Efficient Set Sharing using ZBDDs

Mario Méndez-Lojo[1] Ondřej Lhoták[2] Manuel V. Hermenegildo[1,3]

[1] Dept. of Computer Science, University of New Mexico (USA)
[2] D. R. Cheriton School of Computer Science, University of Waterloo (Canada)
[3] Dept. of Computer Science, Tech. U. of Madrid (Spain) and IMDEA-Software

Abstract. Set sharing is an abstract domain in which each concrete object is represented by the set of local variables from which it might be reachable. It is a useful abstraction to detect parallelism opportunities, since it contains definite information about which variables do not share in memory, i.e., about when the memory regions reachable from those variables are disjoint. Set sharing is a more precise alternative to pair sharing, in which each domain element is a set of all pairs of local variables from which a common object may be reachable. However, the exponential complexity of some set sharing operations has limited its wider application. This work introduces an efficient implementation of the set sharing domain using Zero-supressed Binary Decision Diagrams (ZBDDs). Because ZBDDs were designed to represent sets of combinations (i.e., sets of sets), they naturally represent elements of the set sharing domain. We show how to synthesize the operations needed in the set sharing transfer functions from basic ZBDD operations. For some of the operations, we devise custom ZBDD algorithms that perform better in practice. We also compare our implementation of the abstract domain with an efficient, compact, bitset-based alternative, and show that the ZBDD version scales better in terms of both memory usage and running time.

1 Introduction

Set sharing [11] is an abstract domain aimed at tracking dependency information among sets of variables. In set sharing abstractions, each concrete object is represented by the set of program variables from which it might be reachable. Set sharing-based analyses discover valuable information for parallelizing instructions, statements, function calls, etc. (and are therefore typically used for that purpose), since each abstract state contains definite information about which variables do not share, i.e., which variables cannot reach the same memory location. From this perspective, set sharing analysis can be seen as a compact encoding of the information present in points-to analyses, but in set sharing only the groups of variables that might reach the same object in memory are stored.

Set sharing has been shown to be a more precise alternative to, e.g., *pair* sharing, in which each domain element is a set of all pairs of local variables from which a common object may be reachable. However, some of the intrinsic operations of the set sharing domain are exponential in the number of local variables being tracked, which can become a problem for certain programs and has limited so far wider application. This intrinsic complexity can be dealt with in part by introducing widenings, i.e., simplifying the sharing sets conservatively when they become too large, but of course at the expense losing precision. Finding significantly more efficient implementations reduces the need for resorting to such lossy solutions and consequently improves practicality.

We introduce a new, efficient implementation of the set sharing domain using Zero-supressed Binary Decision Diagrams (ZBDDs). ZBDDs were designed to represent sets of combinations (i.e., sets of sets), so they can represent very naturally the elements of

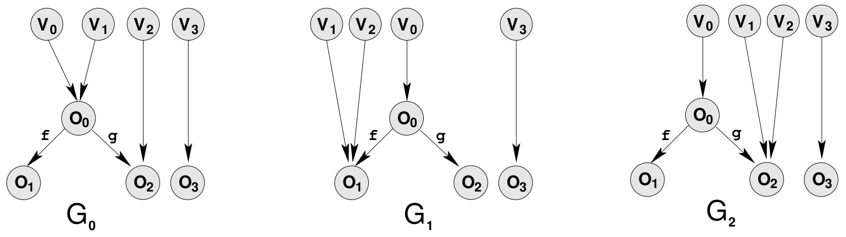

Fig. 1. Three concrete states.

the set sharing domain. To the best of our knowledge this is the first link provided between set sharing and ZBDDs. We start by providing set-sharing transfer functions for a subset of Java.[4] We then show how to express the operations needed for implementing the set sharing transfer functions in terms of basic ZBDD operations. Also, for some of the operations, we propose custom ZBDD algorithms that are more appropriate for these particular cases than those in the standard ZBDD libraries. In particular we provide a design for native ZBDD operations that emulate non-standard set manipulations. The introduction of ZBDDs is done at the implementation level and does not alter the definition of the domain operations, so that the domain designer does not need to be aware of their presence. Finally, we provide performance results comparing two implementations of the set-sharing domain: an efficient, compact, bitset-based alternative (representing a highly-tuned version of the traditional approach) and our ZBDD-based implementation. The results show that the ZBDD version scales better in terms of both memory usage and running time. Our custom ZBDD algorithms are also shown to perform better in practice than the stock ones.

2 Reachability and Sharing

As mentioned before, we will concentrate for concreteness on a subset of Java, although set sharing has been shown to be applicable to different classes of imperative and declarative languages. A concrete state $G = (Var \cup Obj, E)$ is a directed graph where every node can be either a variable $v \in Var$ or an object $o \in Obj$. The edges of the graph have been labeled such that $o_1 \xrightarrow{\texttt{f}} o_2$ means "the field `f` of object o_1 points to o_2." We will assume that edges connecting variables and objects have the special label $-$. An object o is reachable from the variable v in G iff there is a path $v \xrightarrow{-} o_1 \xrightarrow{\texttt{f}} o_1 \xrightarrow{\texttt{g}} o_2 \ldots \xrightarrow{\texttt{h}} o$. The reachability set of a variable v in the state G is the set of all objects that are reachable from it, i.e., $reach(G, v) = \{\, o \in Obj \mid o \text{ is reachable from } v \text{ in } G\}$.

One or more variables *share* in a state G if the intersection of their reachabilty sets is non-empty:

$$share(G, V) \Leftrightarrow \bigcap_{v \in V} reach(G, v) \neq \emptyset.$$

Since null variables have no outgoing edges (conversely, if o.`f` is null, there is no edge in the graph that starts at o and is labeled with `f`), they do not share.

Given graph G, define its set sharing as the set of maximal sets of variables that share:

$$sh(G) = \{\, V' \subseteq Var \mid share(G, V') \text{ and } \nexists W \text{ s.t. } V' \subset W \text{ and } share(G, W)\}$$

[4] As we will see later, these transfer functions, which are independent from the specific way in which the internal set-sharing domain operations are implemented, are in fact themselves improvements over those previously proposed.

The set sharing provides definite information about which variables do not have any memory location in common, i.e., the memory regions reachable from them are disjoint. We can be sure that no object is reachable from more than one variable of a set W if no superset of W is an element of $sh(G)$.

Example 1. Fig. 1 shows three examples of concrete states. We assume that all the variables are of type `Foo`, a class with two fields `f` and `g`, pointing to objects of class `Foo`. In the graph G_0, the reachability sets are $reach(G_0, v_0) = reach(G_0, v_1) = \{o_0, o_1, o_2\}$, $reach(G_0, v_2) = \{o_2\}$ and $reach(G_0, v_3) = \{o_3\}$. The set sharing of G_0 is $sh(G_0) = \{\{v_0, v_1, v_2\}, \{v_3\}\}$. Note that $sh(G_0) = \{\{v_0, v_1\}, \{v_0, v_1, v_2\}, \{v_3\}\}$ is not an acceptable set sharing, even though v_0 shares with v_1, because $\{v_0, v_1\} \subset \{v_0, v_1, v_2\}$, and v_0, v_1, and v_2 all share. The reachability sets of v_1 and v_2 in G_1 and G_2 differ from the ones in G_0; however, the set sharing is the same for all three graphs: $sh(G_0) = sh(G_1) = sh(G_2) = \{\{v_0, v_1, v_2\}, \{v_3\}\}$.

Note that the information provided by set sharing abstract states at program points is instrumental for parallelization: assume that the set sharing of the example, $\{\{v_0, v_1, v_2\}, \{v_3\}\}$, is in fact the abstract state inferred by analysis at the program point just before two consecutive method calls $m(v_0, v_1, v_2)$ and $n(v_3)$. The set sharing represents a number of concrete states (including G_0, G_1, and G_2) in all of which v_3 points to a memory region that is disjoint from the memory regions pointed to by v_0, v_1, or v_2. Since analysis is safe, while actual sharing during execution may be less, there cannot be any concrete states in which there is more sharing than that implied by $\{\{v_0, v_1, v_2\}, \{v_3\}\}$. Thus, under reasonable assumptions regarding the parallel abstract machine, memory management, scheduling, etc., the two method calls can be safely parallelized since they are independent: execution of $m(v_0, v_1, v_2)$ cannot affect that of $n(v_3)$ and they can proceed in parallel without interference. Also, the final state after executing them in parallel will be equivalent to the state obtained after their sequential execution.

3 Sharing Semantics as Set Operations

3.1 Notation

We use double capital letters (like SH) for sets of sets, single capital letters (S) for sets and lowercase letters (for instance, v) to denote elements of a set. We write $SH_V = \{S \in SH \mid V \subseteq S\}$ to denote the subset of SH containing all sets having V as a subset. Conversely, $SH_{-V} = SH - SH_V$. For singleton sets, we define a more concise notation: $SH_v = SH_{\{v\}}$ and $SH_{-v} = SH_{-\{v\}}$.

We define *projecting out* v from SH as removing v from every set in SH: $SH|_{-v} = \{S \setminus \{v\} \mid S \in SH\} \setminus \{\{\}\}$. The *replacement operator* on sets of sets replaces all the ocurrences of variable v_1 with v_2 in every set. Formally, $SH|_{v_1}^{v_2} = \left\{S|_{v_1}^{v_2} \mid S \in SH\right\}$, where

$$S|_{v_1}^{v_2} = \begin{cases} S \text{ if } v_1 \notin S \\ S \setminus \{v_1\} \cup \{v_2\} \text{ else} \end{cases}$$

The *binary union operator* $\uplus$ computes the unions of all pairs of sets taken from two sets of sets: $SH_1 \uplus SH_2 = \{S_1 \cup S_2 \mid S_1 \in SH_1, S_2 \in SH_2\}$.

3.2 Abstract operations

In this section, we review the abstract set sharing semantics that was defined and proven correct in previous work [16]. We also improve the precision for two of the operations: the field load and the field store. Our compositional semantics defines a denotation function for each expression and command. We define the special variable res, which stores

$\mathcal{SE}^I_\pi[\![\texttt{null}]\!](SH)$
$SH' = SH$
$\mathcal{SE}^I_\pi[\![\texttt{new}\ k]\!](SH)$
$SH' = SH \cup \{\{res\}\}$

$\mathcal{SE}^I_\pi[\![v]\!](SH)$
$SH' = (\{\{res\}\} \uplus SH_v) \cup SH_{-v}$
$\mathcal{SE}^I_\pi[\![v\,.\,\texttt{f}]\!](SH)$
$SH' = \begin{cases} \bot \text{ if } \textbf{mustBeNull}(SH, v) \\ SH \cup (\{\{v, res\}\} \uplus \bigcup\limits_{S \in SH_v} \mathcal{P}(S\|_{-v})) \text{ else} \end{cases}$

Fig. 2. Abstract semantics for the expressions as set operations

the result of an expression. Thus, the functions for both expressions and commands are transformers on set sharings. The function for an expression transforms the set sharing to abstract a state in which res points to the result of evaluating the expression.

Figs. 2 and 5 contain the semantics of expressions and commands, respectively. They represent the transition from an initial *abstract state* [6] SH to a final abstract state SH'. In our domain, an abstract state SH approximates all the set sharings of a set of concrete states GG: $SH = \alpha(GG) = \bigcup\limits_{G \in GG} sh(G)$, i.e., SH is a correct abstraction of a set of concrete states $\{G_1, \ldots, G_n\}$ if $sh(G_i) \subseteq SH, i = 1..n$. For instance, given a concrete state G such that $sh(G) = \{\{v_3\}\}$, the abstract state $\{\{v_0, v_1, v_2\}, \{v_3\}\}$ is a valid approximation of G. If a variable is null in the concrete states $\{G_1, \ldots, G_n\}$, it does not appear in SH. Thus, the predicate **mustBeNull**(SH, v) returns true when $SH_v = \emptyset$.

In practice, our abstract state is a pair composed of an abstract set sharing and a type component τ. The objective of this second element is to approximate the set of possible types of each variable. This corresponds to the concept of a "type of class" analysis [1, 7]. In our context, τ helps in determining which variables are non null and which ones may be null. If we consider $null$ as another type [13], then a variable may be null if $null$ is one of its possible types: **mayBeNull**$(\tau, v) = (\overline{\textbf{mustBeNull}}(SH, v)$ and null $\in \tau(v))$. For clarity, we omitted the type component from the transfer functions in Fig. 2 and 5; the full version of the semantics can be found in the Appendix in Fig. 13 and 14.

3.3 Semantics of Expressions

Null, New and Variable Load: The `null` expression loads the null constant into the special variable res, so it has no effect on the abstract state, since res does not point to any object, and therefore does not share with any variable (including itself), both before and after evaluating the expression. The `new` expression adds the singleton $\{res\}$ to the current set sharing, since it creates a fresh object that cannot be reached from any of the existing variables. A variable load `v` forces res to be an alias of v, and therefore res shares with all those variables with which v shares. Sharings in SH_{-v} remain unaffected, since the addition of res cannot change the reachability set of any variable not reachable from v. For instance, given $SH = \{\{v_0, v_1, v_2\}, \{v_3\}\}$, the variable load v_0 results in $SH' = SH_{-v} \cup (\{\{res\}\} \uplus SH_v) = \{\{v_3\}\} \cup (\{\{res\}\} \uplus \{\{v_0, v_1, v_2\}\})$=$\{\{v_3\}\} \cup \{\{res\} \cup \{v_0, v_1, v_2\}\} = \{\{v_0, v_1, v_2, res\}, \{v_3\}\}$.

Field Load: In the case that $v\,.\,\texttt{f}$ is null, there is no change in the existing set sharing. Because the expression of SH' includes SH, that case is correctly approximated. When $v\,.\,\texttt{f}$ is not null, we know that the object being assigned to res is reachable from v. The other variables that share with v in SH may or may not share with res in SH'. In the

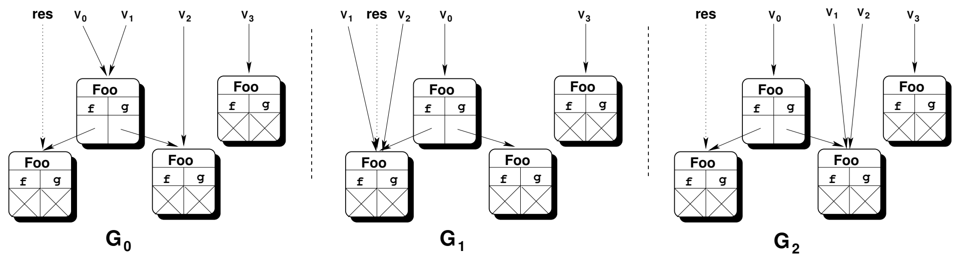

Fig. 3. Three concrete states.

state G_0 of Fig. 3, although v_2 shares with v_0 in the initial and final states, it does not share with res in the final state; however, v_1 will share with both res and v_0 after the load. We write $\{\{v, res\}\} \uplus \bigcup_{S \in SH_v} \mathcal{P}(S|_{-v}))$ to account for objects reachable from v which become also reachable from res, and may be reachable from any subset of the variables that shared with v in SH. Objects not reachable from v (SH_{-v}) are accounted for by the union with SH. For instance, in the same state G_0, if $\{v_3\} \in SH$, then the load of v_0.f does not alter that particular element, which has to also be present in SH'.

Example 2. The graphs in Fig. 3 illustrate three different memory states before the evaluation of v_0.f. They correspond to the graphs in Fig. 1, but this time we indicate the type of every object and the object pointed to by res after the expression evaluation. The initial set sharing is identical in all cases: $sh(G_0) = sh(G_1) = sh(G_2) = \{\{v_0, v_1, v_2\}, \{v_3\}\}$. However, the evaluation results in a different set sharing for each resulting graph G_i': $sh(G_0') = \{\{v_0, v_1, v_2\}, \{v_0, v_1, res\}, \{v_3\}\}$, $sh(G_1') = \{\{v_0, v_1, v_2, res\}, \{v_3\}\}$, and $sh(G_2') = \{\{v_0, v_1, v_2\}, \{v_0, res\}, \{v_3\}\}$. Assume that the abstract state that approximates all the initial concrete states is also $SH = \{\{v_0, v_1, v_2\}, \{v_3\}\}$. The transfer function for v_0.f results in a final abstract state $SH' = SH \cup (\{\{v_0, res\}\} \uplus \mathcal{P}(\{v_1, v_2\}))$ = $\{\{v_0, v_1, v_2\}, \{v_3\}\} \cup (\{\{v_0, res\}\} \uplus \{\{\}, \{v_1\}, \{v_2\}, \{v_1, v_2\}\})$ = $\{\{v_0, v_1, v_2\}, \{v_0, v_1, v_2, res\}, \{v_0, v_1, res\}, \{v_0, v_2, res\}, \{v_0, res\}, \{v_3\}\}$. As required, all the sharings $sh(G_0')$, $sh(G_1')$, and $sh(G_2')$ are included in SH'.

3.4 Semantics of Commands

Variable Store: For a store of the form v=$expr$, the semantics comprises three steps. First, the expression on the right-hand side is evaluated. Second, all ocurrences of v are removed from the current abstract state, since the value of v is being overwritten. Finally, all appearances of res are replaced by v, which deletes res from the abstract state.

Field Store: First, we evaluate the expression whose result is being stored; SH_1 contains that intermediate value. Sharings in SH_1 unrelated to v or res are unaffected by the store and contained in $SH_2 = SH_{1_{-\{v,res\}}}$, which is a subset of the final state. For each sharing in SH_{1_v}, the store might affect the reachability set of each variable involved and result in many smaller sharings. For example, in a memory state like G in Fig. 4, an assignment to v_0.f destroys any sharing between v_0 and v_1 (note that res does not share with v_1), but not the one between v_0 and v_2. All the possible combinations for the final sharings that have to do with v are contained in $SH_3 = \bigcup_{S \in SH_{1_v}} \mathcal{P}(S) \setminus \{\{\}\}$.

$\mathcal{SC}^I_\pi[\![v\texttt{=}expr]\!](SH)$
$SH_1 = \mathcal{SE}^I_\pi[\![expr]\!](SH)$ $SH_2 = SH_1\vert_{-v}$ $SH' = SH_2\vert^{v}_{res}$
$\mathcal{SC}^I_\pi[\![v\,\texttt{.}\,\texttt{f=}expr]\!](SH)$
$SH_1 = \mathcal{SE}^I_\pi[\![expr]\!](SH)$ $SH_2 = SH_{1_{-\{v,res\}}}$ $SH_3 = \bigcup_{S \in SH_{1_v}} \mathcal{P}(S) \setminus \{\{\}\}$ $SH_4 = SH_{1_{res}} \uplus SH_{3_v}$ $SH' = \begin{cases} \bot \text{ if } \textbf{mustBeNull}(SH_1, v) \\ SH_2 \cup (SH_3 \cup SH_4)\vert_{-res} \text{ else} \end{cases}$

$\mathcal{SC}^I_\pi[\![\texttt{if } v\texttt{==null}\ com_1 \texttt{ else } com_2]\!](SH)$
$SH_1 = \mathcal{SC}^I_\pi[\![com_1]\!](SH\vert_{-v})$ $SH_2 = \mathcal{SC}^I_\pi[\![com_2]\!](SH)$ $SH' = \begin{cases} SH_1 \text{ if } \textbf{mustBeNull}(SH, v) \\ SH_1 \cup SH_2 \text{ if } \textbf{mayBeNull}(\tau, v) \\ SH_2 \text{ else} \end{cases}$
$\mathcal{SC}^I_\pi[\![\texttt{if } v\texttt{==}w\ com_1 \texttt{ else } com_2]\!](SH)$
$SH_1 = \mathcal{SC}^I_\pi[\![com_1]\!](SH)$ $SH_2 = \mathcal{SC}^I_\pi[\![com_2]\!](SH)$ $SH' = \begin{cases} SH_1 \text{ if } \textbf{mustAlias}(SH, v, w) \\ SH_1 \cup SH_2 \text{ if } \textbf{mayAlias}(SH, v, w) \\ SH_2 \text{ else} \end{cases}$
$\mathcal{SC}^I_\pi[\![com_1\,\texttt{;}\,com_2]\!](SH)$
$SH' = \mathcal{SC}^I_\pi[\![com_2]\!](\mathcal{SC}^I_\pi[\![com_1]\!](SH))$

Fig. 5. Abstract semantics for the commands.

Now, for every sharing in SH_3 that contains v we have two possibilities: all the variables share also with res (and therefore, with $SH_{1_{res}}$), or none of them does. Note that every possible intermediate case in which just a few of the variables share with $SH_{1_{res}}$ is represented by a smaller subset in SH_3 containing only those variables. While $SH_4 = SH_{1_{res}} \uplus SH_{3_v}$ includes the combinations in which all the variables do share with $SH_{1_{res}}$, SH_3 approximates the situations in which none of them do share with res.

Example 3. Assume an initial state (after evaluating the expression) G depicted in Fig. 4. The dotted edge indicates where $v_0\texttt{.f}$ will point after the execution of $v_0\texttt{.f=}\ expr$. The initial set sharing is $sh(G) = \{\{v_0, v_1\}, \{v_0, v_2\}, \{res\}\}$. After the load, $sh(G') = \{\{v_0, v_2\}, \{v_0, res\}, \{v_1\}\}$. Assume that the starting abstract state, after the evaluation of the expression $expr$, is also $SH_1 = \{\{v_0, v_1\}, \{v_0, v_2\}, \{res\}\}$. Since there is no sharing unrelated to v or res, $SH_2 = \emptyset$. The next step is to calculate $SH_3 = \mathcal{P}(\{v_0, v_1\}) \cup \mathcal{P}(\{v_0, v_2\}) \setminus \{\{\}\}$= $\{\{v_0\}, \{v_0, v_1\}, \{v_1\}\} \cup \{\{v_0\}, \{v_0, v_2\}, \{v_2\}\}$ = $\{\{v_0\}, \{v_0, v_1\}, \{v_0, v_2\}, \{v_1\}, \{v_2\}\}$. Since $SH_{1_{res}} = \{\{res\}\}$ and $SH_{3_{v0}} = \{\{v_0\}, \{v_0, v_1\}, \{v_0, v_2\}\}$, $SH_4 = \{\{v_0, res\}, \{v_0, v_1, res\}, \{v_0, v_2, res\}\}$. The final abstract state $SH' = \{\{v_0\}, \{v_0, v_1\}, \{v_0, v_2\}, \{v_1\}, \{v_2\}\}$ is the union of $SH_3\vert_{-res} = SH_3$ and $SH_4\vert_{-res} \subset SH_3$. As required, $sh(G') \subseteq SH'$ holds after the removal of the auxiliary variable res from G'.

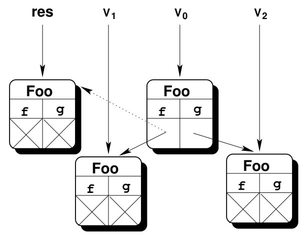

Fig. 4. Graph G.

Conditional Statements: In the case where the guard is (v==`null`), the type component may contain definite information about whether a variable v is not null ($null \notin \tau(v)$). If we cannot determine exactly the nullity of v (i.e., **mayBeNull**(τ, v) is true), then the final state is the least upper bound of the resulting set sharing for the two branches. In particular, $SH_1 \sqcup SH_2 = SH_1 \cup SH_2$.

In the case where the condition is $v{=}{=}w$, the sharing information may be enough to tell that the two variables are definitely equal, because they are both null: **mustAlias**$(SH, v, w) =$ (**mustBeNull**(SH, v) and **mustBeNull**(SH, w)). On the other hand, v and w do not share if they do not appear together within a subset of SH. Therefore **mayAlias**$(SH, v, w) =$ ($\overline{\textbf{mustAlias}}(SH, v, w)$ and $SH_{\{v,w\}} \neq \emptyset$). It is important to see that sharing information does not imply equality: a set sharing like $\{\{v, w\}\}$ indicates that v and w might reach a common object, not that they must be aliases.

Example 4. Given a command like `if (cond)` $v0$ `=` $v1$ `else {`$v0$ `= null;` $v1$ `= null}`, and assuming an initial abstract state $SH = \emptyset$ that does not contain enough information to determine $cond$, the set sharing corresponding to the `if` branch is $SH_1 = \{\{v_0, v_1\}\}$. The abstract state after simulating the `else` branch is $SH_2 = \{\}$. Therefore, the final state is $SH' = SH_1 \cup SH_2 = \{\{v_0, v_1\}\}$. However, SH' does not imply that v_0 necessarily shares with v_1, even when they appear together in SH', but that v_0 *might* reach an object reachable from v_1 in some of the concrete states approximatted by SH'; in the example, if $cond$ would be false, both variables are null and do not share.

4 Semantics as ZBDD operations

Zero-suppressed BDDs (ZBDDs) [8, 9] are a data structure similar to binary decision diagrams (BDDs) [3], but designed to encode sets of combinations (i.e., sets of sets of primitive elements). To encode the set sharing domain using ZBDDs, we define the primitive elements to be the variables in the program being analyzed. ZBDDs have been demostrated to perform better [15, 14] than standard BDDs when encoding sets of combinations that are sparse in the sense that a) the set contains just a small fraction of all the possible combinations, and b) each combination contains just a few literals. A ZBDD is a rooted directed acyclic graph (DAG) of non-terminal and terminal nodes. Each non-terminal ZBDD node is labeled with a variable, and has two outgoing edges to other nodes, called the zero-edge and the one-edge. There are two terminal nodes, the zero node and the one node. They do not have variables or outgoing edges. The universe of all variables is totally ordered, and the order of the variables appearing on the nodes of any path through the ZBDD is consistent with the total order. Each path through the ZBDD that ends at the one terminal node defines a set of variables. The set contains a variable v if the path passes through a node labeled with v, and leaves the node along its one edge. Assuming the variable ordering is fixed, the smallest ZBDD representing a given set of sets is unique, and can be found efficiently.

Example 5. Assume a set of variables $Var = \{v_0, v_1, v_2\}$ and the variable ordering v_0, v_1, v_2. The unique smallest ZBDD representing the set of sets $\{\{v_0, v_2\}, \{v_1\}\}$ is the ZBDD shown in Fig. 6. There are two paths from the root of the ZBDD to the one terminal node. On the path containing the v_0 and v_1, only the node labeled v_1 is exited through the one edge; thus, this path represents the set $\{v_1\}$. On the path containing v_0 and v_2, both nodes are exited through their one edges; thus, this path represents the set $\{v_0, v_2\}$.

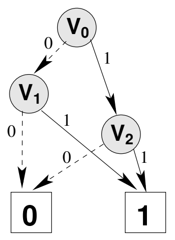

Fig. 6.

Efficient algorithms exist for common operations on the set of sets encoded by a ZBDD, including union (denoted +), intersection, set difference, product ($SH_1 * SH_2 = \{\, S_1 \cup S_2 \mid S_1 \in SH_1 \text{ and } S_2 \in SH_2\}$), and division ($SH/v = \{\, S \setminus \{v\} \mid S \in SH \text{ and } v \in S\}$ and $SH\%v = \{\, S \in SH \mid v \notin S\}$).

$\mathcal{SE}^I_\pi[\![\texttt{null}]\!](SH)$
$SH' = SH$
$\mathcal{SE}^I_\pi[\![\texttt{new}\ k]\!](SH)$
$SH' = SH + res$
$\mathcal{SE}^I_\pi[\![v]\!](SH)$
$SH' =$ **setResEqTo**(SH, v)
$\mathcal{SE}^I_\pi[\![v\,.\,\texttt{f}]\!](SH)$
$SH' = \begin{cases} \bot \text{ if } \textbf{mustBeNull}(SH, v) \\ SH + v * res * \textbf{powUnion}(SH/v) \text{ else} \end{cases}$

```
setResEqTo(P) {
  if ($P = 0$ or $P = 1$ or $P.top > v$)
    return $P$
  if ($P.top < v$))
    return Getnode($P.top$,$P_0$,$P_1$)
  return Getnode($P.top$,$P_0$,res*$P_1$)
}

powUnion(P) {
  if ($P = 0$ or $P = 1$)
    return $P$
  $R_0 \leftarrow$ powUnion($P_0$)
  $R_1 \leftarrow$ powUnion($P_1$)
  return Getnode($P.top$,$R_o + R_1$,$1 + R_1$)
}
```

Fig. 7. Abstract semantics for the expressions as ZBDD operations.

A set sharing like $SH = \{\{v_0, v_2\}, \{v_1\}\}$ is expressed in ZBDD notation as $SH = v_0v_2 + v_1$. Note that we will denote single literal sets by a single lower case letter (like v), while generic ZBDDs will be referred to with double upper case (normally, SH). For instance, given the set sharings $SH = v_0v_2 + v_1$ and v_0 , an expression like $SH * v_0 = v_0v_1 + v_0v_2$ is legal. The empty set is written as 0, and the set containing only the empty set is written as 1.

4.1 Expressions and Commands; Native Operations

Figs. 7 and 9 show the ZBDD version of the transfer functions[5] in Fig. 2 and 5. For most of the set operations, there is an equivalent native ZBDD operation. For instance, $SH_1 \uplus SH_2$ is equivalent to $SH_1 * SH_2$ and SH_{-v} is equivalent to $SH\%v$. This correspondence is useful because it results in no gap between the denotational semantics of Sect. 3 and the implementation. However, we added a number of non-standard ZBDD operators to improve the readability of the equations. The set of elements in SH containing v (SH_v, in set notation) is obtained via $SH//v = SH/v * v$. We delete all the ocurrences of v in SH using **projOut**$(SH, v) = SH/v + SH\%v - 1$. The unit set 1 (which represents the set containing the empty set) has to be deleted because SH might contain the single literal v, as we did in the corresponding *project out* set operator $SH|_{-v}$.

In other occasions, we created new ZBDD operators because of efficiency reasons. For instance, the variable load set equation $SH' = (\{\{res\}\} \uplus SH_v) \cup SH_{-v}$ can be expressed as $SH' = res * (SH//v) + SH\%v$. This combination of standard operators, while intuitive, has the disadvantage of being inefficient in practice. Since we expect this function to be invoked with high frequency (every time a variable is on the right hand side of an assignment), we devised a dedicated ZBDD algorithm that computes the same result, **setResEqTo**(SH, v). The algorithm, shown in Fig. 7, uses the same notation as in [9]: P_0 and P_1 for the graph reachable through the zero-edge and one-edge, respectively, $P.top$ for the current variable, and $Getnode(v, P_0, P_1)$ for the procedure that generates a node with the variable v and subgraphs P_0 and P_1. The correctness of **setResEqTo**(SH, v) is based on a variable order in which res is always the last variable,

[5] The type component is again omitted, although in practice it is updated in an identical fashion to Fig. 13 and 14.

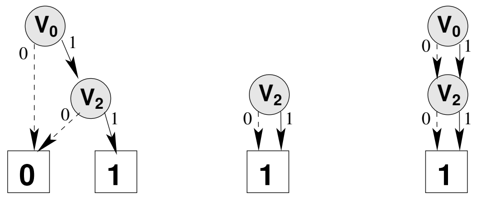

Fig. 8. ZBDDs representing v_0v_2, $1 + v_2$, and $1 + v_0 + v_0v_2 + v_2$.

the one closer to the leaves. Given this precondition, we only need to find v in the graph, and then multiply its one-edge child by res, which will preserve the variable order.

With the basic ZBDD operators and **setResEqTo** we can understand the transfer functions of the null, new, and variable load expressions. The field load, on the other hand, depends on the ZBDD version of the predicate that determines whether a variable is null: **mustBeNull**(SH, v) = $(SH/v = 0)$. It also requires computing the union of the powersets of the elements of a set sharing SH: $\{\mathcal{P}(S) \mid S \in SH\}$. Although this seems to be a complex operation, it has a very natural description in terms of an algorithm in ZBDDs. We have devised a native ZBDD algorithm, **powUnion**(SH), shown in pseucode in Fig. 7. The correctness proof of the algorithm is given in the appendix. This native implementation will prove to be fundamental for the scalability of the analysis (Sect. 5).

Example 6. We show how the native algorithm computes **powUnion**(v_0v_2). Fig. 8 contains the initial ZBDD representing v_0v_2 (left). To compute **powUnion** for the original ZBDD, we first recursively compute **powUnion** for the node labeled v_2. When **powUnion** is applied to the node labeled v_2, which represents the set v_2, we have $R_0 = P_0 = 0$ and $R_1 = P_1 = 1$. The result is a node labeled v_2 with zero successor $R_0 + R_1 = 1$ and one successor $1 + R_1 = 1 + 1 = 1$, shown in the center of the figure. This ZBDD represents the powerset of v_2, namely $1 + v_2$. We will call this ZBDD N. When we compute **powUnion** of the original ZBDD, $R_0 = P_0 = 0$, and $R_1 = N$. This step generates a node with value v_0, zero successor $R_0 + R_1 = 0 + N = N$, and one successor $1 + R_1 = 1 + N = N$. Because both nodes are identical (*reduction rule* applied within $Getnode$), we can delete one of them and change both edges of v_0 to lead to just one N, as shown in the right ZBDD in Fig. 8. The resulting graph represents $1 + v_0 + v_0v_2 + v_2$.

The command semantics (Fig. 9) is described in terms of the operators listed before. We only add a new predicate, used when checking if two variables might be aliases: **mayAlias**(SH, v, w) = $(\overline{\textbf{mustAlias}(SH, v, w)}$ and $SH/(v * w) \neq 0)$. The following example shows how the field store from Example 3 would be calculated using ZBDDs.

Example 7. Assume we start evaluating v_0 `.f=` *expr* in an abstract set sharing $SH_1 = v_0v_1{+}v_0v_2{+}res$. Because all the sharings in SH_1 contain v_0 or res, $SH_2 = 0$. The union of the powersets of $SH_1//v_0 = v_0v_1 + v_0v_2$ is calculated in a very similar fashion to the last example, and results in a set sharing $1{+}v_0{+}v_0v_1{+}v_0v_2{+}v_1{+}v_2$. Therefore, $SH_3 =$ **projOut**$(v_0{+}v_0v_1{+}v_0v_2{+}v_1{+}v_2, res) = v_0{+}v_0v_1{+}v_0v_2{+}v_1{+}v_2$. The last component of the result is $SH_4 = (SH_1/res) * (SH_3//v_0) = 1 * (SH_3//v_0) = v_0 + v_0v_1 + v_0v_2$. The result is $SH' = 0 + SH_3 + SH_4 = SH_3 = v_0 + v_0v_1 + v_0v_2 + v_1 + v_2$, which is the same result obtained in the set example.

$\mathcal{SC}^I_\pi[\![v\texttt{=}expr]\!](SH)$
$SH_1 = \mathcal{SE}^I_\pi[\![expr]\!](SH)$ $SH_2 = \textbf{projOut}(SH_1\%res, v)$ $SH' = SH_1/res * v + SH_2$
$\mathcal{SC}^I_\pi[\![v\texttt{.f=}expr]\!](SH)$
$SH_1 = \mathcal{SE}^I_\pi[\![expr]\!](SH)$ $SH_2 = SH_1\%v\%res$ $SH_3 = \textbf{projOut}($ $\quad \textbf{powUnion}(SH_1/\!/v) - 1, res)$ $SH_4 = (SH_1/res) * (SH_3/\!/v)$ $SH' = \begin{cases} \bot \text{ if } \textbf{mustBeNull}(SH_1, v) \\ SH_2 + SH_3 + SH_4 \text{ else} \end{cases}$

$\mathcal{SC}^I_\pi[\![\texttt{if } v\texttt{==null}\ com_1 \texttt{ else } com_2]\!](SH)$
$SH_1 = \mathcal{SC}^I_\pi[\![com_1]\!](\textbf{projOut}(SH, v))$ $SH_2 = \mathcal{SC}^I_\pi[\![com_2]\!](SH)$ $SH' = \begin{cases} SH_1 \text{ if } \textbf{mustBeNull}(SH, v) \\ SH_1 + SH_2 \text{ if } \textbf{mayBeNull}(\tau, v) \\ SH_2 \text{ else} \end{cases}$
$\mathcal{SC}^I_\pi[\![\texttt{if } v\texttt{==}w\ com_1 \texttt{ else } com_2]\!](SH)$
$SH_1 = \mathcal{SC}^I_\pi[\![com_1]\!](SH)$ $SH_2 = \mathcal{SC}^I_\pi[\![com_2]\!](SH)$ $SH' = \begin{cases} SH_1 \text{ if } \textbf{mustAlias}(SH, v, w) \\ SH_1 + SH_2 \text{ if } \textbf{mayAlias}(SH, v, w) \\ SH_2 \text{ else} \end{cases}$
$\mathcal{SC}^I_\pi[\![com_1\texttt{;}\,com_2]\!](SH)$
$SH' = \mathcal{SC}^I_\pi[\![com_2]\!](\mathcal{SC}^I_\pi[\![com_1]\!](SH))$

Fig. 9. Abstract semantics for the commands.

5 Experiments

To evaluate the scalability (in terms of memory usage and running time) of the ZBDD approach, we compared it to an alternative representation for set sharings based on sets of bitsets. Bitsets are a fast, light representation compared to other ways of representing a set sharing. In a bitset, each bit b_i indicates if the variable v_i is in the sharing ($b_i = 1$) or not ($b_i = 0$). Our first implementation used the Java library where a `BitSet` is an array of double words. However, our first experiments showed that this approach does not scale beyond set sharings with more than a few thousand elements. For this reason, we replaced the library implementation by a lightweight version, which only requires a single word to represent each sharing. This effectively limits the number of variables to be not more than 32 for the bitset approach, which is reasonable when confronted with powerset operations. In all the experiments we assume that the number of variables n is bounded by 32, but note that the ZBDD implementation scales well for larger set sharings, and could handle bigger values of n. Our ZBDD implementation of set sharing is based on the JDD library [21].

Several characteristics of set sharings influence the memory usage and the performance of the data structure representing them. Although the number of variables n seems to be important, our two representations are independent of this parameter. In the case of the bitsets, because we use 32 bits to store every sharing, independently of the number of variables. In the case of ZBDDs, only the statistical distribution of the sharings (i.e., their sparsity) influences the number of nodes required to represent the information, and therefore the memory usage and performance of the ZBDD. For the same reason, the behavior of the two data structures is independent of the *sharing density* of SH, i.e., the proportion of the number of sharings over the maximum possible: $SH_d = |SH|/2^n$.

The most decisive factor is the number of sharings $|SH|$. Because we allocate a new bitset every time a new sharing is added, the performance of the set of bitsets approach is inversely proportional to $|SH|$. In the case of ZBDDs we also have to take into account the *variable density*. This metric is the average number of variables per sharing: $v_d =$

$\frac{1}{n*|SH|} * \sum_{S \in SH} |S|$. A small variable density is synonymous with a sparse set sharing, and therefore we can expect the ZBDD to perform inversely proportional to the metric. We now examine how the number of sharings and the variable density relate to memory consumption and execution times in our experiments.

Memory Usage: We generated random set sharings and measured the space requirements for the Java objects backing the set of bitsets and ZBDD as reported by a profiler [12]. The different memory usages are shown on the left of Fig. 10. The plot shows that the ZBDD scales better than the bitset solution. The differences are more significant (a factor of 5) for large values of $|SH|$. A set of bitsets uses 56 bytes per sharing, less than the 80 required by a set of the JDK 1.5 `BitSet` class. At one million sharings, the set of bitsets requires more than 56Mb, while the same information occupies 12Mb in the ZBDD version ($v_d = 0.28$). The staircase behavior of the ZBDD memory usage function is due to the *capacity* of the array storing the node list (ZBDDs are represented as arrays in JDD), which doubles when the load exceeds a certain threshold.

In the leftmost graph in Fig. 10 we did not take into account the effect of variable density. The other plot in that figure demonstrates how ZBDDs benefit from sparse variable distributions. This time we do not show the number of Kbytes in the y-axis, but rather the number of nodes in the binary decision diagram. As expected, sparse sharings require fewer nodes than those that are more dense in terms of v_d. In the experiments, the number of nodes goes down by an average 38.2% from $v_d = 0.34$ to $v_d = 0.22$.

Speed: We measured the number of milliseconds required to compute the semantics of the most significant operations (variable load/store, and field load/store), given a random initial set sharing. We disabled the JDD cache for the experiments. All the measurements were done on a Pentium M 1.73Ghz with 1Gb of RAM. The virtual machine was Sun's JVM 1.5.0 running on Ubuntu 6.06. The results are in Figs. 11 and 12.

The time required to simulate a variable load presents a similar, linear behavior in both cases; the bitset version is 14.6% faster in the average. Although not reflected in Fig. 11, the native operation **setResEqto** takes half the time of the equivalent composition of ZBDD operations (see Sect. 4). For the variable store, both running times are roughly linear in the number of sharings. However, the lack of a native ZBDD implementation results in running times noticeably slower than those of the set of bitsets. It remains an open question whether a dedicated ZBDD algorithm can be devised for this command.

The powerset operation is a major obstacle for a feasible implementation of set sharing using the sets of bitsets. Both the field load and field store transfer functions depend

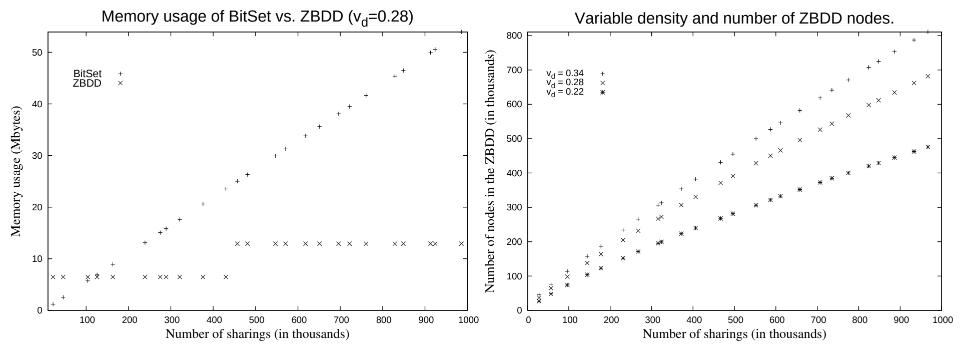

Fig. 10. Memory usage experiments. Over 25 runs.

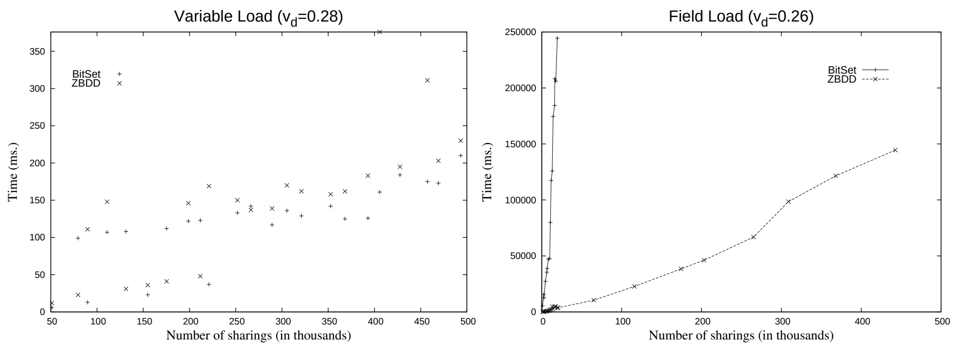

Fig. 11. Performance of a set of bitsets vs ZBDD (expressions). Over 25 runs.

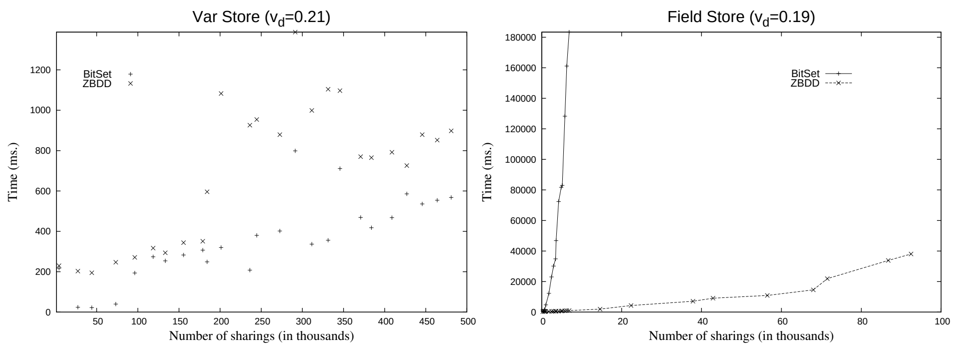

Fig. 12. Performance of a set of bitsets vs ZBDD (commands). Over 25 runs.

on this operation. While the ZBDD **powUnion** algorithm requires reasonable times for calculating the union of many powersets, the bitset implementation presents exponential growth with respect to the number of sharings. For example, it needs half a minute to compute the output state for a field load in which the initial sharing has 5,000 elements. The ZBDD implementation finishes the same operation in less than 600ms. The field store (Fig. 12, right), which is a more complex operation, presents a similar pattern, although the running times are always significantly larger than for the field load.

6 Related Work

The ideas presented in this paper build on one hand on [16], where a first definition of a set sharing-based analysis for Java was introduced and shown to offer advantages in certain cases with respect to pair sharing-based analyses. We offer substantially improved definitions of the abstract semantics, a reduction in the number of components of an abstract state, and in some cases (like the field load and store) more precise abstract operations. In addition, a significant difference with our previous work is of course the use of Zero-suppressed Decision Diagrams to efficiently implement the analysis domain. This is done without having to redesign any of the existing abstract operations. The experiments in [16] involved small set sharings (of at most 50 elements at a time) while in this paper we show how with ZBDDs we can scale up to thousands of sharings and still get reasonable times.

There has been extensive work in recent years on the use of BDDs [24, 2, 22, 25] to represent (abstract) *points-to* information. In these abstractions, information is stored in the form of (v, a) pairs, where each such pair indicates that v may point to the allocation site a. As mentioned before, set sharing information can be interpreted as an *abstraction* of points-to information where instead of representing which exact objects can be pointed to by a variable, the domain captures only which sets of variables may point transitively to the same object. Thus, our analysis works at a different level since the set sharing encoding can result in some loss of precision, but offers the advantage of more compact representation.

ZBDDs were introduced by Minato [8] and applied to a great diversity of problems in model checking (e.g., [23, 5, 10]). More recently, Lhoták *et al.* have applied ZBDDs to the exploration of infinite state spaces [14] in the context of points-to analysis. The main differences between this work and [14] are one hand the abstraction used (set sharing vs. points-to pairs) and on the other that in the approach proposed the domain does not require relational information, i.e., we can use existing ZBDD libraries [20, 21] directly in our implementation.

To the extent of our knowledge, this is the first work that relates set sharing analysis with ZBDDs or presents implementation results for the set-sharing domain using any type of binary decision diagram. In the logic programming realm, there has been a significant amount of work related to set sharing-based analysis for the automatic parallelization of Prolog programs (e.g., [11, 17, 18]). However, the abstract operations show significant differences with the ones required for an imperative/OO language. Furthermore, to the best of our knowledge, all existing implementations use lists of lists to represent set sharings. In [4] a connection between the set sharing domain and standard BDDs is suggested, but no implementation or experimental results are provided and there is no mention of ZBDDs. More recent work [19] for Java presents results for a BDD-based implementation of the less precise pair sharing domain [16]. Because in this case the abstraction is a set of pairs (and not a set of sets), the representation used is quite different from ours.

References

1. David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. *Proc. of OOPSLA'96, SIGPLAN Notices*, 31(10):324–341, October 1996.
2. Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to Analysis Using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 103–114. ACM Press, 2003.
3. Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.
4. Michael Codish, Harald Søndergaard, and Peter J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.
5. O. Coudert. Solving Graph Optimization Problems with ZBDDs, 1997.
6. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
7. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP*, pages 77–101, 1995.
8. Shin ichi Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *DAC*, pages 272–277, 1993.
9. Shin ichi Minato. *Binary Decision Diagrams and Applications for VLSICAD*. Kluwer, Norwell/MA, USA, 1996.

10. Shin ichi Minato. Zero-suppressed BDDs and their Applications. *STTT*, 3(2):156–170, 2001.
11. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *North American Conference on Logic Programming*, 1989.
12. JProfiler. `http://www.ej-technologies.com/products/jprofiler/`.
13. Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.
14. O. Lhoták, S.Curial, and J.N.Amaral. Using ZBDDs in Points-to Analysis. In *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing*, 2007.
15. Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.
16. M. Méndez-Lojo and M. Hermenegildo. Precise Set Sharing Analysis for Java-style Programs. In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.
17. K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.
18. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.
19. É Payet and F. Spoto. Magic-Sets Transformation for the Analysis of Java Bytecode. In *Proceedings of the 14th International Static Analysis Symposium (SAS'07)*, 2007.
20. F. Somenzi. *CUDD: CU Decision Diagram Package*, 2005. `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.
21. A. Vahidi. *JDD: A Pure Java BDD Library*, 2008. `http://javaddlib.sourceforge.net/jdd/index.html`.
22. John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144. ACM, 2004.
23. Tomohiro Yoneda, Hideyuki Hatori, Atsushi Takahara, and Shin ichi Minato. BDDs vs. Zero-Suppressed BDDs: for CTL Symbolic Model Checking of petri nets. In *FMCAD*, pages 435–449, 1996.
24. Jianwen Zhu. Symbolic Pointer Analysis. In *ICCAD*, pages 150–157, 2002.
25. Jianwen Zhu and Silvian Calman. Symbolic Pointer Analysis Revisited. In *PLDI*, pages 145–157, 2004.

A Complete semantics for the expressions and commands

Contained in figures 13 and 14. In the case of the type component, the least upper bound is computed as $\tau_1 \sqcup \tau_2 = \{\,(v, \tau_1(v) \cup \tau_2(v)) \mid v \in Var\}$.

B PowUnion: correctness proof

Proof. **powUnion**(SH) correctly computes $\bigcup\limits_{S \in SH} \mathcal{P}(S)$:

$\textbf{powUnion}(ZBDD(a, P_0, P_1)) = \textbf{powUnion}(P_0 + a * P_1) = \textbf{powUnion}(P_o) + \textbf{powUnion}(a * P_1) = \bigcup\limits_{S \in P_0} \mathcal{P}(S) \cup \bigcup\limits_{S \in P_1} (\mathcal{P}(S \cup \{a\}) = \bigcup\limits_{S \in P_0} \mathcal{P}(S) \cup \{\{a\}\} \cup \bigcup\limits_{S \in P_1} (\mathcal{P}(S) \uplus \{\{\}, \{a\}\}) = \bigcup\limits_{S \in P_0} \mathcal{P}(S) \cup \bigcup\limits_{S \in P_1} \mathcal{P}(S) \cup \{\{a\}\} \cup \bigcup\limits_{S \in P_1} (\mathcal{P}(S) \uplus \{\{a\}\}) = \bigcup\limits_{S \in P_0 \cup P_1} \mathcal{P}(S) \cup \{\{a\}\} \cup (\{\{a\}\} \uplus \bigcup\limits_{S \in P_1} \mathcal{P}(S)) = ZBDD(a, \textbf{powUnion}(P_0 + P_1), 1 + \textbf{powUnion}(P_1))$.

$\mathcal{SE}^I_\pi[\![\texttt{null}]\!](SH,\tau)$
$SH' = SH$
$\tau' = \tau[res \mapsto \{\text{null}\}]$
$\mathcal{SE}^I_\pi[\![\texttt{new}\ k]\!](SH,\tau)$
$SH' = SH\ \cup\ \{\{res\}\}$
$\tau' = \tau[res \mapsto \{\text{k}\}]$
$\mathcal{SE}^I_\pi[\![v]\!](SH,\tau)$
$SH' = (\{\{res\}\} \uplus SH_v) \cup SH_{-v}$
$\tau' = \tau[res \mapsto \tau(v)]$
$\mathcal{SE}^I_\pi[\![v\,\texttt{.}\,\texttt{f}]\!](SH,\tau)$
$SH' = \begin{cases} \bot \text{ if } \textbf{mustBeNull}(SH,v) \\ SH \cup (\{\{v,res\}\} \uplus \bigcup_{S\in SH_v} \mathcal{P}(S\vert_{-v})) \text{ else} \end{cases}$
$\tau_1 = \tau[v \mapsto (\tau(v) \setminus \{\text{null}\}), res \mapsto (\downarrow F(v.f) \cup \{\text{null}\})]$

Fig. 13. Abstract semantics for the expressions as set operations

$\mathcal{SC}^I_\pi[\![v\texttt{=}expr]\!](SH,\tau)$
$(SH_1,\tau_1) = \mathcal{SE}^I_\pi[\![expr]\!](SH,\tau)$
$SH_2 = SH_1\vert_{-v}$
$SH' = SH_2\vert^v_{res}$
$\tau' = \tau_1[v \mapsto \tau_1(res)] \setminus (res, \tau_1(res))$
$\mathcal{SC}^I_\pi[\![v\,\texttt{.}\,\texttt{f=}expr]\!](SH,\tau)$
$(SH_1,\tau_1) = \mathcal{SE}^I_\pi[\![expr]\!](SH,\tau)$
$SH_2 = SH_{1_{-\{v,res\}}}$
$SH_3 = \bigcup_{S\in SH_{1_v}} \mathcal{P}(S)\ \setminus \{\{\}\}$
$SH_4 = SH_{1_{res}} \uplus SH_{3_v}$
$SH' = \begin{cases} \bot \text{ if } \textbf{mustBeNull}(SH_1,v) \\ SH_2 \cup (SH_3 \cup SH_4)\vert_{-res} \text{ else} \end{cases}$
$\tau' = \tau_1[v \mapsto (\tau_1(v) \setminus \{\text{null}\})] \setminus (res, \tau_1(res))$

$\mathcal{SC}^I_\pi[\![\texttt{if}\ v\texttt{==null}\ com_1\ \texttt{else}\ com_2]\!](SH,\tau)$
$SH_1 = SH\vert_{-v}$
$\tau_1 = \tau[v \mapsto \{\text{null}\}]$
$\sigma_1 = \mathcal{SC}^I_\pi[\![com_1]\!](SH_1,\tau_1)$
$\tau_2 = \tau[v \mapsto (\tau(v) \setminus \{\text{null}\})]$
$\sigma_2 = \mathcal{SC}^I_\pi[\![com_2]\!](SH,\tau_2)$
$(SH',\tau') = \begin{cases} \sigma_1 \text{ if } \textbf{mustBeNull}(SH,v) \\ \sigma_1 \sqcup \sigma_2 \text{ if } \textbf{mayBeNull}(\tau,v) \\ \sigma_2 \text{ else} \end{cases}$
$\mathcal{SC}^I_\pi[\![\texttt{if}\ v\texttt{==}w\ com_1\ \texttt{else}\ com_2]\!](SH,\tau)$
$\sigma_1 = \mathcal{SC}^I_\pi[\![com_1]\!](SH,\tau)$
$\sigma_2 = \mathcal{SC}^I_\pi[\![com_2]\!](SH,\tau)$
$(SH',\tau') = \begin{cases} \sigma_1 \text{ if } \textbf{mustAlias}(SH,v,w) \\ \sigma_1 \sqcup \sigma_2 \text{ if } \textbf{mayAlias}(SH,v,w) \\ \sigma_2 \text{ else} \end{cases}$
$\mathcal{SC}^I_\pi[\![com_1\texttt{;}\,com_2]\!](SH,\tau)$
$(SH',\tau') = \mathcal{SC}^I_\pi[\![com_2]\!](\mathcal{SC}^I_\pi[\![com_1]\!](SH,\tau))$

Fig. 14. Abstract semantics for the commands.