Negative Ternary Set-Sharing*

Eric Trias,[1,2,**] Jorge Navas,[1] Elena S. Ackley,[1] Stephanie Forrest[1], and M. Hermenegildo[1,3]

[1] University of New Mexico, USA
[2] Air Force Institute of Technology, USA
[3] Technical U. of Madrid (Spain) and IMDEA-Software

Abstract. The Set-Sharing domain has been widely used to infer at compile-time interesting properties of logic programs such as occurs-check reduction, automatic parallelization, and finite-tree analysis. However, performing abstract unification in this domain requires a closure operation that increases the number of sharing groups exponentially. Much attention has been given to mitigating this key inefficiency in this otherwise very useful domain. In this paper we present a novel approach to Set-Sharing: we define a new representation that leverages the complement (or negative) sharing relationships of the original sharing set, without loss of accuracy. Intuitively, given an abstract state $sh_{\mathcal{V}}$ over the finite set of variables of interest $\mathcal{V}$, its negative representation is $\wp(\mathcal{V}) \setminus sh_{\mathcal{V}}$. Using this encoding during analysis dramatically reduces the number of elements that need to be represented in the abstract states and during abstract unification as the cardinality of the original set grows toward $2^{|\mathcal{V}|}$. To further compress the number of elements, we express the set-sharing relationships through a set of ternary strings that compacts the representation by eliminating redundancies among the sharing sets. Our experiments show that our approach can compress the number of relationships, reducing significantly the memory usage and running time of all abstract operations, including abstract unification.

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 that contain a common variable. A variable in a logic program is said to be *ground* if it is bound to a term that does not contain free variables in all possible executions of the program. *Set-Sharing* is an important type of combined sharing and groundness analysis. It was originally introduced by Jacobs and Langen [17,19] and its abstract values are sets of sets of variables that keep track in a compact way of the sharing patterns among variables.

* 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*.

** The views expressed in this article are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

M. Garcia de la Banda and E. Pontelli (Eds.): ICLP 2008, LNCS 5366, pp. 301–316, 2008.

Example 1 (Set-Sharing abstraction). Let $V = \{X_1, X_2, X_3, X_4\}$ be a set of variables. The abstraction in Set-Sharing of a substitution $\theta = \{X_1 \mapsto f(U_1, U_2, V_1, V_2, W_1), X_2 \mapsto g(V_1, V_2, W_1), X_3 \mapsto g(W_1, W_1), X_4 \mapsto a\}$ 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) shared run-time variables is abstracted away.

Set-Sharing has been used to infer several interesting properties and perform optimization and verification of programs at compile-time, most notably but not limited to: occurs-check reduction (e.g., [27]), automatic parallelization (e.g., [25,6]), 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 [24,17,25,9,15], and also information about *term structure* [25,18,3,23]. 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 e.g. *Pair-Sharing* [27] due to better groundness propagation and other factors that are relevant in some of its applications [5]. In addition, Set-Sharing has attracted much attention [7,10] because its algebraic properties allow elegant encodings into other efficient implementations (e.g., *ROBDDs* [4]). In [25],the first comparatively efficient algorithms were presented for the basic operations needed for set sharing-based analyses.

However, Set-Sharing has a key computational disadvantage: the *abstract unification* (*amgu*, for short) implies 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 [29], Zaffanella et al. extended the Set-Sharing domain for inferring pair-sharing to support *widening*. 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 [21] 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 [20] 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 the Set-Sharing domain is isomorphic to the dual negative of *Pos* [1], denoted by $\overline{coPos}$. This insight improved the understanding of Set-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, although this point was not investigated further.

In this paper we introduce a novel approach to Set-Sharing: we define a new representation that leverages the complement (or negative) sharing relationships of the original sharing set, without loss of accuracy. Intuitively, given an abstract state $sh_{\mathcal{V}}$ over the finite set of variables of interest $\mathcal{V}$, its negative representation is $\wp(\mathcal{V}) \setminus sh_{\mathcal{V}}$. Using

this encoding during analysis dramatically reduces the number of elements that need to be represented in the abstract states and during abstract unification as the cardinality of the original set grows toward $2^{|\mathcal{V}|}$. To further compress the number of elements, we express the set-sharing relationships through a set of ternary strings that compacts the representation by eliminating redundancies among the sharing sets. 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 compressing 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.

2 Set-Sharing Encoded by Binary Strings

The presentation here follows that of [29,10] since the notation used and the abstract unification operation obtained are rather intuitive, but adapted for handling binary strings rather than sets of sets of variables.

Therefore, 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. An algorithm for this conversion and examples are presented in [28].

Definition 1 (Binary sharing domain, bSH). Let alphabet $\Sigma = \{0, 1\}$, $\mathcal{V}$ be a fixed and finite set of variables of interest in 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 power set* (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 by $\hat{t}$ 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 by 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.

Definition 2 (Binary relevant sharing $rel(bsh, t)$, 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 3 (Binary cross-union, $\bowtie$). Given $bsh_1, bsh_2 \in bSH^l$, their *cross-union* is a function $\bowtie : bSH^l \times bSH^l \to bSH^l$ 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 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 $(.)^* : bSH^l \to bSH^l$ 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 5 (Binary abstract unification, $amgu$). The abstract unification is a function $amgu : \mathcal{V} \times Term \times bSH^l \to bSH^l$ defined as

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

■

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 because the other three operations are trivial. We refer the reader to [28] for the rest of operations.

Definition 6 (Binary projection, $bsh|_t$). The *binary projection* is a function $bsh|_t$: $bSH^l \times Term \to bSH^k$ ($k \leq l$) 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

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 encoding discussed above to the 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 represented (i.e., without loss of accuracy). To handle the ternary alphabet, we redefine the binary operations covered in Sec. 2.

Definition 7 (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. 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. 8 and Def. 9, 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 8 (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 $\forall i\ 1 \leq i \leq l$,

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

■

Definition 9 (Subsumed_By $\trianglelefteq$ and Subsumed_In $\trianglelefteqq$). Given two ternary strings s_1, $s_2 \in \Sigma_*^l$, $\trianglelefteq : \Sigma_*^l \times \Sigma_*^l \rightarrow \mathcal{B}$ is a function such that $s_1 \trianglelefteq s_2$ if and only if every string matched by s_1 is also matched by s_2 ($s_1 \trianglelefteq s_2 \iff \forall s \in tSH^l,\ if\ s_1 \mathcal{M} s\ then\ s_2 \mathcal{M} s$). 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, $tsh \in tSH^l$, $\trianglelefteqq : \Sigma_*^l \times tSH^l \rightarrow \mathcal{B}$ is a function such that $s \trianglelefteqq tsh$ if and only if there exists some element $s' \in tsh$ such that $s \trianglelefteq s'$. ■

Figure 1 gives the pseudo code for an algorithm which converts 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 $0 \leq k \leq l$ (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 a non-empty bsh, $k = 1$ introduces the maximum number of $*$ symbols. For now, we will assume that $k = 1$, and experimental results in Sec. 5 shows 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, y replaces all the redundant strings.

Example 2 (Conversion from bSH to tSH). 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 $tsh = \{100*, 0010, 010*, *001\}$.

Definition 10 (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:

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

■

```
0  Convert(bsh, k)
1  tsh ← ∅
2  foreach s ∈ bsh
3      y ← PatternGenerate(tsh, s, k)
4      tsh ← ManagedGrowth(tsh, y)
5  return tsh
6  ManagedGrowth(tsh, y)
7  S_y = {s | s ∈ tsh, s ⊴ y}
8  if S_y = ∅ then
9      if y ⋬ tsh then
10         append y to tsh
11 else
12     remove S_y from tsh
13     append y to tsh
14 return tsh
```

```
15 PatternGenerate(tsh, x, k)
16 m ← Specified(x)
17 i ← 0
18 x' ← x
19 l ← length(x)
20 while m > k and i < l
21     Let b_i be the value of x' at position i
22     if b_i = 0 or b_i = 1 then
23         x' ← x' with position i replaced by ¬b_i
24         if x' ⋬ tsh then
25             x' ← x' with position i replaced by ∗
26         else
27             x' ← x' with position i replaced by b_i
28     m ← Specified(x')
29     i ← i + 1
30 return x'
```

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

Definition 11 (Ternary set intersection, $\cap$). Given $tsh_1, tsh_2 \in tSH^l$, $\cap : tSH^l \times tSH^l \rightarrow tSH^l$ is defined as

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

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 also takes 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 12 (Ternary relevant sharing $rel(tsh, t)$, 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

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

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

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

Ternary cross-union, $\bowtie$, and ternary up-closure, $(.)^*$, operations are as defined in Def. 3 and in Def. 4, respectively, except the binary version of the bitwise OR operator is replaced with its ternary counterpart defined in Def. 10 in order to account for the $*$ symbol. In addition, the ternary abstract unification $(amgu)$ is defined exactly as the binary version, Def.5, using the corresponding ternary definitions.

Example 3 (Ternary abstract unification). Let $tsh = \{100*, 010*, 0010, *001\}$ as in Example 2. Consider again the analysis of $X_1 = f(X_2, X_3)$, the result is:

$$
\begin{array}{ll}
A = rel(tsh, X_1) & = \{100*\} \\
B = rel(tsh, f(X_2, X_3)) & = \{010*, 0010\} \\
A \bowtie B & = \{110*, 101*\} \\
(A \bowtie B)^* & = \{110*, 101*, 111*\} \\
C = irrel(tsh, X_1 = f(X_2, X_3)) & = \{0001\} \\
amgu(X_1, f(X_2, X_3), tsh) = C \cup (A \bowtie B)^* & = \{0001, 110*, 101*, 111*\}
\end{array}
$$

Here briefly, we describe the ternary projection. The other ternary operations required by any analysis framework can be be found in [28]. The ternary projection, $tsh|_t$, is defined similarly as binary projection, see Def. 6. However, the projection domain and range is extended to accommodate the $*$ symbol. For example, let $tsh = \{$100*, 010*, 0010, *001$\}$ as in Example 2. 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 all zeros is meaningless in a set-sharing representation, it is not included here.

4 Negative Ternary Set-Sharing

In this section, we extend the use of the ternary representation discussed in the previous section.[1] 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, $\overline{tsh}$. The same reasoning is applicable to binary strings (i.e., bSH). 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 exploit. In Set-Sharing analysis, as we consider programs with larger numbers of variables of interest, the potential number of sharing groups grows exponentially toward $2^{|\mathcal{V}|}$, whereas 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 to the conversion algorithms shown in Figs. 1 and 2. However, they produce a negative set with unspecified bits in random positions and with less emphasis on managing the growth of the resulting set. The technique was originally introduced as a means of generating Boolean satisfiability (SAT) formulas where, by leveraging the difficulty of finding solutions to hard SAT instances, the contents of the original set are obscured without using encryption [12]. In addition, these hard-to-reverse negative sets are still able to answer membership queries efficiently while remaining intractable to reverse (i.e., to obtain the contents of the original set). In this paper, we are not interested in this security property, but use the negative approach simply to address the efficiency issues faced by traditional Set-Sharing domain.

[1] Note that we could have also used the binary representation described in Sec. 2 but we chose the ternary encoding in order to achieve more compactness.

```
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 with position q_i replaced by 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     V_P ← every possible bit assignment of length |P|
25     foreach v ∈ V_P
26         y ← x with positions P replaced by v
27         tnsh' ← ManagedGrowth(tnsh, y)
28 else
29     y ← PatternGenerate(tnsh, x, k)
30     tnsh' ← ManagedGrowth(tnsh, y)
31 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

The conversion to the negative set can be accomplished using the two algorithms shown 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 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. 7. For clarity, we will denote it by $tNSH$ such that $tNSH \equiv tSH$.

For simplicity, we describe only NegConvert since NegConvertMissing uses the same machinery. Assume a transformation from bsh to $tnsh$ calling 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

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 $bnsh$

Transformation	Time Complexity	Size Complexity
$bSH \rightarrow tSH$	$O(\|bsh\|\alpha l)$	$O(\|bsh\|)$
$bSH/tSH \rightarrow tNSH$	$O(\|bsh\|\alpha(\alpha 2^\delta + 1))$	$O(\|tnsh\|(l-m)2^\delta)$
$tNSH \rightarrow tSH$	$O(\|tnsh\|\alpha(\alpha 2^\delta + 1))$	$O(\|tsh\|(l-m)2^\delta)$
$bSH \rightarrow tNSH$	$O(\beta + \|bnsh\|(\alpha 2^\delta + 1))$	$O(\|bnsh\|2^\delta)$

$tnsh$ (line 11-12). If the set of matched strings, D_x, contains unspecified bit values (* symbols), then all string combinations *not* matching x must be re-inserted 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 (line 28) and appends it while removing any redundancy in the resulting set using ManagedGrowth (line 29).

Example 4 (Conversion from bSH to tNSH). Consider the same sharing set as in Example 2: $bsh = \{1000, 1001, 0100, 0010, 0101, 0001\}$. 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. Thus, $tnsh = \{11**, 1*1*, *11*, **11\}$.

NegConvertMissing would return the same result for Example 4, and, in general, an equivalent negative representation. Table 1 illustrates the different transformation functions and their complexities for a given input. Transformation $bSH \rightarrow tSH$ can be performed by the Convert algorithm described in Fig. 1. Transformations $bSH/tSH \rightarrow tNSH$ and $bSH \rightarrow tNSH$ are done by NegConvert and NegConvertMissing, respectively. Both transformations show that we can convert a positive representation into negative with corresponding difference in time and memory complexity. Depending on the size of the original input we may prefer one transformation over another. If the input size is relatively small, less than 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. Finally, transformation $tNSH \rightarrow tSH$ is performed by NegConvert to revert back to the ternary positive from a negative representation.

Consider now the same set of variables and order among them as in Example 4 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 can be greater than the positive, $|tnsh| = 6 > |bsh| = 3$ and $|tsh| = 2$, unlike Example 4 where $|bsh| = 6$,

and $|tnsh| = 4 < |bsh|$. As the size of $|bsh|$ increases, the complement set that the negative must represent $(2^{|\mathcal{V}|} - |bsh|)$ decreases. This illustrates how selecting the appropriate set-sharing representation affects the size of the converted result. Thus, the size of the original sharing set at specific program points will be used by the analysis to produce the most compact working set. The negative sharing set representation allows us to represent more variables of interest enabling larger problem instances to be evaluated.

We now define the negative abstract unification operations, along with key ancillary operations required by our engine to use the negative representation.

Definition 13 (Negative relevant sharing and irrelevant sharing). 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 $\overline{rel}(tnsh, t) : tNSH^l \times Term \rightarrow tNSH^l$ defined as:

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

In addition, $\overline{irrel}(tnsh, t)$ is defined as:

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

where $\overline{\cap} \equiv \cup$ and defined in [13]. ■

Because the negative representation is the complement, it is not only more compact for large positive set-sharing instances, but also, and perhaps more importantly, it enables us to use inverse operations that are more memory- and computationally efficient than in the positive representation. However, the negative representation does have its limitations. 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 formulas 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. This is known to be an $\mathcal{NP}$-Hard problem. As mentioned before, this fact is exploited in [12] for privacy enhancing applications. In [28] we show that negative cross-union, $\overline{\otimes}$, is $\mathcal{NP}$-Complete.

Due to the interdependent nature of the relationship between the elements of a negative set, it is unclear how 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, $\overline{\otimes}$, 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 the negative upclosure operation, $\overline{*}$, suffers the same drawback as cross-union. Therefore, it is handled the same way as negative cross-union.

Definition 14 (Negative union, $\overline{\cup}$). Given two negative sets with same length strings, $tnsh_1$ and $tnsh_2$, the *Negative Union* returns a negative set representing the set union of $tnsh_1 \,\overline{\cup}\, tnsh_2$, and is defined in [13] as:

$$tnsh_1 \,\overline{\cup}\, tnsh_2 = \{z | (x \mathcal{M} y) \Rightarrow z = x \bigwedge y, x \in tnsh_1, y \in tnsh_2\}$$

where $\bigwedge$ is the ternary AND operator. ■

Definition 15 (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

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

■

Example 5 (Negative abstract unification). Let $tnsh = \{11^{**}, 1^*1^*, ^*11^*, ^{**}11\}$ be the same sharing set as in Example 4. Consider the analysis of $X_1 = f(X_2, X_3)$:

$$\begin{array}{ll}
A = \overline{rel}(tnsh, X_1) & = \{11**, 1*1*, *11*, **11, 0***\} \\
B = \overline{rel}(tnsh, f(X_2, X_3)) & = \{11**, 1*1*, *11*, **11, *00*\} \\
A \,\overline{\otimes}\, B & = \{00**, 01**, 0*0*, *00*\} \\
(A \,\overline{\otimes}\, B)^{\overline{*}} & = \{01**, 0*1*, 100*\} \\
C = \overline{irrel}(tnsh, X_1 = f(X_2, X_3)) & = \{11**, 1*1*, *11*, **11, 1***, \\
 & \quad *1**, **1*\} \\
 & = \{1***, *1**, **1*\} \\
\overline{amgu}(X_1, f(X_2, X_3), tnsh) = C \,\overline{\cup}\, (A \,\overline{\otimes}\, B)^{\overline{*}} & = \{01**, 0*1*, 0**0, 100*\}
\end{array}$$

Here, we define the negative projection and refer the reader to [28] for the remaining operations:

Definition 16 (Negative projection, $\overline{tnsh|_t}$). The *negative projection* is a function $\overline{tnsh|_t}$: $tNSH^l \times Term \rightarrow tNSH^k$ ($k \leq l$) that selects elements of $tnsh$ projected onto the binary representation of $t \in Term$ and is defined as

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

Υ_t = positions where $\hat{t}[i] = 1, \forall i 1 \leq i \leq l$ and *Negative Project* $\overline{\pi}$ as defined in [13]. ■

We find that the resulting negative set will contain strings that have a bit value projected in column(s) specified by Υ if and only if all possible binary combination of all strings created with the projected column(s) appear in the negative set. For example, given $tnsh = \{000, 011, 10^*, 11^*\}$, the $\overline{\pi}_{\Upsilon=1,2}(tnsh) = \{10, 11\}$.

5 Experimental Results

We developed a proof-of-concept implementation to measure experimentally the relative efficiency in terms of running time and memory usage obtained with the two new representations, tSH and $tNSH$. 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 and their conversion times can vary significantly from one to another. An essential issue is to determine experimentally the best overall k parameter for the conversion algorithms. Second, we study the core abstract operation of the traditional set-sharing, $amgu$, under two different metrics. One is the running time to perform the abstract unification. The other metric expresses the memory usage through the size of the representation in terms

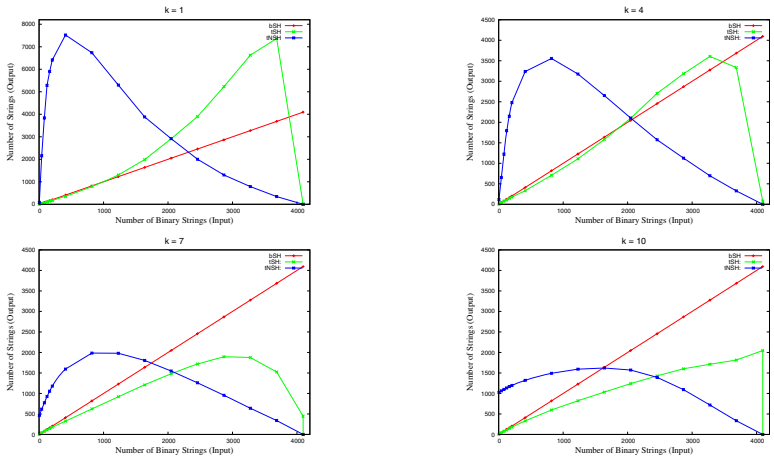

Fig. 3. Compression level after conversions from bSH to tSH and $tNSH$ for k = 1, 4, 7 & 10

of number of strings during key steps in the unification. All experiments have been conducted on an Intel^R Core^{TM} Duo CPU T2350 at 1.86GHz with 1GB of RAM running Ubuntu 7.04, and were performed with 12-bit strings since we consider this value large enough to show all the relevant features of our approach. In general, within some upper bound, the more variables considered the better the expected efficiency.

The first experiment determines the best k value suitable for the conversion algorithms, shown in Figs. 1 and 2. We submit a set of 12-bit strings in random order using different k values. We evaluate size of the output (see Fig. 3) for a given k value. As expected, bSH ($x = y$ line) results in no compression; tSH slowly increases with increasing input size, remaining below bSH (for $k = 7$ and $k = 10$) 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 past 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 $*$ symbols 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 strings that must be represented by the converted result, see line 13-17 of Fig. 2. In addition, a small k value results in a set with more ternary strings than the number of binary strings represented. This occurs when multiple ternary strings, none of which subsumes any other, represent the same binary string. This redundancy in the ternary representation is not prevented by ManagedGrowth, and is apparent in Fig. 3 when $|tSH|$ and $|tNSH|$ exceed the maximum size of binary sharing relationships (i.e., 4096). One way to reduce the number of redundant strings is to sort the binary input by *Hamming distance* before conversion. In the subsequent tests, sorting was performed to maximize compression. We have found empirically that a k setting near (or slightly larger than) $l/2$ is the best

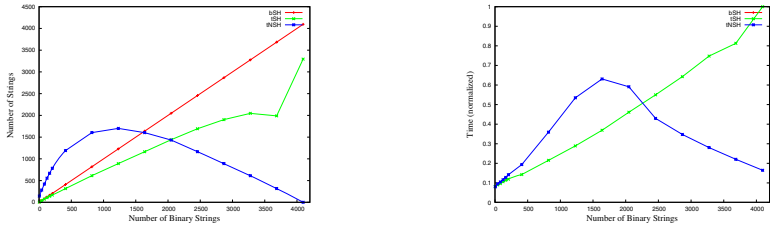

Fig. 4. Memory usage (avg. # of strings) and time normalized for conversions with $k = 7$

overall value considering both the result size and time complexity. We use $k = 7$ in the following experiments. 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 the comparison in terms of memory usage (Fig. 4, left) and running time (Fig. 4, right) of the conversion algorithms for transforming an initial set of binary strings, bSH, into its corresponding set of ternary strings, tSH, or its complement (negative), $tNSH$. We generated random sets of binary strings (over 30 runs) using $k = 7$ and we converted the set of binary strings using the Convert algorithm described in Fig. 1 for tSH, and NegConvertMissing in Fig. 2 for $tNSH$. The plot on the left shows that the number of positive ternary strings,$|tSH|$, used for encoding the input binary strings always remains below $|bSH|$, and this number increases slowly with increasing input size. It important to notice that for large values of $|bSH|$, tSH compacts worse than expected and the compression factor is lower. The main cause is the use of the parameter $k = 7$ that implies only the use of 5 or less $*$ symbols for compression. Conversely, the number of negative sharing relationships, $|tNSH|$, is greater than $|bSH|$ and $|tSH|$ up to between 40% and 50%, respectively. However, when the load exceeds those thresholds $tNSH$ compresses much better than its alternatives. For instance, for the maximum number of binary sharing relationships, $tNSH$ compresses them to only one negative string. On the other hand, the rightmost plot shows the average time consumed over 30 runs for both conversion algorithms. Again, $tNSH$ scales better than the positive ternary solution, tSH, after a threshold established around 50% of the maximum number of binary sharing relationships. Our proof-of-concept implementation is not really optimized, since our objective is to study the *relative* performance between the three representations, and thus times are normalized to the range $[0, 1]$. We argue that comparisons that we report between representations are fair since the three cases have been implemented with similar efficiency, and useful since the absolute performance of the base representation is well understood.

Finally, our third experiment shows the efficiency in terms of the memory usage (in Fig. 5, left) and running time (in Fig. 5, right) when performing the abstract unification for $k = 7$. Several characteristics of the abstract unification influence the memory usage and its performance. Given an arbitrary set of variables of interest $\mathcal{V}$ ($|\mathcal{V}| = 12$),

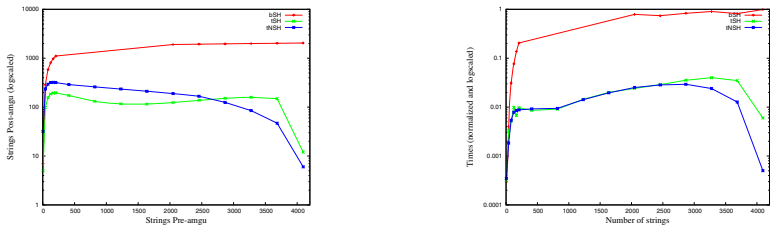

Fig. 5. Memory usage (avg. # of strings) and time normalized for amgu over 30 runs with $k = 7$

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. Another important aspect is the input sharing set, bSH. Again, we reduced the influence of this factor by generating randomly 30 different sets. In the leftmost plot, the x-axis illustrates the number of input binary strings considered during the $amgu$. In the case of the positive and negative ternary $amgu$, the input binary strings were first converted to their corresponding compressed representations. The y-axis shows the number of strings after the unification. The plot shows that exceeding a threshold lower than 500 in the number of input binary sharing relationships, both tSH and $tNSH$ yield a significant smaller number of strings than the binary solution after unification. Moreover, when the number of the input binary strings is smaller than 50% of its maximum value, tSH compresses more efficiently than $tNSH$. However, if this value is exceeded then this trend is reversed: the negative encoding yields a better compression as the cardinality of the original set grows toward $2^{|\mathcal{V}|}$. The rightmost plot shows the size of the random binary input sets in the x-axis, and the average time consumed for performing the abstract unification in its y-axis, normalized again from 0 to 1. This graph shows that the execution times behave similarly to the memory usage during abstract unification. Both tSH and $tNSH$ run much faster than bSH. The differences are significant (a factor of 10) for most x-values, reaching a factor of 1000 for large values of $|bSH|$. When the load exceeds a $50-60\%$-threshold, $tNSH$ scales better than tSH by a factor of 10. The main difference with respect to the memory usage depicted in the leftmost plot is that for a smaller load, tSH runs as fast as $tNSH$ during unification. The main reason is that the ternary relevant and irrelevant sharing operations are less efficient than their negative counterparts, i.e., intersection is an expensive operation in the positive whereas negative intersection is very efficient (positive union).

6 Conclusions

We have presented a novel approach to Set-Sharing that leverages the complement (negative) sharing relationships of the original sharing set, without any loss of accuracy. In this work, we based the negative representation on ternary strings. We also showed that the same ternary representation can be used as a positive encoding to efficiently compact the original binary sharing set. This provides the user the option of working with whichever set sharing representation is more efficient for a given problem instance.

The capabilities of our negative approach to compress sharing relationships are orthogonal to the use of the ternary representation. Hence, the negative relationships may be encoded using other representations such as BDDs [16]. Concretely, *Zero-suppressed BDDs* [16] are particularly interesting because they were designed to represent sets of combinations (i.e., sets of sets). In addition, ZBDDs may be also applicable to similar sharing-related analyses in object-oriented languages (e.g., [22]).

Our experimental evaluation has shown that our approach can reduce significantly the memory usage of the sharing relationships and the running time of the abstract operations, including the abstract unification. Our experiments also show how to set up key parameters in our algorithms in order to control the desired compression and 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 can contribute to the practical, scalable application of Set-Sharing.

References

1. Armstrong, T., Marriott, K., Schachte, P., Søndergaard, H.: Boolean functions for dependency analysis: Algebraic properties and efficient representation. In: LeCharlier, B. (ed.) SAS 1994. LNCS, vol. 864. Springer, Heidelberg (1994)
2. Bagnara, R., Gori, R., Hill, P.M., Zaffanella, E.: Finite-tree analysis for constraint logic-based languages. Information and Computation 193(2), 84–116 (2004)
3. Bruynooghe, M., Codish, M., Mulkers, A.: Abstract unification for a composite domain deriving sharing and freeness properties of program variables. Verification and Analysis of Logic Languages (1994)
4. Bryant, R.E.: Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. ACM Comput. Surv. 24(3), 293–318 (1992)
5. Bueno, F., García de la Banda, M.: Set-Sharing is not always redundant for Pair-Sharing. In: Kameyama, Y., Stuckey, P.J. (eds.) FLOPS 2004. LNCS, vol. 2998. Springer, Heidelberg (2004)
6. Bueno, F., García de la Banda, M., Hermenegildo, M.: Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In: 1994 Intl. Symposium on Logic Programming (1994)
7. Codish, M., Lagoon, V., Bueno, F.: An algebraic approach to sharing analysis of logic programs. In: Proc. of the Fourth Intl. Static Analysis Symposium (1997)
8. Codish, M., Mulkers, A., Bruynooghe, M., García de la Banda, M., Hermenegildo, M.: Improving Abstract Interpretations by Combining Domains. In: PEPM 1993 (1993)
9. Codish, M., Dams, D., Filé, G., Bruynooghe, M.: On the design of a correct freeness analysis for logic programs. The Journal of Logic Programming 28(3), 181–206 (1996)
10. Codish, M., Søndergaard, H., Stuckey, P.J.: Sharing and groundness dependencies in logic programs. ACM Transactions on Prog. Languages and Systems 21(5), 948–976 (1999)
11. Cousot, P., Cousot, R.: Abs Interp: a Unified Lattice Model for Static Analysis of Programs by Construction or Approx of Fixpoints. In: POPL 1977 (1977)
12. Esponda, F., Ackley, E.S., Forrest, S., Helman, P.: On-line negative databases (with experimental results). Intl. Journal of Unconventional Computing 1(3), 201–220 (2005)
13. Esponda, F., Trias, E.D., Ackley, E.S., Forrest, S.: A relational algebra for negative databases. Technical Report TR-CS-2007-18, University of New Mexico (2007)
14. Fecht, C.: An efficient and precise sharing domain for logic programs. In: Kuchen, H., Swierstra, S.D. (eds.) PLILP 1996. LNCS, vol. 1140, pp. 469–470. Springer, Heidelberg (1996)

15. Hill, P.M., Zaffanella, E., Bagnara, R.: A correct, precise and efficient integration of set-sharing, freeness and linearity for the analysis of finite and rational tree languages. In: TPLP 2004 (2004)
16. Minato, S.: ZBDDs for Set Manipulation in Combinatorial Problems. In: DAC 1993 (1993)
17. Jacobs, D., Langen, A.: Static Analysis of Logic Programs for Independent And-Parallelism. Journal of Logic Programming 13(2, 3), 291–314 (1992)
18. King, A., Soper, P.: Depth-k Sharing and Freeness. In: ICLP 1994 (1994)
19. Langen, A.: Advanced techniques for approximating variable aliasing in Logic Programs. PhD thesis, Computer Science Dept., University of Southern CA (1990)
20. Li, X., King, A., Lu, L.: Collapsing Closures. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079. Springer, Heidelberg (2006)
21. Li, X., King, A., Lu, L.: Lazy Set-Sharing Analysis. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945. Springer, Heidelberg (2006)
22. Méndez-Lojo, M., Hermenegildo, M.: Precise Set Sharing Analysis for Java-style Programs. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905. Springer, Heidelberg (2008)
23. Mulkers, A., Simoens, W., Janssens, G., Bruynooghe, M.: On the Practicality of Abstract Equation Systems. In: ICLP 1995 (1995)
24. Muthukumar, K., Hermenegildo, M.: Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In: ICLP 1991 (1991)
25. Muthukumar, K., Hermenegildo, M.: Compile-time Derivation of Variable Dependency Using Abstract Interpretation. JLP 13(2/3), 315–347 (1992)
26. Navas, J., Bueno, F., Hermenegildo, M.: Efficient top-down set-sharing analysis using cliques. In: Van Hentenryck, P. (ed.) PADL 2006. LNCS, vol. 3819. Springer, Heidelberg (2005)
27. Søndergaard, H.: An application of abstract interpretation of logic programs: occur check reduction. In: Robinet, B., Wilhelm, R. (eds.) ESOP 1986. LNCS, vol. 213. Springer, Heidelberg (1986)
28. Trias, E., Navas, J., Ackley, E.S., Forrest, S., Hermenegildo, M.: Efficient Representations for Set-Sharing Analysis. TR-CLIP9/2008.0, Univ. of New Mexico (2008)
29. Zaffanella, E., Bagnara, R., Hill, P.M.: Widening Sharing. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702. Springer, Heidelberg (1999)