypothesis that once conditional tests hold they may continue doing so recursively. Specialized versions of predicates are generated to detect and exploit those cases in which the invariance may hold. Abstract interpretation is then used again to verify the truth of such hypotheses and confirm the proposed simplification. This allows optimizations that go beyond those possible with only one pass of the abstract interpreter over the original program, as is normally the case. It also allows selective program specialization using a standard abstract interpreter not specifically designed for this purpose, thus simplifying the design of this already complex module of the compiler. In the paper, a class of programs amenable to such optimization is presented, along with some examples and an evaluation of the proposed techniques in some application areas such as floundering detection and reducing run-time tests in automatic logic program parallelization. The analysis of the examples presented has been performed automatically by an implementation of the technique using existing abstract interpretation and program transformation tools.

Keywords: Logic Programming, Abstract Interpretation, Program Transformation, Program Specialization, Parallel Logic Programming, Cycle Invariant Detection, Compile-time Optimization.

1 Introduction

This paper presents a technique for achieving a class of optimizations related to the reduction of conditionals within cycles by performing selective program specialization. Much work has been done on techniques such as partial evaluation which allow safe program transformations [1, 9, 25, 24, 5, 2]. Much work has also been done in abstract interpretation, which has been used to achieve several types of high level optimizations: mode inference analysis [8, 21], efficient backtracking [6], garbage collection [19], aliasing and sharing analysis [16, 22, 23] type inferencing [4, 20], etc. It has also been proposed to use the results from abstract interpretation to produce specialized versions of predicates for different run-time instantiation situations [10, 11]. One important issue in program specialization is when to create specialized versions of predicates and how to select the different versions. In [26] and [17] an elegant framework is proposed for abstract interpretation-based program specialization. An automaton is generated which allows run-time selection of the appropriate version. Specialization is controlled by a "wish-list" of optimizations which would be generated by the compiler. Although quite powerful, the framework as described requires the construction of a complex, specialized abstract interpreter with knowledge of the specialization process being done and capable of understanding the "wish-list" from the compiler.

This paper presents a technique (developed independently of [26] and [17]) which attempts similar results but by quite different means. We assume the existence of an abstract interpreter. We also assume that this interpreter uses a domain that is adequate for the type of optimizations that the compiler performs. Based on these assumptions, and rather than asking the compiler for a "wish-list" of desired optimizations, we develop an abstract domain related notion ("abstract executability") which will guide the process of specialization and invariant detection. Also, rather than modifying the abstract interpreter to be aware of the specialization process we leave the interpreter unmodified.[3] Rather, we propose to perform the program specialization and simplification steps externally to the interpreter while still achieving our objective of extracting repetitive run-time tests to the outermost possible level. This is achieved by using program transformation to build a hypothetical situation that would reduce the predicates to be executed in the cycle according to their abstract executability and then using the abstract interpreter again to verify the truth of the hypothesis and (possibly) confirm the proposed simplification.

Consider the following conditional:

$$p(X) \leftarrow q(X), cond(test(X), p(X), r(X)).$$

Program transformation can be used to build a hypothetical situation that reduces the predicates to be executed in the cycle. For example, we can hypothesize that once the test in the conditional succeeds, it will always succeed. A correct program transformation under this hypothesis would be

$$\begin{gathered} p(X) \leftarrow q(X), cond(test(X), p1(X), r(X)) \\ p1(X) \leftarrow q(X), p1(X). \end{gathered}$$

Of course, this transformation is legal if we are sure that $test(X)$ will remain true in all the recursive calls of p in the *then branch*. If that is the case the relevance of the obtained optimization will depend on the complexity of $test(X)$ and on the number of nested recursive calls in the *then branch*. The interesting issue is whether the abstract interpretater can derive if $test(X)$ will be true in all the recursive calls. This depends on the capabilities of the abstract interpreter and precisely these capabilities can be used as a guideline for when to perform the transformation. I.e. given an abstract interpreter, a class of predicates that can be executed directly (reduced to true or false) on the information generated by the abstract interpreter can be identified. Then, rather than blindly performing hypothesis and transformations this class is used to select only potentially useful transformations. The abstract interpreter, run for a second time on the transformed program to verify the truth of the hypothesis formulated, has then a chance of being successful in its task. Our conviction is that such classes of predicates can be easily found for each abstract interpreter.

The idea of leaving the abstract interpreter unmodified is motivated by the consideration that the interpreter is probably already a quite complex module which may be quite difficult to modify and that therefore there is practical advantage in using this module as is. This appears to be the case with most current implementations. In addition, this allows the use of several different abstract interpreters with only minor modifications to the rest of the system. Our description, thus, will be quite independent from the abstract interpreter, which will be considered as a "black box."

The paper is organized as follows: the following section (section 2) recalls the basic ideas of Abstract Interpretation and introduces the concept of and-or graph to represent the result of an abstract interpretation process. Section 2 presents a class of predicates that may be executed at compile-time by using the information collected by a generic abstract interpreter. In sections 3 and 4 the and-or graph representation is exploited to describe the basic program transformation and optimization techniques proposed, based on the concept of abstract executability. The possibility of performing these optimizations using abstract interpretation and program transformation occurred to us while considering their implementation in the context of the abstract interpreter of the &-Prolog system [13].[4] Section 5 is dedicated to some examples illustrating the applicability of such techniques

[3]Including any other type of program specialization and optimization that it may be doing.

[4]Although the techniques that will be proposed are of general applicability, examples from the &-Prolog system will

to several programs, including optimizing the automatic parallelization process in the &-Prolog system. Finally, section 6 presents our conclusions.

Some knowledge of Prolog and Abstract Interpretation is assumed.

2 Abstract Interpretation of Logic Programs

Abstract interpretation is a useful technique for performing 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 interesting aspect of abstract interpretation vs. classical types of compile-time analyses is that it offers a well founded framework which can be instanciated to produce a rich variety of types of analysis with guaranteed correctness with respect to a particular semantics [3, 4, 7, 8, 16, 18, 21, 23].

In abstract interpretation a program is "executed" using *abstract substitutions* instead of actual substitutions. An abstract substitution is a finite representation of a, possibly infinite, set of actual substitutions in the concrete domain. The set of all possible terms that a variable can be bound to in an abstract substitution represents an "abstract domain" which is usually a complete lattice or cpo which is ascending chain finite (such finiteness required, in principle, for termination of fixpoint computation).

Abstract substitutions and sets of concrete substitutions are related via a pair of functions referred to as the *abstraction* (α) and *concretization* (γ) functions. In addition, each primitive operation u of the language (unification being a notable example) is abstracted to an operation u' over the abstract domain. Soundness of the analysis requires that each concrete operation u be related to its corresponding abstract operation u' as follows: for every x in the concrete computational domain, $u(x)$ is "contained in" $\gamma(u'(\alpha(x)))$.

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) the 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. Different names distinguish abstract substitutions depending on the point in a clause to which they correspond: *abstract call substitution* and the *abstract success substitution* are meant for the literal level while the terms *abstract entry substitution* and *abstract exit substitution* refer to the clause level.

A general mechanism for representing the output of the abstract interpretation process is the *and-or graph* associated with a logic program for a given entry point. It is a finite representation of a usually infinite *and-or tree* adorned with the *abstract substitutions*. It is worth noting that, even if the notion of and-or graph is typical of the *top-down* approaches to abstract interpretation, it also encompasses the *bottom-up* approaches which increasingly emphasize recording the *call patterns* of a program. Thus or_nodes are decorated with the abstract call and success substitutions for the associated predicate, whereas and_nodes are decorated with the abstract entry and exit substitutions for the associated clause. Such nodes of an and-or graph are defined by the following equations:

$$\mathit{and\_node} ::= A \times 2^{Sub} \times 2^{Sub} \times \mathit{or\_node}^*$$
$$\mathit{or\_node} ::= \mathit{rec\_call}\ \textbf{of}\ A \mid A \times 2^{Sub} \times 2^{Sub} \times \mathit{and\_node}^*$$

where A is the set of all atoms and Sub is the set of concrete substitutions, | denotes discriminated union, *rec_call* is the constructor for recursive calls and "*" stands for sequence construction.

be used throughout the paper for motivational purposes.

Intuitively, an and_node represents a clause in the program, and it is composed by an atom that is the head of the clause, the entry and the exit substitutions for the clause, and the list of children or_nodes each corresponding to a literal in the body of the clause.

An or-node represents a literal in the body of a clause; if the literal is a descendent of a literal that calls the same predicate then the construction of the and-or tree is suspended and the or-node refers back to the ancestor or-node. Otherwise, the or-node is composed by a literal, the call and the success substitutions for the literal, and the list of children and-nodes each corresponding to a clause for the predicate of the literal.

The various actual abstract interpreters differ in the way the finite and-or graph is constructed, i.e. when they decide to prune recursion and whether they allow different instances of the same clause. These options affect the accuracy of the computed abstract substitutions for the *recursive cliques*, i.e. the path on the graph that connects a predicate with one of its recursive calls, if there are.

Note that if the abstract interpreter itself already performs a certain degree of program specialization (as is sometimes the case) the abstract and-or tree represents the transformed program. In this case we assume that the specialized predicates have been renamed appropriately.

3 Abstract Executability and Program Transformation

In this section the concept of abstract executability is defined. As mentioned before, the recognition of potentially abstractly executable goals will be the guiding heuristic in guiding program specialization to obtain that optimizations desired. For example, sometimes Prolog programs (and very often the programs resulting from the transformations performed in the first stages of a Prolog compiler) mix logic predicates with meta-predicates which deal with program variables characteristics or types. Examples of such predicates are $var(X)$, $nonvar(X)$, $number(X)$, $list(X)$, $integer(X)$, $ground(X)$, $atomic(X)$, $independent(X, Y)$, etc. Some can be based on others: for example $independent(X, Y)$, which explores the two terms to which X and Y are bound to check that they do not have any common variables, can be written in terms of $var/1$ and $== /2$. Such predicates can be part of the original program or they may have been introduced by compilation stages performing tasks such as type-checking, indexing, program parallelization, etc. A characteristic of these predicates is that they seldom modify the current substitution. They are mostly used as “tests” to have an effect on the control of the program, rather than on the logic (which would obviously not be appropriate, given their generally extralogical nature).

Another important characteristic of such predicates is that they can potentially be executed on practical abstract domains, that is, they can be reduced to true or false by simply operating on the abstract representation of the possible bindings of their arguments. Consider for instance the abstract domain consisting of the three elements $\{int, free, any\}$. These elements respectively correspond to the set of all integers, the set of all unbound variables, and the set of all terms. An abstract substitution is then defined as a mapping from program variables to elements of the abstract domain. Assume a correct abstract interpreter whose estimates of integer type and freeness are conservative in the sense that it infers these values only when it is possible to guarantee that all possible substitutions bind respectively to integers and unbound variables. Consider the following clause containing the predicate $ground(X)$:

$$p(X, Y) \leftarrow q(Y), ground(X), r(X, Y).$$

Assume now that such an interpreter infers the call substitution for $ground(X)$ to be $\{Y/free, X/int\}$. This means, that if S is the set of all possible concrete substitutions corresponding to the program point just before $ground(X)$, then $\forall \theta_i \in S,\ X/n \in \theta_i$ where n is an integer. Since given any term t $integer(t) \rightarrow ground(t)$, then, and knowing that $ground/1$ doesn't modify its arguments, $\forall \theta_i \in S,\ ground(X) \equiv ground(\gamma(int)) \equiv true$. Therefore we can “execute on the abstract domain” the $ground(X)$ literal and reduce it to true. Note that this also trivially holds in the case where the abstract domain directly captures the information required by the literal for executability. This would be the case in the previous example had the abstract domain been for example $\{ground, free, any\}$ and the call substitution for $ground(X)$ been $\{Y/free, X/ground\}$.

We call the characteristic of a predicate sketched above "abstract executability." In general the condition for *abstract executability* according to an abstract domain D equipped with the concretization function γ and the abstraction function α is the following:

Definition (Abstractly executable goal): Given an abstract call substitution λ and a literal A, A is said to be abstractly executable and succeed (resp. fail) if $\forall \theta \in \gamma(\lambda)$ the concrete evaluation of $A\theta$ succeeds with empty concrete answer substitution (resp. finitely fails).

Trivially, we can state that a literal which is abstractly executable and succeeds (rep. fails) can be substituted by *true* (resp. *false*).

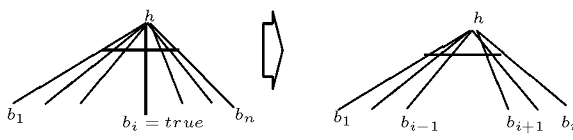
In other words, the (potentially) abstractly executable predicates behave as recognizers of the abstract values. In fact, if a predicate *testT* is a recognizer for the abstract value T then $\forall t \in \gamma(T)$ $testT(t) = true$. Now, if the abstract call substitution for an atom $testT(X)$ contains X/T then $testT(T)$ is a tautology and we are authorized to reduce it to true.

For instance the literal $ground(X)$ can be abstractly reduced to *true* w.r.t. the abstract call substitution $\{X/ground\}$, it can be abstractly reduced to *false* w.r.t. the abstract call substitution $\{X/free\}$ while it cannot be abstractly executed w.r.t. the abstract call substitution $\{X/any\}$.

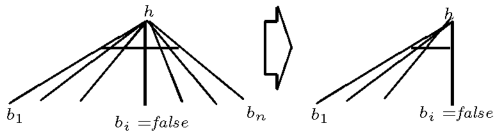
Let us remark that the notion of abstract executability is only meaningful with respect to the final outcome of the abstract interpreter since only the final abstract substitutions represent *all* the possible concrete substitutions. Nevertheless, it is a design option of an abstract interpreter to let the abstractly executable predicates affect the abstract interpretation itself. In fact if recognizers of the abstract values are considered built-ins then an abstract counterpart already exists. As an example, if the abstract interpreter exploits the semantics of the built-in ground, it will include the binding $X/ground$ in the exit substitution of a literal $ground(X)$. In other words, the abstractly executable predicates play two different roles. At the end of the abstract interpretation process, they can be simplified by examining their final abstract call substitutions. During the abstract interpretation process they are clearly useful, as any other builtin, for enriching the information being inferred through the knowledge of their abstract exit substitutions.

The possibility of abstractly executing predicates reducing them to true or false allows for a further analysis phase aimed at performing simplifications of the final and-or graph and, consequently, of the program itself. This is done using standard program transformation and partial evaluation techniques. It is not the purpose of this paper to give an overview of such techniques. Rather, a simple (and non-exhaustive) repertoire of simplifications is proposed, described by the following rewriting rules on the and-or graph (head unification is assumed to be expressed in the form of unification goals at the beginning of the body, i.e. clauses are in the "normal form" of [3]):

• Case 1 – true in and-node:



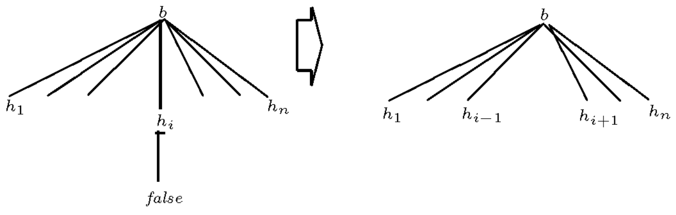
• Case 2 – false in and-node:



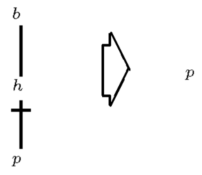
note that if the literals to the left of the literal reduced to *false* are pure, then they can also be eliminated.[5] In general, all literals to the right of the rightmost impure literal can be eliminated.

[5]Ignoring infinite computations. This transformation in fact has the arguably beneficial effect of augmenting the finite

• Case 3 – Simplifying an or-node:



• Case 4 – Collapsing an or-node:



Note that, since no unification appears in the head, the call from g to h cannot produce any bindings as is essentially just parameter passing.

Further optimization of the resulting programs can be done using well known program transformation and partial evaluation techniques.

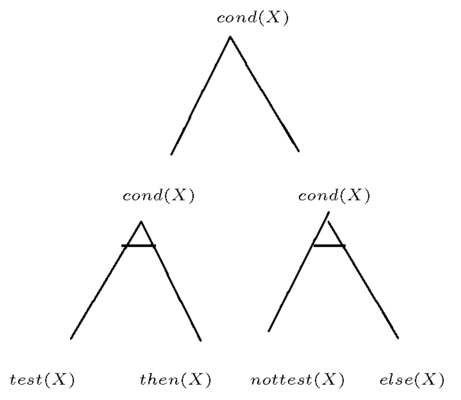
As an example, let's consider the if-then-else construct of Prolog. First, note that, provided $test(X)$ has no side-effects, is sufficiently instantiated (or *nottest* can be safely defined), and modulo additional computation time,[6]

$$cond(X) \leftarrow (test(X) \rightarrow then(X); else(X)).$$

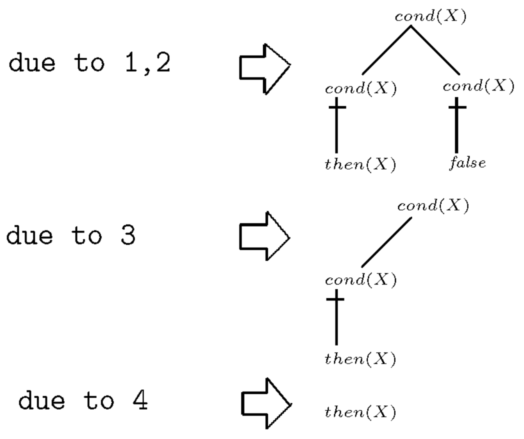
is essentially an (efficient) abbreviation of:

$$cond(X) \leftarrow test(X), then(X).$$
$$cond(X) \leftarrow nottest(X), else(X).$$

The associated and-or graph is labeled "a)" in the following figure:



a)



b)

Assuming that $test(X)$ can be abstractly reduced to *true* and hence that $nottest(X)$ can be reduced to *false*, the sequence of simplifications in figure "b)" above takes place. As a consequence, the original call to $cond(X)$ in the and-or graph can be replaced with the call to $then(X)$.

An interesting application of the conditional simplification is during a test for *floundering* in the evaluation of negated goals (or, better stated, *warning* against floundering). A conditional goal like $(ground(X) \rightarrow not\ p(X); error, abort)$ warns against incorrectness with respect to the negation as failure rule. If ground(X) is simplified according to the above observation, correctness is preserved. The optimization is relevant because the cost of a groundness check is proportional to the size of the term being checked.

As a final remark, notice that the correctness of the proposed simplifications is a straightforward consequence of the correctness of the associated Abstract Interpretation framework (w.r.t. the

failure set of the program.

[6]For conciseness, the actual parameter passing has been abstracted out, as in the rest of this section, into a single variable X. The extension of the transformation to handle actual arguments is trivial and is illustrated in the examples shown in section 5.

semantics that the abstract interpreter is derived from).

4 A transformation for detecting deeper invariants

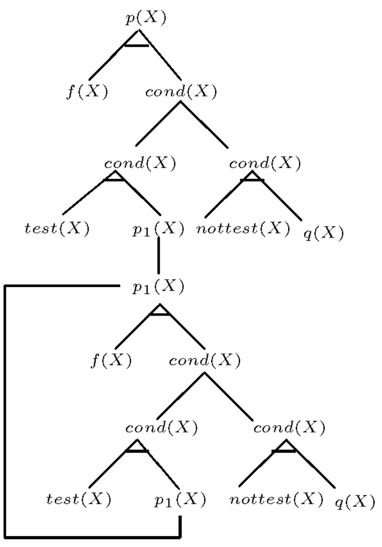
Although the direct applicability of the simplification criteria introduced in the previous section is in practice pretty rare, they do suggest a further possibility: that further optimization is possible in the case of recursive predicates. The idea is to transform the program into an equivalent form which gives more chances to obtain a simplification by highlighting possible invariant predicates which can then be extracted out of cycles. Consider the following recursive clause for a predicate p:

$$p(X) \leftarrow f(X), (test(X) \rightarrow p(X); q(X)).$$

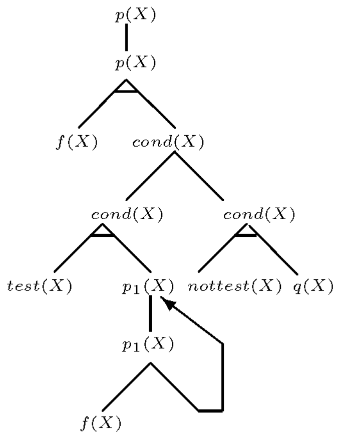
and assume that the abstract interpreter cannot reduce $test(X)$ to either *true* or $false$. It is still possible that during the concrete interpretation, the atom $test(X)$ after its first evaluation to true will also succeed in all the subsequent recursive calls through the *then branch*. This consideration brings us to define a transformation of the clause which separates the first iteration from the subsequent ones:

$$p(X) \leftarrow f(X), (test(X) \rightarrow p1(X); q(X)).$$
$$p1(X) \leftarrow f(X), (test(X) \rightarrow p1(X); q(X)).$$

The above transformation introduces a copy of the predicate p with a new name, and the recursive calls refer to new predicate $p1$. Picture "c)" below represents the transformation at and-or graph level hiding at this stage the abstract substitutions:



c)



d)

It is now possible to run the abstract interpreter to check whether $test(X)$ can be simplified to *true* in $p1$ exploiting the fact the the abstract interpreter reached $p1$ under the hypothesis $test(X) = true$ in p, i.e. the initial call substitution of $p1$ has been extended with the new binding $\{X/T\}$ if $test/1$ is the recognizer for T. Note that if $test(X)$ is a builtin "understood" by the abstract interpretation framework, the extension of abstract substitution is for free by the application of the abstract interpreter. Now, if when running again the abstract interpreter, the and-or tree contains again the binding $\{X/T\}$ as abstract call substitution for $p1$, the basic simplification for the conditional in the clause for $p1$ applies yielding the final and-or graph of figure "d)" above, corresponding to the following optimized code:

$$p(X) \leftarrow f(X), (test(X) \rightarrow p1(X); q(X)).$$
$$p1(X) \leftarrow f(X), p1(X).$$

The above optimization corresponds to the detection of the invariant $test(X)$ for the recursive predicate $p1$; the abstract interpreter actually proved that $test(X)$ is invariant for $p1$: in fact $test(X)$ is *true* in the preconditions for $p1$ and the abstract interpreter derives that it is *true* in all the derivations from $p1$. Thus, in the final code the invariant has been extracted out of the cycle: $test(X)$, if *true*, will be executed only the first time; otherwise, the control remains to p (executing the else branch) with the open possibility of capturing possible invariance later on.

The generalization of the ideas presented above hinges on the and-or graph definition. We designate the critical portions of the graph, i.e. recursive cliques containing abstractly executable predicates. The transformation then consists in the duplication of such clique, and, after the simplification process, rebuilding a minimal logic program from the transformed graph. The full algorithm is given in [12]. The overall process includes the following phases: Abstract Interpretation (decorates the and-or graph with the computed abstract substitutions), Simplification, Transformation (subgraphs containing executable predicates within cycles are duplicated), second Abstract Interpretation (aimed at confirming the invariants), Simplification, and Final Transformation (new parts of the graph that have not been affected by simplifications are eliminated and the final program is generated).

It should be mentioned that although a minimum of two passes of the abstract interpreter is needed to achieve the desired result additional iterations might improve the program further. This cycle could be repeated until fixpoint. However, two passes appear to be a good compromise between precision and efficiency.

A final consideration worth noting is that the correctness of the process relies on two observations: a) the transformations are equivalence preserving since they perform copies of the graph, connecting them by proper renaming; b) the simplifications on the transformed graph are correct provided that the Abstract Interpreter is sound in the sense of section 2.

5 Examples on an Implementation based on the &-Prolog System

As mentioned in the introduction, the possibility of performing optimizations using multiple abstract interpretation and program transformation occurred to us while considering their implementation in the context of the abstract interpreter of the &-Prolog system [13]. In &-Prolog, abstract interpretation is primarily concerned with the detection of argument groundness and goal independence in logic programs. This information is used to achieve automatic parallelization of such programs, by exploiting independent and-parallelism[14]. &-Prolog uses two layers of static analysis: a local one (at clause level — the "annotator") and a global one (based on an abstract interpretation using the sharing/freeness domain [23], capable of inferring independence, groundness, and freeness of variables — the "analyzer"). The former determines conditions under which goals can be "safely" executed in parallel (i.e. while preserving the search space of the original program [14, 15]). It generates an annotated version of the program which contains parallel conjunctions, sometimes inside conditionals. The run-time tests in these conditionals are generally groundness and independent checks on clause variables. The latter layer analyses the global flow of the program and eliminates unnecessary checks. This is an important efficiency issue since these tests can be quite expensive. Although the abstract interpreter has proven quite powerful, there are situations where such checks can be eliminated only in particular cases. Indeed, we have often discovered further opportunities for optimization of conditionals inside cycles by performing program transformation and running the abstract interpreter a second time on the transformed programs. In order to illustrate the practical application of the techniques presented in the previous sections several experiments performed with this system are presented.

An interesting practical example, in the context of program parallelization, is the multiplication of matrices of integers. This program is sometimes used as a benchmark for evaluating the performance of parallel logic programming languages. The following is a part of the program relevant to our example:

```
main:- read(M), read(V), multiply(M,V,Result), write(Result).
```

```
multiply([],_,[]).
multiply([V0|Rest], V1, [Result|Others]):-
      multiply(Rest, V1, Others), vmul(V0,V1,Result).

vmul([],[],0).
vmul([H1|T1], [H2|T2], Result):-
      vmul(T1,T2, Newresult), scalar_mult(H1,H2,Product),
      add(Product,Newresult,Result).
```

The tests introduced by the Annotator on the procedure multiply are simplified by the first pass of the abstract interpreter resulting in the following program (note that the abstract interpreter has inferred that Result and Others are free variables):

```
multiply([],_,[]).
multiply([V0|Rest],V1,[Result|Others]) :-
      ( ground(V1), indep([[V0,Rest]]) ->
        multiply(Rest,V1,Others) & vmul(V0,V1,Result)
      ; multiply(Rest,V1,Others),  vmul(V0,V1,Result) ).
```

Where `&` means that two goals can be executed in parallel and `ground(V1), indep([[V0,Rest]])` are dynamic tests for groundness and independence that express the sufficient condition under which the two goals can be parallelized.

A relevant hypothesis is that the test in the conditional will be an invariant in the recursive loop. Another hypothesis is built to detect whether there is an invariant in the else branch. The transformed program is the following:

```
multiply([],_,[]).
multiply([V0|Rest],V1,[Result|Others]) :-
      ( ground(V1), indep([[V0,Rest]]) ->
        multiply_then(Rest,V1,Others) & vmul(V0,V1,Result)
      ; multiply_else(Rest,V1,Others),  vmul(V0,V1,Result) ).

multiply_then([],_,[]).
multiply_then([V0|Rest],V1,[Result|Others]) :-
      ( ground(V1), indep([[V0,Rest]]) ->
        multiply_then(Rest,V1,Others) & vmul(V0,V1,Result)
      ; multiply(Rest,V1,Others),       vmul(V0,V1,Result) ).

multiply_else([],_,[]).
multiply_else([V0|Rest],V1,[Result|Others]) :-
     ( ground(V1), indep([[V0,Rest]]) ->
       multiply(Rest,V1,Others) & vmul(V0,V1,Result)
     ; multiply_else(Rest,V1,Others), vmul(V0,V1,Result) ).
```

The second pass of the abstract interpreter then determines that V1 is always ground in multiply_then. With this information we see that the multiply_else transformation was not successful in determining an invariant, and that the multiply_then transformation was partially successful: the first half of the test (ground(V1)) is found to be an invariant. The program is therefore rewritten as:

```
multiply([],_,[]).
multiply([V0|Rest],V1,[Result|Others]) :-
     ( ground(V1), indep([[V0,Rest]]) ->
       multiply_then(Rest,V1,Others) & vmul(V0,V1,Result)
```

```
      ; multiply(Rest,V1,Others),       vmul(V0,V1,Result) ).

multiply_then([],_,[]).
multiply_then([V0|Rest],V1,[Result|Others]) :-
      ( indep([[V0,Rest]])  ->
        multiply_then(Rest,V1,Others) & vmul(V0,V1,Result)
      ; multiply(Rest,V1,Others),       vmul(V0,V1,Result) ).
```

The repeated tests for ground(V1) are eliminated once the test succeeds in the first iteration. Note that in the normal use of the multiply predicate the remaining test checks for the independence of two free variables. This is a quite inexpensive check, the resulting program thus being quite efficient.

A different example which is unrelated to parallelization and takes advantage of the fact that "nonvar" is executable on the sharing/freeness abstract domain is the following:

```
map_add1(_,[]).
map_add1(X,[Y|TY]):-
     ( nonvar(X) -> Y is X+1, map_add1(X,TY)
                 ;  X is Y-1, map_add1(X,TY) ).
```

The first abstract interpretation pass cannot determine which side of the conditional will be taken. Again the hypothesis is that once a branch is taken the condition will hold. The specialized version of the program for testing the hypothesis is:

```
map_add1(_,[]).
map_add1(X,[Y|TY]):-
     ( nonvar(X)-> Y is X+1, map_add1_nvX(X,TY)
                ;  X is Y-1, map_add1(X,TY) ).

map_add1_nvX(_,[]).
map_add1_nvX(X,[Y|TY]):-
     ( nonvar(X)-> Y is X+1, map_add1_nvX(X,TY)
                ;  X is Y-1, map_add1(X,TY) ).
```

The abstract interpreter can now determine that X is always non-var in the body of map_add1_nvX and generates the simplified version:

```
map_add1(_,[]).
map_add1(X,[Y|TY]):-
     ( nonvar(X)-> Y is X+1, map_add1_nvX(X,TY)
                ;  X is Y-1, map_add1(X,TY) ).

map_add1_nvX(_,[]).
map_add1_nvX(X,[Y|TY]):-
     Y is X+1, map_add1_nvX(X,TY).
```

which dynamically and inexpensively traps the invariant avoiding run-time checking.

6 Conclusions

The paper has presented a technique for achieving a class of optimizations related to the reduction of conditionals within cycles. The technique is somewhat analogous to cycle invariant detection and removal in traditional compilers. The application of the techniques results in a specialized program and code which detects when it is legal to use the specialized version. A important feature of the technique presented is that it does not require any changes to an existing abstract interpreter to achieve a quite useful form of program specialization. Rather, the technique makes use of repeated applications (generally two) of the abstract interpretater, with program transformations interspersed. The transformations are used to build a hypothetical situation that could bring to the simplification of some predicates that couldn't be simplified by the first pass of the abstract interpreter. The abstract interpreter is the used to verify the truth of such hypothesis and possibly confirm the proposed simplification.

The technique was applied to a series of prototypical benchmarks, showing its usefulness. We have also applied these techniques to other programs, such as the Boyer and Moore theorem prover. The obtained performance figures are appealing enough to motivate further investigation, in particular regarding how to extend the class of predicates that can be executed at abstract interpretation-time and also which other types of control constructs are amenable to similar transformations.

References

[1] K. Benkerimi and J. Lloyd. A partial evaluation procedure for logic programs. In *Proceedings of the North American Conference on Logic Programming*, pages 343–358. MIT Press, October 1990.

[2] D. Bjorner, A.P. Ershov, and N.D. Jones, editors. *Partial Evaluation and Mixed Computation – Proceedings of the Gammel Avernaes Workshop*. Noth-Holland, October 1987.

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

[4] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *Fifth International Conference and Symposium on Logic Programming*, pages 669–683, Seattle, Washington, August 1988. MIT Press.

[5] M. Bugliesi and F. Russo. Partial evaluation in prolog: Some improvements about cut. In *Proc. of the 1989 North American Conference on Logic Programming*, pages 645–660. MIT Press, 1989.

[6] J.-H. Chang and Alvin M. Despain. Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis. In *International Symposium on Logic Programming*, pages 10–22. IEEE Computer Society, July 1985.

[7] 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.

[8] S.K. Debray. Register allocation in a prolog machine. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 267–275, Salt Lake City, 1986.

[9] J. Gallagher. Transforming logic programs by specializing interpreters. In *Proc. of the 7th. European Conference on Artificial Intelligence*, 1986.

[10] J. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialization. In *1990 International Conference on Logic Programming*, pages 732–746. MIT Press, June 1990.

[11] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing*, 6(2–3):159–186, 1988.

[12] F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. Technical Report CLIP7/91.0, U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, February 1991.

[13] 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.

[14] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[15] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[16] 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.

[17] D. Jacobs, A. Langen, and W. Winsborough. Multiple specialization of logic programs with run-time tests. In *1990 International Conference on Logic Programming*, pages 718–731. MIT Press, June 1990.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[24] D. Sahlin. The mixtus approach to the automatic evaluation of full prolog. In *Proceedings of the North American Conference on Logic Programming*, pages 377–398. MIT Press, October 1990.

[25] P. Sestoft. A bibliography on partial evaluation and mixed computation. In *Proceedings of the Workshop on Partial Evaluation and Mixed Computation*. North-Holland, October 1987.

[26] W. Winsborough. Path-dependent reachability analysis for multiple specialization. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.