EXPLOITING GOAL INDEPENDENCE IN THE ANALYSIS OF LOGIC PROGRAMS

MICHAEL CODISH
MAURICE BRUYNOOGHE
MARIA GARCÍA DE LA BANDA
MANUEL HERMENEGILDO

▷ This paper illustrates the use of a top-down framework to obtain goal independent analyses of logic programs, a task which is usually associated with the bottom-up approach. While it is well known that the bottom-up approach can be used, through the magic set transformation, for goal dependent analysis, it is less known that the top-down approach can be used for goal independent analysis. The paper describes two ways of doing the latter. We show how the results of a goal independent analysis can be used to speed up subsequent goal dependent analyses. However this speed-up may result in a loss of precision. The influence of domain characteristics on this precision is discussed and an experimental evaluation using a generic top-down analyzer is described. ◁

1. INTRODUCTION

The framework of abstract interpretation [12] provides the basis for a semantic approach to data-flow analysis. A program analysis is viewed as a non-standard

Address correspondence to M. Codish, Department of Mathematics and Computer Science, Ben-Gurion University, Israel or email: codish@bengus.bgu.ac.il; mbanda@cs.monash.edu.au; maurice@cs.kuleuven.ac.be; herme@fi.upm.es. A preliminary version of this paper appeared as [5] This research was supported in part by CEC DGXIII ESPRIT Project "PRINCE", CEC HCM-project ABILE (CHRX-CT94-0624), CEC DGIII EC-Israel collaborative activity, ISC-IL-90-PARFORCE and CICYT project IPL-D. M. Codish was supported by a post doctoral fellowship from K.U. Leuven. María José García de la Banda was supported in part by a Spanish Ministry of Education Grant. M. Bruynooghe is supported by the Belgium National Fund for Scientific Research.

THE JOURNAL OF LOGIC PROGRAMMING

© Elsevier Science Inc., 1997
655 Avenue of the Americas, New York, NY 10010 97/$7.00

semantics defined over a domain of data descriptions where the syntactic constructs in the program are given corresponding non-standard interpretations. For a given language, different choices of a semantic basis for abstract interpretation may lead to different approaches to analysis of programs in that language. For logic programs we distinguish between two main approaches which have been termed: "bottom-up analysis" and "top-down analysis" [20]. Bottom-up analyses are typically based on abstractions of bottom-up semantics such as the classic T_P semantics, while top-down analyses are typically based on abstractions of top-down semantics such as the SLD semantics. In addition, we distinguish between "goal dependent" and "goal independent" analyses. Intuitively, a goal dependent analysis provides information about the possible behaviors of a specified set of initial goals and a given logic program. In contrast, a goal independent analysis considers the program in isolation.

Traditionally, the standard meaning of a logic program P is given as the set of ground atoms in P's vocabulary which are implied by P. The development of top-down analysis frameworks was originally driven by the need to abstract not only the declarative meaning of programs, but also their behavior. To this end it is straightforward to enrich the operational SLD semantics into a collecting semantics which captures call patterns (i.e. how particular predicates are activated while searching for refutations), and success patterns (i.e. how call patterns are instantiated by the refutation of the involved predicate). Consequently, it is quite natural to apply a top-down approach to derive goal dependent analyses.

Falaschi *et al.* [14] introduce the s-semantics which bridges the gap between the declarative bottom-up semantics and the operational top-down semantics for logic programs. This semantics basically consists of a non-ground version of the bottom-up T_P operator. The meaning of a program is a set of possibly non-ground atoms which can later be applied to determine the answers for arbitrary initial goals. This semantics is the basis for a number of frameworks for the bottom-up analysis of logic programs [1, 4]. An analysis based on the abstraction of this semantics is naturally viewed as goal independent. It computes an abstraction of the answers to most general queries which in turn can be used to determine abstract answers to arbitrary queries.

Bottom-up computations have also been used for query evaluation in the context of deductive databases where "magic sets" and related transformation techniques are applied to make the evaluation process goal dependent. These same techniques have also been applied to enable bottom-up frameworks of abstract interpretation to support goal dependent analysis (see [4] for a list of references). In contrast, the practical application of top-down frameworks for goal independent analysis has received little attention.

This paper describes the application of a top-down framework of abstract interpretation to the goal independent analysis of logic programs. An immediate benefit is to make goal independent analyses readily available using existing top-down frameworks.

2. TOP-DOWN GOAL INDEPENDENT ANALYSIS

Falaschi *et al.* [14] illustrate that the computed answers for an arbitrary initial goal G with a program P can be obtained by solving G in the s-semantics of P. Various bottom-up frameworks of abstract interpretation for logic programs (e.g. [1, 4]) take advantage of this fact to provide for the goal independent bottom-up analysis of logic programs. It is straightforward to apply also a top-down framework to provide for such goal independent analyses. This follows from the observation [14] that the s-semantics, $[\![P]\!]$ of a program P is determined by:

$$[\![P]\!] = \left\{ p(\bar{x})\theta \;\middle|\; \begin{array}{l} p/n \in pred(P) \;\; and\; \theta\; is\; a \\ computed\; answer\; for\; p(\bar{x}) \end{array} \right\}$$

where $\bar{x}$ is an n-tuple of distinct variables and $pred(P)$ is the set of predicate symbols defined in P. An approximation of the s semantics of a program can be obtained in a top-down framework by analyzing the set of "most general" initial goal descriptions $\langle p(\bar{x}); \kappa_\epsilon \rangle$ where p/n is a predicate in P and κ_ϵ is the (most precise) description of the empty substitution, for the abstract domain at hand. The same result can be obtained with a single application of the top-down framework by adding to P the set of clauses

$$\{\, analyze \leftarrow p(\bar{x}) \,|\, p/n \in pred(P) \,\}$$

where $analyze/0 \notin pred(P)$. In this way, starting the analysis with the initial call pattern $(analyze; \kappa_\epsilon)$ there is a call pattern $(p(\bar{x}); \kappa_\epsilon)$ for every $p/n \in pred(P)$. We will refer to this transformation as the *naive* transformation and the corresponding analysis as the *naive* analysis.

The experimental results described in this paper are obtained using the top-down framework, PLAI, described in [23]. However, the proposed techniques are general and typically described in terms of source level transformations. Consequently, it is straightforward to provide similar functionalities using other top-down frameworks based on [2] such as for example those described in [19] (GAIA) and in [17] (AMAI).

The experiments described in this paper are based on three well known abstract domains: *Prop* [9, 11, 18], *Sharing* [16, 23] and *ASub* [24]. For sharing analysis, data descriptions are represented as lists of lists of variables which appear as comments in the text of the program. The information describes properties of possible substitutions when execution reaches different points in the clause. The information given after the clause head describes properties of variables after performing head unification. The information given after each subgoal describes properties of variables after executing the clause body up to and including that subgoal.

Example 1. Consider the following simple program **P**:

```
length(Y,N):- length(Y,0,N).
length([ ],N,N).
length([X|Xs],N1,N):- N2 is N1+1, length(Xs,N2,N).
```

The naive transformation adds the following clauses to **P**:

```
analyze:- length(X,Y).
analyze:- length(X,Y,Z).
```

A top-down *Sharing* analysis of the transformed program with the initial call pattern $\langle analyse; [\,] \rangle$ gives the following annotations:

```
(1)  analyze :-                 %[[X],[Y]]
           length(X,Y).         %[[X]]
(2)  analyze :-                 %[[X],[Y],[Z]]
```

```
            length(X,Y,Z).          %[[X],[Y,Z]]
(3)   length(Y,N) :-                %[[Y],[N]]
            length(Y,0,N).          %[[Y]]
(4)   length([ ],N,N).              %[[N]]
(5)   length([X|Xs],N1,N) :-        %[[N1],[N],[X],[X,Xs],[Xs],[N2]]
            N2 is N1+1,             %[[N],[X],[X,Xs],[Xs]]
            length(Xs,N2,N).        %[[X],[X,Xs],[Xs]]
```

Intuitively, each list $[v_1, \ldots, v_n]$ in an annotation represents a set of clause variables and specifies that there may be a runtime environment in which these are exactly the variables which are bound to terms containing a common variable x. If a variable v does not occur in any list, then there is no variable that may occur in the terms to which v is bound and thus those terms are definitely ground. If a variable v appears only in a singleton list, then the terms to which it is bound may contain only variables which do not appear in any other term. For example, after executing the recursive call in clause (5) the variables N, $N1$ and $N2$ are ground while X and Xs possibly share.

The analysis provides also the following information indicating the set of call and success patterns:

Atom	Call Pattern	Success Pattern
analyze	[ ]	[ ]
length(A,B,C)	[[A],[B],[C]]	[[A],[B,C]]
length(A,B)	[[A],[B]]	[[A]]
length(A,0,B)	[[A],[B]]	[[A]]
length(A,B,C)	[[A],[C]]	[[A]]

The first three rows in this table provide the goal independent information as obtained in a bottom-up analysis. The other two rows correspond to information inferred for additional call patterns which arise in the course of the analysis. For a more detailed description of the *Sharing* domain see [16] and [23]. □

Observe that the analysis described in Example 1 is inefficient in that it provides information concerning call patterns which are not required in a goal independent analysis. A more efficient goal independent analysis is obtained by transforming the program so that all of the calls in the body of a clause are "flat" and involve only fresh variables. As a consequence, any call encountered in the top-down analysis is in its most general form and corresponds to the most general call patterns required by a goal independent analysis. In the sequel this transformation is referred to as the *efficient* transformation and involves replacing each call of the form $q(\bar{t})$ in a clause body by $q(\bar{x}), \bar{x} = \bar{t}$[1] where $\bar{x}$ are fresh variables. The corresponding analysis is called the *efficient* analysis.

Example 2. Applying the efficient transformation to the program in Example 1 gives:

[1] Note however that, due to the transformation, the abstraction of built-ins such as $is/2$ has to be adapted. See the discussion at the end of Section 4.

```
analyze:- length(X,Y).        length([ ],N,N).
analyze:- length(X,Y,Z).      length([X|Xs],N1,N) :-
                                      N2 is N1+1,
length(Y,N) :-                       length(Xsa,N2a,Na),
      length(Ya,Ma,Na),               <Xsa,N2a,Na> = <Xs,N2,N>.
      <Y,0,N> = <Ya,Ma,Na>.
```

A goal independent analysis of this program eliminates the last two rows in the table of Example 1. □

This paper illustrates that the "efficient" transformation often provides a substantial speed-up over the "naive" approach. However, for some types of domains, there can be a loss of precision which can exceptionally also increase the cost of the analysis. This is discussed in Section 4.

Finally, we would like to point out the strong similarities between the efficient analysis described above, and a bottom-up analysis which traverses the clause bodies from left to right. Consider the analysis of a call $p(\bar{t})$ in some clause body under a data description κ_i. The bottom-up analyser solves the atom $p(\bar{t})$ against the abstraction of the s-semantics of the atom p/n (by analysing an equality $\bar{x} = \bar{t}$) and uses the result to update κ_i into a data description κ_j. The top-down efficient analysis solves $p(\bar{x}), \bar{x} = \bar{t}$ under a data description $\kappa_{i'}$ which differs from κ_i in expressing that $\bar{x}$ are fresh variables. In doing this, it first analyses $p(\bar{x})$ by computing an abstraction of the s-semantics of p/n (or looks it up if it has been computed before) and using this result to update the description of $\bar{x}$ in $\kappa_{i'}$. Then, it performs the analysis of $\bar{x} = \bar{t}$ which has the effect of solving the call $p(\bar{t})$ against the abstracted s-semantics of p/n and of updating the data description into a $\kappa_{j'}$. Assuming that the same abstraction of the s-semantics of p/n is used, one can expect that $\kappa_{j'}$, after projecting out the variables $\bar{x}$, is the same as κ_j.

3. REUSING GOAL INDEPENDENT INFORMATION

In this section we illustrate how the results of a goal independent analysis can be used (and reused) to derive goal dependent information. There are two issues involved: (1) using the result of the analysis to obtain abstract answers for an abstract call; and (2) using the result of the analysis to obtain an approximation of the set of call patterns which arise in the computation of a given initial call pattern. The first issue is extensively discussed in the literature, both for top-down frameworks as proposed by [16] and in the context of bottom-up frameworks as described in [1] and [4]. Basically, the abstract answers for a given call pattern are obtained by "solving" the call using the result of the goal independent phase. Also the second issue is considered in the literature. The basic concept, underlying the Magic-set transformation, is a recursive specification of the set of activated calls, for example as described in [1] and as formalized in [15]: (1) if $a_1, \ldots, a_i, \ldots, a_m$ is an initial goal then $a_i\theta$ is a call if θ is an answer for $a_1, \ldots, a_{i-1}$ (in particular a_1 is a call); and (2) if $h \leftarrow b_1, \ldots, b_i, \ldots, b_n$ is a (renamed) program clause, a is a call, $mgu(a, h) = \theta$ and φ is an answer of $(b_1, \ldots, b_{i-1})\theta$ then $b_i\theta\varphi$ is a call.

Our contribution is to perform this collection of activated call patterns efficiently from within a top-down analysis framework. Given the results of a goal independent analysis for P and an initial call pattern G, the call patterns for P and G are collected in a single pass over the program without performing any form of fixpoint iteration.

We illustrate the approach with an example.

Example 3. Consider a *Sharing* analysis of the following simple Prolog program.

```
q(0,_,_,_,_,V,V).
q(s(A),X,Y,Z,W,U,V):- q(A,Z,W,U,V,X,Y).
```

The result of the goal independent analysis is:

Atom	Call Pattern
q(A,X,Y,Z,W,U,V)	[[A],[X],[Y],[Z],[W],[U],[V]]
	Success Pattern
	[[X],[X,Y],[Y],[Z],[Z,W],[W],[U],[U,V],[V]]

This result is obtained after three iterations, for both the naive and the efficient analysis. In the first iteration, the analysis of the base clause yields sharing groups [X],[Y],[Z],[W] and [U,V]. During the analysis of the recursive clause, it is observed that the recursive call q(A,Z,W,U,V,X,Y) has the same call pattern as the original query. Thus, in order not to go into an infinite loop, the success pattern obtained so far (from the base clause) is used to estimate its success pattern. This yields the additional sharing groups [U],[V] and [X,Y]. The second iteration reanalyzes the second clause, now using the success pattern of the first iteration to handle the recursive call and finds the additional share group [Z,W]. After the third iteration no new sharing groups are found and, therefore, a fixpoint is reached.

Now consider a goal dependent analysis for a query q(A,X,Y,Z,W,U,V) with the call pattern [[X],[Y],[Z],[W],[U],[V]] (i.e., A is ground). A standard top-down analysis will exhibit the same behavior illustrated above, i.e., it will require three iterations deriving as success patterns the sharing groups [X],[Y],[Z],[W] and [U,V] for the base case, plus [U],[V] and [X,Y] for the first iteration, and [Z,W] for the second iteration. However, if the results of a goal independent analysis are available, the goal dependent analysis can be sped-up as follows. The analysis of the base case proceeds as usual, obtaining the sharing groups [X],[Y],[Z],[W] and [U,V]. During the analysis of the recursive clause it is observed that the recursive call has the same call pattern as the original query. Hence, rather than using the success pattern of the base case to proceed, the analysis can use the goal independent analysis to derive the final result by performing an abstract conjunction of the goal indpendent information ([[X],[X,Y],[Y],[Z],[Z,W],[W],[U],[U,V],[V]])[2]with the call pattern ([[X],[Y],[Z],[W],[U],[V]]). The result of the abstract conjunction ([[X],[X,Y],[Y],[Z],[Z,W],[W],[U],[U,V],[V]]) is known to be a safe data description for the program point following the recursive call. This information is propagated to the query and no iteration is required.

This does not imply that each predicate is analyzed only once. Consider the

[2]The result of the goal-independent analysis is stored as a pair call pattern-success pattern, e.g. for a binary predicate $p/2$, a pair could be [[X1][X2]] [[X1][X1,X2]]. For a normalized call, e.g. $p(A, B)$, the success pattern is simply renamed into [[A][A,B]]. For an unnormalized call, additionally an abstract unification has to be performed, e.g. for $P(f(A), B)$ the success pattern is renamed into [[X1][X1,B]] and, additionally, the unification X1=f(A) is abstracted yielding [[X1,A],[X1,A,B]].

same query, but with call pattern `[[X,Y],[Z],[W],[U],[V]]`. During the analysis of the recursive clause, the goal dependent analysis (both the standard as well as our "reuse" version) creates a new call pattern `[[X],[Y],[Z],[W],[U,V]]`. Analyzing the predicate for this pattern yields yet another call pattern, namely `[[X],[Y],[Z,W],[U],[V]]`. The analysis of the recursive clause for this third pattern creates for the recursive call the same pattern as the initial call. At this point the traditional goal dependent analysis would use the result for the base clause and start iterations for each of the nested calls created during the analysis (A quite complex process as the calls are nested, but which a system as PLAI performs in a clever way to minimize the overall work.). However, if goal independent information is available, the analysis can reuse such information yielding a safe data description that will be propagated to the rest of the calls without the need for any iteration.

4. DOMAIN DEPENDENT ISSUES

There are several domain-dependent issues which significantly affect the precision of a program analysis. The following example illustrates that a naive top-down analysis can provide a more precise analysis for some programs.

Example 4. Consider the following program:

```
naive :-                          efficient :-
   Y=f(X,_), Z=f(X,_),               Y=f(X,_), Z=f(X,_),
   q(Y,Z).                           q(U,V), <U,V> = <Y,Z>

          q(A,B) :- A = f(a,a).
          q(A,B) :- B = f(a,a).
```

where the predicates `naive/0` and `efficient/0` correspond to our two different approaches for goal independent analysis.

A top-down analysis based on the *Sharing* domain infers the groundness of `X` in `naive/0` but not in `efficient/0`. The reason is that $q(Y,Z)$ is called with pattern $[[Y],[Z],[X,Y,Z]]$. After the analysis of $q(Y,Z)$, although the *Sharing* domain cannot express that either Y or Z are ground, it definitely knows that they cannot share, and thus X must be ground. On the other hand $q(U,V)$ is called with pattern $[[U],[V],[Y],[Z],[X,Y,Z]]$. If the groundness of either U or V could be inferred after $q(U,V)$, then the groundness of X could have been inferred due to $\langle U,V\rangle = \langle Y,Z\rangle$. Unfortunately, the fact that U and V do not share after $q(U,V)$ does not imply the groundness of X, and therefore this information is lost. □

The above example illustrates that the precision of an analysis is highly dependent on the ability of the underlying abstract domain to capture information which enables a good propagation of the property being analyzed.

Jacobs and Langen [16] prove that analyzing $p(\bar{t})$ and analyzing $p(\bar{x}), \bar{x} = \bar{t}$ are guaranteed to be equally precise when they involve an abstract unification function which is *idempotent*, *commutative* and *additive*. Consequently, under these conditions, the naive and efficient goal independent analysis are equally precise as well as the standard one phase and our two phase goal dependent analysis. Idempotence implies that repeating abstract unification does not change the result. Commutativity allows abstract unification to be performed in any order. Finally, additivity guarantees that precision is not lost when performing least upper bounds. These conditions impose a restriction on the abstract domain which must support an abstract unification algorithm satisfying these properties. Marriott and Søndergaard refine the terminology introducing the notion of a *condensing domain* [21]. It is interesting to note that most of the domains used in practice are not additive, and many not even commutative or idempotent. Consequently, the answer to the question *can we benefit from goal independent analyses (top-down or bottom-up)* remains an issue for practical experimentation.

In the remainder of the paper we describe an experimental investigation involving the three well known abstract domains, *Prop*, *Sharing* and *ASub*. Note that *Prop* comes equipped with an abstract unification operation which is *idempotent*, *commutative* and *additive*; *Sharing* with an operation which is *idempotent* and *commutative*; and *ASub* with an operation which is *additive*. Our choice of domains is intended to illustrate the influence of domain properties on its ability to support precise and efficient goal independent analysis. For a comparison of these three domains see [10].

It is interesting to note that domain properties such as idempotence, commutativity and additivity have more influence on goal independent than on goal dependent analyses. This is because, operations in a goal independent analysis involve "more general" substitutions as there is no propogation of inputs from an initial goal. Consequently, accuracy can be lost in weaker domains and may also slow down analyses in domains where loss of accuracy incurs larger representations. As an example, in *ASub*, when groundness information propagates from an initial goal, the inability of the domain to capture groundness dependencies has less effect on accuracy than in a goal independent analysis. In fact we observe in [7] that the groundness information obtained with *ASub* is essentially the same as that obtained with *Sharing* in a goal dependent setting (for a rich set of benchmarks). We reason that most real Prolog programs tend to propagate groundness in a top-down manner. However, the absence of such properties becomes more relevant in goal-independent analyses, although less important in naive top-down analyses than in bottom-up or efficient top-down analyses.

Another important issue concerns the analysis of Prolog built-ins. In standard top-down analysis, the data descriptions in a program point describe the substitutions which are possible at that point during the actual execution of the program. This can be exploited in defining the abstraction of built-ins. Consider for example an abstract domain which captures definite freeness information. In a standard top-down analysis if we know that the clause `p(X,Y) :- ground(X), Y=a` is called with X a free variable then we may assume that the clause fails. This is no longer the case when performing a goal independent analysis (whether naive or efficient). Here one has to abstract the built-ins under the assumption that the substitutions which occur during the execution are not only those described by the data descriptions, but also their instances. As a free variable can have ground instances, failure cannot be assumed in the above example. However, it remains valid to claim that X is ground after executing the built-in. So, when doing a goal independent analysis, all abstractions of built-ins have to be reconsidered.

5. OBJECTIVES, EXPERIMENTS AND RESULTS

Our objective is to illustrate the relative impact of the issues discussed in the previous sections on efficiency and accuracy of goal independent analyses. We compare the standard top-down, goal dependent analysis with the alternative two phase analysis which first infers goal independent information and then reuses it to obtain goal dependent information for given initial goals. For goal independent analyses we compare the *naive* and *efficient* approaches described in Section 2. The experiments focus on the domains *ASub*, *Sharing*, and *Prop*. For *Prop* the analyzer is run on an abstract version of the program as described in [6]. The benchmark programs are the same as those used in [7][3]and they range in size from 2 clauses with 5 variables (occurrences) to 227 clauses with 869 variables. All analyses are obtained using SICStus 2.1 (native code) on a SPARC10. All times are in seconds.

	Prop			*Sharing*				*ASub*			
Name	GI^{ef}	GI^n	$Size^n$	GI^{ef}	GI^n	$Size^n$	Δ	GI^{ef}	GI^n	$Size^n$	Δ
init	0.2	3.3	2.9/7	0.9	173.5	6.7/12	0	0.2	0.4	2.7/6	0
seri	0.5	9.1	2.8/8	0.7	3.0	5.3/12	0	0.2	0.2	2.0/4	0
map	0.1	1.1	2.2/4	1.4	1.9	5.2/7	0	0.2	0.3	2.0/5	0
gram	0.1	0.1	1.4/2	0.1	0.1	3.7/5	0	0.0	0.0	0.7/1	0
brow	0.3	2.5	1.8/3	3.9	14.0	5.2/12	0	0.3	1.1	1.3/4	12
bid	0.2	1.9	1.7/3	0.5	1.4	3.8/8	0	0.3	1.0	0.8/3	5
derv	0.6	2.1	2.3/6	0.8	1.9	5.4/9	0	0.6	2.1	1.7/3	0
rdtk	0.3	1.2	1.7/4	0.7	1.5	4.8/8	0	0.7	1.0	1.3/3	33
read	2.3	93.7	3.1/26	10.6	206.0	8.4/67	4	2.1	9.3	1.3/9	4
boyr	0.7	6.3	2.6/9	3.7	7.5	6.1/35	0	0.7	1.1	2.3/11	0
peep	2.4	15.7	4.6/11	33.4	19.4	10.8/24	0	1.8	2.9	3.4/6	0
ann	1.8	69.2	2.9/10	418.1	381.8	11.0/60	6	2.9	11.5	4.2/19	3

TABLE 1. *Goal Independent* results

Table 1 presents the results of the goal independent experiments for the three domains considered. For each benchmark program (in the **Name** column) the table describes the following information:

- $\mathbf{GI}^{ef}$: time for the efficient top-down goal independent analysis.
- $\mathbf{GI}^{n}$: time for the naive top-down goal independent analysis.
- $\mathbf{Size}^{n}$: A measure of the average/maximal sizes of the results given by the naive goal independent analyses: For *Prop*, the number of disjuncts in the resulting disjunctive normal forms; for *Sharing*, the number of lists of variables in the lists of lists representations; and for *ASub*, the number of pairs of variables in the corresponding abstract substitutions.

[3]Benchmark names abbreviated as follows: init (init_susbt), seri (serialize), map (map-color), gram (grammar), brow (browse), derv (deriv), rdtk (rdtok), boyr (boyer), peep (peephole).

		GD^{reuse}			GD^{standard}		
Name	Query	Tm	LU	Size	Tm	>1	>2
init(X,Y,Z,W)	X∧Y	1.1/1.3	58	6.6/7	1.0	0	0
	X	1.2/1.4	66	6.0/7	1.2	3	1
	true	1.2/1.4	66	6.0/7	1.2	3	1
seri(X,Y)	X	5.7/6.2	161	6.9/8	13.7	74	32
	true	6.0/6.5	189	6.3/8	14.5	88	41
map(X,Y,Z,W)	X	1.0/1.1	64	3.0/4	1.5	26	11
gram(X,Y)	true	0.1/0.2	0	0.0/0	0.1	0	0
brow(X,Y)	X∧Y	1.8/2.1	123	2.2/3	2.9	49	17
	true	1.0/1.3	93	2.1/3	1.9	48	17
bid(X,Y,Z)	X∧Y∧Z	0.3/0.5	14	2.4/3	0.3	0	0
derv(X,Y,Z)	X∧Y	0.4/1.0	30	2.3/6	0.4	0	0
	X	0.4/1.0	30	2.3/6	0.4	0	0
rdtk(X,Y)	true	1.2/1.5	55	2.3/3	2.2	28	12
read(X,Y)	X	1.7/4.0	57	2.4/5	2.4	15	7
	true	17.0/19.3	241	2.7/26	135.7	753	424
boyr(X)	X	2.5/3.2	92	2.8/9	4.6	122	72
	true	2.6/3.3	89	2.8/9	4.8	120	67
peep(X,Y)	X	2.8/5.2	146	3.1/5	6.0	98	47
	true	10.2/12.6	412	3.6/5	24.7	202	104
ann(X,Y)	true	12.7/14.5	488	3.0/6	58.4	217	95

TABLE 2. *Prop* results

- Δ: the percentage of *predicates* for which the analysis using $\mathbf{GI}^{ef}$ is less accurate than that obtained by $\mathbf{GI}^{n}$.[4]

Tables 2, 3, and 4 present the results of the goal dependent experiments for the *Prop*, *Sharing*, and *Asub* domains respectively. For each benchmark program the **Name** and **Query** columns describe the program, the arguments of its top-level predicate and several initial goal patterns (for *Prop*, a propositional formula on the variables of the top-level predicate). The results for the goal dependent analyses (with look-up and standard) are given under the headings $\mathbf{GD}^{reuse}$ and $\mathbf{GD}^{standard}$. The other columns describe:

- **Tm**: the time for the respective analyses, for $\mathbf{GD}^{reuse}$, times exclusive / inclusive the time for the efficient goal independent analysis are given;
- **LU**: the number of look-ups into the goal independent phase;
- **Size**: a measure of the average/maximal sizes of the answers for the looked up queries (gives a rough idea of the complexity of the abstract unification operations involved);
- >1 and >2: the number of fixed point computations that take more than **one** and **two** iterations. These are the non-trivial computations. Note that the last iteration usually takes much less time than the others. Thus, the >2 computations are bound to be more costly than those which involve only two iterations;

[4] Only for *Sharing* and *ASub* (for *Prop* both techniques give identical results).

		GD^{reuse}			$GD^{standard}$			Δ
Name	Query	Tm	LU	Size	Tm	>1	>2	%
init(X,Y,Z,W)	[[Z],[W]]	0.2/1.1	9	8.0/10	0.2	0	0	0
	[[Y],[Z],[W]]	0.7/1.6	15	9.5/16	0.9	6	1	0
	[[X],[Y],[Z],[W]]	98.1/99.0	21	32.4/70	193.7	21	4	0
seri(X,Y)	[[Y]]	2.8/3.5	14	12.4/23	3.0	8	0	0
	[[X],[Y]]	2.9/3.6	14	12.7/23	3.1	8	0	0
	[[X],[X,Y],[Y]]	2.9/3.6	14	12.7/23	3.1	8	0	0
map(X,Y,Z,W)	[[Y],[Z],[W]]	1.5/2.9	5	7.4/10	3.1	8	0	0
gram(X,Y)	[[X],[Y]]	0.1/0.2	0	0.0/0	0.1	0	0	0
	[[X],[X,Y],[Y]]	0.1/0.2	0	0.0/0	0.1	0	0	0
brow(X,Y)	[ ]	13.6/17.5	18	5.1/10	16.4	9	0	0
	[[X],[Y]]	0.2/4.1	10	4.2/7	0.4	8	0	0
	[[X],[X,Y],[Y]]	0.2/4.1	9	4.7/7	0.3	6	0	0
bid(X,Y,Z)	[ ]	0.3/0.8	7	3.9/6	0.3	0	0	0
derv(X,Y,Z)	[[Z]]	0.9/1.7	35	3.8/7	0.9	0	0	0
	[[Y],[Z]]	0.9/1.7	35	4.2/7	0.9	0	0	0
rdtk(X,Y)	[[X],[Y]]	1.2/1.9	47	5.1/6	2.0	25	13	0
	[[X],[X,Y],[Y]]	1.2/1.9	47	5.1/6	2.0	25	13	0
read(X,Y)	[[Y]]	1.5/12.1	22	8.3/11	1.5	18	11	0
	[[X],[Y]]	66.4/77.0	73	11.5/25	257.9	270	115	0
boyer(X)	[ ]	1.7/5.4	15	7.8/14	4.0	45	19	0
	[[X]]	1.7/5.4	13	8.5/14	4.0	44	18	0
peep(X,Y)	[[Y]]	4.1/37.5	60	4.7/12	7.3	28	7	0
	[[X],[Y]]	11.1/44.5	63	6.0/12	19.8	36	10	0
ann(X,Y)	[[X],[Y]]	22.2/440.3	69	9.3/33	27.8	40	11	2.4
	[[X],[X,Y],[Y]]	22.1/440.2	69	9.4/33	27.7	39	10	2.4

TABLE 3. *Sharing* results

- Δ: the % of program *points* at which the information inferred by the $\mathbf{GD}^{reuse}$ is less accurate than that obtained by the standard $\mathbf{GD}^{standard}$ approach.[4]

6. DISCUSSION

Consider first the two alternatives for goal independent top-down analyses. Table 1 indicates that for *Prop* and *Asub*, GI^{ef} is consistently faster than GI^{n}. On the other hand, for *Sharing* there are cases where this difference is not as large, and a few in which GI^{n} is faster. To this end we note that the abstract conjunction functions for *Prop* and *Asub* are relatively simple. Hence while the cost of the additional conjunctions introduced by the *efficient* schema is relatively small, the cost of analyzing the extra call patterns introduced by the *naive* schema is avoided. For *Sharing*, this is not the case. Data descriptions can become very large in which case the abstract operations can become very time consuming. There are three reasons why the efficient schema can cause a slow-down. (1) The extra variables which are introduced can sometimes cause substantially larger data descriptions. (2) The loss of precision with respect to the naive schema can sometimes cause substantially larger data descriptions. (3) Computing the result for a more instantiated call pattern first can sometimes reduce the number of iterations needed for

Name	Query	GD^{reuse} Tm	LU	Size	$GD^{standard}$ Tm	>1	>2	Δ %
init(X,Y,Z,W)	([X,Y],[ ])	0.3/0.5	9	3.9/5	0.4	5	0	0
	([X],[ ])	0.3/0.5	12	3.3/5	0.5	8	0	0
	([ ],[ ])	0.3/0.5	9	3.9/5	0.4	6	0	0
seri(X,Y)	([X],[ ])	0.1/0.3	8	2.5/4	0.2	5	1	12.5
	([ ],[ ])	0.1/0.3	8	2.5/4	0.2	5	1	12.5
	([ ],[[X,Y]])	0.4/0.6	9	4.4/7	0.4	6	1	12.5
map(X.Y,Z,W)	([X],[ ])	0.3/0.5	6	5.0/11	0.3	2	0	0
gram(X,Y)	([ ],[ ])	0.0/0.0	0	0.0/0	0.0	0	0	0
	([ ],[[X,Y]])	0.1/0.1	0	0.0/0	0.1	0	0	0
brow(X,Y)	([ ],[ ])	0.6/0.9	18	3.1/5	0.5	7	0	71.4
	([ ],[ ])	0.1/0.4	9	0.8/2	0.2	6	0	0
	([ ],[[X,Y]])	0.6/0.9	13	3.1/5	0.7	9	0	0
bid(X,Y,Z)	([ ],[ ])	0.5/0.8	8	1.9/3	0.3	0	0	76.2
derv(X,Y,Z)	([X,Y],[ ])	3.1/3.7	72	2.7/5	0.8	0	0	91.1
	([X],[ ])	3.1/3.7	72	2.7/5	0.8	0	0	91.1
rdtk(X,Y)	([ ],[ ])	1.0/1.7	43	1.8/3	1.4	23	12	17.9
	([ ],[[X,Y]])	1.0/1.7	43	1.8/3	1.4	23	12	17.9
read(X,Y)	([X],[ ])	4.8/6.9	61	2.3/3	1.8	18	11	74.5
	([ ],[ ])	3.7/5.8	46	2.3/3	10.4	121	50	0
boyr(X)	([X],[ ])	0.8/1.5	15	2.0/3	1.4	45	19	0
	([ ],[ ])	0.8/1.5	15	2.0/3	1.4	45	19	0
peep(X,Y)	([X],[ ])	1.7/3.5	58	3.3/10	2.5	21	3	0
	([ ],[ ])	1.9/3.7	58	3.3/10	3.0	25	6	0
ann(X,Y)	([ ],[ ])	3.9/6.8	79	4.8/16	5.1	37	9	6.5
	([ ],[[X,Y]])	5.4/8.3	94	4.7/16	6.6	40	10	6.5

TABLE 4. *Asub* results

the more general call pattern, giving an overall reduction in the time needed to analyze the predicate.

Concerning precision, for *Prop* both techniques give identical results; for *Sharing*, relatively high precision is maintained by GI^{ef}, while for *Asub* there is some loss when using GI^{ef}. Given the fact that *Asub* is a weaker domain GI^{ef} presents a reasonable precision / cost compromise.

To compare the standard goal dependent analysis ($GD^{standard}$) with the two phase approach using GI^{ef} and GD^{reuse}, the accumulated cost of both phases (GI^{ef}+GD^{reuse}) must be considered. On this comparison, the results are mixed. While almost consistently favorable for *Prop*, the results are very erratic for *Sharing* and almost consistently worse for the fast *Asub* analysis. We attribute this to the fact that a very efficient fixed point is being used in $GD^{standard}$ which, by keeping track of data dependencies and incorporating several other optimizations, performs very few fixed point iterations – often none. The real advantage of a goal independent analysis is for cases when we are interested in the analysis for more than one initial query pattern.

Having performed already the goal independent phase the cost of GD^{reuse} is almost consistently faster than $GD^{standard}$, although not as much as one might expect. The advantage of GD^{reuse} over $GD^{standard}$ is proportional to the number of fixed points avoided by performing look-up's in GD^{reuse}. A measure of this can be observed from the ">1" and ">2" columns which indicate the number of "heavy" fixed point computations in the $\mathrm{GD}^{standard}$ approach. Any time the number in these columns is high the advantage of performing a two phase analysis is significant. The exception is for the weakest *Asub* domain where the loss in precision in the two phase analysis has its influence also on the cost for several of the benchmark programs.

As for precision, both techniques give identical results for *Prop* and almost identical results for *Sharing*. This is quite surprising and indicates that in practice the least upper bound operation does not cause much loss of information in the *Sharing* domain. For the *read* benchmark some information is lost by GI^{ef} however there is no loss of information with respect to $\mathrm{GD}^{standard}$ after the GD^{reuse} pass. This is due to the fact that the predicate in which loss of precision occurs are not used in the goal dependent computation for the given query patterns. Less surprising is the fact that the weaker *Asub* domain presents a more relevant loss of precision.

Overall one can say that the two-phase analysis is beneficial for domains such as *Prop* where there is no loss of precision, almost no slow down and often a substantial speed-up, in particular for programs requiring a rather high analysis time. As another example in this class, we mention the analysis aiming at detecting possible aliases between memory cells which is part of the liveness analysis of [22]. A two phase analysis for this domain is described in [3]. Substantial speed-ups are expected. For domains such as *Sharing*, the results are mixed. The lack of additivity sometimes incurs a small loss of precision. More importantly, the goal independent analysis can sometimes be expensive, due to the much larger data descriptions which can show up during a goal independent analysis. Finally, the results are negative for a domain as *Asub* which also lacks commutativity, there is a substantial loss of precision, while there is also a slow down.

In addition we can mention that the combined two phase analyses described in this paper are particularly beneficial in situations where the results of a goal independent phase are reused many times. One such case is when programs reuse their predicates in several ways and with different call patterns. However, while this does happen sometimes in typical programs, it is not frequent. A more typical example is for library modules which may be pre-analyzed to obtain goal independent information that can be stored with the module. Then, only the GD^{reuse} pass is needed to specialize that information for the particular goal pattern corresponding to the use of the library performed by the program that calls it.

Because the look-up operation in GD^{reuse} uses a safe approximation of the success pattern, the analysis computes information which is guaranteed to be a post fixpoint. It might be interesting to investigate whether narrowing of this post-fixpoint [13] allows obtaining the same precision as $\mathrm{GD}^{standard}$.

REFERENCES

1. R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, (1993).
2. M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *The Journal of Logic Programming*, 10(2):91–124, (1991).
3. M. Bruynooghe, G. Janssens and A. Kagedal. Live-structure analysis for logic programming languages with declarations. CW report nr. 231, Dept. Comp. Sc, K.U.Leuven, 1996.

4. M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Journal of Theoretical Computer Science*, 124:93–125, (1994).
5. M. Codish, García de la Banda M.,M. Bruynooghe, and M. Hermenegildo. Goal dependent vs goal independent analysis of logic programs. *Proceedings, Fifth Int'l Conf. on Logic Programming and Automated Reasoning*, *LNAI* 822:305–320, 1994.
6. M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. *The Journal of Logic Programming*, 25(3):249–274, (1995).
7. M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving abstract interpretations by combining domains. *ACM Transactions on Programming Languages and Systems*, 17(1):28–44, (1995).
8. M. Corsini, K. Musumbu, A. Rauzy, and B. Le Charlier. Efficient bottom-up abstract interpretation of Prolog by means of constraint solving over symbolic finite domains. In *Proceedings of the Fifth International Symposium on Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science, Talin, August 1993. Springer Verlag.
9. A. Cortesi, G. Filé, and W. Winsborough. *Prop* revisited: Propositional formula as abstract domain for groundness analysis. *Proceedings, Sixth IEEE Symposium on Logic in Computer Science*, pp 322–327, 1991. IEEE Press.
10. A. Cortesi, G. Filè, and W. Winsborough. Comparison of Abstract Interpretations. In M. Kuich, editor, *Proc. 19th International; Colloquium on Automata, Languages and Programming (ICALP'92)*, volume 623 of *LNCS*, pages 521–532, Wien, Austria, 1992.
11. A. Cortesi, G. Filé, and W. Winsborough. Optimal groundness analysis using propositional logic. To appear, *The Journal of Logic programming*.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the Fourth ACM Symp. on Principles of Programming Languages*, pp 238–252, 1977.
13. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 and 3):103–179, (1992).
14. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, (1989).
15. J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6:159–186, (1988).
16. D. Jacobs and A. Langen. Static analysis of logic programs for independent and parallelism. *The Journal of Logic Programming*, 13(2 and 3):291–314, (1992).
17. G. Janssens, M. Bruynoooghe, and V. Dumortier. A blueprint for an abstract machine for abstract interpretation of (constraint) logic programs. In *Proceedings of the 1995 Int'l Symposium on Logic Programming*, pp 336–350, MIT Press, 1995.
18. B. Le Charlier and P. Van Hentenryck. Groundness analysis for Prolog: Implementation and evaluation of the domain Prop. *The Journal of Logic Programming*, 23(3):237–278, (1995).
19. B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, (1994).
20. K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Washington, Seattle, August 1988.
21. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems, ACM-LOPLAS*,

2(1–4):181–196, (1993).

22. A. Mulkers, W. Winsborough and M. Bruynooghe. Live-structure data-flow analysis for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(2):205–258, (1994).
23. K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *The Journal of Logic Programming*, 13(2 and 3):315–347, (1992).
24. H. Søndergaard. An application of abstract interpretation of logic programs: Occur-check reduction. *Proceedings, ESOP 86*, LNCS 213: 327–338, Springer-Verlag, 1986.