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 México
Albuquerque, NM, USA

and Manuel V. Hermenegildo
Departments of Computer Science
University of New México, 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
a substitution such as $\theta = \{ X_1 \rightarrow f(U_1, U_2, V_1, V_2, W_1), X_2 \rightarrow g(V_1, V_2, W_1), X_3 \rightarrow g(W_1, W_1), X_4 \rightarrow a \}$ 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 (amu, 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 amu 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 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 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 shy be a sharing set over the set of variables of interest \( \mathcal{V} \), then tNSH keeps track of \( \varphi(\mathcal{V}) \setminus \text{shy} \). This new capability of tNSH dramatically reduces the number of elements to represent as the cardinality of the original set grows toward \( 2^{lV} \). It is important to notice that our work is not based on [10]. Although they define the dual negated positive Boolean functions, coPos does not represent the entire complement of the positive set. Moreover, they do not use 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 \( \Sigma^l \) the finite set of all strings over \( \Sigma \) with length \( l \), \( 0 \leq l \leq |\mathcal{V}| \). Let \( bSH^l = \varphi^0(\Sigma^l) \) be the proper powerset (i.e., \( \varphi(\Sigma^l) \setminus \{\emptyset\} \) ) that contains all possible combinations over \( \Sigma \) 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) function symbols 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 \( g \in \text{Term} \) 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 \( t \in \text{Term} \) 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)** \( \text{rel}(bsh, t) \) and **irrelevant sharing** \( \text{irrel}(bsh, t) \). Given \( t \in \text{Term} \), 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 \( \text{rel}(bsh, t) : bSH^l \times \text{Term} \rightarrow bSH^l \) defined as:

\[
\text{rel}(bsh, t) = \{ s \mid s \in bsh, (s \land t) \neq \theta^l \}
\]

where \( \land \) represents the bitwise AND operation and \( \theta^l \) 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 \( \text{irrel}(bsh, t) : bSH^l \times \text{Term} \rightarrow bSH^l \) where \( \text{irrel}(bsh, t) \) is the complement of \( \text{rel}(bsh, t) \), i.e., \( bsh \setminus \text{rel}(bsh, t) \).

**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:

\[
bsh_1 \bowtie bsh_2 = \{ s \mid s = s_1 \lor s_2, s_1 \in bsh_1, s_2 \in bsh_2 \}
\]

where \( \lor \) represents the bitwise OR operation.

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

\[
bsh^* = \{ s \mid \exists n \geq 1 \exists t_1, \ldots, t_n \in bsh, s = t_1 \lor \ldots \lor t_n \}
\]

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

\[
\text{amgu}(x, t, bsh) = \text{irrel}(bsh, x = t) \cup (\text{rel}(bsh, x) \bowtie \text{rel}(bsh, t))^*
\]

**Example 2.6 (Binary abstract unification)**. Let \( V = \{ X_1, X_2, X_3, X_4 \} \) 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: \( X_1 < X_2 < X_3 < X_4 \). 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 \( V \) 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:

1. \( A = \text{rel}(bsh, X_1) = \{ 1000 \} \) and \( B = \text{rel}(bsh, f(X_2, X_3)) = \{ 0100, 0010 \} \)
2. \( A \bowtie B = \{ 1100, 1010 \} \)
3. \( (A \bowtie B)^* = \{ 1100, 1010, 1110 \} \)
4. \( C = \text{irrel}(bsh, X_1 = f(X_2, X_3)) = \{ 0001 \} \)
5. \( \text{amgu}(X_1, f(X_2, X_3), bsh) = C \cup (A \bowtie B)^* = \{ 0001, 1100, 1010, 1110 \} \)

The design of the analysis must be completed by defining the following abstract operations that are required by an analysis engine: \( \text{init} \) (initial abstract state), \( \text{equivalence} \) (between two abstract substitutions), \( \text{join} \) (defined as the union), and
In the interest of brevity, we define only the project operation since the other three operations are trivial.

**Definition 2.7 (Binary projection, \(bsh|_i\)).** The binary projection is a function \(bsh|_i : bSH^l \times \text{Term} \rightarrow bSH^k\) \((k \leq l)\) 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 \(t\) (denoted by \(t[i]\)) is 0, and it is defined as

\[
bsh|_i = \{s' \mid s \in bsh, s' = \pi(s, t)\}
\]

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

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

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 \(\Sigma_3 = \{0, 1, *\}\), 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 \(\Sigma_3 = \{0, 1, *\}\), \(\mathcal{V}\) be a fixed and finite set of variables of interest in an arbitrary order as in Def. 2.1, and \(\Sigma_3^l\) the finite set of all strings over \(\Sigma_3\) with length \(l\), \(0 \leq l \leq |\mathcal{V}|\). Then, \(tSH^l = \varphi(\Sigma_3^l)\) 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, \(M\)).** Given two ternary strings, \(x, y \in \Sigma_3^l\), of length \(l\), match is a function \(M : \Sigma_3^l \times \Sigma_3^l \rightarrow \mathbb{B}\), such that \(\forall i \ 1 \leq i \leq l,\)

\[
xM y = \begin{cases} 
\text{true}, & \text{if } (x[i] = y[i]) \lor (x[i] = *) \lor (y[i] = *) \\
\text{false}, & \text{otherwise}
\end{cases}
\]

**Definition 3.3 (Subsumed_By \(\subseteq\) and Subsumed_In \(\subseteq\)).** Given two ternary strings \(s_1, s_2 \in \Sigma_3^l\), \(\subseteq : \Sigma_3^l \times \Sigma_3^l \rightarrow \mathbb{B}\) is a function such that \(s_1 \subseteq s_2\) if and only if every string matched by \(s_1\) is also matched by \(s_2\). More formally, \(s_1 \subseteq s_2 \iff\)
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-\( \ast \)) to remain.

\[ \forall s \in tsh^l, \text{ if } s_1 \not\subset s_2 \text{ then } s_2 \not\subset s_1. \]

For convenience, we augment this definition to deal with sets of strings. Given a ternary string \( s \in \Sigma_3^l \) and a ternary sharing set, \( tsh \in tsh^l, \), \( \subseteq : \Sigma_3^l \times tsh^l \rightarrow B \) is a function such that \( s \subseteq tsh \) if and only if there exists some element \( s' \in tsh \) such that \( s \subseteq s' \).

The example above begins with \( \text{Convert}(bsh,k = 1) \). Since \( tsh = \emptyset \) initially (line 1), the first string 1000 is appended to \( tsh \) only if it 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 \( V \) 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 \( \text{Convert} \) is \( tsh = \{100^*, 0010, 010^*, 0001\} \).
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 $x'_0 \not\subseteq tsh$, $x'_1 \not\subseteq tsh$, and $x'_2 \not\subseteq tsh$, we reset each $i^\text{th}$ 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 $x'_3 \not\subseteq tsh$, 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 $\lor$ and Ternary-and $\land$).** Given two ternary strings, $x, y \in \Sigma_t^l$ of length $l$, ternary-or and ternary-and are two bitwise-or functions defined as $\lor, \land : \Sigma_t^l \times \Sigma_t^l \rightarrow \Sigma_t^l$ such that $z = x \lor y$ and $w = x \land y$, $\forall 1 \leq i \leq l$, where:

$$
\begin{align*}
z[i] &= \begin{cases} 
* & \text{if } (x[i] = *) \land (y[i] = *) \\
0 & \text{if } (x[i] = 0) \land (y[i] = 0) \\
1 & \text{otherwise}
\end{cases} \\
w[i] &= \begin{cases} 
* & \text{if } (x[i] = *) \land (y[i] = *) \\
1 & \text{if } (x[i] = 1) \land (y[i] = 1) \\
\lor (x[i] = 1) \land (y[i] = *) \\
\lor (x[i] = * \land (y[i] = 1) \\
0 & \text{otherwise}
\end{cases}
\end{align*}
$$

**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

$$
tsh_1 \cap tsh_2 = \{r | r = s_1 \land s_2, s_1, s_2 \in tsh_1, s_1, s_2 \in tsh_2\}
$$

For convenience, we define two binary patterns, $0\text{-mask}$ and $1\text{-mask}$, 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, $\forall 1 \leq i \leq l$. 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, $\forall 1 \leq i \leq l$. For instance, $0\text{-mask}(0110)$ and $1\text{-mask}(0110)$ return $\{*00*\}$ and $\{*1*\}$, respectively.

**Definition 3.7 (Ternary relevant sharing $\text{rel}(tsh, t)$ and irrelevant sharing $\text{irrel}(tsh, t)$).** Given $t \in \text{Term}$ 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 $\text{rel}(tsh, t) : tSH^l \times \text{Term} \rightarrow tSH^l$ defined as

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

In addition, $\text{irrel}(tsh, t)$ is defined as
\[ irrel(tsh, t) = (tsh \cap 1\text{-}\text{mask}(t)) \cap 0\text{-}\text{mask}(t) \]

Ternary cross-union, \(\otimes\), 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 \(\ast\) 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 \(tsh = \{100\ast, 010\ast, 0010, \ast001\}\) as in Example 3.4. Consider again the analysis of \(X_1 = f(X_2, X_3)\), the result is:

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

(ii) \(A \otimes B = \{110\ast, 101\ast\}\)

(iii) \((A \otimes B)^* = \{110\ast, 101\ast, 111\ast\}\)

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

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

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 \(\ast\) symbol. So, the function definition remains the same except that ternary string projection is now defined as a function \(\pi(s, t) : \Sigma^t \times \text{Term} \to \Sigma^k, k \leq l\). For example, let \(tsh = \{100\ast, 010\ast, 0010, \ast001\}\) 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 \(\text{init} : \mathcal{V} \times \mathbb{T}^+ \to tSH|_{\mathcal{V}}\) describes an empty substitution given a set of variables of interest. Assuming the binary initial state operation defined as \(\text{init}_{bSH} : \mathcal{V} \to bSH|_{\mathcal{V}}\), the ternary initial state can be defined using the Convert algorithm in Fig. 1 as:

\[ \text{init}(\mathcal{V}, k) = \text{Convert}(\text{init}_{bSH}(\mathcal{V}), k) \]

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

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, \( \overline{t} \). 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., \( 2^{|V|}-1 \)), then the size of its complement will not be greater than \( 2^{|V|}-1 \). 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 \( 2^{|V|} \), 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 \( U \), the set of all \( l \)-bit strings \( U = \{**\} \), and then, the **Insert** operation to return \( U \setminus sh \) 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 \( tnsb \) calling to **NegConvert** with \( k = 1 \). We begin with \( tnsb = U = \{***\} \) (line 1), then incrementally delete each element of \( bsh \) from \( tnsb \) (line 2-3). **Delete** removes all strings matched by \( x \) from \( tnsb \) (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 \( tnsb \) (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**
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, \( tnsn = \{11**, 1*1*, *11*, **11\} \).

For Example 4.1, the first string 1000 is deleted from \( U = \{ *** \} \). So, \( D_x = \{ *** \} \) (line 11) and \( tnsn' = \emptyset \) (line 12). For each \( i^{th} \) bit of \( x \), a new \( y' \) is evaluated for insertion into the result set. So, \( \text{Insert} (\emptyset, y'_0 = 0***, k = 1) \) is called (line 17). Since \( \text{Specified}(y') \geq k \) and \( tnsn' = \emptyset \), the result returned is \( tnsn' = 0*** \) (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, \( y' = \overline{x_i} \). So, \( \text{Insert} (y'_i = *1**, k = 1) \) is called next and \( y'_i \) is appended to \( tnsn' \). The process continues with \( y'_2 \) and \( y'_3 \) resulting in \( tnsn = \{0***, *1**, *1*, **11\} \). Notice that \( tnsn = U \setminus (bsh \cup \{0000\}) \).

NegConvertMissing would return the same result for Example 4.1, and in gen-
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 $2^{l} - 1$, i.e., $|bsh| = 3 < 2^{3}$, then its complement set must represent $(2^{l} - |bsh|) = 13$ 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 $t \in \text{Term}$ and $tnsh \in tNSH^{t}$ with strings of length $l$, the set of strings in $tnsh$ that are negative relevant with respect to $t$ is obtained by a function $\overline{rel}(tnsh,t) : tNSH^{t} \times \text{Term} \rightarrow tNSH^{t}$ defined as

$$\overline{rel}(tnsh,t) = tnsh \cap 0\text{-mask}(i),$$

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

$$\overline{irrel}(tnsh,t) = tnsh \cap 1\text{-mask}(i).$$
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 \( \text{tnsh} = \{11**, 1*1*, *11*, **11\} \) be the same sharing set as in Example 4.1. Its equivalent Boolean formula \( \phi \equiv \neg((x_1 \text{ and } x_2) \text{ or } (x_1 \text{ and } x_3) \text{ or } (x_2 \text{ and } x_3)) \) is defined over the set of variables \( \{x_1, x_2, x_3, x_4\} \). The formula \( \phi \) 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 \( \phi \) to an equivalent formula in conjunctive normal form. Then, it is easy to see that a satisfying assignment of the formula such as \( \{x_1 = \text{true}, x_2 = \text{false}, x_3 = \text{false}, x_4 = \text{true}\} \) 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 \( U \) and performing cross-union on strings not in \( \text{tnsh} \), we achieve a more efficient negative cross-union, \( \mathcal{U} \), by converting \( \text{tnsh} \) to \( \text{tsh} \) first, i.e., using \text{NegConvert} from Table 1 and performing ternary cross-union on strings \( t \in \text{tsh} \). In this way, the ternary representation continues to provide a compressed representation of the sharing set. Note that negative up-closure operation, *, 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, \text{amgu}).** The negative abstract unification is a function \( \text{amgu} : V \times \text{Term} \times tNSH^I \rightarrow tNSH^I \) defined as

\[
\text{amgu}(x, t, \text{tnsh}) = \text{irrel}(\text{tnsh}, x = t) \cup (\text{rel}(\text{tnsh}, x) \ominus \text{rel}(\text{tnsh}, t))^\top,
\]

where \( \cup \) is the negative set union as defined in \([13]\).
\[ C = \text{irrel}(tnsh, X_1 = f(X_2, X_3)) = \{11**^*, 1**1*, *11*, **11, **1**, 1**1**1*, 1**1**1*\} = \{1**1**, 1**1**, 1**1**\} \]

(iv) \( \text{amryn}(X_1, f(X_2, X_3), tnsh) = C \cup (A \uplus B) = \{01**, 0*1*, 0**0, 100*\} \)

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

\[ \overline{tnsh}_i = \pi(tnsh, \Upsilon_i), \]

where \( \Upsilon_i \) is equal to all \( i \)-th bit positions of \( i \) where \( i[i] = 1 \) and \( \pi \) 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 \( \overline{tnsh}_i \). 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 \( \overline{init} : \mathcal{V} \times \mathcal{I}^+ \rightarrow tNSH^{\mathcal{V}} \) describes an initial substitution given a set of variables of interest. Assuming as in Def. 3.9 the binary initial state operation \( \overline{init}_{bSH} : \mathcal{V} \rightarrow bSH^{\mathcal{V}} \), the negative initial state can be defined using both the NegConvert and NegConvertMissing algorithms described in Fig. 2 (denoted by Convert ) as follows:

\[ \overline{init}(\mathcal{V}, k) = \text{Convert}(\overline{init}_{bSH}(\mathcal{V}), k) \]

**Definition 4.8** (Negative set equivalence, \( \equiv \)). Given \( tnsh_1, tnsh_2 \in tNSH^l \), they are equivalent if and only if (\( \forall t_1 \in tnsh_1, \forall s_1 \subseteq t_1, s_1 \not\subseteq tnsh_2 \) \( \land \) (\( \forall t_2 \in tnsh_2, \forall s_2 \subseteq t_2, s_2 \not\subseteq tnsh_1 \)).

**Definition 4.9** (Negative join, \( \Box \)). Given \( tnsh_1, tnsh_2 \in tNSH^l \), the negative join function \( \Box : tNSH^l \times tNSH^l \rightarrow \wp(tNSH^l) \) is defined as the negative set union of the two sets, i.e., \( tnsh_1 \cup tnsh_2 \).

**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
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 $|U|$. 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
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
Table 3

<table>
<thead>
<tr>
<th>Pre-amgu</th>
<th>Post-amgu</th>
<th>$t_1$</th>
<th>$t_{m/4}$</th>
<th>$t_{m/2}$</th>
<th>$t_{3m/4}$</th>
<th>$t_m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Converted Size (% of $</td>
<td>V</td>
<td>)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$bSH$</td>
<td>2048 (50%)</td>
<td>1535(14)</td>
<td>1909(9)</td>
<td>2017(8)</td>
<td>2028(7)</td>
<td>2032(8)</td>
</tr>
<tr>
<td></td>
<td>2457 (60%)</td>
<td>1642(14)</td>
<td>1942(8)</td>
<td>2029(7)</td>
<td>2038(6)</td>
<td>2040(6)</td>
</tr>
<tr>
<td></td>
<td>2867 (70%)</td>
<td>1742(12)</td>
<td>1968(7)</td>
<td>2035(4)</td>
<td>2042(3)</td>
<td>2044(4)</td>
</tr>
<tr>
<td></td>
<td>3276 (80%)</td>
<td>1835(8)</td>
<td>1995(6)</td>
<td>2042(2)</td>
<td>2045(2)</td>
<td>2047(4)</td>
</tr>
<tr>
<td></td>
<td>3685 (90%)</td>
<td>1945(6)</td>
<td>2031(4)</td>
<td>2041(1.4)</td>
<td>2046(1.7)</td>
<td>2047(7)</td>
</tr>
<tr>
<td></td>
<td>4095 (99%)</td>
<td>2047(0)</td>
<td>2047(5)</td>
<td>2047(17)</td>
<td>2047(0)</td>
<td>2047(0)</td>
</tr>
<tr>
<td>$tSH$</td>
<td>1397 (50%)</td>
<td>408(14)</td>
<td>117(12)</td>
<td>39(13)</td>
<td>33(10)</td>
<td>28(11)</td>
</tr>
<tr>
<td></td>
<td>1645 (60%)</td>
<td>494(15)</td>
<td>130(11)</td>
<td>31(11)</td>
<td>21(7)</td>
<td>18(6)</td>
</tr>
<tr>
<td></td>
<td>1846 (70%)</td>
<td>568(16)</td>
<td>136(11)</td>
<td>25(8)</td>
<td>15(5)</td>
<td>13(4)</td>
</tr>
<tr>
<td></td>
<td>1986 (80%)</td>
<td>620(21)</td>
<td>133(12)</td>
<td>19(4)</td>
<td>12(2)</td>
<td>11(1)</td>
</tr>
<tr>
<td></td>
<td>1913 (90%)</td>
<td>586(32)</td>
<td>119(15)</td>
<td>18(4)</td>
<td>11(1)</td>
<td>10(0)</td>
</tr>
<tr>
<td></td>
<td>3285 (99%)</td>
<td>15(6)</td>
<td>13(4)</td>
<td>11(0)</td>
<td>11(0)</td>
<td>11(0)</td>
</tr>
<tr>
<td>$tNSH$</td>
<td>1370 (50%)</td>
<td>745(88)</td>
<td>200(19)</td>
<td>43(10)</td>
<td>26(7)</td>
<td>16(8)</td>
</tr>
<tr>
<td></td>
<td>1123 (60%)</td>
<td>619(31)</td>
<td>163(17)</td>
<td>31(7)</td>
<td>19(4)</td>
<td>12(6)</td>
</tr>
<tr>
<td></td>
<td>860 (70%)</td>
<td>462(22)</td>
<td>123(14)</td>
<td>24(5)</td>
<td>16(3)</td>
<td>11(4)</td>
</tr>
<tr>
<td></td>
<td>587 (80%)</td>
<td>310(14)</td>
<td>83(10)</td>
<td>18(2)</td>
<td>13(2)</td>
<td>12(4)</td>
</tr>
<tr>
<td></td>
<td>299 (90%)</td>
<td>162(10)</td>
<td>47(6)</td>
<td>14(2)</td>
<td>12(1)</td>
<td>12(3)</td>
</tr>
<tr>
<td></td>
<td>1 (99%)</td>
<td>5(1)</td>
<td>6(1)</td>
<td>9(4)</td>
<td>11(0)</td>
<td>13(0)</td>
</tr>
</tbody>
</table>

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 $V$ such that $|V| = l = 12$, we constructed $x \in V$ by selecting one variable and $t \in Term$ as a term consisting of a subset of the remaining variables, i.e., $V \setminus \{x\}$. We tested with different values of $t$. Let $m = l - 1$ and $|\text{ones} : BS \to \mathbb{Z}^+|$ a function that returns the number of 1’s in a binary string, then $t_1$ represents $|\text{ones} = 1$, $t_{m/4}$ means $|\text{ones} = [11/4]$, 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 $|U|$. 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 $|U|$) and the term $t$ involves most variables of $V$, $tSH$ 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


