Efficient Top-Down Set-Sharing Analysis Using Cliques

Jorge Navas[1], Francisco Bueno[2], and Manuel Hermenegildo[1,2]

[1] D. of C.S. and Electr. and Comp. Eng., U. of New Mexico, Albuquerque, NM, USA
[2] School of Computer Science, T.U. Madrid (UPM), Madrid, Spain
jorge@cs.unm.edu, herme@unm.edu
{bueno, herme}@fi.upm.es

Abstract. We study the problem of efficient, scalable set-sharing analysis of logic programs. We use the idea of representing sharing information as a pair of abstract substitutions, one of which is a worst-case sharing representation called a clique set, which was previously proposed for the case of inferring pair-sharing. We use the clique-set representation for (1) inferring actual set-sharing information, and (2) analysis within a top-down framework. In particular, we define the new abstract functions required by standard top-down analyses, both for sharing alone and also for the case of including freeness in addition to sharing. We use cliques both as an alternative representation and as widening, defining several widening operators. Our experimental evaluation supports the conclusion that, for inferring set-sharing, as it was the case for inferring pair-sharing, precision losses are limited, while useful efficiency gains are obtained. We also derive useful conclusions regarding the interactions between thresholds, precision, efficiency and cost of widening. At the limit, the clique-set representation allowed analyzing some programs that exceeded memory capacity using classical sharing representations.

1 Introduction

In static analysis of logic programs the tracking of variables shared among terms is essential. Arguably, the most accurate abstract domain defined for tracking sharing is the so called Sharing domain [JL92, MH92], which represents variable occurrences, i.e., the possible occurrences of run-time variables within the terms to which program variables will be bound. In this paper we study an alternative representation for this domain.

Example 1. Let $V = \{x, y, z\}$ be a set of variables of interest. A substitution such as $\{x/f(u_1, u_2, v_1, v_2, w), y/g(v_1, v_2, w), z/g(w, w)\}$ will be abstracted in Sharing as $\{x, xy, xyz\}$.[1] Sharing group x in the abstraction represents the occurrence of run-time variables u_1 and u_2 in the concrete substitution, xy represents v_1 and v_2, and xyz represents w. Note that the number of (occurrences of) run-time variables shared is abstracted away.

[1] To simplify notation, we denote a sharing group (a set of variables representing sharing) by the concatenation of its variables, e.g., xyz is $\{x, y, z\}$.

Sharing analysis has been used for inferring several interesting properties of programs; most notably (but not only), variable and goal independence. Several program variables are said to be independent if the terms they are bound to do not have (run-time) variables in common. Variable independence is the counterpart of sharing: program variables share when the terms they are bound to do have run-time variables in common. When we are talking of only two variables then we refer to pair-sharing, and when we track relations among more than two variables we refer to set-sharing. Sharing abstract domains are used to infer *possible* sharing, i.e., the possibility that shared variables exist, and thus, in the absence of such possibility, *definite* information about independence.

Example 2. Let $V = \{x, y, z\}$ be the variables of interest. A Sharing abstract substitution such as $\{x, y, z\}$ (which denotes the set of the singleton sets containing each variable) represents that all three variables are independent.

The Sharing domain has deserved a lot of attention in the literature in the past. It has been enhanced in several ways [Fil94, ZBH99]. It has also been extended with other kinds of information, the most relevant of which being freeness and linearity [JL92, CDFB96, HZB04], but also for example information about term structure [KS94, BCM94, MSJB95]. Its combination with other abstract domains has also been studied [CMB$^+$93, Fec96]. In particular, in [ZBH99] an alternative representation for Sharing is proposed for the non-redundant domain of [BHZ02] and this representation is thoroughly studied for inferring pair-sharing. A new component is added to abstract substitutions that represents sets of variables, the powerset of which would have been part of the original abstract substitution. Such sets are called *cliques*.

Example 3. Let V be as above. Consider the abstraction $\{x, xy, xyz, xz, y, yz, z\}$, i.e., the powerset of V (without the empty set). Such an abstraction conveys no information: there might be run-time variables shared by any pair of the three program variables, by the three of them, or not shared at all. However, abstractions such as this one are expensive to process during analysis: they penalize efficiency for no benefit at all. The clique that will convey the same information is simply the set V.

A clique is thus a compact representation for a piece of sharing which in fact does not convey any useful information. The precision and efficiency of using cliques for the case of inferring pair-sharing were reported in [ZBH99]. In [Zaf01] cliques were incorporated into the original Sharing domain, but precision and efficiency are again studied for the case of inferring pair-sharing. Here, we are interested in studying the substantially different case of inferring set-sharing. Another important difference with previous work is that we develop the analysis for a top-down framework. This requires the definition of additional and non-trivial abstract functions in the domain. Such functions were not defined in [ZBH99, Zaf01], since bottom-up analyses were used there. We use the PLAI/CiaoPP framework [HBPLG99], which is an efficient implementation of a top-down analyzer using the fixpoint algorithms and optimizations described in [MH90, MH92, HPMS00].

The rest of the paper proceeds as follows. Notation and preliminaries are presented in Section 2, together with the representation based on cliques and the clique-domains for set-sharing and set-sharing with freeness. In Section 3 the required functions for top-down analysis are defined. In Section 4 we present an algorithm for detecting cliques, in Section 5 we introduce the use of the representation based on cliques as widening, and in Section 6 our experimental evaluation of the proposed analyses. Finally, Section 7 concludes.

2 Preliminaries

Let $\wp(S)$ denote the powerset of set S, and $\wp^0(S)$ denote the *proper powerset* of set S, i.e., $\wp^0(S) = \wp(S) \setminus \{\emptyset\}$. Let also $|S|$ denote the cardinality of a set S.

Let V be a set of variables of interest; e.g., the variables of a program. A *sharing group* is a set of variables of interest, which represents the possible sharing among them (i.e., that they might be bound to terms which have a common variable). Let $SG = \wp^0(V)$ be the set of all sharing groups. A *sharing set* is a set of sharing groups. The Sharing domain is $SH = \wp(SG)$, the set of all sharing sets.

Let F and P be sets of ranked (i.e., with a given arity) functors of interest; e.g., the function symbols and the predicate symbols of a program. We will use $Term$ to denote the set of terms constructed from V and $F \cup P$. Although somehow unorthodox, this will allow us to simply write $g \in Term$ whether g is a term or a predicate atom, since all our operations apply equally well to both classes of syntactic objects. We will denote $\hat{t}$ the set of variables of $t \in Term$. For two elements $s \in Term$ and $t \in Term$, $\hat{st} = \hat{s} \cup \hat{t}$.

For two elements $s_1 \in SH$, $s_2 \in SH$, let $s_1 \uplus s_2$ be their *binary union*, i.e., the result of applying union to each pair in their Cartesian product $s_1 \times s_2$. Let also s_1^* be the *star union* of s_1, i.e., its closure under union. Given terms s and t, and $sh \in SH$, we denote by sh_t the set of sets in sh which have non-empty intersection with $\hat{t}$, the set of variables of t. By extension, in sh_{st} we use $\hat{st}$ as the set of variables. Also, $\overline{sh_t}$ is the complement of sh_t, i.e., $sh \setminus sh_t$.

Analysis of a program proceeds by abstractly solving unification equations of the form $t_1 = t_2$, $t_1 \in Term$, $t_2 \in Term$. Let $solve(t_1 = t_2)$ denote the solved form of unification equation $t_1 = t_2$. The results of analysis are abstract substitutions which approximate the concrete substitutions that may occur during execution of the program. Let U be a denumerable set of variables (e.g., the variables that may occur during execution of a program). Concrete substitutions that occur during execution are mappings from V to the set of terms constructed from $U \cup V$ and F. Abstract substitutions are sharing sets.

2.1 Clique Domains

When a sharing set $sh \in SH$ includes the proper powerset of some set C of variables, the representation can be made more compact by using C to represent the same sharing that its powerset represents in the sharing set sh [ZBH99]. A *clique* is, thus, a set of variables of interest, much the same as a sharing group, but a clique C represents all the sharing groups in $\wp^0(C)$. For a clique C, we will use $\downarrow C = \wp^0(C)$. Note that $\downarrow C$ denotes all the sharing that is implicitly represented in a clique C. A *clique set* is a set of cliques. Let $CL = SH$ denote the set of all clique sets. For a clique set $cl \in CL$ we define $\Downarrow cl = \cup\{\downarrow C \mid C \in cl\}$. Note that $\Downarrow cl$ denotes all the sharing that is implicitly represented in a clique set cl. For a pair (cl, sh) of a clique set cl and a sharing set sh, the sharing that the pair represents is $\Downarrow cl \cup sh$.

The Clique-Sharing domain is $SH^W = \{(cl, sh) \mid cl \in CL, sh \in SH\}$, i.e., the set of pairs of a clique set and a sharing set [ZBH99]. An abstract unification operation $amgu^W$ is defined in [Zaf01] which uses a function $\overline{rel} : \wp(V) \times CL \longrightarrow CL$ (complement of rel), defined as:

$$\overline{rel}(S, cl) = \{\ C \setminus S \mid C \in cl\ \} \setminus \{\emptyset\}$$

which approximates the sharing not related to variables in S. We have used an equivalent definition of $amgu^W$ to the one in [Zaf01] (see [BNH05]).

Freeness can be introduced to the Clique-Sharing domain in the usual way [MH91], by including a component which tracks the variables which are known to be free. The Clique-Shfr domain is thus $SHF^W = SH^W \times V$. A method to define an abstract unification function for SH^W with freeness and linearity is outlined in [Zaf01]. We have used an abstract unification operation $amgu^{sf}$ for SH^W with freeness which is a simplification of the corresponding operation which results from the application of such method.

3 Abstract Functions Required by Top-Down Analysis

In top-down frameworks, the analysis of a clause $Head$:-$Body$ proceeds as follows. There is a goal $Goal$ for the predicate of $Head$, which is called in a context represented by abstract substitution $Call$ on a set of variables (distinct from $\hat{Head} \cup \hat{Body}$) which contains those of $Goal$. Then the success of $Goal$ by executing the above clause is represented by abstract substitution $Succ$ given by:

$$\begin{aligned}
Succ &= extend(Call, Goal, Prime)\\
Prime &= exit2succ(project(Head, Exit), Goal, Head)\\
Exit &= entry2exit(Body, Entry)\\
Entry &= augment(F, call2entry(Proj, Goal, Head))\\
Proj &= project(Goal, Call)
\end{aligned}$$

where F is any term with the variables $\hat{Body} \setminus \hat{Head}$. Function $project$ approximates the projection of a substitution on the variables of a given term. Function $augment$ extends the domain of an abstract substitution to the variables of a given term, which are assumed to be new fresh variables. Function $entry2exit$ is given by the framework, and basically traverses the body of a clause, analyzing each atom in turn. The other three domain-dependent abstract functions which are essential are:

- *call2entry*$(Proj, Goal, Head)$ yields a substitution on the variables of $Head$ which represents the effects of unification $Goal = Head$ in a context represented by substitution $Proj$ on the variables of $Goal$.
- *exit2succ*$(Exit', Goal, Head)$ yields a substitution on the variables of $Goal$ which represents the effects of unification $Goal = Head$ in a context represented by substitution $Exit'$ on the variables of $Head$.
- *extend*$(Call, Goal, Prime)$ yields a substitution for the success of $Goal$ when it is called in a context represented by substitution $Call$ on a set of variables which contains those of $Goal$, given that in such context the success of $Goal$ is already represented by substitution $Prime$ on the variables of $Goal$. The domain of the resulting substitution is the same as the domain of $Call$.

In fact, the first two can be defined from the abstract unification operation *amgu*. The third one, however, is specific to the top-down framework and needs to be defined specifically for a given domain.

Given an operation $amgu(x = t, ASub)$ of abstract unification for equation $x = t$, $x \in V, t \in Term$, and $ASub$ an abstract substitution (the domain of which contains variables $\hat{t} \cup \{x\}$), abstract unification for equation $t_1 = t_2$, $t_1 \in Term$, $t_2 \in Term$, is given by:

$$unify(ASub, t_1, t_2) = project(t_1, Amgu(solve(t_1 = t_2), augment(t_1, ASub)))$$

$$Amgu(Eq, ASub) = \begin{cases} ASub & \text{if } Eq = \emptyset \\ Amgu(Eq', amgu(x = t, ASub)) & \text{if } Eq = Eq' \cup \{x = t\} \end{cases}$$

Functions *call2entry* and *exit2succ* can defined as follows:

$$\begin{aligned} call2entry(ASub, Goal, Head) &= unify(ASub, Head, Goal) \\ exit2succ(ASub, Goal, Head) &= unify(ASub, Goal, Head) \end{aligned}$$

However, *extend*, together with *project*, *augment*, and *amgu* are all domain-dependent. In the Sharing domain, *extend* [MH92], *project*, and *augment* are defined as follows:

$$\begin{aligned} &extend(Call, g, Prime) = \overline{Call_g} \cup \{\ s \mid s \in Call^*_g,\ \ (s \cap \hat{g}) \in Prime\ \} \\ &project(g, sh) = \{s \cap \hat{g} \mid s \in sh\} \setminus \{\emptyset\} \\ &augment(g, sh) = sh \cup \{\{x\} \mid x \in \hat{g}\} \end{aligned}$$

In the Shfr domain, these functions are defined as follows [MH91]:

$$\begin{aligned} &project^f(g, (sh, f)) = (project(g, sh), f \cap \hat{g}) \\ &augment^f(g, (sh, f)) = (augment(g, sh), f \cup \hat{g}) \\ &extend^f((sh_1, f_1), g, (sh_2, f_2)) = (extend(sh_1, g, sh_2), f') \\ &f' = f_2 \cup \{x \mid x \in (f_1 \setminus \hat{g}), ((\cup sh'_x) \cap \hat{g}) \subseteq f_2\} \end{aligned}$$

3.1 Abstract Functions for Top-Down Analysis in the Clique-Domains

Functions *call2entry* and *exit2succ* have usually been defined in a way which is specific to the domain and for top-down analysis (see, e.g., [MH92] for a definition for set-sharing). We have chosen instead to present here a formalization of a way to use the *amgu* in top-down frameworks. Thus, the definitions of *call2entry* and *exit2succ* based on *amgu* given above. Our intuition in doing this is that the results should be (more) comparable to goal-dependent bottom-up analyses, where *amgu* is used directly.

Note, however, that such definitions imply a possible loss of precision. Using *amgu* in the way explained above does not allow to take advantage of the fact that all variables in the head of the clause being entered during analysis are free. Alternative definitions of *call2entry* can be obtained that improve precision from this observation.[2] The overall effect would be equivalent to using the *amgu* function for the Sharing domain coupled with freeness, with the head variables as free variables, and then throwing out the freeness component of the result. For example, for the Clique-Sharing domain a function $call2entry^s$ that takes advantage of freeness information can be defined as follows, where $unify^{sf}$ is the version of $unify$ that uses $amgu^{sf}$:

$$\begin{array}{ll} call2entry^s(ASub, Goal, Head) = ASub' & \\ \text{where} \qquad (ASub', Free) = unify^f((ASub, \emptyset), Head, Goal) & \end{array}$$

However, for the reasons mentioned above, we have used the definitions of *call2entry* and *exit2succ* based on *amgu*. The rest of the top-down functions are defined below. For the Clique-Sharing domain, let $g \in Term$, and $(cl, sh) \in SH^W$. Functions $project^s$ and $augment^s$ are defined as follows:

$$\begin{array}{l} project^s(g, (cl, sh)) = (project(g, cl), project(g, sh)) \\ augment^s(g, (cl, sh)) = (cl, augment(g, sh)) \end{array}$$

Function $extend^s(Call, g, Prime)$ is defined as follows. Let $Call = (cl_1, sh_1)$ and $Prime = (cl_2, sh_2)$. Let *normalize* be a function which normalizes a pair (cl, sh) so that no powersets occur in sh (all are "transferred" to cliques in cl; Section 4 presents a possible implementation of such a function). Let *Prime* be already normalized, and:

$$(cl', sh') = normalize((cl_{1_g}^* \cup (cl_{1_g}^* \otimes sh_{1_g}^*), sh_{1_g}^*))$$

The following two functions lift the classical *extend* [MH92] respectively to the cases of the two clique sets (clique groups of the *Call* allowed by the clique component of the *Prime*) and of the two sharing sets (sharing groups belonging to the *Call* allowed by the sharing part of the *Prime*):

$$\begin{array}{l} extsh(sh_1, g, sh_2) = \overline{sh_{1_g}} \cup \{\ s \mid s \in sh',\ (s \cap \hat{g}) \in sh_2\ \} \\ extcl(cl_1, g, cl_2) = \overline{rel}(\hat{g}, cl_1) \cup \{\ (s' \cap s) \cup (s' \setminus \hat{g}) \mid s' \in cl',\ s \in cl_2\ \} \end{array}$$

The following two functions account respectively for the sharing sets belonging to the clique component of the *Call* allowed by the sharing part of the *Prime*,

[2] For example, one such definition (developed independently) can be found in [AS05].

and the sharing sets of the sharing component of the $Call$ allowed by the clique part of the $Prime$:

$$\begin{array}{l} clsh(cl', g, sh_2) = \{\ s \mid s \subseteq c \in cl',\ (s \cap \hat{g}) \in sh_2\ \} \\ shcl(sh', g, cl_2) = \{\ s \mid s \in sh',\ (s \cap \hat{g}) \subseteq c \in cl_2\ \} \end{array}$$

The *extend* function for the Clique-Sharing domain is thus:

$$\begin{array}{l} extend^s((cl_1, sh_1), g, (cl_2, sh_2))\ = \\ (\ extcl(cl_1, g, cl_2) \\ ,\ extsh(sh_1, g, sh_2) \cup clsh(cl', g, sh_2) \cup shcl(sh', g, cl_2)\ ) \end{array}$$

Example 4. Let $Call = (cl_1, sh_1) = (\{xyz\}, \{u, v\})$, $Prime = (cl_2, sh_2) = (\{x\}, \{uv\})$, and $\hat{g} = \{x, u, v\}$. Then we have $(cl', sh') = (\{xyzuv\}, \emptyset)$. The $extend^s$ function is computed as follows:

$$\begin{array}{l} extsh(sh_1, g, sh_2) = extsh(\{u, v\}, g, \{uv\}) = \emptyset \\ extcl(cl_1, g, cl_2) = extcl(\{xyz\}, g, \{x\}) = \{xyz, yz\} \\ clsh(cl', g, sh_2) = clsh(\{xyzuv\}, g, \{uv\}) = \{yzuv, yuv, zuv, uv\} \\ shcl(sh', g, cl_2) = shcl(\emptyset, g, \{x\}) = \emptyset \end{array}$$

Thus, $extend^s(Call, g, Prime) = (\{xyz, yz\}, \{yzuv, yuv, zuv, uv\})$, which after regularization yields $(\{xyz\}, \{yzuv, yuv, zuv, uv\})$.

Note how the result is less precise than the exact result $(\{xyz\}, \{uv\})$. This is due to overestimation of sharing implied by the cliques; in particular, for the case of *extend*, overestimations stem mainly from the necessary worst-case assumption given by (cl', sh'), which is then "pruned" as much as possible by the functions defined above. The resulting operation, however, is correct: the sharing implied by $extend^s$ on two abstract substitutions $Call$ and $Prime$ is an over-approximation of that given by *extend* on the sharing set substitutions corresponding to $Call$ and $Prime$.

Theorem 1. *Let $Call \in SH^W$, $Prime \in SH^W$, and $g \in Term$, such that the conditions for the extend function are satisfied. Let $Call = (cl_1, sh_1)$, $Prime = (cl_2, sh_2)$, and $extend^s(Call, g, Prime) = (cl', sh')$. Then*

$$(\ \Downarrow cl' \cup sh') \supseteq extend(\ \Downarrow cl_1 \cup sh_1, g,\ \Downarrow cl_2 \cup sh_2)\ .$$

For the Clique-Shfr domain, let $g \in Term$, and $s \in SHF^W$, $s = ((cl, sh), f)$. Functions $project^{sf}$ and $augment^{sf}$ are defined as follows:

$$\begin{array}{l} project^{sf}(g, s) = (project^s(g, (cl, sh)), f \cap \hat{g}) \\ augment^{sf}(g, s) = (augment^s(g, (cl, sh)), f \cup \hat{g}) \end{array}$$

Function $extend^{sf}(Call, g, Prime)$ is defined as follows. Let $Call = ((cl_1, sh_1), f_1)$ and $Prime = ((cl_2, sh_2), f_2)$, $extend^{sf}(Call, g, Prime) = ((cl', sh'), f')$, where:

$$\begin{array}{l} (cl', sh') = extend^s((cl_1, sh_1), g, (cl_2, sh_2)) \\ f' = f_2 \cup \{x \mid x \in (f_1 \setminus \hat{g}),\ ((\cup(sh'_x \cup cl'_x)) \cap \hat{g}) \subseteq f_2\} \end{array}$$

Operation $extend^{sf}$ is correct: it gives safe approximations. The resulting sharing it implies when applied on two abstract substitutions $Call$ and $Prime$ is no less than that given by $extend^{f}$ on the sharing set substitutions corresponding to $Call$ and $Prime$; and the freeness is no more than what $extend^{f}$ would have computed.

Theorem 2. *Let $Call \in SHF^{W}$, $Prime \in SHF^{W}$, and $g \in Term$, such that the conditions for the extend function are satisfied. Let $Call = ((cl_1, sh_1), f_1)$, $Prime = ((cl_2, sh_2), f_2)$, and $extend^{sf}(Call, g, Prime) = ((cl', sh'), f')$. Let also $s_1 = \Downarrow cl_1 \cup sh_1$, $s_2 = \Downarrow cl_2 \cup sh_2$, and $extend^{f}((s_1, f_1), g, (s_2, f_2)) = (sh, f)$. Then $(\Downarrow cl' \cup sh') \supseteq sh$ and $f' \subseteq f$.*

4 Detecting Cliques

Obviously, to minimize the representation in SH^{W} it pays off to replace any set S of sharing groups which is the proper powerset of some set of variables C by including C as a clique. Once this is done, the set S can be eliminated from the sharing set, since the presence of C in the clique set makes S redundant. This is the normalization mentioned in Section 3.1 when defining *extend* for the Clique-Sharing domain, and denoted there by a *normalize* function. In this section we present an algorithm for such a normalization.

Given an element $(cl, sh) \in SH^{W}$, sharing groups might occur in sh which are already implicit in cl. Such groups are redundant with respect to the sharing represented by the pair. We say that an element $(cl, sh) \in SH^{W}$ is *minimal* if $\Downarrow cl \cap sh = \emptyset$. An algorithm for minimization is straightforward: it should delete from sh all sharing groups which are a subset of an existing clique in cl. But normalization goes a step further by "moving sharing" from the sharing set of a pair to the clique set, thus forcing redundancy of some sharing groups (which can therefore be eliminated).

While normalizing, it turns out that powersets may exist which can be obtained from sharing groups in the sharing set plus sharing groups implied by existing cliques in the clique set. The representation can be minimized further if

```
1. Let $n = |sh|$; if $n < 3$, stop.
2. Compute the maximum $m$ such that $n \geq 2^m - 1$.
3. Let $i = m$.
4. If $i = 1$, stop.
5. Let $C = \{s \mid s \in sh, |s| = i\}$.
6. If $C = \emptyset$ then decrement $i$ and go to 4.
7. Take $S \in C$ and delete it from $C$.
8. Let $SS = \{s \mid s \in sh, s \subseteq S\}$.
9. Compute $[S]$.
10. If $|SS| = 2^i - 1 - [S]$ then:
    (a) Add $S$ to $cl$ (regularize $cl$).
    (b) Subtract $SS$ from $sh$.
11. Go to 6.
```

Fig. 1. Algorithm for detecting cliques

such sharing groups are also "transferred" to the clique set by adding the adequate clique. We say that an element $(cl, sh) \in SH^W$ is *normalized* if whenever there is an $s \subseteq (\Downarrow cl \cup sh)$ such that $s = \downarrow c$ for some set c then $s \cap sh = \emptyset$.

Our normalization algorithm is presented in Figure 1. It starts with an element $(cl, sh) \in SH^W$, which is already minimal, and obtains an equivalent element (w.r.t. the sharing represented) which is normalized and minimal. First, the number m is computed, which is the length of the longest possible clique. Then the sharing set sh is traversed to obtain candidate cliques of the greatest possible length i (which starts in m and is iteratively decremented). Existing subsets of a candidate clique S of length i are extracted from sh. If there are $2^i - 1 - [S]$ subsets of S in sh then S is a clique: it is added to cl and its subsets deleted from sh. Note that the test is performed on the number of existing subsets, and requires the computation of a number $[S]$, which is crucial for the correctness of the test.

The number $[S]$ stands for the number of subsets of S which may not appear in sh because they are already represented in cl (i.e., they are already subsets of an existing clique). In order to correctly compute this number it is essential that the input to the algorithm be already minimal; otherwise, redundant sharing groups might bias the calculation: the formula below may count as not present in sh a (redundant) group which is in fact present. The computation of $[S]$ is as follows. Let $I = \{S \cap C \mid C \in cl\} \setminus \{\emptyset\}$ and $A_i = \{\cap A \mid A \subseteq I,\ |A| = i\}$. Then:

$$[S] = \sum_{1 \le i \le |I|} (-1)^{i-1} \sum_{A \in A_i} (2^{|A|} - 1)$$

Note that the representation can be minimized further by eliminating cliques which are redundant with other cliques. This is the regularization mentioned in step 10 of the algorithm. We say that a clique set cl is *regular* if there are no two cliques $c_1 \in cl$, $c_2 \in cl$, such that $c_1 \subset c_2$. This can be tested while adding cliques in step 10 above.

Finally, there is a chance for further minimization by considering as cliques candidate sets of variables such that not all of their subsets exist in the given element of SH^W. Note that the algorithm preserves precision, since the sharing represented by the element of SH^W input to the algorithm is the same as that represented by the element which is output. However, we could set up a threshold for the number of subsets of the candidate clique that need be detected, and in this case the output element may in general represent more sharing. This might in fact be useful in practice in order to use the normalization algorithm as a widening operation. Note that, although the complexity of this algorithm is exponential since it is actually the problem of solving all the maximal cliques of an undirected graph (NP-complete), it is not a practical problem due to the small size of these graphs.

5 Widening Set-Sharing

A *widen* function for SH^W is based on a widening operator $\bigtriangledown : SH^W \to SH^W$, which must guarantee that for each $clsh \in SH^W$, $\bigtriangledown clsh \supseteq clsh$. The following theorem is necessary to establish the correctness of the widenings used:

Theorem 3. *Let $clsh \in SH^W$ and equation $x = t$, $x \in V$, $t \in Term$, we have*

$$amgu^s(\bigtriangledown clsh, x = t) \supseteq amgu^s(clsh, x = t)$$

For our experiments we start defining two widenings. The first of them, by [Fec96], is of an intermediate precision and it is as follows:

$$\bigtriangledown^F(cl, sh) = (cl \cup sh, \emptyset)$$

The second widening was defined in [ZBH99] as a cautious widening (because it did not introduce new sharing sets, although obviously information was lost as soon as the operations for the Clique-Sharing domain were used) and the idea was to define an undirected graph from an element $clsh \in SH^W$ and compute the maximal cliques of that graph:

$$\bigtriangledown^G(cl, sh) = (\{C_1, \ldots, C_k\}, sh)$$

where $C_1, \ldots, C_k$ are all the maximal cliques of the induced graph from (cl, sh). For the experimental evaluation in [ZBH99] a version of this cautious widening $\bigtriangledown^g$ was used which is equivalent to the previous one but disregarding the singletons. It is easy to see that our normalization process is totally equivalent to the computation of the maximal cliques of a graph and thus we will use the normalization process as a cautious widening $\bigtriangledown^N$. In the same way as [ZBH99], we use a more precise version of $\bigtriangledown^N$ which is based on disregarding the singletons called $\bigtriangledown^n$.

Since cliques should only be used when it is strictly necessary to keep the analysis from running out of memory, its application is guarded by a condition. We use the simplest possible condition based on cardinality of the sets in SH^W, imposing a threshold n on cardinality which triggers the widening. We have tuned the threshold in order to be able to achieve a reasonable trade-off between the objective of triggering widening only when strictly required and preventing running out of memory in all cases. For each widening, the triggering condition is defined as follows:

$$widen(cl, sh) = \begin{cases} \bigtriangledown(cl, sh) & \text{if } (\sum_{s \in sh} |s|) > n \\ (cl, sh) & \text{otherwise} \end{cases}$$

6 Experimental Results

We have measured experimentally the relative efficiency and precision obtained with the inclusion of cliques both as an alternative representation in the Sharing and Shfr domains and as a widening in the Shfr domain. Our first objective is to study the implications of the change in representation for analysis: although the introduction of cliques does not by itself imply a loss of precision, the abstract operations for cliques are not precise. We first want to measure such loss in practice. Second, to minimize precision loss, the clique representation should ideally be used only whenever necessary, i.e., when the classical representation cannot deal with the analysis of the program at hand. In this case, we will be using the clique representation as a widening to guarantee (smooth) termination of the analysis, i.e., that analysis does not abort because of running out of memory. It turns out that this is not a trivial task: it is not easy to determine beforehand when analysis will need more memory than is available.

Benchmarks are divided into three groups. Because of space limitations, for each group we only show a reduced number of the benchmarks actually used: those which are more representative. The first group, append (app in the tables) through serialize (serial), is a set of simple programs, used as a testbed for an analysis: they have only direct recursion and make a straightforward use of unification (basically, for input/output of arguments i.e., they are moded). The second group, aiakl through zebra, are more involved: they make use of mutual recursion and of elaborate aliasing between arguments to some extent; some of them are parts of "real" programs (aiakl is part of an analyzer of the AKL language; prolog_read (plread) and rdtok are Prolog parsers). The benchmarks in the third group are all (parts of) "real" programs: ann is the &-prolog parallelizer, peephole (peep) is the peephole optimizer of the SB-Prolog compiler, qplan is the core of the Chat-80 application, and witt is a conceptual clustering application.

Our results are shown in Tables 1 and 2. Columns labeled **T** show analysis times in milliseconds, on a medium-loaded Pentium IV Xeon 2.0Ghz with two processors, 4Gb of RAM memory, running Fedora Core 2.0, and averaging several runs after eliminating the best and worst values. Ciao version 1.11#326 and CiaoPP 1.0#2292 were used. Columns labeled **P** (precision) show the number of sharing groups in the information inferred and, between parenthesis, the number of sharing groups for the worst-case sharing. Columns labeled **#W** show the number of widenings performed and columns labeled **#C** show the number of clique groups. Since our top-down framework infers information at all program points (before and after calling each clause body atom), and also several variants for each program point, it is not trivial to provide a good absolute measure of precision: changes in precision may cause more variants during analysis, which in turn affect the precision measure. Instead, we have chosen to provide the accumulated number of sharing groups in all variants for all program points, in order to be able to compare results in different situations.

6.1 Cliques as Alternative Representation

Table 1 shows the results for Sharing, Clique-Sharing, Shfr, and Clique-Shfr, for the cases in which cliques are used as an alternative representation.

In order to understand the results it is important to note an existing synergy between normalization, efficiency, and precision when cliques are used as an alternative representation. If normalization causes no change in the sharing representation (i.e., sharing groups are not moved to cliques), usually because powersets do not really occur during analysis, then the clique part is empty. Analysis is the same as without cliques, but with the extra overhead due to the use of the normalization process. Then precision is the same but the time spent in analyzing the program is a little longer. This also occurs often if the use of normalization is kept to a minimum: only for correctness (in our implementation, normalization is required for correctness at least for the *extend* function and other functions used for comparing abstract substitutions). This should not be surprising, since the fact that powersets occur during analysis at a given time does not necessarily mean that they keep on occurring afterward: they can disappear because of groundness or other precision improvements during subsequent analysis (of, e.g., builtins).

Table 1. Precision and Time-efficiency

	Sh		SH^W			**Shfr**		SH^W **fr**		
	T	**P**	**T**	**P**	$\#C$	**T**	**P**	**T**	**P**	$\#C$
app	11	29 (60)	8	44 (60)	4	6	7 (30)	6	7 (30)	0
deriv	35	27 (546)	27	27 (546)	0	27	21 (546)	27	21 (546)	0
mmat	13	14 (694)	11	14 (694)	0	9	12 (694)	11	12 (694)	0
qsort	24	30 (1716)	25	30 (1716)	0	25	30 (1716)	27	30 (1716)	0
query	11	35 (501)	13	35 (501)	5	12	22 (501)	14	22 (501)	0
serial	306	1734 (10531)	90	2443 (10531)	88	61	545 (5264)	55	736 (5264)	41
aiakl	35	145 (13238)	42	145 (13238)	0	37	145 (13238)	43	145 (13238)	0
boyer	369	1688 (4631)	267	1997 (4631)	158	373	1739 (5036)	278	2074 (5036)	163
brow	30	69 (776)	29	69 (776)	0	29	69 (776)	31	69 (776)	0
plread	400	1080 (408755)	465	1080 (408755)	10	425	1050 (408634)	481	1050 (408634)	0
rdtok	325	1350 (11513)	344	1391 (11513)	182	335	1047 (11513)	357	1053 (11513)	2
wplan	3261	8207 (42089)	1430	8191 (26857)	420	1320	3068 (23501)	1264	5705 (25345)	209
zebra	25	280 ($67 \cdot 10^7$)	34	280 ($67 \cdot 10^7$)	0	41	280 ($67 \cdot 10^7$)	42	280 ($67 \cdot 10^7$)	0
ann	2382	10000 ($31 \cdot 10^4$)	802	19544 ($31 \cdot 10^4$)	700	1791	7811 ($40 \cdot 10^4$)	968	14108 ($39 \cdot 10^4$)	510
peep	831	2210 (12148)	435	2920 (12118)	171	508	1475 (9941)	403	2825 (12410)	135
qplan	-	-	860	$42 \cdot 10^4$ ($38 \cdot 10^5$)	747	-	-	2181	$23 \cdot 10^4$ ($31 \cdot 10^5$)	529
witt	405	858 ($45 \cdot 10^5$)	437	858 ($45 \cdot 10^5$)	25	484	813 ($45 \cdot 10^5$)	451	813 ($45 \cdot 10^5$)	0

When the normalization process is used more often (like for example at every call to *call2entry* as we have done), then sharing groups are moved more often to cliques. Thus, the use of the operations that compute on clique sets produces efficiency gains, and also precision losses, as it was expected. However, precision losses are not high. Finally, if normalization is used too often, then the analysis process suffers from heavy overhead, causing too high penalty in efficiency that it makes the analysis intractable. Therefore it is very clear that a thorough tuning of the use of the normalization process is crucial to lead analysis to good results in terms of both precision and efficiency.

As usual in top-down analysis, the *extend* function plays a crucial role. In our case, this function is a very important bottleneck for the use of normalization. As we have said, we use the normalization for correctness at the beginning of the *extend* function. Additionally, it would be convenient to use it also at the end of such function, since the number of sharing groups can grow too much. However, this is not possible in practice due to the *clsh* function, which can generate so many sharing groups that, at the limit, the normalization process itself cannot run. Alternative definitions of $clsh$ have been studied, but because of the precision losses incurred, they have been found impractical.

Table 1 shows that there are always programs whose analysis of which does not produce cliques. This occurs in some of the benchmarks (like all of the first group but serialize and some of the second one such as aiakl, browse (brow), prolog_read, and zebra). In this case, precision is maintained as expected but there is a small loss of efficiency due to the extra overhead discussed above. The same thing happens with benchmarks which produce cliques (append, query, prolog_read, and witt, in the case of Sharing without freeness), but this does not affect precision.

On the other hand, for those benchmarks which do generate cliques (like serialize, boyer, warplan (wplan), ann, and peephole) the gain in efficiency is considerable at the cost of a small precision loss. As usual, efficiency and precision correlate inversely: if precision increases then efficiency decreases and vice versa. A special case is, to some extent, that of rdtok, since precision losses are not coupled with efficiency gains. The reason is that for this benchmark there are extra success substitutions (which do not convey extra precision and, in fact, the result is less precise) that make the analysis times larger.

In general, the same effects are maintained with the addition of freeness, although the efficiency gains are lower whereas the precision gains are a little higher. The reason is that the $amgu^{sf}$ function is less efficient than $amgu^{s}$ (but more precise). Overall, however, the trade-off between precision and efficiency is beneficial. Moreover, the more compact representation of the clique domain makes possible to analyze benchmarks (e.g., qplan) which ran out of memory with the standard representation.

6.2 Widening Set-Sharing Via Cliques

As mentioned before, the intention of the widening operator is to limit the use of cliques only to the cases where it is necessary in order to avoid analysis running out of memory. This is not a trivial task, as explained below. Table 2 shows results from our experiments for Shfr, Clique-Shfr using widening. The widenings have been applied before each abstract unification and at the end of the *extend* function, and they are guarded by the condition discussed in Section 5.

The choice of a suitable value of the threshold is a key issue, since this threshold is responsible for triggering widening only for the cases where it is needed. In a top-down framework the choice of threshold is further complicated by the *extend* function. As commented above, this function and, in particular, the $clsh$ function defined in Section 3.1 can make the number of sharing groups grow excessively after every call, since that function generates powersets of the given cliques. In order to solve this problem we studied two different alternatives.

First, we tried a more efficient version of the $clsh$ function, which moved some extra sharing groups to cliques. This, however, resulted in excessive precision losses which reduced the usefulness of the analysis. Given this, we also developed a hybrid approach for the case of $\bigtriangledown^{n}$, where $\bigtriangledown^{n}$ is used in unifications but the more aggressive $\bigtriangledown^{F}$ is used after calling $clsh$. We call this version $\bigtriangledown^{nF}$.

As for practical thresholds, we have concluded experimentally that an appropriate value for the guard for the widenings in our test platform is 250. This is the highest value that prevents analysis from running out of memory. However, as we will see, it also triggers widening for a few cases where it is not needed. For the additional threshold used in the ∇^{nF} operations (Section 4) we have determined that 40% is an appropriate value since, although low, it gives surprisingly good results. The results in Table 2 thus correspond to ∇^{F}_{250} and ∇^{nF}_{250-40}.

Table 2. Precision and Time-efficiency with Widening

	Shfr		SH^W**fr**$+\nabla^{F}_{250}$			SH^W**fr**$+\nabla^{nF}_{250-40}$		
	T	**P**	**T**	**P**	$\#W$	**T**	**P**	$\#W$
app	6	7 (30)	11	7 (30)	0	10	7 (30)	0
deriv	27	21 (546)	48	21 (546)	0	35	21 (546)	0
mmat	9	12 (694)	16	12 (694)	0	16	12 (694)	0
qsort	25	30 (1716)	40	30 (1716)	0	43	30 (1716)	0
query	12	22 (501)	23	22 (501)	0	25	22 (501)	0
serial	61	545 (5264)	74	722 (5264)	6	70	703 (5264)	10
aiakl	37	145 (13238)	63	145 (13238)	6	61	145 (13238)	33
boyer	373	1739 (5036)	561	1744 (5036)	2	536	1743 (5036)	4
brow	29	69 (776)	44	69 (776)	0	42	69 (776)	0
plread	425	1050 (408634)	3419	24856 (1754310)	198	593	1050 (408634)	103
rdtok	335	1047 (11513)	472	1047 (11513)	0	466	1047 (11513)	0
wplan	1320	3068 (23501)	1878	5376 (21586)	42	1394	5121 (20894)	60
zebra	41	280 ($67{\cdot}10^7$)	42	280 ($67{\cdot}10^7$)	1	56	280 ($67{\cdot}10^7$)	48
ann	1791	7811 (401220)	751	16122 (394800)	17	726	16122 (394800)	34
peep	508	1475 (9941)	453	2827 (12410)	8	512	2815 (12410)	16
qplan	-	-	1722	238426 (3141556)	26	1897	233070 (3126973)	55
witt	484	813 (4545594)	2333	259366 (23378597)	110	736	813 (4545594)	140

As expected, the use of widening allows executing programs which the Shfr domain could not due to exceeded memory capacity. However, as mentioned in the discussion of the threshold, we do also widen for some benchmarks which the original domain could handle. Fortunately, the precision losses are limited.

Widening operator ∇^{nF}_{250-40} results at least as precise as ∇^{F}_{250} and, for most of the cases, better. In fact, the results obtained for prolog_read and witt using ∇^{F}_{250} are remarkable since the information obtained is very poor.

The difference in time efficiency between ∇^{F}_{250} and ∇^{nF}_{250-40} is acceptable, and in fact for some programs ∇^{nF}_{250-40} is more efficient than ∇^{F}_{250}. Note that for prolog_read and witt the difference is considerable in favor of ∇^{nF}_{250-40}. There appears to be a clear correspondence between number of widenings and efficiency gains. This holds even if the widening operations are expensive, such as with ∇^{nF}_{250-40}, because the widening expense is offset by efficiency gains in the abstract operations due to the reduction in the size of the abstract substitutions being processed.

7 Conclusions

We have studied the problem of efficient, scalable set-sharing analysis of logic programs using cliques both as alternative representation and as widenings. We have concentrated on the previously unexplored case of inferring set-sharing information in the context of top-down analyses. To this end, we have proposed all the operations required for top-down analyses for the cases of combining cliques with both Sharing and Sharing+Freeness. We have also proposed and studied several widenings, providing different levels of precision and efficiency tradeoff.

Our experimental evaluation supports the conclusion that, for inferring set-sharing, the use of cliques as an alternative representation results in limited precision losses (due to normalizations) while useful efficiency gains are obtained. We have also derives useful conclusions regarding the interactions between thresholds, precision, efficiency and cost of widening which have resulted in the proposal of a hybrid widening which resulted quite useful in practice. In fact, the new representations allowed analyzing some programs that exceeded memory capacity using classical sharing representations. Thus, we believe our results contribute to the practical application of top-down analysis of set sharing.

References

[AS05] Gianluca Amato and Francesca Scozzari. Optimality in goal-dependent analysis of sharing. Technical Report TR-05-06, Dipartimento di Informatica, Università di Pisa, 2005.

[BCM94] M. Bruynooghe, M. Codish, and A. Mulkers. Abstract unification for a composite domain deriving sharing and freeness properties of program variables. In F.S. de Boer and M. Gabbrielli, editors, *Verification and Analysis of Logic Languages*, pages 213–230, 1994.

[BHZ02] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Set-sharing is redundant for pair-sharing. *Theoretical Computer Science*, 277(1-2):3–46, 2002.

[BNH05] F. Bueno, J. Navas, and M. Hermenegildo. Sharing, freeness, linearity, redundancy, widenings, and cliques. Technical Report CLIP5/2005.0, Technical University of Madrid (UPM), School of Computer Science, UPM, April 2005.

[CDFB96] Michael Codish, Dennis Dams, Gilberto Filé, and Maurice Bruynooghe. On the design of a correct freeness analysis for logic programs. *The Journal of Logic Programming*, 28(3):181–206, 1996.

[CMB$^+$93] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 194–206. ACM, June 1993.

[Fec96] Christian Fecht. An efficient and precise sharing domain for logic programs. In Herbert Kuchen and S. Doaitse Swierstra, editors, *PLILP*, volume 1140 of *Lecture Notes in Computer Science*, pages 469–470. Springer, 1996.

[Fil94] G. Filé. Share x Free: Simple and correct. Technical Report 15, Dipartamento di Matematica, Universita di Padova, December 1994.

[HBPLG99] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 Int'l. Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[HPMS00] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[HZB04] P. M. Hill, E. Zaffanella, and R. Bagnara. A correct, precise and efficient integration of set-sharing, freeness and linearity for the analysis of finite and rational tree languages. *Theory and Practice of Logic Programming*, 4(3):289–323, 2004.

[JL92] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2 and 3):291–314, July 1992.

[KS94] A. King and P. Soper. Depth-k Sharing and Freeness. In *International Conference on Logic Programming*. MIT Press, June 1994.

[MH90] K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[MH91] 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.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[MSJB95] A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the Practicality of Abstract Equation Systems. In *International Conference on Logic Programming*. MIT Press, June 1995.

[Zaf01] Enea Zaffanella. *Correctness, Precision and Efficiency in the Sharing Analysis of Real Logic Languages*. PhD thesis, School of Computing, University of Leeds, Leeds, U.K., 2001.

[ZBH99] E. Zaffanella, R. Bagnara, and P. M. Hill. Widening Sharing. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 414–431, Paris, France, 1999. Springer-Verlag, Berlin.