A study of set-sharing analysis via cliques

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

jorge@cs.unm.edu, herme@unm.edu,
{bueno, herme}@fi.upm.es

[1]Depts. of Comp. Science and Electr. and Comp. Eng., Univ. of New Mexico, Albuquerque, NM, USA.
[2]School of Computer Science, T. U. Madrid (UPM), Madrid, Spain

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 abstract functions required by standard top-down analyses, both for sharing alone and also for the case of including freeness in addition to sharing. 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. 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 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 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 it is 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 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 to a great extent [CMB$^+$93,Fec96]. In particular, in [ZBH99] an alternative representation for Sharing is proposed for the non-redundant domain of [BHZ97] 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 resulting precision and efficiency results for the case of inferring pair-sharing were reported in [ZBH99]. In [Zaf01] cliques are incorporated to the original Sharing domain, but precision and efficiency are again studied for the case of inferring pair-sharing. Here, we are interested in studying precision and efficiency for the different case of inferring set-sharing. Another difference with previous work is that we develop the analysis for a top-down analysis framework, which requires the definition of additional abstract functions in the domain. Such functions were not defined in the previous works cited, since bottom-up analyses were used there.

The rest of the paper proceeds as follows. Notation and preliminaries are presented in Section 2. Then Section 3 introduces the representation based on cliques and the clique-domains for set-sharing and set-sharing with freeness. In Section 4 the required functions for top-down analysis are defined. In Section 5 we present an algorithm for detecting cliques, 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.

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 the set of variables of t. By extension, in sh_{st} st acts as a single term. Also, $\overline{sh_t}$ is the complement of sh_t, i.e., $sh \setminus sh_t$.

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 $t \in Term$, $\hat{st} = \hat{s} \cup \hat{t}$.

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.

3 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]. The proper powerset of C can then be eliminated from sh, since it is already represented by C. In fact, we will be using pairs (cl, sh) of two sharing sets. The second one represents sharing as in SH. However, in the first one, each element $C \in cl$ represents the sharing that in SH would be represented by $\wp^0(C)$.

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 $\overline{rel} : \wp(V) \times CL \longrightarrow CL$, defined as:

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

and $(amgu^W)$ is equivalent to the following definition:

$$amgu^s(x = t, (cl, sh)) = \begin{cases} (\ cl\ ,\ \overline{sh_{xt}} \cup (sh_x^* \bowtie sh_t^*)\ ) & \text{if } cl_x = cl_t = \emptyset \\ (\ \overline{rel}(\hat{xt}, cl)\ ,\ \overline{sh_{xt}}\ ) & \text{if } cl_x = sh_x = \emptyset \\ & \text{or } cl_t = sh_t = \emptyset \\ (\ \overline{rel}(\hat{xt}, cl) \cup \{\cup(cl_x \cup cl_t \cup sh_x \cup sh_t)\} & \\ \ ,\ \overline{sh_{xt}}\ ) & \text{otherwise} \end{cases}$$

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-Sharing+Freeness domain is thus $SHF^W = SH^W \times V$.

Abstract unification $amgu^{sf}$ for equation $x = t$, $x \in V$, $t \in Term$, and $s \in SHF^W$, $s = ((cl, sh), f)$, is given by $amgu^{sf}(x = t, s) = ((cl', sh'), f')$, with:

$$(cl', sh') = \begin{cases} amgu^{sff}(x = t, (cl, sh)) & \text{if } x \in f \text{ or } t \in f \\ amgu^{sfl}(x = t, (cl, sh)) & \text{if } x \notin f,\ t \notin f \text{ and } lin^s(t) \\ amgu^s(x = t, (cl, sh)) & \text{otherwise} \end{cases}$$

where $lin^s(t)$ holds iff t is a linear term and[2] for all $\{y, z\} \subseteq \hat{t}$ such that $y \neq z$, $sh_y \cap sh_z = \emptyset$ and $cl_y \cap cl_z = \emptyset$; and:

$$\begin{aligned} amgu^{sff}(x = t, (cl, sh)) = (\ & \overline{rel}(\hat{xt}, cl) \cup \\ & ((cl_x \cup sh_x) \bowtie cl_t) \cup (cl_x \bowtie sh_t) \\ ,\ & \overline{sh_{xt}} \cup (sh_x \bowtie sh_t) \qquad) \end{aligned}$$

$$amgu^{sfl}(x = t, (cl, sh)) = \begin{cases} (\ \overline{rel}(\hat{xt}, cl) \cup (cl_x \bowtie \{\cup sh_t\}) & \\ ,\ \overline{sh_{xt}} \cup (sh_x \bowtie sh_t^*)\ ) & \text{if } cl_t = \emptyset \\ (\ \overline{rel}(\hat{xt}, cl) \cup ((cl_x \cup sh_x) \bowtie \{\cup(cl_t \cup sh_t)\}) & \\ ,\ \overline{sh_{xt}}\ ) & \text{if } cl_t \neq \emptyset \end{cases}$$

[2] Note that checking this second condition can be rather expensive. Instead, the following, which is more efficient, can be checked: for all $s \in (sh_t \cup cl_t)$, $|s \cap \hat{t}| = 1$.

$$f' = \begin{cases} f & \text{if } x \in f, t \in f \\ f \setminus (\cup(sh_x \cup cl_x)) & \text{if } x \in f, t \notin f \\ f \setminus (\cup(sh_t \cup cl_t)) & \text{if } x \notin f, t \in f \\ f \setminus (\cup((sh_x \cup cl_x) \cup (sh_t \cup cl_t))) & \text{if } x \notin f, t \notin f \end{cases}$$

The operation $amgu^{sf}$ defined above is a simplification of the corresponding operation which results from the method outlined in [Zaf01] to obtain an abstract unification for SH^W plus freeness and linearity.

4 Abstract functions required by top-down analysis

In top-down analysis frameworks, the analysis of a clause $Head$:-$Body$ is 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 the variables 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. The rest of the functions are as follows:

$call2entry(Proj, Goal, Head)$
: 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 $Goal$.

$entry2exit(Body, Entry)$
: yields a substitution which represents the success of $Body$ when called in a context represented by substitution $Entry$. Both substitutions have a domain which includes the variables of $Body$, and the domain of the resulting substitution includes the domain of $Entry$.

$exit2succ(Exit', Goal, Head)$
: 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 the variables 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 $Call$.

Function $entry2exit$ is given by the framework, and basically traverses the body of a clause, analyzing each atom in turn. The three domain-dependent abstract functions which are essential are: $call2entry$, $exit2succ$, and $extend$. 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:

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

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:

$$extend(Call, g, Prime) = \overline{Call_g} \cup \{\ s \mid s \in Call_g^*,\ (s \cap \hat{g}) \in Prime\ \}$$

$$project(g, sh) = \{s \cap \hat{g} \mid s \in sh\} \setminus \{\emptyset\}$$

$$augment(g, sh) = sh \cup \{\{x\} \mid x \in \hat{g}\}$$

In the Sharing+Freeness domain, these functions are defined as follows [MH91]:

$$project^f(g, (sh, f)) = (project(g, sh), f \cap \hat{g})$$

$$augment^f(g, (sh, f)) = (augment(g, sh), f \cup \hat{g})$$

$$extend^f((sh_1, f_1), g, (sh_2, f_2)) = (sh', f')$$

$$sh' = extend(sh_1, g, sh_2)$$

$$f' = f_2 \cup \{x \mid x \in (f_1 \setminus \hat{g}), ((\cup sh'_x) \cap \hat{g}) \subseteq f_2\}$$

4.1 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 (see, e.g., [MH92] for a definition for set-sharing). We have chosen instead to present here a formalization of a way to use *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. 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 $call2entry^s$ can be defined as follows, where $unify^{sf}$ is the version of $unify$ that uses $amgu^{sf}$:

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

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:

$$project^s(g, (cl, sh)) = (project(g, cl), project(g, sh))$$

$$augment^s(g, (cl, sh)) = (cl, augment(g, sh))$$

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 5 presents a possible implementation of such a function). Let $Prime$ be already normalized, and:

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

The following two functions lift the classical *extend* [MH92] respectively to the cases of the two clique sets and of the two sharing sets occurring in each of the pairs in $Call$ and $Prime$:

$$extsh(sh_1, g, sh_2) = \overline{sh_{1g}} \cup \{\ s \mid s \in sh',\ (s \cap \hat{g}) \in sh_2\ \}$$

$$extcl(cl_1, g, cl_2) = \overline{rel}(\hat{g}, cl_1) \cup \{\ (s' \cap s) \cup (s' \setminus \hat{g}) \mid s' \in cl',\ s \in cl_2\ \}$$

The following two functions account respectively for the cases of the clique set of $Call$ and the sharing set of $Prime$, and the other way around:

$$clsh(cl', g, sh_2) = \{\ s \mid s \subseteq c \in cl',\ (s \cap \hat{g}) \in sh_2\ \}$$

$$shcl(sh', g, cl_2) = \{\ s \mid s \in sh',\ (s \cap \hat{g}) \subseteq c \in cl_2\ \}$$

The function extend 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 function $extend^s$ is computed as follows:

$$extsh(sh_1, g, sh_2) = extsh(\{u, v\}, g, \{uv\}) = \emptyset$$

$$extcl(cl_1, g, cl_2) = extcl(\{xyz\}, g, \{x\}) = \{xyz, yz\}$$

$$clsh(cl', g, sh_2) = clsh(\{xyzuv\}, g, \{uv\}) = \{yzuv, yuv, zuv, uv\}$$

$$shcl(sh', g, cl_2) = shcl(\emptyset, g, \{x\}) = \emptyset$$

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.

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

$$project^{sf}(g, s) = (project^s(g, (cl, sh)), f \cap \hat{g})$$

$$augment^{sf}(g, s) = (augment^s(g, (cl, sh)), f \cup \hat{g})$$

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:

$$(cl', sh') = extend^s((cl_1, sh_1), g, (cl_2, sh_2))$$

$$f' = f_2 \cup \{x \mid x \in (f_1 \setminus \hat{g}),\ ((\cup(sh'_x \cup cl'_x)) \cap \hat{g}) \subseteq f_2\}$$

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*

$$(\ \Downarrow cl' \cup sh') \supseteq sh \quad and \quad f' \subseteq f\ .$$

5 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 4.1 when defining *extend* for the Clique-Sharing domain, and denoted there by a function *normalize*. 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 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 $s \cap sh = \emptyset$.

It is important to stress the fact that neither minimization nor normalization change the precision of the sharing representation. They are both *reductions*, or compressions of the representation of a substitution, in the sense that the substitution is the same (i.e., conveys the same information) but its representation is smaller. Thus, they are not a widening operation, in the sense, widely used, of a change in domain or representation with the objective of improving efficiency at the cost of losing precision. This is not the case in the above operations.

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. 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 $[S]$ corresponds to 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 is already minimal; otherwise, redundant sharing groups might bias the calculation: the formula below may count

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

as not present in sh a (redundant) group which is in fact present. The computation of $[S]$ is as follows. Take cl in its state at step 9 of the algorithm. Let $I = \{S \cap C \mid C \in cl\} \setminus \{\emptyset\}$ and $A_i = \{\cap A \mid A \subseteq I,\ |A| = i\}$. Then:

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

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. This opens up the possibility of using the above algorithm as a widening. 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.

6 Experimental results

We have measured experimentally the relative efficiency and precision obtained with the inclusion of cliques in the Sharing and Sharing+Freeness domains. We measure absolute precision of a sharing set by the number of its sharing groups relative to the number of sharing groups in the worst-case for the set of variables in its domain. The number of sharing groups in the worst-case sharing for n variables is given by $2^n - 1$.

Our results are shown in Tables 1 for Sharing and 2 for Sharing+Freeness. Columns labeled **time** 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 **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 **#C** show the number of clique groups. In both tables, first the numbers for the original domain are shown, then the numbers for the clique-domain. Since our analyses infer information at all program points (before and after calling each clause body atom), and also several variants for each program point, we show the accumulated number of sharing groups in all variants for all program points.

	Sharing			**Clique-Sharing**		
	time	**precision**	**#C**	**time**	**precision**	**#C**
append	11	29 (60)	0	8	44 (60)	4
deriv	35	27 (546)	0	27	27 (546)	0
mmatrix	13	14 (694)	0	11	14 (694)	0
qsort	24	30 (1716)	0	25	30 (1716)	0
query	11	35 (501)	0	13	35 (501)	5
serialize	306	1734 (10531)	0	90	2443 (10531)	88
aiakl	35	145 (13238)	0	42	145 (13238)	0
boyer	369	1688 (4631)	0	267	1997 (4631)	158
browse	30	69 (776)	0	29	69 (776)	0
prolog_read	400	1080 (408755)	0	465	1080 (408755)	10
rdtok	325	1350 (11513)	0	344	1391 (11513)	182
warplan	3261	8207 (42089)	0	1430	8191 (26857)	420
zebra	25	280 (671088746)	0	34	280 (671088746)	0
ann	2382	10000 (314354)	0	802	19544 (313790)	700
peephole	831	2210 (12148)	0	435	2920 (12118)	171
qplan	-	-	-	860	420203 (3826458)	747
witt	405	858 (4545564)	0	437	858 (4545564)	25

Table 1. Precision and Time-efficiency for Sharing

Benchmarks are divided into three groups. Of each group we only show a reduced number of the benchmarks actually used: those which are more representative. The first group, append through serialize, 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). The second group, aiakl through zebra, are more involved: they make use of mutual recursion and of elaborated 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 and rdtok are parsers of Prolog). The benchmarks in the third group are all (parts of) "real" programs: ann is the &-prolog parallelizer, peephole 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.

	Sharing+Freeness			Clique-Sharing+Freeness		
	time	precision	#C	time	precision	#C
append	6	7 (30)	0	6	7 (30)	0
deriv	27	21 (546)	0	27	21 (546)	0
mmatrix	9	12 (694)	0	11	12 (694)	0
qsort	25	30 (1716)	0	27	30 (1716)	0
query	12	22 (501)	0	14	22 (501)	0
serialize	61	545 (5264)	0	55	736 (5264)	41
aiakl	37	145 (13238)	0	43	145 (13238)	0
boyer	373	1739 (5036)	0	278	2074 (5036)	163
browse	29	69 (776)	0	31	69 (776)	0
prolog_read	425	1050 (408634)	0	481	1050 (408634)	0
rdtok	335	1047 (11513)	0	357	1053 (11513)	2
warplan	1320	3068 (23501)	0	1264	5705 (25345)	209
zebra	41	280 (671088746)	0	42	280 (671088746)	0
ann	1791	7811 (401220)	0	968	14108 (394800)	510
peephole	508	1475 (9941)	0	403	2825 (12410)	135
qplan	-	-	-	2181	233070 (3126973)	529
witt	484	813 (4545594)	0	451	813 (4545594)	0

Table 2. Precision and Time-efficiency for Sharing+Freeness

In order to understand the results shown in the tables above it is important to note an existing synergy between normalization, efficiency, and precision. 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).

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. 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 function *extend*. 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 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.

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

On the other hand, for those benchmarks which do generate cliques (like serialize, boyer, warplan, 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 runs longer.

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 function $amgu^{sf}$ 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 run out of memory with the standard representation.

Effectiveness. We have also tested how relevant precision losses can be when the analysis is used as part of another application. In particular, we have used the Clique-Sharing+Freeness domain for inferring non-failure information [BLGH04]. We have selected a representative subset of our benchmarks. Results for them are shown in Table 3. Columns marked **Total** show the number of predicates. Columns marked **NF** show the number of predicates which the analysis can infer that they will not fail. Columns marked **Cov** show the number of predicates that the analysis can infer that they are covered (a necessary condition for guaranteeing non-failure). The results obtained suggest that the precision losses caused by the use of the clique domain are not relevant when the information from analysis is used as input in this particular application.

	Sharing+Freeness			Clique-Sharing+Freeness		
	Total	NF (%)	Cov (%)	Total	NF (%)	Cov (%)
append	1	1 (100)	1 (100)	1	1 (100)	1 (100)
deriv	1	1 (100)	1 (100)	1	1 (100)	1 (100)
qsort	3	3 (100)	3 (100)	3	3 (100)	3 (100)
serialize	5	0 (0)	2 (40)	5	0 (0)	2 (40)
rdtok	22	8 (36)	13 (59)	22	8 (36)	13 (59)
zebra	6	1 (16)	4 (66)	6	1 (16)	4 (66)

Table 3. Accuracy of the non-failure analysis

7 Conclusions and Future work

We have reported on a study of efficiency and precision of the clique representation of sharing when used for inferring proper set-sharing, as opposed to pair-sharing. We have also included the case of Clique-Sharing plus freeness information. Besides the abstract unification operations for both domains with the clique representation (equivalent definitions of which were already proposed in the literature), we have contributed other operations required for top-down analyses, in particular, the extend function. Experiments reported aim specifically at the use of cliques as an alternative representation, not as a widening (as opposed to similar experiments reported in [Zaf01], where a threshold on the number of allowed sharing groups was imposed that triggered their move into cliques). We are currently working on using the clique representation as a widening in order to solve the mentioned limitations of the *extend* function. In line with the conclusions from previous experiments, our experimental evaluation also supports the conclusion that precision losses are reasonable. This is also supported additionally by our experiments in actually using the information inferred, as we have showed for inferring non-failure. Efficiency gains have also been shown, to the extreme case of being able to analyze programs that exceeded memory capacity using the classical sharing representation.

References

[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.

[BHZ97] R. Bagnara, P. M. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. In *Static Analysis Symposium*, pages 53–67. Springer-Verlag, 1997.

[BLGH04] F. Bueno, P. López-García, and M. Hermenegildo. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.

[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.

[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.

[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.