Two Efficient Representations for Set-Sharing Analysis in Logic Programs

Eric Trias, Jorge Navas, Elena S. Ackley, Stephanie Forrest

Department of Computer Science
University of New Mexico
Albuquerque, NM, USA

and Manuel V. Hermenegildo

Departments of Computer Science
University of New Mexico, Albuquerque, NM (USA),
Technical University of Madrid, Madrid (Spain) and IMDEA-Software.

Abstract

Set-Sharing analysis, the classic Jacobs and Langen's domain, has been widely used to infer several interesting properties of programs at compile-time such as occurs-check reduction, automatic parallelization, finite-tree analysis, etc. However, performing abstract unification over this domain implies the use of a closure operation which makes the number of sharing groups grow exponentially. Much attention has been given in the literature to mitigate this key inefficiency in this otherwise very useful domain. In this paper we present two novel alternative representations for the traditional set-sharing domain, *tSH* and *tNSH*, which compress efficiently the number of elements into fewer elements enabling more efficient abstract operations, including abstract unification, without any loss of accuracy. Our experimental evaluation supports that both representations can reduce dramatically the number of sharing groups showing they can be more practical solutions towards scalable set-sharing.

1 Introduction

In abstract interpretation [11] of logic programs sharing analysis has received considerable attention. Two or more variables in a logic program are said to *share* if in some execution of the program they are bound to terms which contain a common variable. A variable in a logic program is said to be *ground* if it is bound to a ground term in all possible executions of the program. A type of sharing analysis that has received significant attention is *set-sharing* analysis. *Set-sharing* analysis was originally introduced by Jacobs and Langen [16,18] and its abstract values are sets of sets of variables that keep track in a compact way of the sharing patterns among variables.

Example 1.1 (**Set-sharing using set of sets of variables**). Let $V = \{X_1, X_2, X_3, X_4\}$ be a set of variables of interest. The abstraction in *set-sharing* of

[1] The authors gratefully acknowledge the support of the National Science Foundation (grants CCR-0331580 and CCR-0311686, and DBI-0309147), the Santa Fe Institute, the Air Force Institute of Technology, the Prince of Asturias Chair in Information Science and Technology at UNM, and by EU projects 215483 *S-Cube*, IST-15905 *MOBIUS*, Spanish projects ITEA2/PROFIT FIT-340005-2007-14 *ES_PASS*, MEC TIN2005-09207-C03-01 *MERIT/COMVERS*, and Comunidad de Madrid project S-0505/TIC/0407 *PROMESAS*.

[2] Email: trias@cs.unm.edu

[3] Email: jorge@cs.unm.edu

[4] Email: elenas@cs.unm.edu

[5] Email: forrest@cs.unm.edu

[6] Email: herme@unm.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

a substitution such as $\theta = \{X_1 \mapsto f(U_1, U_2, V_1, V_2, W_1),\ X_2 \mapsto g(V_1, V_2, W_1), X_3 \mapsto g(W_1, W_1), X_4 \mapsto a\}$ will be $\{\{X_1\}, \{X_1, X_2\}, \{X_1,\ X_2, X_3\}\}$. Sharing group $\{X_1\}$ in the abstraction represents the occurrence of run-time variables U_1 and U_2 in the concrete substitution, $\{X_1, X_2\}$ represents V_1 and V_2, and $\{X_1, X_2, X_3\}$ represents W_1. Note that X_4 does not appear in the sharing groups because X_4 is *ground*. Note also that the number of (occurrences of) run-time variables shared is abstracted away.

Sharing and *groundness* have been used to infer several interesting properties of programs at compile-time; most notably but not limited to: occurs-check reduction (e.g., [27]), automatic parallelization (e.g., [25,24]), and finite-tree analysis (e.g., [2]). The accuracy of *set-sharing* has been improved by extending it with other kinds of information, the most relevant being *freeness* and *linearity* information [16,24,9,15], and also information about *term structure* [17,4,23]. Sharing in combination with other abstract domains has also been studied [8,14,10]. The significance of *set-sharing* is that it keeps track of *sharing* among *sets* of variables more accurately than other abstract domains such as *pair-sharing* [27] due to better groundness propagation and other factors that are relevant in some of its applications [6]. In addition, *set-sharing* has attracted much attention [7,10,3,6] because its algebraic properties allow elegant encodings into other efficient implementations (e.g., *ROBDDs* [5]). In [25,24], the first comparatively efficient algorithms were presented for performing the basic operations needed for implementing set sharing-based analyses.

However, *set-sharing* has intrinsically a key computational disadvantage: the *abstract unification* (*amgu*, for short) implies a potentially exponential growth in the number of sharing groups due to the *up-closure* (also called *star-union*) operation which is the heart of that operation. Considerable attention has been given in the literature to reducing the impact of the complexity of this operation. In [28], Zaffanella et al. extend the *set-sharing* domain for inferring *pair-sharing* from a set of sets of variables to a pair of sets of sets of variables in order to support widening. The key concept is that the set of sets in the first component (called *clique*) is reinterpreted as representing all sharing groups that are contained within it. Although significant efficiency gains are achieved, this approach loses precision with respect to the original *set-sharing*. A similar approach is followed in [26] but for inferring set-sharing in a top-down framework. Other relevant work was presented in [20] in which the *up-closure* operation was delayed and full sharing information was recovered lazily. However, this interesting approach shares some of the disadvantages of Zaffanella's widening. Therefore, the authors refined the idea in [19] reformulating the *amgu* in terms of the *closure under union* operation, collapsing those closures to reduce the total number of closures and applying them to smaller descriptions without loss of accuracy. In [10] the authors show that Jacobs and Langen's sharing domain is isomorphic to the dual negative of Pos [1], denoted by $\overline{coPos}$. This insight improved the understanding of sharing analysis, and led to an elegant expression of the combination with groundness dependency analysis based on the reduced product of Sharing and Pos. In addition, this work pointed out the possible implementation of $\overline{coPos}$ through ROBDDs leading to more efficient implementations of set-sharing analyses.

In this paper, we present a different approach in order to mitigate the computational inefficiencies of the *set-sharing* domain. We propose two novel representations that compress efficiently the number of elements into fewer elements enabling more efficient abstract operations without any loss of accuracy. The first representation, tSH, compacts the sharing relationships by eliminating redundancies among them. The second, $tNSH$, leverages the complement (or negative) sharing relationships of the original sharing set. Intuitively, let $sh_{\mathcal{V}}$ be a sharing set over the set of variables of interest $\mathcal{V}$, then $tNSH$ keeps track of $\wp(\mathcal{V}) \setminus sh_{\mathcal{V}}$. This new capability of $tNSH$ dramatically reduces the number of elements to represent as the cardinality of the original set grows toward $2^{|\mathcal{V}|}$. It is important to notice that our work is not based on [10]. Although they define the dual negated positive Boolean functions, $\overline{coPos}$ does not represent the entire complement of the positive set. Moreover, they do not use $\overline{coPos}$ as a means of compacting relationships but as a way of representing Sharing through Boolean functions. We also represent Sharing through Boolean functions, but that is where the similarity ends.

In the remainder of the paper we first describe Jacobs and Langen's *set-sharing* domain, bSH, adapted for handling binary strings in Section 2 and we extend it in Section 3 presenting tSH, a more compact representation. In Section 4, we introduce our next novel representation, $tNSH$, the complement (or negative) of the original set-sharing. Finally, we show our experimental evaluation of these representations in Section 5 and conclude in Section 6.

2 Set-Sharing Abstract Domain

The set-sharing domain was first presented by Jacobs and Langen in [16]. The presentation here follows that of [28,10], since the notation used and the abstract unification operation obtained are rather intuitive. Unless otherwise stated here and in the rest of paper we will represent the set-sharing domain using a set of strings rather than a set of sets of variables.

Definition 2.1 (Binary Sharing Domain, bSH). Let alphabet $\Sigma = \{0, 1\}$, $\mathcal{V}$ be a fixed and finite set of variables of interest in an arbitrary order, and Σ^l the finite set of all strings over Σ with length l, $0 \leq l \leq |\mathcal{V}|$. Let $bSH^l = \wp^0(\Sigma^l)$ be the *proper powerset* (i.e., $\wp(\Sigma^l) \setminus \{\emptyset\}$) that contains all possible combinations over Σ with length l. Then, the *binary sharing domain* is defined as $bSH = \bigcup_{0 \leq l \leq |\mathcal{V}|} bSH^l$.

Let $\mathcal{F}$ and $\mathcal{P}$ 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 $\mathcal{V}$ and $\mathcal{F} \cup \mathcal{P}$. 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 $\hat{t}$ by the binary representation of the set of variables of $t \in Term$ according to a particular order among variables. Since $\hat{t}$ will be always used through a bitwise operation with some string of length l, the length of $\hat{t}$ must be l. If not, $\hat{t}$ is adjusted with 0's in those positions associated with variables represented in the string but not in t.

The following definitions are an adaptation for the binary representation of the standard definitions for the sharing domain [16]:

Definition 2.2 (Binary relevant sharing $rel(bsh,t)$ and irrelevant sharing $irrel(bsh,t)$). Given $t \in Term$, the set of binary strings in $bsh \in bSH^l$ of length l that are relevant with respect to t is obtained by a function $rel(bsh,t) : bSH^l \times Term \rightarrow bSH^l$ defined as:

$$rel(bsh,t) = \{s \mid s \in bsh, (s \bigwedge \hat{t}) \neq 0^l\}$$

where $\bigwedge$ represents the bitwise AND operation and 0^l is the all-zeros string of length l. Consequently, the set of binary strings in $bsh \in bSH^l$ that are *irrelevant with respect to t is a function $irrel(bsh,t) : bSH^l \times Term \rightarrow bSH^l$ where $irrel(bsh,t)$ is the complement of $rel(bsh,t)$, i.e., $bsh \setminus rel(bsh,t)$.*

Definition 2.3 (Binary cross-union, $\bowtie$). Given $bsh_1, bsh_2 \in bSH^l$, their *cross-union* is a function $\bowtie : bSH^l \times bSH^l \rightarrow bSH^l$ defined as

$$bsh_1 \bowtie bsh_2 = \{s \mid s = s_1 \bigvee s_2, s_1 \in bsh_1, s_2 \in bsh_2\}$$

where $\bigvee$ represents the bitwise OR operation.

Definition 2.4 (Binary up-closure, $(.)^*$). Let l be the length of strings in $bsh \in bSH^l$, then the *up-closure* of bsh, denoted bsh^* is a function $(.)^* : bSH^l \rightarrow bSH^l$ that represents the smallest superset of bsh such that $s_1 \bigvee s_2 \in bsh^*$ whenever $s_1, s_2 \in bsh^*$:

$$bsh^* = \{s \mid \exists n \geq 1\ \exists t_1, \ldots, t_n \in bsh,\ s = t_1 \bigvee \ldots \bigvee t_n\}$$

Definition 2.5 (Binary abstract unification, $amgu$). The abstract unification is a function $amgu : \mathcal{V} \times Term \times bSH^l \rightarrow bSH^l$ defined as

$$amgu(x,t,bsh) = irrel(bsh, x = t) \ \cup \ (rel(bsh,x) \bowtie rel(bsh,t))^*$$

Example 2.6 (Binary abstract unification). Let $\mathcal{V} = \{X_1, X_2, X_3, X_4\}$ be the set of variables of interest and let $sh = \{\{X_1\}, \{X_2\}, \{X_3\}, \{X_4\}\}$ be a sharing set. Assume the following order among variables: $X_1 \prec X_2 \prec X_3 \prec X_4$. Then, we can easily encode each sharing group $sg \in sh$ into a binary string s such that $s[i] = 1$, $(1 \leq i \leq |sg|)$ if and only if the i-th variable of $\mathcal{V}$ appears in sg. In this example, sh is encoded as the following set of binary strings $bsh = \{1000, 0100, 0010, 0001\}$. Consider the analysis of $X_1 = f(X_2, X_3)$, the result is:

(i) $A = rel(bsh, X_1) = \{1000\}$ and
$B = rel(bsh, f(X_2, X_3)) = \{0100, 0010\}$

(ii) $A \bowtie B = \{1100, 1010\}$

(iii) $(A \bowtie B)^* = \{1100, 1010, 1110\}$

(iv) $C = irrel(bsh, X_1 = f(X_2, X_3)) = \{0001\}$

(v) $amgu(X_1, f(X_2, X_3), bsh) = C \ \cup \ (A \bowtie B)^* = \{0001, 1100, 1010, 1110\}$

The design of the analysis must be completed by defining the following abstract operations that are required by an analysis engine: *init* (initial abstract state), *equivalence* (between two abstract substitutions), *join* (defined as the union), and

project. In the interest of brevity, we define only the *project* operation since the other three operations are trivial.

Definition 2.7 (Binary projection, $bsh|_t$). The *binary projection* is a function $bsh|_t$: $bSH^l \times Term \rightarrow bSH^k$ $(k \leq l)$ that removes the i-th positions from all strings (of length l) in $bsh \in bSH^l$, if and only if the i-th positions of $\hat{t}$ (denoted by $\hat{t}[i]$) is 0, and it is defined as

$$bsh|_t = \{s' \mid s \in bsh, s' = \pi(s,t)\}$$

where $\pi(s,t)$ is the binary string projection defined as

$$\pi(s,t) = \begin{cases} \epsilon, & \text{if } s = \epsilon, \text{ the empty string} \\ \pi(s',t), & \text{if } s = s'a_i \text{ and } \hat{t}[i] = 0 \\ \pi(s',t)a_i, & \text{if } s = s'a_i \text{ and } \hat{t}[i] = 1 \end{cases}$$

and $s'a_i$ is the concatenation of character a to string s' at position i.

3 Ternary Set-Sharing Abstract Domain

In this section, we introduce a more efficient representation for the set-sharing domain defined in Sec. 2 to accommodate a larger number of variables for analysis. We extend the binary string representation discussed above to use a ternary alphabet $\Sigma_* = \{0, 1, *\}$, where the $*$ symbol denotes both 0 and 1 bit values. This representation effectively compresses the number of elements in the set into fewer strings without changing what is being represented (i.e., without loss of accuracy). To handle the ternary alphabet, we redefine the binary operations covered in Sec. 2.

Definition 3.1 (Ternary Sharing Domain, tSH). Let alphabet $\Sigma_* = \{0, 1, *\}$, $\mathcal{V}$ be a fixed and finite set of variables of interest in an arbitrary order as in Def. 2.1, and Σ_*^l the finite set of all strings over Σ_* with length l, $0 \leq l \leq |\mathcal{V}|$. Then, $tSH^l = \wp^0(\Sigma_*^l)$ and hence, the *ternary sharing domain* is defined as $tSH = \bigcup_{0 \leq l \leq |\mathcal{V}|} tSH^l$.

Prior to defining how to transform the binary string representation into the corresponding ternary string representation, we introduce two core definitions, Def. 3.2 and Def. 3.3, for comparing ternary strings. These operations are essential for the conversion and set operations. In addition, they are used to eliminate redundant strings within a set and to check for equivalence of two ternary sets containing different strings.

Definition 3.2 (Match, $\mathcal{M}$). Given two ternary strings, $x, y \in \Sigma_*^l$, of length l, *match* is a function $\mathcal{M} : \Sigma_*^l \times \Sigma_*^l \rightarrow \mathcal{B}$, such that $\forall i \ 1 \leq i \leq l$,

$$x\mathcal{M}y = \begin{cases} true, \text{if } (x[i] = y[i]) \vee (x[i] = *) \vee (y[i] = *) \\ false, \text{otherwise} \end{cases}$$

Definition 3.3 (Subsumed_By $\trianglelefteqq$ and Subsumed_In $\trianglerighteqq$). Given two ternary strings $s_1, s_2 \in \Sigma_*^l$, $\trianglelefteqq : \Sigma_*^l \times \Sigma_*^l \rightarrow \mathcal{B}$ is a function such that $s_1 \trianglelefteqq s_2$ if and only if every string matched by s_1 is also matched by s_2. More formally, $s_1 \trianglelefteqq s_2 \iff$

```
0  Convert(bsh, k)
1  tsh ← ∅
2  foreach s ∈ bsh
3      y ← PatternGenerate(tsh, s, k)
4      tsh ← ManagedGrowth(tsh, y)
5  return tsh

10 PatternGenerate(tsh, x, k)
11 m ← Specified(x)
12 i ← 0
13 x' ← x
14 l ← length(x)
15 while m > k and i < l
16     Let b_i be the value of x' at position i
17     if b_i = 0 or b_i = 1 then
18         x' ← x' · b̄_i
19         if x' ⊑ tsh then
20             x' ← x' · *_i
21         else
22             x' ← x' · b_i
23     m ← Specified(x')
24     i ← i + 1
25 return x'

30 ManagedGrowth(tsh, y)
31 S_y = {s | s ∈ tsh, s ⊑ y}
32 if S_y = ∅ then
33     if y ⋢ tsh then
34         append y to tsh
35 else
36     remove S_y from tsh
37     append y to tsh
38 return tsh
```

Fig. 1. A deterministic algorithm for converting a set of binary strings bsh into a set of ternary strings tsh, where k is the desired minimum number of specified bits (non-$*$) to remain.

$\forall s \in tSH^l,\ if\ \ s_1 \mathcal{M} s\ \ then\ \ s_2 \mathcal{M} s$. For convenience, we augment this definition to deal with sets of strings. Given a ternary string $s \in \Sigma_*^l$ and a ternary sharing set, $tsh \in tSH^l$, $\sqsubseteq : \Sigma_*^l \times tSH^l \rightarrow \mathcal{B}$ is a function such that $s \sqsubseteq tsh$ if and only if there exists some element $s' \in tsh$ such that $s \sqsubseteq s'$.

Figure 1 details the pseudo code for converting a set of binary strings into a set of ternary strings. The function `Convert` evaluates each string of the input and attempts to introduce $*$ symbols using `PatternGenerate`, while eliminating redundant strings using `ManagedGrowth`.

`PatternGenerate` evaluates the input string bit-by-bit to determine where the $*$ symbol can be introduced. The number of $*$ symbols introduced depends on the sharing set represented and k, the desired minimum number of specified bits, where $1 \leq k \leq l$ (the string length). For a given set of strings of length l, parameter k controls the compression of the set. For $k = l$ (all bits specified), there is no compression and $tsh = bsh$. For $k = 1$, the maximum number of $*$ symbols are introduced. For now, we will assume that $k = 1$, and some experimental results in Section 5 will show the best overall k value for a given l. The `Specified` function returns the number of specified bits (0 or 1) in x.

`ManagedGrowth` checks if the input string y subsumes other strings from tsh. If no redundant string exists, then y is appended to tsh only if y itself is not redundant to an existing string in tsh. Otherwise, all such redundant strings are removed from the set and replaced by y.

Example 3.4 (**Conversion from bSH to tSH**). Let $\mathcal{V}$ be the set of variables of interest with the same order as Example 2.6. Assume the following sharing set of binary strings bsh = {1000, 1001, 0100, 0101, 0010, 0001}. Then, a ternary string representation produced by applying `Convert` is tsh ={100*, 0010, 010*, *001}.

The example above begins with `Convert`(bsh,$k = 1$). Since $tsh = \emptyset$ initially (line 1), the first string 1000 is appended to tsh, so $tsh = \{1000\}$. Next, 1001 from bsh is evaluated. In `PatternGenerate`, with x' at iteration i (denoted as x'_i), $i = 3$ and $b_3 = 1$, we test $x'_3 = 1000$ if the i^{th} position of x can be replaced with a $*$ (line 15-24). In this case, since $x'_3 \sqsubseteq tsh$ (line 19), $x'_3 = 100*$ is returned

(line 25). Next, `ManagedGrowth` evaluates 100* and since it subsumes 1000 (S_y = {1000}), 100* replaces 1000 leaving tsh = {100*} (line 38). The process continues with `PatternGenerate`({100*},0100) (line 3). In `PatternGenerate`, since $x'_0 \not\sqsubseteq tsh$, $x'_1 \not\sqsubseteq tsh$, $x'_2 \not\sqsubseteq tsh$, and $x'_3 \not\sqsubseteq tsh$, we reset each i^{th} bit to its original value (line 22) and $x' = x = 0100$ is returned. Next, `ManagedGrowth`({100*},0100) is called and since 0100 is not redundant to any string in tsh, it is appended to tsh resulting in tsh = {100*,0100}. The process continues with `PatternGenerate`({100*,0100},0101). In `PatternGenerate`, when $x'_3 = 0100$ and since $x'_3 \sqsubseteq tsh$, then $x'_3 = 010^*$ is returned. `ManagedGrowth`({100*, 0100}, 010*) is called next and since 010* subsumes 0100 in tsh, it is replaced leaving tsh ={100*,010*} (line 38). The process continues similarly, for the remaining input strings in bsh obtaining the final result of tsh = {100*, 0010, 010*, *001}.

Next, we redefine the binary string operations to account for the $*$ symbol in a ternary string. Note that since the ternary representation extends the binary alphabet (i.e., binary is a subset of the ternary alphabet), ternary operations can also operate over strictly binary strings. For sake of simplicity, we will overload certain operators to denote operations involving both binary and ternary strings.

Definition 3.5 (Ternary-or $\bigvee$ and Ternary-and $\bigwedge$). Given two ternary strings, $x, y \in \Sigma_*^l$ of length l, *ternary-or* and *ternary-and* are two bitwise-or functions defined as $\bigvee, \bigwedge : \Sigma_*^l \times \Sigma_*^l \rightarrow \Sigma_*^l$ such that $z = x \bigvee y$ and $w = x \bigwedge y$, $\forall i\ 1 \leq i \leq l$, where:

$$z[i] = \begin{cases} * \text{ if } (x[i] = * \wedge y[i] = *) \\ 0 \text{ if } (x[i] = 0 \wedge y[i] = 0) \\ 1\ otherwise \end{cases} \qquad w[i] = \begin{cases} * \text{ if } (x[i] = * \wedge y[i] = *) \\ 1 \text{ if } (x[i] = 1 \wedge y[i] = 1) \\ \quad \vee\ (x[i] = 1 \wedge y[i] = *) \\ \quad \vee\ (x[i] = * \wedge y[i] = 1) \\ 0\ otherwise \end{cases}$$

Definition 3.6 (Ternary set intersection, $\cap$). Given $tsh_1,\ tsh_2 \in tSH^l$, $\cap : tSH^l \times tSH^l \rightarrow tSH^l$ is defined as

$$tsh_1\ \cap\ tsh_2 = \{r \mid r = s1\ \bigwedge\ s2, s1 \mathcal{M} s2, s1 \in tsh1, s2 \in tsh2\}$$

For convenience, we define two binary patterns, `0-mask` and `1-mask`, in order to simplify further operations. The former takes an l-length binary string s and returns a set with a single string having a 0 where $s[i] = 1$ and $*$'s elsewhere, $\forall i\ 1 \leq i \leq l$. The latter takes also an l-length binary string s, but returns a set of strings with a 1 where $s[i] = 1$ and $*$'s elsewhere, $\forall i\ 1 \leq i \leq l$. For instance, `0-mask`(0110) and `1-mask`(0110) return $\{*00*\}$ and $\{*1**, **1*\}$, respectively.

Definition 3.7 (Ternary relevant sharing $rel(tsh, t)$ and irrelevant sharing $irrel(tsh, t)$). Given $t \in Term$ with length l and $tsh \in tSH^l$ with strings of length l, the set of strings in tsh that are *relevant* with respect to t is obtained by a function $rel(tsh, t) : tSH^l \times Term \rightarrow tSH^l$ defined as

$$rel(tsh, t) = tsh \cap \texttt{1-mask}(\hat{t})$$

In addition, $irrel(tsh, t)$ is defined as

$$irrel(tsh, t) = (tsh \ \cap \ \texttt{1-mask}(\bar{\hat{t}})) \cap \texttt{0-mask}(\hat{t})$$

Ternary cross-union, $\bowtie$, and ternary up-closure, $(.)^*$, operations are as defined in Def. 2.3 and in Def. 2.4, respectively, except the binary version of the bitwise OR operator is replaced with its ternary counterpart defined in Def. 3.5 in order to account for the $*$ symbol. In addition, the ternary abstract unification ($amgu$) is defined exactly as the binary version, Def.2.5, using the corresponding ternary definitions.

Example 3.8 (**Ternary abstract unification**). Let tsh = {100*, 010*, 0010, *001} as in Example 3.4. Consider again the analysis of $X_1 = f(X_2, X_3)$, the result is:

(i) $A = rel(tsh, X_1) = \{100^*\}$ and
$B = rel(tsh, f(X_2, X_3)) = \{010^*, 0010\}$

(ii) $A \bowtie B = \{110^*, 101^*\}$

(iii) $(A \bowtie B)^* = \{110^*, 101^*, 111^*\}$

(iv) $C = irrel(tsh, X_1 = f(X_2, X_3)) = \{0001\}$

(v) $amgu(X_1, f(X_2, X_3), tsh) = C \ \cup \ (A \bowtie B)^* = \{0001, 110^*, 101^*, 111^*\}$

Ternary projection, $tsh|_t$, is defined similarly as binary projection, see Def. 2.7. However, the projection domain and range is extended to accommodate the $*$ symbol. So, the function definition remains the same except that *ternary* string projection is now defined as a function $\pi(s, t)$: $\Sigma_*^l \times Term \rightarrow \Sigma_*^k$, $k \leq l$. For example, let tsh = {100*, 010*, 0010, *001} as in Example 3.4. Then, the projection of tsh over the term $t = f(X_1, X_2, X_3)$ is $tsh|_t$ = {100, 010, 001}. Note that since a string of all 0's is meaningless in a set-sharing representation, it is not included here.

Definition 3.9 (Ternary initial state, $init$). The *initial state* $init : \mathcal{V} \times \mathcal{I}^+ \rightarrow tSH^{|\mathcal{V}|}$ describes an empty substitution given a set of variables of interest. Assuming the binary initial state operation defined as $init_{bSH} : \mathcal{V} \rightarrow bSH^{|\mathcal{V}|}$, the ternary initial state can be defined using the `Convert` algorithm in Fig. 1 as:

$$init(\mathcal{V}, k) = \texttt{Convert}(init_{bSH}(\mathcal{V}), k)$$

Definition 3.10 (Ternary equivalence, $\equiv$). Given tsh_1, $tsh_2 \in tSH^l$, the sets are *equivalent* if and only $(\forall t_1 \in tsh_1, \forall s_1 \subseteqq t_1, \ s_1 \subseteqq tsh_2) \ \wedge \ (\forall t_2 \in tsh_2, \forall s_2 \subseteqq t_2, \ s_2 \subseteqq tsh_1)$.

Finally, the ternary join is defined as its binary counterpart, i.e., union.

4 Negative Ternary Set-Sharing Abstract Domain

In this section, we describe a further step using the ternary representation discussed in the previous section. In certain cases, a more compact representation of sharing relationships among variables can be captured equivalently by working with the complement (or negative) set of the original sharing set. A ternary string t can either be in or *not in* the set $tsh \in tSH$. This mutual exclusivity together with the finiteness of $\mathcal{V}$ allows for checking t's membership in tsh by asking if t is in tsh, or,

equivalently, if t is *not in* its complement, $\overline{tsh}$. Given a set of l-bit binary strings, its complement or negative set contains *all* the l-bit ternary strings *not* in the original set. Therefore, if the cardinality of a set is greater than half of the maximum size (i.e., $2^{|\mathcal{V}|-1}$), then the size of its complement will not be greater than $2^{|\mathcal{V}|-1}$. It is this size differential that we leverage to our advantage. In set-sharing analysis, as we consider programs with larger numbers of variables of interest, the potential number of sharing groups grows exponentially, toward $2^{|\mathcal{V}|}$, and the number of sharing groups *not in* the sharing set decreases toward 0.

The idea of a negative set representation and its associated algorithms extends the work by Esponda et al. in [12,13]. In that work, a negative set is generated from the original set in a similar manner as the conversion algorithms shown in Fig. 1 and 2. However, they produce a negative set with unspecified bits in random positions and with less integrated emphasis in managing the growth of the resulting set. The technique was originally introduced as a means to generate Boolean satisfiability (SAT) formulas. By leveraging the difficulty of finding solutions to hard SAT instances, they essentially are able to secure the contents of the original set, without the use of encryption [12]. In addition, these hard-to-reverse negative sets are still able to answer membership queries efficiently but remain intractable to reverse (to obtain the contents of the original set). In this paper, we disregard this security property, and use the negative approach to address the efficiency issues faced by the traditional set-sharing domain.

The conversion to the negative set can be accomplished using the two algorithms shown in pseudo code in Figure 2. `NegConvert` uses the `Delete` operation to remove input strings of the set sh from $\mathcal{U}$, the set of all l-bit strings $\mathcal{U} = \{*^l\}$, and then, the `Insert` operation to return $\mathcal{U} \setminus sh$ which represents all strings *not* in the original input. Alternatively, `NegConvertMissing` uses the `Insert` operation directly to append each string *missing* from the input set to an empty set resulting in a representation of all strings *not* in the original input. Although as shown in Table 1 both algorithms have similar time complexities, depending on the size of the original input, it may be more efficient to find all the strings missing from the input and transform them with `NegConvertMissing`, rather than applying `NegConvert` to the input directly. Note that the resulting negative set will use the same ternary alphabet described in Def. 3.1. For clarity, we will denote it by $tNSH$ such that $tNSH \equiv tSH$.

For simplicity, we only describe `NegConvert` since `NegConvertMissing` uses the same machinery. Assume a transformation from bsh to $tnsh$ calling to `NegConvert` with $k = 1$. We begin with $tnsh = \mathcal{U} = \{****\}$ (line 1), then incrementally `Delete` each element of bsh from $tnsh$ (line 2-3). `Delete` removes all strings matched by x from $tnsh$ (line 11-12). If the set of matched strings, D_x, contains unspecified bit values (* symbol), then all string combinations *not* matching x must be reinserted back into $tnsh$ (line 13-17). Each string y' not matching x is found by setting the unspecified bit to the opposite bit value found in $x[i]$ (line 16). Then, `Insert` ensures string y' has at least k specified bits (line 22-26). This is done by specifying $k - m$ unspecified bits (line 23) and appending each to the result using `ManagedGrowth` (line 24-26). If string x already has at least k specified bits, then the algorithm attempts to introduce more $*$ symbols using `PatternGenerate`

```
0  NegConvert(sh, k)
1  tnsh ← U
2  foreach t ∈ sh
3      tnsh ← Delete(tnsh, t, k)
4  return tnsh
```

```
0  NegConvertMissing(bsh, k)
1  tnsh ← ∅
2  bnsh ← U \ bsh
3  foreach t ∈ bnsh
4      tnsh ← Insert(tnsh, t, k)
5  return tnsh
```

```
10 Delete(tnsh, x, k)
11 D_x ← ∀t ∈ tnsh, x ℳ t
12 tnsh' ← tnsh with D_x removed
13 foreach y ∈ D_x
14     foreach unspecified bit position q_i of y
15         if b_i (the i^th bit of x) is specified, then
16             y' ← y · (q_i = b̄_i)
17             tnsh' ← Insert(tnsh', y', k)
18 return tnsh'
```

```
20 Insert(tnsh, x, k)
21 m ← Specified(x)
22 if m < k then
23     P ← select (k − m) unspecified bit positions in x
24     foreach possible bit assignment V_P of the selected positions
25         y ← x · V_P
26         tnsh' ← ManagedGrowth(tnsh, y)
27 else
28     y ← PatternGenerate(tnsh, x, k)
29     tnsh' ← ManagedGrowth(tnsh, y)
30 return tnsh'
```

Fig. 2. `NegConvert`, `NegConvertMissing`, `Delete` and `Insert` algorithms used to transform positive to negative representation; k is the desired number of specified bits (non-*'s) to remain.

(line 28) and appends it while removing any redundancy in the resulting set using `ManagedGrowth` (line 29).

Example 4.1 (**Conversion from bSH to tNSH**). Given the same sharing set as in Example 3.4: bsh = {1000, 1001, 0100, 0010, 0101, 0001}. A negative ternary string representation is generated by applying the `NegConvert` algorithm to obtain {0000, 11**, 1*1*, *11*, **11}. Since a string of all 0's is meaningless in a set-sharing representation, it is removed from the set. So, $tnsh$ = {11**, 1*1*, *11*, **11}.

For Example 4.1, the first string 1000 is deleted from $\mathcal{U} = \{****\}$. So, $D_x = \{****\}$ (line 11) and $tnsh' = \emptyset$ (line 12). For each i^{th} bit of x, a new $y'_i \mathcal{M} x$ is evaluated for insertion into the result set. So, `Insert` ($\emptyset$, $y'_0 = 0***$, $k = 1$) is called (line 17). Since `Specified`(y') $\geq k$ and $tnsh' = \emptyset$, the result returned is $tnsh'$ ={0***} (line 27-30). For all other unspecified positions (line 14) of y, a new string is created with a bit value opposite of x_i's value, ($\overline{b_i}$). So, `Insert` ({0***}, $y'_1 = *1**$, $k = 1$) is called next and y'_1 is appended to $tnsh'$. The process continues with y'_2 and y'_3 resulting in $tnsh$ = {0***, *1**, **1*, ***1}.

Next, 1001 from bsh is deleted (line 2) resulting in D_x ={***1} and $tnsh'$ = {0***, *1**, **1*} (line 11,12). Then, `Insert` ({0***, *1**, **1*}, y' = 0**1, $k = 1$) is called. Since 0**1 $\sqsubseteq$ $tnsh'$, then $tnsh'$ remains unchanged. The process continues with y'_1 =*1*1, y'_2 =**11 being subsumed by $tnsh'$; so the result returned is $tnsh$ = {0***, *1**, **1*}. Next, 0100 is deleted resulting in $tnsh$ = {00**, 0**1, 11**, *1*1, **1*}. Next, 0010 is deleted resulting in $tnsh$ = {000*, 0**1, 11**, 1*1*, *11*, *1*1, **11}. Next, 0101 is deleted resulting in $tnsh$ = {000*, 00*1, 11**, 1*1*, *11*, **11}. Finally, 0001 is deleted resulting in $tnsh$ = {0000, 11**, 1*1*, *11*, **11}. Removing the all 0 string, we get the final $tnsh$ = {11**, 1*1*, *11*, **11}. Notice that $tnsh = \mathcal{U} \setminus (bsh \cup \{0000\})$.

`NegConvertMissing` would return the same result for Example 4.1, and in gen-

Input	Convert Operation	Result	Description	Time Complexity	Size Complexity
bsh	`Convert`	tsh	bSH to tSH	$O(\|bsh\|\alpha l)$	$O(\|bsh\|)$
bsh/tsh	`NegConvert`	$tnsh$	pos. to neg.	$O(\|bsh\|\alpha(\alpha 2^{\delta}+1))$	$O(\|tnsh\|(l-m)2^{\delta})$
$tnsh$	`NegConvert`	tsh	neg. to pos.	$O(\|tnsh\|\alpha(\alpha 2^{\delta}+1))$	$O(\|tsh\|(l-m)2^{\delta})$
bsh	`NegConvertMissing`	$tnsh$	pos. to neg.	$O(\beta+\|bnsh\|(\alpha 2^{\delta}+1))$	$O(\|bnsh\|2^{\delta})$

Table 1
Summary of conversions: l-length strings; $\alpha = |Result| \cdot l$; if $m < k$ then $\delta = k - m$ else $\delta = 0$, where $m =$ minimum specified bits in entire set, $k =$ number of specified bits desired; $bnsh = \mathcal{U} \setminus bsh$; $\beta = O(2^l)$ time to find $bnsh$.

eral, an equivalent negative representation. Table 1 illustrates the different transformation functions and their results for a given input and convert operation. Rows 3 and 5 show that both `NegConvert` and `NegConvertMissing` can convert a positive representation into negative with corresponding difference in time complexity. Depending on the size of the original input we may prefer one transformation over another. If the input size is relatively small $< 50\%$ of the maximum size, then `NegConvert` is often more efficient than `NegConvertMissing`. Otherwise, we may prefer to insert those strings missing in the input set. In our implementation, we continuously track the size of the relationships to choose the most efficient transformation.

Consider now the same set of variables and order among them as in Example 4.1 but with a slightly different set of sharing groups encoded as $bsh = \{1000, 1100, 1110\}$ or $tsh = \{1*00, 1110\}$. Then, a negative ternary string representation produced by `NegConvert` is $tnsh = \{00**, 01**, 0*1*, 0**1, 1**1, *01*\}$. This example shows that the number of elements, or size, of the negative result, $|tnsh| = 6 > |bsh| = 3$ and $|tsh| = 2$. However, in Example 4.1 when $|bsh| = 6$, $|tnsh| = 4 < |bsh|$. This is because when $|bsh|$ is less than $2^{|\mathcal{V}|-1}$, i.e., $|bsh| = 3 < 2^3$, then its complement set must represent $(2^{|\mathcal{V}|} - |bsh|) = 13$ elements. Depending on the strings in the positive set, the size of negative result may indeed be greater. This is a good illustration of how selecting the appropriate set-sharing representation will affect the size of the converted result. We want to leverage the size of the original sharing set at specific program points in the analysis to produce the most compact working set. The negative sharing set representation results in the ability to represent more variables of interest enabling larger problem instances to be evaluated.

We now define negative operations in order to perform abstract unification and the rest of the abstract operations required by our engine using this negative representation.

Definition 4.2 (Negative relevant sharing $\overline{rel}(tnsh, t)$ and irrelevant sharing $\overline{irrel}(tnsh, t)$) Given $t \in Term$ and $tnsh \in tNSH^l$ with strings of length l, the set of strings in $tnsh$ that are *negative relevant* with respect to t is obtained by a function $\overline{rel}(tnsh, t) : tNSH^l \times Term \to tNSH^l$ defined as

$$\overline{rel}(tnsh, t) = tnsh \mathbin{\overline{\cap}} \texttt{0-mask}(\hat{t}),$$

where $\overline{\cap}$ is the negative intersection of two negative sets, as defined in [13]. In addition, $\overline{irrel}(tnsh, t)$ is defined as

$$\overline{irrel}(tnsh, t) = tnsh \mathbin{\overline{\cap}} \texttt{1-mask}(\hat{t}).$$

The negative representation, the complement of a set, provides a more compact representation for large positive set-sharing instances. This has enabled us to efficiently conduct operations in the negative that are more memory and computationally expensive in the positive. However, the negative representation does have its own drawbacks. Certain operations that are straightforward in the positive representation are $\mathcal{NP}$-Hard in the negative representation [12,13]. A key observation given in [12] is that there is a mapping from Boolean formulae to the negative set-sharing domain such that finding which strings are not represented is equivalent to finding satisfying assignments to the corresponding Boolean formula, which is known to be an $\mathcal{NP}$-Hard problem. This mapping is defined as follows.

Let $tnsh = \{11^{**}, 1^{*}1^{*}, {}^{*}11^{*}, {}^{**}11\}$ be the same sharing set as in Example 4.1. Its equivalent Boolean formula $\phi \equiv$ not $[(x_1$ and $x_2)$ or $(x_1$ and $x_3)$ or $(x_2$ and $x_3)$ or $(x_3$ and $x_4)]$ is defined over the set of variables $\{x_1, x_2, x_3, x_4\}$. The formula ϕ is mapped into a negative set-sharing instance where each clause corresponds to a string and each variable in the clause is represented as a 0 if it appears negated, as a 1 if it appears un-negated, and as a * if it does not appear in the clause. By applying DeMorgan's law, we can convert ϕ to an equivalent formula in conjunctive normal form. Then, it is easy to see that a satisfying assignment of the formula such as $\{x_1 = true, x_2 = false, x_3 = false, x_4 = true\}$ corresponding to the string 1001 is not represented in the negative set-sharing instance.

Due to the interdependent nature of the relationship between the elements of a negative set, it is unclear how or how efficiently a precise negative cross-union can be accomplished without going through a positive representation. Therefore, we accomplish the negative cross-union by first identifying the represented positive strings and then applying cross-union accordingly.

Rather than iterating through all possible strings in $\mathcal{U}$ and performing cross-union on strings not in $tnsh$, we achieve a more efficient negative cross-union, $\overline{\boxtimes}$, by converting $tnsh$ to tsh first, i.e., using `NegConvert` from Table 1 and performing ternary cross-union on strings $t \in tsh$. In this way, the ternary representation continues to provide a compressed representation of the sharing set. Note that negative up-closure operation, $\overline{{}^{*}}$, suffers the same drawback as cross-union. Therefore, we deal with it in the same way as the negative cross-union.

Definition 4.3 (Negative abstract unification, $\overline{amgu}$). The *negative abstract unification* is a function $\overline{amgu} : \mathcal{V} \times Term \times tNSH^l \rightarrow tNSH^l$ defined as

$$\overline{amgu}(x, t, tnsh) = \overline{irrel}(tnsh, x = t) \,\overline{\cup}\, (\overline{rel}(tnsh, x) \;\overline{\boxtimes}\; \overline{rel}(tnsh, t))^{\overline{*}},$$

where $\overline{\cup}$ is the negative set union as defined in [13].

Example 4.4 (Negative abstract unification). Let $tnsh = \{11^{**}, 1^{*}1^{*}, {}^{*}11^{*}, {}^{**}11\}$ be the same sharing set as in Example 4.1. Consider the analysis of $X_1 = f(X_2, X_3)$, the result is:

(i) $A = \overline{rel}(tnsh, X_1) = \{11^{**}, 1^{*}1^{*}, {}^{*}11^{*}, {}^{**}11, 0^{***}\}$
$B = \overline{rel}(tnsh, f(X_2, X_3)) = \{11^{**}, 1^{*}1^{*}, {}^{*}11^{*}, {}^{**}11, {}^{*}00^{*}\}$

(ii) $A \,\overline{\boxtimes}\, B = \{00^{**}, 01^{**}, 0^{*}0^{*}, {}^{*}00^{*}\}$

(iii) $(A \,\overline{\boxtimes}\, B)^{\overline{*}} = \{01^{**}, 0^{*}1^{*}, 100^{*}\}$

(iv) $C = \overline{irrel}(tnsh, X_1 = f(X_2, X_3)) = \{11**,1*1*,*11*,**11,1***,*1**,**1*\}$
$= \{1***,*1**,**1*\}$

(v) $\overline{amgu}(X_1, f(X_2, X_3), tnsh) = C \,\overline{\cup}\, (A \bowtie B)^{\overline{*}} = \{01**,\ 0*1*,\ 0**0,\ 100*\}$

Definition 4.5 (Negative projection, $\overline{tnsh|_t}$). The *negative projection* is a function $\overline{tnsh|_t}$: $tNSH^l \times Term \to tNSH^k$ $(k \leq l)$ that selects elements of $tnsh$ projected onto the binary representation of $t \in Term$ and is defined as

$$\overline{tnsh|_t} = \overline{\pi}(tnsh, \Upsilon_t),$$

where Υ_t is equal to all i^{th}-bit positions of $\hat{t}$ where $\hat{t}[i] = 1$ and $\overline{\pi}$ is the negative project operation, as defined in [13].

Example 4.6 (**Negative projection**). Let $tnsh$ = {11**, 1*1*, *11*, **11} be the same sharing set as in Example 4.1. The negative projection of $tnsh$ over the term $t = f(X_1, X_2, X_3)$ is $\overline{tnsh|_t}$ = {11*, 1*1, *11}. String **1 is not in the result because it represents the following strings when fully specified {001, 011, 101, 111} and not all these strings are in the complement, e.g., 001 is in the positive result of the same projection over bsh.

Definition 4.7 (Negative initial state, $\overline{init}$). The *negative initial state* $\overline{init} : \mathcal{V} \times \mathcal{I}^+ \to tNSH^{|\mathcal{V}|}$ describes an initial substitution given a set of variables of interest. Assuming as in Def. 3.9 the binary initial state operation $init_{bSH} : \mathcal{V} \to bSH^{|\mathcal{V}|}$, the negative initial state can be defined using both the `NegConvert` and `NegConvertMissing` algorithms described in Fig. 2 (denoted by $\overline{\texttt{Convert}}$) as follows:

$$\overline{init}(\mathcal{V}, k) = \overline{\texttt{Convert}}(init_{bSH}(\mathcal{V}), k)$$

Definition 4.8 (Negative set equivalence, $\equiv$). Given $tnsh_1, tnsh_2 \in tNSH^l$, they are *equivalent* if and only if $(\forall t_1 \in tnsh_1, \forall s_1 \trianglelefteq t_1, s_1 \trianglelefteq\!\!\!\!/\ tnsh_2) \ \wedge\ (\forall t_2 \in tnsh_2, \forall s_2 \trianglelefteq t_2, s_2 \trianglelefteq\!\!\!\!/\ tnsh_1)$.

Definition 4.9 (Negative join, $\overline{\sqcup}$). Given $tnsh_1, tnsh_2 \in tNSH^l$, the *negative join* function $\overline{\sqcup} : tNSH^l \times tNSH^l \to \wp^0(tNSH^l)$ is defined as the negative set union of the two sets, i.e., $tnsh_1 \,\overline{\cup}\, tnsh_2$.

5 Experimental Results

We have developed a proof-of-concept implementation, which is currently being optimized, in order to measure experimentally the relative efficiency obtained with the inclusion of the two new representations presented in this paper, tSH and $tNSH$, as alternatives to the traditional set-sharing domain. In this preliminary prototype we have used *Patricia tries* [22] to handle efficiently binary and ternary strings, and a naive bottom-up fixpoint for testing real programs.

Our first objective is to study the implications of the conversions in the representation for analysis. Note that although both tSH and $tNSH$ do not imply a loss

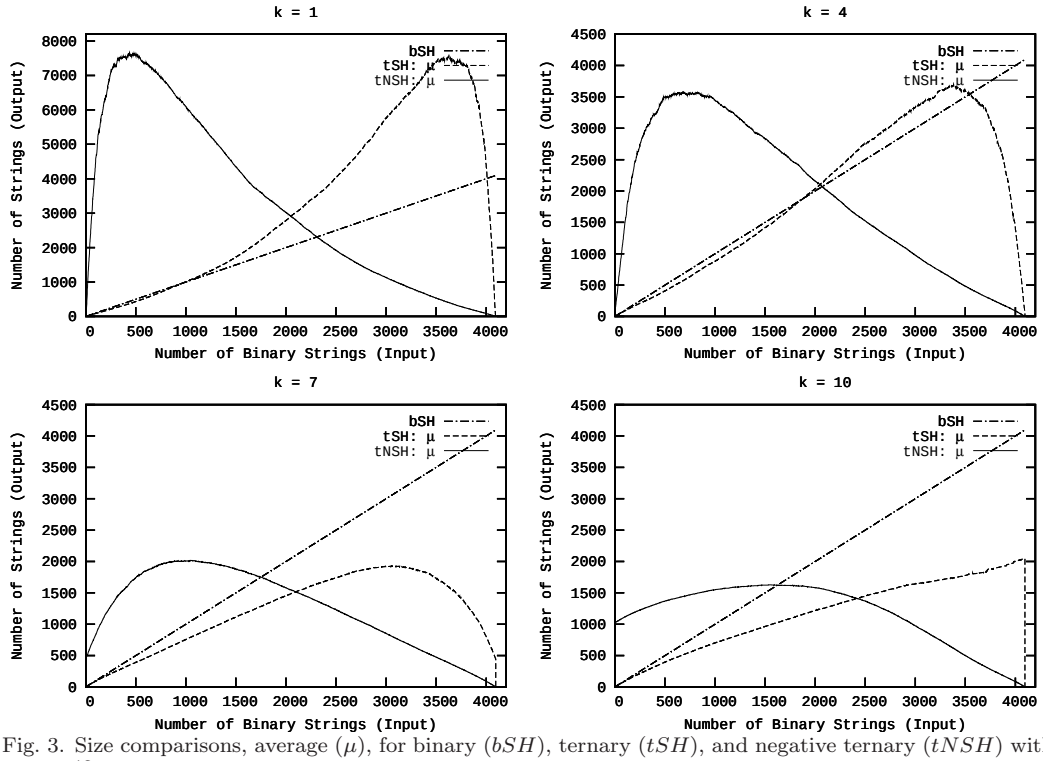

Fig. 3. Size comparisons, average (μ), for binary (bSH), ternary (tSH), and negative ternary ($tNSH$) with up to 2^{12} sharing relationships for k = 1, 4, 6, and 9.

of precision, the sizes of the resulting representations can vary significantly from one to another. An essential part will be to show experimentally the best overall k parameter for the conversion algorithms. Second, we study the core abstract operation of the traditional set-sharing, $amgu$, expressing its performance considering a notion of memory consumption, size of the representation (in terms of number of strings) during key steps in the unification. All experiments were performed with up to 2^{12} sharing relationships since we consider this value characteristic enough to show all the relevant features of our representations. In general, within some upper bound, the more variables considered the better the efficiency expected.

Our first experiment determines the best k value suitable for the conversion algorithms, shown in Figs. 1 and 2. We proceed by submitting a set of 12-bit strings in random order using different k values. We evaluate the size of the results for the smallest output size (see Fig. 3) for a given k value. As expected, bSH ($x = y$ line) results in no compression; tSH slowly increases from left to right remaining below bSH (for $k = 6$ and $k = 9$) due to the compression provided by the $*$ symbol and by having little redundancy; $tNSH$, the complement set, starts larger than bSH but quickly tapers off as the input size increases pass 50% of $|\mathcal{U}|$. Since the k parameter helps determine the minimum number of specified bits in the set, there is a direct relationship between the k parameter and the size of the output due to compression by the $*$ symbol. A smaller k value, i.e., $k = 1$, introduces the maximum number of $*$'s in the set. However, for a given input, a small k value does not necessarily result in the best compression factor (see $k = 1$ of Fig. 3). This result may be counter-intuitive, but it is due to the potentially larger number of unmatched strings that must be re-inserted back into the set determined by all the

bSH	tSH	$tNSH$
Initial Num Strings	**Num. Strings**	**Num. Strings**
2048	1397	1379
2457	1645	1123
2867	1846	860
3276	1986	587
3685	1913	300
4095	3285	1

Table 2
Size for conversion algorithms with up to 2^{12} sharing relationships, $k = 7$.

strings that must be represented by the converted result, see line 13-17 of Fig. 2.

We have found empirically that a k setting near (or slightly larger than) $l/2$ is the best *overall* value considering both the result size and time complexity. We use $k = 7$ in the following experiments below. It is interesting to note that a k value of $log_2(l)$ results in polynomial time conversion of the input (see the Complexity column of Table 1) but it may not result in the maximum compression of the set (see $k = 4$ of Fig. 3). Therefore, k may be adjusted to produce results based on acceptable performance level depending on which parameter is more important to the user, the level of compression (memory constraints) or execution time.

Our second experiment shows in Table 2 the comparison between the conversion algorithms to transform an initial set of binary strings, bSH, into its corresponding set of ternary strings, tSH, or its complement (negative), $tNSH$. Recall that the number of variables used is 12, hence the size of the input binary set might vary from 0 to 4095 (there is no representation for the zero string). Since a basic assumption in this work is the analysis of programs in which there is a large set of sharing relationships (i.e., scalable set-sharing), we measure our experiments by starting at 50% of the maximum size (i.e. 2048). The first column shows the size of the input binary set which varies approximately from 50% (2048) to 100% (4095). The second and third columns illustrate the sizes of the sets after the conversions from bSH into tSH and $tNSH$, respectively. These conversions are performed by the `Convert` algorithm described in Fig. 1 for tSH, and `NegConvertMissing` in Fig. 2 for $tNSH$, using $k = 7$. Table 2 shows that our two representations proposed can reduce dramatically the size of the input set. For example, at 90% (3685) of the binary set size, tSH compacts by 51% and $tNSH$ by 92% of the initial input size. This difference between tSH and $tNSH$ is even larger when the binary set size is 4095 since using $k = 7$ a more compression for tSH is not possible. Note also the efficiency of tSH and $tNSH$ compressing the initial input depends on its input size. If the size of bSH is approximately 50% of the total, then the level of compression is relatively similar. This fact makes sense since it was expected these two representations would behave similarly when the size of the positive and negative images were close to 50%. Significant gains in compression of $tNSH$ with respect to tSH are observed when the input size increases above 50%. Once again, notice at the 90% (3685) point, the compression ratio from bSH to $tNSH$ is almost seven times more compact as compared to bSH to tSH. Again, at 100% (4095) this difference between $tNSH$ and tSH is remarkably significant, $1 : 3285$.

Our third experiment shows in Table 3 the efficiency in terms of the level of

	Pre-*amgu*	**Post-*amgu***				
	Converted Size (% of $\|\mathcal{U}\|$)	t_1	$t_{m/4}$	$t_{m/2}$	$t_{3m/4}$	t_m
bSH	2048 (50%)	1535(14)	1909(9)	2017(8)	2028(7)	2033(8)
	2457 (60%)	1642(14)	1942(8)	2029(7)	2038(6)	2040(6)
	2867 (70%)	1742(12)	1968(7)	2035(4)	2042(3)	2044(4)
	3276 (80%)	1843(8)	1995(6)	2042(2)	2045(2)	2047(4)
	3685 (90%)	1945(6)	2021(4)	2044(1.4)	2046(.7)	2047(.7)
	4095 (99%)	2047(0)	2047(.5)	2047(.17)	2047(0)	2047(0)
tSH	1397 (50%)	408(14)	117(12)	39(13)	33(10)	28(11)
	1645 (60%)	494(18)	130(11)	31(11)	21(7)	18(6)
	1846 (70%)	568(16)	136(11)	25(8)	15(5)	13(4)
	1986 (80%)	620(21)	133(12)	19(4)	12(2)	11(1)
	1913 (90%)	586(32)	119(15)	18(4)	11(1)	11(0)
	3285 (99%)	15(6)	13(4)	11(0)	11(0)	11(0)
tNSH	1379 (50%)	745(88)	200(19)	43(10)	26(7)	16(8)
	1123 (60%)	619(31)	163(17)	31(7)	19(4)	12(6)
	860 (70%)	462(22)	123(14)	24(5)	16(3)	11(4)
	587 (80%)	310(14)	83(10)	18(2)	13(2)	12(4)
	299 (90%)	162(10)	47(6)	14(2)	12(1)	12(3)
	1 (99%)	5(1)	6(1)	9(.4)	11(0)	13(0)

Table 3
For up to 2^{12} sharing relationships with various t values (30 runs each): comparing average size, and standard deviation before and after *amgu* with $k = 7$.

compression of tSH and $tNSH$ performing the major abstract operation of the Jacobs and Langen's set-sharing domain: the abstract unification *amgu*. Another reason for testing *amgu*, rather than others such as *projection*, *join*, etc., is because *amgu* may affect more significantly the size of the abstract substitutions than those operations. The experiment has been carried out as follows. Given an arbitrary set of variables of interest $\mathcal{V}$ such that $|\mathcal{V}| = l = 12$, we constructed $x \in \mathcal{V}$ by selecting one variable and $t \in Term$ as a term consisting of a subset of the remaining variables, i.e., $\mathcal{V} \setminus \{x\}$. We tested with different values of t. Let $m = l - 1$ and $|.|_{ones} : BS \to \mathcal{I}^+$ a function that returns the number of 1's in a binary string, then t_1 represents $|\hat{t}|_{ones} = 1$, $t_{m/4}$ means $|\hat{t}|_{ones} = \lfloor 11/4 \rfloor$, and so on. Another important aspect that affects the *amgu* performance is the input sharing set, bSH. In order to reduce the effect of the input set in the *amgu* results we generated randomly 30 different sets which varies from 50% to 100% of the total size, 4095. Column **Pre-*amgu*** shows the number of input strings for bSH, and for tSH and $tNSH$, after the conversion. The data shown in this column is the same as in Table 2, but it is given again for clarity. Column **Post-*amgu*** provides the average number of strings and its standard deviation (in parenthesis) for each values of t, after running the abstract unification using 30 different input sets (bSH, tSH, and $tNSH$).

Firstly, Table 3 shows clearly that after *amgu* both tSH and $tNSH$ always yield dramatically less number of strings than bSH. In our experiment, the level of compression for tSH and $tNSH$ varies from 50% until 99% as compared to bSH. We also experienced that the bigger the size of the input and more variables are involved in the *amgu*, and the smaller the size after the *amgu* for tSH and $tNSH$. However, this trend is inverse in bSH: the bigger is the size of the input, the bigger is the size after the *amgu*.

The second relevant component of this experiment is to compare the performance between tSH and $tNSH$. For values of t_1, $t_{m/4}$, and $t_{m/2}$, the break-even point p is

around between 60% and 70% of $|\mathcal{U}|$. That is, tSH compresses more effectively the number of strings after unification at a size smaller than p, but it is significantly improved by $tNSH$ with sizes bigger than p. However, for the rest of t values ($t_{3m/4}$ and t_m), $tNSH$ compacts more effectively than tSH between 50% and 80% in most cases, but they offer very similar performances after 80% and even sometimes, tSH compacts more than $tNSH$. After some investigation, we discovered that when unifications imply large input sets (close to $|\mathcal{U}|$) and the term t involves most variables of $\mathcal{V}$, tSH yields sets with very few strings because of the large amount of redundancies captured by the representation. Conversely, $tNSH$ represents those strings which are in the complement of tSH also resulting in few strings. The remarkable implication is that both numbers of strings have very close values.

6 Conclusions

We have presented two novel alternative representations to Jacobs and Langen's domain, tSH and $tNSH$, which in certain cases provide a more compact representation of the sharing relationships. The first representation, tSH, compacts the sharing relationships by eliminating redundancies among them. The second, $tNSH$, leverages the complement or negative sharing relationships of the original sharing set. Note also that the representations presented here can be potentially used to improve other sharing-related analyses (e.g., [21]). Our experimental evaluation has shown that both representations can reduce dramatically the size of the sharing representation. Our experiments also show how to set up some key parameters in our algorithms in order to control the desired compression and also their time complexities. We have shown that we can obtain a reasonable compression in polynomial time by tuning appropriately those parameters. Thus, we believe our results contribute to the practical application of scalable set-sharing.

References

[1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In Springer-Verlag, editor, *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 266–280, Namur, Belgium, September 1994.

[2] R. Bagnara, R. Gori, P. M. Hill, and E. 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] 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.

[5] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[6] F. Bueno and M. García de la Banda. Set-Sharing is not always redundant for Pair-Sharing. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, Heidelberg, Germany, April 2004. Springer-Verlag.

[7] M. Codish, V. Lagoon, and F. Bueno. An algebraic approach to sharing analysis of logic programs. In *Proc. of the Fourth International Static Analysis Symposium*, number 1302 in LNCS, pages 68–82. Springer Verlag, 1997.

[8] 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.

[9] 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.

[10] 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.

[11] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[12] F. Esponda, E. S. Ackley, S. Forrest, and P. Helman. On-line negative databases (with experimental results). *International Journal of Unconventional Computing*, 1(3):201–220, 2005.

[13] F. Esponda, E. D. Trias, E. S. Ackley, and S. Forrest. A relational algebra for negative databases. Technical Report TR-CS-2007-18, University of New Mexico, 2007.

[14] 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.

[15] 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.

[16] 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.

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

[18] A. Langen. *Advanced techniques for approximating variable aliasing in Logic Programs*. PhD thesis, Computer Science Dept., University of Southern California, 1990.

[19] X. Li, A. King, and L. Lu. Collapsing Closures. In *ICLP'06*, pages 148–162. Springer-Verlag.

[20] X. Li, A. King, and L. Lu. Lazy Set-Sharing Analysis. In *International Symposium on Functional and Logic Programming, 2006*. Springer-Verlag.

[21] M. Méndez-Lojo and M. Hermenegildo. Precise Set Sharing Analysis for Java-style Programs. In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.

[22] Donald R. Morrison. Patricia: Practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[23] 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.

[24] 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.

[25] Kalyan Muthukumar. *Compile-time Algorithms for Efficient Parallel Implementation of Logic Programs*. PhD thesis, University of Texas at Austin, August 1991.

[26] J. Navas, F. Bueno, and M. Hermenegildo. Efficient top-down set-sharing analysis using cliques. In *Eight International Symposium on Practical Aspects of Declarative Languages*, number 2819 in LNCS, pages 183–198. Springer-Verlag, January 2006.

[27] H. Søndergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[28] 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 414–432. Springer-Verlag, Berlin, 1999.