Lazy Set-Sharing Analysis

Xuan Li[1], Andy King[2], Lunjin Lu[1]

[1] Oakland University Rochester, MI 48309, USA
{x2li,l2lu}@oakland.edu
[2] University of Kent, Canterbury, UK
a.m.king@kent.ac.uk

Abstract. Sharing analysis is widely deployed in the optimisation, specialisation and parallelisation of logic programs. Each abstract unification operation over the classic Jacobs and Langen domain involves the calculation of a closure operation that has exponential worst-case complexity. This paper explores a new tactic for improving performance: laziness. The idea is to compute partial sharing information eagerly and recover full sharing information lazily. The net result is an analysis that runs in a fraction of the time of the classic analysis and yet has comparable precision.

1 Introduction

Sharing analysis is one of the most well-studied analyses within logic programming. Two variables are said to share if they are bound to terms that contain a common variable; otherwise they are independent. Independence information has many applications in logic programming that include occurs-check reduction [24, 10], automatic parallelisation [11] and finite-tree analysis [2]. Set-sharing analysis [17, 18], as opposed to pair-sharing analysis [24], has particularly attracted much attention. This is because researchers have been repeatedly drawn to the algebraic properties of the domain since these offer tantalising opportunities for implementation [3, 7, 8, 12].

The reoccurring problem with the set-sharing domain is the closure under union operation that lies at the heart of the abstract unification (*amgu*) operation [18]. A sharing abstraction is constructed from a set of sharing groups each of which is a set of program variables. Closure under union operation repeatedly unions together sets of sharing groups, drawn from a given sharing abstraction, until no new sharing group can be obtained. This operation is exponential, hence the interest in different, and possibly more tractable, encodings of set-sharing [7, 8]. One approach to curbing the problem of closure is to schedule the solving of a sequence of equations so as to first apply *amgu* to equations that involve ground terms [21]. To this end, Muthukumar and Hermenegildo [22] detail an queueing/dequeueing mechanism for maximally propagating groundness among systems of equations. This tactic exploits the commutativity of the *amgu* operation [13, 21][Lemma 62]. A refinement of this idea is to schedule the *amgu* operations using an estimate of the cost of a closure. The *amgu* operations are applied in order of increasing closure cost and then widening is invoked [25, 21][Section 4.3] when the application of a closure becomes prohibitively expensive. To support widening, the set-sharing domain has been extended [25, 21][Section 4.3] from a set of sharing groups, to a pair of sets of sharing groups. A sharing group in the first component (that is called a clique [25]) is then reinterpreted as representing all sharing groups that are contained within it.

This paper addresses the question of whether closure calculation is actually required at all. This paper shows that cliques can be reinterpreted as pending closure operations. Under this reinterpretation, the complexity of *amgu* can be reduced to a quadratic time operation on sharing groups and a constant time operation on cliques. If necessary, a classic set-sharing abstraction can be recovered using the cliques without incurring a precision loss by applying a closure operation for the groups that are relevant to each clique. Quite apart from being rather surprising, this reinterpretation of cliques can avoid computing closures all together in some circumstances. Firstly, if one clique is contained within another, then the smaller clique is redundant as is its associated closure operation. Secondly, if a clique contains those variables that appear in a call, then it can be passed from one procedure to another without applying closure. Thirdly, if closure is prohibitively expensive, then it can be retained as clique, albeit incurring a possible loss of precision in the projection and merge operations. This leads to an analysis that is parameterised by a cost threshold k: a clique is only created if the resulting number of sharing groups exceed k. The resulting analysis is polynomial and benchmarking suggests that it realises a good tradeoff between precision and efficiency.

This paper is organised as follows. Section 2 contains basic concepts and recalls the *Sharing* domain. Section 3 informally introduces the motivation via examples. Sections 4-7 present the abstract domain and abstract operators in details. Section 8 discusses an experimental implementation. Section 9 discusses related work and section 10 concludes.

2 Preliminaries

Let $\mathcal{V}$ denote a denumerable universe of variables. Let $var(o)$ be the set of variables in the syntactic object o. If S is a set then $|S|$ is its cardinality and $\wp(S)$ is the powerset of S. The (classic) set-sharing domain *Sharing* is defined by $Sharing = \{S \in \wp(\wp(\mathcal{V})) \mid \emptyset \in S\}$. We call a set of variables $G \subseteq \mathcal{V}$ a sharing group and a set $S \in Sharing$ a sharing abstraction. The restriction $G \upharpoonright V$ is defined by $G \upharpoonright V = G \cap V$ where $V \subseteq \mathcal{V}$. Moreover, if $S \in Sharing$ then $S \upharpoonright V$ is defined by $S \upharpoonright V = \{G \upharpoonright V \mid G \in S\}$. The function $max : \wp(\wp(\mathcal{V})) \to \wp(\wp(\mathcal{V}))$ is defined by $max(S) = \{G \in S \mid \forall H \in S.G \subseteq H \to G = H\}$.

The most important operator in *Sharing* is the abstract unification operator $amgu_{x=t}$ where x is a variable and t is a term not containing x. The operator $amgu_{x=t}$ is defined in terms of three auxiliary operations. The closure under union of a sharing abstraction S, denoted by S^*, is the smallest superset of S satisfying if $X \in S^*$ and $Y \in S^*$ then $X \cup Y \in S^*$. The set of sharing groups in

S that are relevant to a syntactic object o is $rel(o, S) = \{G \in S \mid var(o) \cap G \neq \emptyset\}$. We say that a sharing abstraction $S \in Sharing$ is relevant to a term t iff $rel(t, S) \neq \emptyset$. The cross union of two sharing abstractions S_1 and S_2 is defined by $S_1 \uplus S_2 = \{G \cup H \mid G \in S_1 \wedge H \in S_2\}$. The following definition of the abstract unification is adapted from [8] that is equivalent to classic definition in [17, 18].

Definition 1. $amgu_{x=t}(S) = S \setminus rel(x, S) \setminus rel(t, S) \cup (rel(x, S) \uplus rel(t, S))^*$ where $\setminus$ is the set minus operator.

The set of all clique-sets CL [25] is defined by $CL = \wp(\{G \in \wp(\mathcal{V}) \mid G \neq \emptyset\})$. A clique is an element of a clique-set. In this paper, a clique is the abstraction of an equation of the e where e takes the form $x = t$.

3 Motivation Example

This section informally introduces the basic idea using several examples. We demonstrate how to use cliques and part of sharing abstraction to represent the whole sharing abstraction.

Example 1. Suppose $S = \{\emptyset, \{X_1\}, \{X_2\}, \{X_3\}, \{X_4\}, \{X_5\}, \{X_6\}, \{X_7\}\}$ and equation e is $X_1 = f(X_2, X_3, X_4, X_5, X_6, X_7)$. If we apply the classical abstract unification $amgu$, we obtain

$$amgu_e(S) = \{\emptyset\} \cup \left\{ \begin{array}{l} \{X_1, X_2\}, \{X_1, X_3\}, \{X_1, X_4\}, \\ \{X_1, X_5\}, \{X_1, X_6\}, \{X_1, X_7\} \end{array} \right\}^*$$

There are 63 groups in $amgu_e(S) \setminus \{\emptyset\}$ in all. Note that we only list 6 groups instead of enumerating all 63 groups for the sake of saving space. We shall refer to these 6 groups as the basis of the 63 groups because the whole closure (the 63 groups) can be recovered from the basis (the 6 groups) by applying closure under union operator to the basis. In other words, we can use the basis to represent the whole closure. The intuition behind the analysis is to record which groups participate in the closure under union and avoid computing closure under union operation until it is called for. We record which groups participate in closure under union by keeping a record of the set of variables that are contained within the equation e. Moreover, we only consider the equations that contain at least one variable and thus we can use a clique [25] to represent these variable sets. Therefore we use a pair $\langle E, S\rangle \in CL \times Sharing$ to represent a sharing abstraction: the first component is the set of cliques; the second component is the partially computed sharing information. In this way, we can avoid closure under union operation during analysis.

Example 2. Continuing with example 1, the state of sharing after abstract unification is expressed as

$$\left\langle \begin{array}{l} \{\{X_1, X_2, X_3, X_4, X_5, X_6, X_7\}\}, \\ \{\emptyset, \{X_1, X_2\}, \{X_1, X_3\}, \{X_1, X_4\}, \{X_1, X_5\}, \{X_1, X_6\}, \{X_1, X_7\}\} \end{array} \right\rangle$$

In this example, this part of sharing information is $S \setminus rel(x = t, S) \cup rel(x, S) \uplus rel(t, S)$. We will show that this is not a coincident in the later sections.

When we need to recover full sharing information, we can just apply the closure under union to those groups that are relevant to each clique.

Example 3. Continuing with example 2, there is only one clique in first components. If we denote $\{X_1, X_2, X_3, X_4, X_5, X_6, X_7\}$ by e and $\{\emptyset, \{X_1, X_2\}, \{X_1, X_3\}, \{X_1, X_4\}, \{X_1, X_5\}, \{X_1, X_6\}, \{X_1, X_7\}\}$ by S, then the full sharing information is obtained by

$$S \cup rel(e, S)^* = \{\emptyset\} \cup \left\{ \begin{array}{l} \{X_1, X_2\}, \{X_1, X_3\}, \{X_1, X_4\}, \\ \{X_1, X_5\}, \{X_1, X_6\}, \{X_1, X_7\} \end{array} \right\}^*$$

which coincides with the sharing information derived in example 1.

If there is more than one clique, then sharing is recovered by considering each clique in turn.

Example 4. Consider $\langle \{e_1, e_2\}, S \rangle$ where $e_1 = \{X, Y\}$, $e_2 = \{Y, Z\}$ and $S = \{\emptyset, \{X\}, \{Y\}, \{Z\}\}$. First, applying e_1, we obtain $S_1 = S \cup rel(e_1, S)^* = \{\emptyset, \{X\}, \{Y\}, \{Z\}, \{X, Y\}\}$. Next applying e_2, we have $S_1 \cup rel(e_2, S_1)^* = \{\emptyset, \{X\}, \{Y\}, \{Z\}, \{X, Y\}, \{Y, Z\}, \{X, Y, Z\}\}$. In fact, as is shown latter, the order in which the cliques are consider is incidental.

Although we call closure under union operator when we recover full sharing information, we only apply it on demand. Even if the client wants to get full sharing information at every program point, a lazy evaluation strategy is still better because we avoid redundant computation during analysis. For an instance, when clique e_1 is a subset of e_2, we can disregard e_1 because the effect of e_1 is subsumed by that of e_2. Moreover, the number of groups involved in least fixed point computation is much less than the number of groups involved in classical *Sharing* because only part of closure is involved.

In summary, we only compute part of sharing information at each program point during analysis and can recover full sharing information on demand after analysis. As a result, the closure under union operator is avoided for abstract unification. In following sections, we will present this new domain formally.

4 Abstract Domain

Reducing the input data size and improving the efficiency of operators over the domain are two ways to improve the efficiency of an analysis. Recall definition 1:

$$amgu_{x=t}(S) = S \setminus rel(x, S) \setminus rel(t, S) \cup (rel(x, S) \uplus rel(t, S))^*$$

For the ensuing discussion, denote $rel(x, S) \uplus rel(t, S)$ by S_b. Rather than directly compute S_b^* and use this closure to calculate $amgu_{x=t}(S)$, we compute the partial sharing information $S \setminus rel(x, S) \setminus rel(t, S) \cup S_b$ and record information that enables S_b^* to be computed on demand. The size of the partial sharing information inferred using this strategy is never more, and is often significantly less, than that inferred using full sharing. The key issue is to resolve which groups belong to S_b so that we can recover S_b^* on demand. One way to deduce S_b, and hence S_b^*, is to save the equation $x = t$ because if $G \in S \setminus rel(x, S) \setminus rel(t, S) \cup S_b$ then $G \in S_b$ iff $rel(x = t, G) \neq \emptyset$. Since only variables in the equation actually matter in $rel(x = t, G)$, it is sufficient to record the clique $var(x = t)$ to recover S_b and thereby derive full sharing. To summarise, the computational domain of *Sharing* is replaced with pairs of the form $\langle E, S\rangle \in CL \times Sharing$ where each clique in the first component is an abstraction of an equation and is used to recover the full sharing information that $\langle E, S\rangle$ describes. The following function formalises this idea since it maps each abstract values in $CL \times Sharing$ to the sharing abstractions they describe:

Definition 2. $\langle E, S\rangle \propto S'$ iff $r(\langle E, S\rangle) = S'$ where $r(\langle E, S\rangle)$ is defined as follows:

$$r(\langle \emptyset, S\rangle) = S$$

$$r(\langle \{e\} \cup E', S\rangle) = S'' \cup rel(e, S'')^*$$

where $e \in E$, $E' = E \setminus \{e\}$ and $S'' = r(\langle E', S\rangle)$.

This function is called the recovery function. Although this recovery function calls closure under union operator, the recovery function is itself only called on demand. Henceforth, we abbreviate $r(\langle E, S\rangle)$ by $r(E, S)$.

Example 5. Let $\langle E, S\rangle = \langle \{e_1, e_2, e_3\}, S\rangle$, where $e_1 = \{X, Y\}$, $e_2 = \{Y, Z\}$, $e_3 = \{Y, W\}$ and $S = \{\emptyset, \{X\}, \{Y\}, \{Z\}, \{W\}\}$. Thus we can recover the full sharing information S' that $\langle E, S\rangle$ describes by unfolding definition 2 in the following fashion:

$$S' = r(\{e_1, e_2, e_3\}, S) = r(\{e_3\}, r(\{e_1, e_2\}, S)) = r(\{e_3\}, r(\{e_2\}, r(\{e_1\}, S)))$$

$$S_1 = r(\{e_1\}, S) = S \cup rel(e_1, S)^* = \{\emptyset, \{X\}, \{Y\}, \{Z\}, \{W\}, \{X, Y\}\}$$

$$S_2 = r(\{e_2\}, S_1) = S_1 \cup rel(e_2, S_1)^* = \left\{ \begin{array}{l} \emptyset, \{X\}, \{Y\}, \{Z\}, \{W\}, \\ \{X, Y\}, \{Y, Z\}, \{X, Y, Z\} \end{array} \right\}$$

$$S' = r(\{e_3\}, S_2) = S_2 \cup rel(e_3, S_2)^* = \left\{ \begin{array}{l} \emptyset, \{X\}, \{Y\}, \{Z\}, \{W\}, \{X, Y\}, \\ \{Y, Z\}, \{X, Y, Z\}, \{Y, W\}, \\ \{X, Y, W\}, \{Y, Z, W\}, \{X, Y, Z, W\} \end{array} \right\}$$

One natural question is whether the order in which cliques are considered in the recovery process actually matters. The following lemma is a partial answer to this question: it shows that the ordering does not matter when there are only two cliques in the first component.

Lemma 1.

$$S \cup rel(e_1, S)^* \cup rel(e_2, S \cup rel(e_1, S)^*)^* = S \cup rel(e_2, S)^* \cup rel(e_1, S \cup rel(e_2, S)^*)^*$$

The following proposition lifts this result to an arbitrary number of cliques; it also means that the recovery function satisfies the diamond property.

Proposition 1. $r(E, S) = r(P, S)$ whenever P is a permutation of E.

The abstract domain $CL \times Sharing$ is a non-canonical representation because different pairs $\langle E_1, S_1\rangle \neq \langle E_2, S_2\rangle$ can map to the same sharing abstraction during recovery. To obtain a partially ordered domain, it is necessary to induce an equivalence relation on $CL \times Sharing$. We therefore define $\langle E_1, S_1\rangle \equiv \langle E_2, S_2\rangle$ iff $r(E_1, S_1) = r(E_2, S_2)$ and consider the abstract domain $XSharing = (CL \times Sharing)_{\equiv}$, that is, the computational domain of sets of equivalence classes induced by $\equiv$. The partial ordering over $XSharing$, denoted by $\sqsubseteq$, can then by defined as $\langle E_1, S_1\rangle \sqsubseteq \langle E_2, S_2\rangle$ iff $r(E_1, S_1) \subseteq r(E_2, S_2)$. It then follows that $\langle XSharing, \sqsubseteq\rangle$ is a complete lattice.

For notational simplicity, we blur the distinction between elements of $XSharing$ and the elements of $CL \times Sharing$ and interpret the pair $\langle E_1, S_1\rangle \in CL \times Sharing$ as representing the equivalence class $[\langle E_1, S_1\rangle]_{\equiv} \in XSharing$. For example, the recovery function over $XSharing$ is the natural lifting of this function to the domain of equivalence classes.

5 Abstract unification

The rationale for our work is to avoid applying the calculating closure under union operator in abstract unification. In the following definition, we detail how to avoid the closures by instead creating cliques in the first component. An important consequence of this scheme, is that the operator is polynomial in the number of groups. In what follows, we prove that this new notion of abstract unification is both sound and precise.

Definition 3.

$$xamgu_{x=t}(\langle E, S\rangle) = \left\langle \begin{array}{c} max(E \cup \{var(x = t)\}), \\ (S \setminus rel(x = t, S)) \cup (rel(x, S) \uplus rel(t, S)) \end{array} \right\rangle$$

Henceforth, we abbreviate $xamgu_{x=t}(\langle E, S\rangle)$ by $xamgu_{x=t}(E, S)$. Note that we only need to maintain maximal cliques in the first component because the effect of a smaller clique is subsumed by that of a larger one (one that strictly contains it). Therefore, we say a clique e_i is redundant in $E \in CL$ if there exists $e_j \in E$ such that $e_i \subseteq e_j$. In that this is just one way of detecting redundancy: other more sophisticated ways are also possible.

Example 6. Consider $\langle E, S\rangle$ where $E = \{\{X, Y, Z\}\}$ and $S = \{\emptyset, \{X, Y\}, \{X, Z\}, \{Z, W\}\}$. Then $xamgu_{Y=Z}(E, S) = \langle \{\{X, Y, Z\}\}, \{\emptyset, \{X, Y, Z\}, \{X, Y, Z, W\}\}\rangle$

Lemma 2 explains how $xamgu$ coincides with $amgu$ when there is no clique in the first component.

Lemma 2. $r(xamgu_{x=t}(\emptyset, S)) = amgu_{x=t}(S)$.

Proposition 2 also builds towards the main correctness theorem by showing the equivalence of of two instances of abstract unification that operate on pairs whose clique sets differ by just one clique.

Proposition 2. $r(xamgu_{x=t}(E,S)) = r(xamgu_{x=t}(E \setminus \{e\}, r(\{e\},S)))$ where $e \in E$.

By applying proposition 2 inductively, it follows that $r(xamgu_{x=t}(E,S)) = r(xamgu_{x=t}(\emptyset, r(E,S)))$, that is, the result of applying $xamgu$ to $\langle E,S\rangle$ coincides with the sharing abstraction obtained by applying $xamgu$ to $\langle \emptyset, r(E,S)\rangle$. This result leads into Theorem 1 which states that $xamgu$ neither compromises correctness not precision with respect to the classic approach of set-sharing.

Theorem 1. $r(xamgu_{x=t}(E,S)) = amgu_{x=t}(r(E,S))$.

Example 7. Recall from example 6 that $xamgu_{Y=Z}(E,S) = \langle\{\{X,Y,\ Z\}\}, \{\emptyset, \{X,Y,Z\}, \{X,Y,Z,W\}\}\rangle$. By applying r to the $\langle E,S\rangle$ pair we obtain $r(E,S) = \{\emptyset, \{X,Y\}, \{X,Z\}, \{Z,W\}, \{X,Y,Z\}, \{X,Y,Z,W\}, \{X,Z,W\}\}$. Then applying $amgu$ to $r(E,S)$ yields $amgu_{Y=Z}(r(E,S)) = \{\emptyset, \{X,Y,Z\}, \{X,Y,Z,W\}\}$. Observe $r(xamgu_{Y=Z}(E,S)) = amgu_{Y=Z}(r(E,S))$ as predicted by the theorem.

6 Projection

Projection on *Sharing* is relatively simple. In our proposal, projection is not straightforward as one might think. If projection was defined as $\langle E,S\rangle \upharpoonright\upharpoonright V = \langle E \upharpoonright V, S \upharpoonright V\rangle$, then cliques would be possibly lost by projection which could affect recovery and can possibly compromise the soundness of the analysis as a whole. For example, suppose $\langle E,S\rangle = \langle\{\{X,Y\}\}, \{\emptyset, \{Z,X\}, \{Y,W\}\}\rangle$ and $V = \{Z,W\}$. Then $\langle E \upharpoonright V, S \upharpoonright V\rangle = \langle \emptyset, \{\emptyset, \{Z\}, \{W\}\}\rangle$ and hence we lose the information that Z and W could possibly share. In our projection operator, we divide the cliques into three classes. We need to apply a polynomial transformation for all cliques in the first class before applying the classical projection operator $\upharpoonright$. It is easy to do projection for the cliques in the second class. No projection is needed for the cliques in the third class.

6.1 First Class

Let $V \subseteq \mathcal{V}$ be a set of variables of interest. Every clique e such that $e \cap V = \emptyset$ is in the first class and it will be totally projected away. Usually the first class cliques are generated by renaming. One way to keep the effect of a clique e in the first class is applying the recovery function. This is precise but inefficient because of the closure under union operator in the recovery function. To avoid this, we maintain the effect of e by replacing e with a new clique e' that simulates the effect of e. This guarantees soundness but could lose precision. First, we define two auxiliary functions.

Observe that if we use $\cup rel(e,S)$ to replace e, we will ensure $rel(e,S)$ is selected because $rel(e,S) \subseteq rel(\cup rel(e,S), S)$.

Definition 4. Function $Replace : CL \times Sharing \to CL$ is defined by

$$Replace(E,S) = max(\{\cup rel(e,S) | e \in E\})$$

We say that two cliques e_1 and e_2 are related with respect to a sharing abstraction S if $rel(e_2, rel(e_1, S)) \neq \emptyset$. We also say e_i and e_j are related with respect to S if e_i is related to e_l with respect to S and e_l is related to e_j with respect to S. We say E is related with respect to S if every $e \in E$ is related to each other with respect to S.

Definition 5. Let E_1 be a set of cliques in the first class. We define a rewriting rule $\Longrightarrow_S$ by $E_1 \Longrightarrow_S E_2$ if $e_1, e_2 \in E_1 \wedge rel(e_2, rel(e_1, S)) \neq \emptyset \wedge E_2 = E_1 \setminus \{e_1, e_2\} \cup \{e_1 \cup e_2\}$; otherwise $E_1 \Longrightarrow_S E_1$.

We write $E \Longrightarrow_S^* E'$ if there exist $E_1, E_2, \ldots, E_n$ with $E_1 = E, E_n = E'$, $E_i \Longrightarrow_S E_{i+1}$ for $0 \leq i \leq n-1$ and $E' \Longrightarrow_S E'$. That is we apply $\Longrightarrow_S$ repeatedly until all the related cliques are combined into one. In other words, there are no two cliques that are related with respect to S in E'. Once we have E', we can apply the *Replace* function to replace all the cliques in E'. Note that the order in which cliques are considered and combined does not effect the final result. Thus this definition satisfies the diamond property and is well-defined.

Let $E \Longrightarrow_S^* E'$. Then here exists $E_1, E_2, \ldots, E_n$ with $E_1 = E, E_n = E'$, $E_i \Longrightarrow_S E_{i+1}$ for $0 \leq i \leq n-1$. Note that n is bounded by the number of groups. Each combination of two related cliques is polynomial. Thus the transformation is polynomial in the number of groups too.

Example 8. Let $E = \{\{X_5, X_8\}, \{X_5, X_6\}\}$, $S = \{\emptyset, \{X_1, X_6, X_8\}, \{X_4, X_8\}, \{X_5\}, \{X_2, X_6\}\}$ and $V = \{X_1, X_2, X_3, X_4\}$. Then $E' = \{\{X_5, X_6, X_8\}\}$ where $E \Longrightarrow_S^* E'$.

Every $e \in E$ is replaced by a corresponding $e' \in Replace(E', S)$ where $E \Longrightarrow_S^* E'$. There exists $e'' \in E'$ such that $e' = \cup rel(e'', S)$. Observe that $e \subseteq e''$ follows from the definition of $\Longrightarrow_S^*$. Thus $rel(e, S) \subseteq rel(e'', S)$ and $\cup rel(e, S) \subseteq \cup rel(e'', S) = e'$. Thus e' actually includes the effect of e. If e' is not relevant to V then it says nothing about V and hence it does not matter at all. Therefore we can delete all the cliques that are not relevant to V in $Replace(E', S)$ where $E \Longrightarrow_S^* E'$.

Definition 6. Let E be a set of cliques in the first class and V is a set of variables. The function $T : CL \times Sharing \rightarrow CL$ is defined by $T(E, S) = Replace(E', S) \setminus E''$ where $E \Longrightarrow_S^* E'$, $E'' = \{e' \in Replace(E', S) | e' \cap V = \emptyset\}$.

Note that $T(\emptyset, S) = \emptyset$.

Example 9. Continuing with example 8, observe that $Replace(E', S) = \{\{X_1, X_2, X_4, X_5, X_6, X_8\}\}$ where $E \Longrightarrow_S^* E'$. Hence $Replace(E', S) \setminus E'' = Replace(E', S)$ because $E'' = \emptyset$. Thus, $T(E, S) = \{\{X_1, X_2, X_4, X_5, X_6, X_8\}\}$.

Proposition 3. (Soundness)
Let E be a set of cliques in the first class and $E' = T(E, S)$. Then

$$r(E, S) \upharpoonright V \subseteq r(E' \upharpoonright V, S \upharpoonright V)$$

6.2 Second Class and Third Class

Let V be a set of variables, $\langle E, S\rangle \in \mathit{XSharing}$ and $e \in E$. If $e \cap V \neq \emptyset$ and $e \not\subseteq V$, we say that the clique e is in the second class. If the clique e is a subset of V then we say it is in the third class.

Proposition 4. Let V be a set of variables. If $rel(e, S) = rel(e \upharpoonright V, S)$ for each $e \in E$ then $r(E, S) \upharpoonright V = r(E \upharpoonright V, S \upharpoonright V)$.

Example 10. Let $\langle E, S\rangle = \langle\{\{X, Y\}, \{Z, W\}\}, \{\emptyset, \{X, Y, Z\}, \{Z, W\}\}\rangle$ and $V = \{X, Z\}$. Then $r(E, S) = \{\emptyset, \{X, Y, Z\}, \{Z, W\}, \{X, Y, Z, W\}\}$ and hence $r(E, S) \upharpoonright V = \{\emptyset, \{X, Z\}, \{Z\}\}$. On the other hand, $\langle E \upharpoonright V, S \upharpoonright V\rangle = \langle\{\{X\}, \{Z\}\}, \{\emptyset, \{X, Z\}, \{Z\}\}\rangle$. Therefore $r(E, S) \upharpoonright V = r(E \upharpoonright V, S \upharpoonright V)$.

Lemma 3. If a clique e is in second class then $rel(e, S) = rel(e \upharpoonright V, S)$.

According to lemma 3 and proposition 4, we can directly project those cliques in the second class without losing soundness and precision. It is obvious that $rel(e, S) = rel(e \upharpoonright V, S)$ is true for each clique in the third class. Note that the second class and the third class are both relevant to V while the first class is not. Thus we can conclude that we can directly apply classical projection to all those cliques that are relevant to V. This is useful for implementation.

6.3 Projection

Therefore, we define the projection as following.

Definition 7. Let $\mathcal{V}$ be the set of all variables and $V \in \wp(\mathcal{V})$. The projection function $\upharpoonright\upharpoonright$: $\mathit{XSharing} \times \wp(\mathcal{V}) \rightarrow \mathit{XSharing}$ is defined as $\langle E, S\rangle \upharpoonright\upharpoonright V = \langle(T(E_1, S) \cup E_2 \cup E_3) \upharpoonright V, S \upharpoonright V\rangle$ where E_1 contains the first class cliques in E, E_2 contains the second class cliques in E and E_3 contains the third class cliques in E.

Theorem 2. (Soundness)
Let V be a set of variables and $\langle E, S\rangle \in \mathit{XSharing}$. Then $r(E, S) \upharpoonright V \subseteq r(\langle E, S\rangle \upharpoonright\upharpoonright V)$.

Although the projection operator needs to transform the first class cliques, it is still faster than applying the recovery function because T is polynomial in the number of groups.

7 Other Operations and Parameter Throughout

The design of the analysis is completed by defining other operators that are required by an analysis engine. The initial state is $\langle\emptyset, init(\varepsilon)\rangle$ where $init(\varepsilon)$ describes the empty substitution [18]. The renaming operator $\mathit{Rename} : \mathit{XSharing} \rightarrow \mathit{XSharing}$ is defined by $\mathit{Rename}(\langle E, S\rangle) = \langle tag(E), tag(S)\rangle$ where tag is classical tagging function defined in [18]. The join operator $\sqcup : \mathit{XSharing} \times \mathit{XSharing} \to \mathit{XSharing}$ is defined by $\langle E_1, S_1\rangle \sqcup \langle E_2, S_2\rangle = \langle E', S'\rangle$ where $r(E', S') = r(E_1, S_1) \cup r(E_2, S_2)$. In practice, we use $\langle E_1, S_1\rangle \bar{\sqcup} \langle E_2, S_2\rangle = \langle max(E_1 \cup E_2), S_1 \cup S_2\rangle$ as an approximation. Observe $r(E_1, S_1) \cup r(E_2, S_2) \subseteq r(max(E_1 \cup E_2), S_1 \cup S_2)$. Thus it is sound but may lose precision.

Example 11. Let $e_1 = \{x, y\}$, $e_2 = \{y, z\}$, $S_1 = \{\emptyset, \{x, y\}, \{z\}\}$ and $S_2 = \{\emptyset, \{y, z\}, \{x\}\}$. Then $r(\{e_1\}, S_1) \cup r(\{e_2\}, S_2) = \{\emptyset, \{x\}, \{z\}, \{x, y\}, \{y, z\}\}$ while $r(\{e_1, e_2\}, S_1 \cup S_2) = \{\emptyset, \{x\}, \{z\}, \{x, y\}, \{y, z\}, \{x, y, z\}\}$. This is not precise because it says variables x, y, z share a variable.

Basically, more cliques mean more efficiency and less cliques mean more precision. In fact, we can think *Sharing* is the most precise case of *XSharing* because it saves no clique in first component. We can control the complexity/precision ratio in two way:

– When do we create clique? The analysis is parameterised by a cost threshold k: a clique is only created if the resulting number of sharing groups exceed a predefined threshold k.
– Which clique do we create or remove? A practical analyser requires a grounding operator whose effect is to ground given set of variables in a state. To this end, a grounding operator $ground : \mathit{XSharing} \times \mathcal{V} \to \mathit{XSharing}$ is introducing that is defined by

$$ground(\langle E, S\rangle, x) = \langle E \setminus rel(x, E), r(rel(x, E), S) \setminus rel(x, r(rel(x, E), S))\rangle$$

Observe that cliques are possibly removed from first component and that the overall size of the abstraction is not increased by this operator. This operator is applied to model the effect of a built-in predicate such as $>/2$. The effect of $>/2$ is to ground the variables associated with this operator. For instance, the effect of $X > Y$ is to ground both X and Y.

8 Implementation

We have implemented the new abstract domain and incorporated it to a top down framework which is based on Nilsson [23]. The implementation is written in C++. To obtain a credible experimental comparison, both the classical *Sharing* and *XSharing* are implemented using the same language, the same framework, the same data structure and the same primitive operators. Table 1 compares *Sharing* and *XSharing* on cost and precision using some standard benchmark programs. The first column and the fifth column list the names of programs. The second column, the third column, the sixth column and the seventh column are time performance for *XSharing* and *Sharing* respectively. The fourth and the last column are the loss of precision in percentage considering pair sharing information. Suppose there are n program points in a program. For program point i, the pair sharing information computed by *Sharing* is PS_i while *XSharing* computes XPS_i. Pair-sharing PS_i must be a subset of XPS_i since *XSharing* is approximate. It follows that $|XPS_i| \geq |PS_i|$. Then pair sharing precision lost at program point i can be defined by $(|XPS_i| - |PS_i|)/|PS_i|$. The pair sharing precision lost for whole program is $\sum_{i=1}^{n}((|XPS_i| - |PS_i|)/|PS_i|)/n$.

Benchmarks are divided into two groups in table 1. *XSharing* and $Sharing$ are both acceptable for the programs in the first group. For the programs in the second group, *XSharing* is much faster. Classical $Sharing$ takes hundreds seconds for *peepl.pl*, *sdda.pl*, *ga.pl* and *read.pl* while *XSharing* only takes several seconds. This is not a surprise because all key operators are polynomial.

Table 1. Time Performance Considering Effects of Built-in Predicates; k=0

CPU: Intel(R) Pentium(R) 4 CPU 2.40GHz. Operating System: Windows XP							
	XSharing	$Sharing$	Precision		*XSharing*	$Sharing$	Precision
Program	Time	Time	Pair lose	Program	Time	Time	Pair lose
append.pl	0.008	0.016	0	qsort.pl	0.008	0.015	0
merge.pl	0.015	0.016	0	path.pl	0.008	0.008	0
zebra.pl	0.031	0.016	0	life.pl	0.016	0.041	0
disj_r.pl	0.015	0.015	0	queens.pl	0.008	0.047	0
browse.pl	0.032	0.047	0	gabriel.pl	0.016	0.031	0
treesort.pl	0.015	0.172	0.5	dnf.pl	0.11	0.062	11
boyer.pl	0.047	0.062	0	tsp.pl	0.047	0.032	5
peephole.pl	0.078	0.125	0	kalah.pl	0.047	0.281	0
aiakl.pl	0.094	0.468	3	treeorder.pl	0.14	1.172	0
cs_r.pl	0.078	1.187	0	boyer.pl	1.547	7.86	0
peep.pl	1.86	137.797	0	read.pl	2.469	129.859	0
sdda.pl	5.453	151.68	0	ga.pl	0.281	209.687	0

There are 24 programs in table 1. The pair sharing information computed by *XSharing* is exactly same as $Sharing$ for 20 programs. For the programs *peepl.pl*, *sdda.pl*, *ga.pl* and *read.pl*, the performance is remarkably improved while we have exactly same pair sharing information. There is loss of precision for 4 programs. This is caused by ordering, join and projection operators. For program *treesort.pl*, the precision loss is 0.5% that is almost equal to zero. An interesting program is $dnf.pl$. We lost 11% precision for this program while $Sharing$ is faster than *XSharing*. This is a negative result for our analysis. We investigated this program and found out that there are many variables that are ground and thus closure under union operations are applied on small sets. This explains why $Sharing$ is actually faster. Therefore, in our opinion, our analysis is more suitable for complex programs that involve many variables and closure under union operations.

We can adjust the threshold k to control the complexity/precision ratio of the system. The threshold is 0 for all programs in table 1 and there are four programs that lose precision. We can find the minimal k that guarantees these programs do not lose precision. The minimal k are listed in the last column in table 2. The other columns are same as table 1.

Table 2. The Minimal Threshold without Losing Precision

	XSharing	*Sharing*	Precision	Threshold
Program	Time	Time	Pair lose	k
treesort.pl	0.328	0.172	0	16
dnf.pl	0.047	0.062	0	16
aiakl.pl	0.483	0.468	0	171
tsp.pl	0.020	0.032	0	23

There are many built-in predicates in a real program and it is important to consider the effect of them. Thus we have supported all built-in predicates in the benchmark programs. For example, we have considered built-in predicates such as $=/2$ and $sort/2$ whose effect is to apply abstract unification. We also processed built-in predicates such as $>/2$ and $ground/1$ whose effect is to ground variables. Other control built-in predicates such as $;/2$ and $\rightarrow/2$ are also processed. Note that considering the effect of built-in predicates with grounding effect can improve the efficiency significantly because many groups are removed.

9 Related work

There have been much research in sharing analysis of logic programs [17, 24, 4, 20, 8, 19, 9, 22, 14, 5] that have attempted to tame the computational aspects of this domain without compromising too much precision. One approach is reducing the number of groups that can possibly arise is due to Fecht [12] and Zaffanella et al. [25] who use maximal elements to represent downward closed powersets of variables. In fact Zaffanella et al. [25] apply a pair of sharing abstractions, which is almost same as our proposal. The key difference is interpretation. In [25], a clique represents all sharing groups that contained within it. In this paper, a clique is used to select the basis of a closure.

Bagnara et al. [3] also argue that *Sharing* is redundant for pair-sharing based on the following assumption: that the goal of sharing analysis for logic program is to detect which pairs of variables are definitely independent. A group G is redundant with respect to a sharing abstraction S if S already contains the pair sharing information that G contains. Thus they obtain a simpler domain and, perhaps more importantly, reduce complexity of abstract unification. That is, closure under union is not needed. This simpler domain is as precise as *Sharing* on pair-sharing. On the other hand, Bueno et al. [6] argue that the assumption in [3] is not always valid. First, there are applications that use *Sharing* other than retrieving sharing information between pairs of variables. Second, *Sharing* is more precise when combined with other source of information. In this paper, we consider a group is redundant if it can be regenerated using the cliques saved in first component. Moreover, the redundant concept is applied to cliques.

Howe and King [16] present three optimisations for *Sharing*. These optimisations can be used when combine *Sharing* with freeness and linearity. One principle in [16] is that the number of sharing groups should be minimised. This principle is also used in our approach, though we do not seek to exploit either freeness or linearity.

Another thread of work is in the use different encodings of the *Sharing* domain. In [8], Codish et al. show that *Sharing* is isomorphic to the domain of positive Boolean functions [1]. The closure under union operator over *Sharing* corresponds to downward closure operator $\downarrow$ over *Pos* that maps a positive Boolean function to a definite Boolean function. A definite Boolean function thus corresponds to a sharing that is closed under union. This domain is efficient because of efficient data structure of Boolean function. Howe et al. propose a downward closure operator $\downarrow$ in [15]. Codish et al. [7] also propose an algebraic approach to the sharing analysis using set logic programs. This domain is isomorphic to *Sharing* and leads to intuitive definitions for abstract operators.

10 Conclusion

This paper presents a new domain for sharing analysis based on a new efficient representation for *Sharing*. All key operators are polynomial with respect to the number of groups and soundness proofs are given. Moreover, by remembering the cliques, we provide a scheme to reason about the relationship between cliques such as finding redundant cliques. Lastly, there is a parameter to control the complexity/precision ratio of the system. For the future work, we will devote more efforts on the precision and find more redundant cliques. It looks promising to explore how a pair can be recovered to non-redundancy sharing [3] by changing the representation of first component too.

References

1. Tania Armstrong, Kim Marriott, Peter Schachte, and Harald Søndergaard. Two Classes of Boolean Functions for Dependency Analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
2. Roberto Bagnara, Roberta Gori, Patricia M. Hill, and Enea Zaffanella. Finite-Tree Analysis for Constraint Logic-Based Languages. *Information and Computation*, 193(2):84–116, 2004.
3. Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Set-Sharing is Redundant for Pair-Sharing. *Theoretical Computer Science*, 277(1-2):3–46, 2002.
4. Roberto Bagnara, Enea Zaffanella, and Patricia M. Hill. Enhanced Sharing Analysis Techniques: A Comprehensive Evaluation. *Theory and Practice of Logic Programming*, 5(1&2):1–43, 2005.
5. Maurice Bruynooghe, Bart Demoen, Dmitri Boulanger, Marc Denecker, and Anne Mulkers. A Freeness and Sharing Analysis of Logic Programs Based on A Preinterpretation. In Radhia Cousot and David A. Schmidt, editors, *Proceedings of the Third International Symposium on Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 128–142, London, UK, 1996. Springer-Verlag.

6. Francisco Bueno and Maria J. García de la Banda. Set-Sharing Is Not Always Redundant for Pair-Sharing. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Proceedings of 7th International Symposium on Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 117–131. Springer, 2004.
7. Michael Codish, Vitaly Lagoon, and Francisco Bueno. An Algebraic Approach to Sharing Analysis of Logic Programs. *Journal of Logic Programming*, 42(2):111–149, 2000.
8. Michael Codish, Harald Søndergaard, and Peter J. Stuckey. Sharing and Groundness Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.
9. Agostino Cortesi and Gilberto Filé. Sharing Is Optimal. *Journal of Logic Programming*, 38(3):371–386, 1999.
10. Lobel Crnogorac, Andrew D. Kelly, and Harald Sondergaard. A Comparison of Three Occur-Check Analysers. In Radhia Cousot and David A. Schmidt, editors, *Proceedings of Third International Symposium on Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 1996.
11. Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel Logic Programming Systems. *ACM Computing Surveys*, 26(3):295–336, 1994.
12. Christian Fecht. An Efficient and Precise Sharing Domain for Logic Programs. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 469–470. Springer, 1996.
13. Patricia M. Hill, Roberto Bagnara, and Enea Zaffanella. Soundness, Idempotence and Commutativity of Set-Sharing. *Theory and Practice of Logic Programming*, 2(2):155–201, 2002.
14. Patricia M. Hill, Enea Zaffanella, and Roberto 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.
15. J. M. Howe and A. King. Efficient Groundness Analysis in Prolog. *Theory and Practice of Logic Programming*, 3(1):95–124, January 2003.
16. Jacob M. Howe and Andy King. Three Optimisations for Sharing. *Theory and Practice of Logic Programming*, 3(2):243–257, 2003.
17. Dean Jacobs and Anno Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In Ross A. Overbeek Ewing L. Lusk, editor, *Proceedings of the North American Conference on Logic Programming*, pages 154–165. MIT Press, 1989.
18. Dean Jacobs and Anno Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2&3):291–314, 1992.
19. Andy King. Pair-Sharing over Rational Trees. *Journal of Logic Programming*, 46(1-2):139–155, 2000.
20. Vitaly Lagoon and Peter J. Stuckey. Precise Pair-Sharing Analysis of Logic Programs. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 99–108. ACM Press, 2002.
21. Anno Langen. *Advanced Techniques for Approximating Variables Aliasing in Logic Programs.* PhD thesis, 1991.
22. Kalyan Muthukumar and Manuel V. Hermenegildo. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2-3):315–347, 1992.

23. Ulf Nilsson. Towards a Framework for the Abstract Interpretation of Logic Programs. In Pierre Deransart, Bernard Lorho, and Jan Maluszynski, editors, *Proceedings of the Programming Language Implementation and Logic Programming*, volume 348 of *Lecture Notes in Computer Science*, pages 68–82. Springer, 1989.
24. Harald Søndergaard. An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction. In Bernard Robinet and Reinhard Wilhelm, editors, *Proceedings of the 1st European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 1986.
25. Enea Zaffanella, Roberto Bagnara, and Patricia M. Hill. Widening Sharing. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 414–432, London, UK, 1999. Springer-Verlag.