Precise Set Sharing Analysis for Java-Style Programs

Mario Méndez-Lojo[1] and Manuel V. Hermenegildo[1,2]

[1] University of New Mexico (USA)
[2] Technical University of Madrid (Spain)

Abstract. Finding useful *sharing* information between instances in object-oriented programs has recently been the focus of much research. The applications of such static analysis are multiple: by knowing which variables definitely do not share in memory we can apply conventional compiler optimizations, find coarse-grained parallelism opportunities, or, more importantly, verify certain correctness aspects of programs even in the absence of annotations. In this paper we introduce a framework for deriving precise sharing information based on abstract interpretation for a Java-like language. Our analysis achieves precision in various ways, including supporting multivariance, which allows separating different contexts. We propose a combined *Set Sharing + Nullity + Classes* domain which captures which instances do not share and which ones are definitively null, and which uses the classes to refine the static information when inheritance is present. The use of a set sharing abstraction allows a more precise representation of the existing sharings and is crucial in achieving precision during interprocedural analysis. Carrying the domains in a combined way facilitates the interaction among them in the presence of multivariance in the analysis. We show through examples and experimentally that both the set sharing part of the domain as well as the combined domain provide more accurate information than previous work based on pair sharing domains, at reasonable cost.

1 Introduction

The technique of Abstract Interpretation [8] has allowed the development of sophisticated program analyses which are at the same time provably correct and practical. The semantic approximations produced by such analyses have been traditionally applied to high- and low-level optimizations during program compilation, including program transformations. More recently, promising applications of such semantic approximations have been demonstrated in the more general context of program development, such as verification and static debugging.

Sharing analysis [14,20,26] aims to detect which variables do not share in memory, i.e., do not point (transitively) to the same location. It can be viewed as an abstraction of the graph-based representations of memory used by certain classes of alias analyses (see, e.g., [31,5,13,15]). Obtaining a safe (over-) approximation of which instances might share allows parallelizing segments of code,

F. Logozzo et al. (Eds.): VMCAI 2008, LNCS 4905, pp. 172–187, 2008.

improving garbage collection, reordering execution, etc. Also, sharing information can improve the precision of other analyses.

Nullity analysis is aimed at keeping track of null variables. This allows for example verifying properties such as the absence of null-pointer exceptions at compile time. In addition, by combining sharing and null information it is possible to obtain more precise descriptions of the state of the heap.

In type-safe, object-oriented languages *class* analysis [1,3,10,22], (sometimes called *type* analysis) focuses on determining, in the presence of polymorphic calls, which particular implementation of a given method will be executed at runtime, i.e., what is the specific class of the called object in the hierarchy. Multiple compilation optimizations benefit from having precise class descriptions: inlining, dead code elimination, etc. In addition, class information may allow analyzing only a subset of the classes in the hierarchy, which may result in additional precision.

We propose a novel analysis which infers in a combined way *set sharing*, *nullity*, and *class* information for a subset of Java that takes into account most of its important features: inheritance, polymorphism, visibility of methods, etc. The analysis is multivariant, based on the algorithm of [21], which allows separating different contexts, thus increasing precision. The additional precision obtained from context sensitivity has been shown to be important in practice in the analysis of object-oriented programs [30].

The objective of using a reduced cardinal product [9] of these three abstract domains is to achieve a good balance between precision and performance, since the information tracked by each component helps refine that of the others. While in principle these three analyses could be run separately, because they interact (we provide some examples of this), this would result in a loss of precision or require an expensive iteration over the different analyses until an overall fixpoint is reached [6,9]. In addition note that since our analysis is multivariant, and given the different nature of the properties being tracked, performing analyses separately may result in different sets of abstract values (contexts) for each analysis for each program point. This makes it difficult to relate which abstract value of a given analysis corresponds to a given abstract value of another analysis at a given point. At the other end of things, we prefer for clarity and simplicity reasons to develop directly this three-component domain and the operations on it, rather than resorting to the development of a more unified domain through (semi-)automatic (but complex) techniques [6,7]. The final objectives of our analysis include verification, static debugging, and optimization.

The closest related work is that of [26] which develops a *pair*-sharing [27] analysis for object-oriented languages and, in particular, Java. Our description of the (set-)sharing part of our domain is in fact based on their elegant formalization. The fundamental difference is that we track *set* sharing instead of *pair* sharing, which provides increased accuracy in many situations and can be more appropriate for certain applications, such as detecting independence for program parallelization. Also, our domain and abstract semantics track additionally nullity and classes in a combined fashion which, as we have argued above, is

$$
\begin{array}{lll}
\mathit{prog} & ::= & \mathit{class\_decl}^* \\
\mathit{class\_decl} & ::= & \texttt{class}\ \mathrm{k}_1\ [\texttt{extends}\ \mathrm{k}_2]\ \mathit{decl}^*\ \mathit{meth\_decl}^* \\
\mathit{meth\_decl} & ::= & \mathit{vbty}\ (\mathrm{t}_{ret}|\texttt{void})\ \mathrm{meth}\ \mathit{decl}^*\ \mathit{com} \\
\mathit{vbty} & ::= & \texttt{public} \mid \texttt{private} \\
\mathit{com} & ::= & \mathrm{v} = \mathit{expr} \mid \mathrm{v}.f = \mathit{expr} \\
 & & \mid \mathit{decl} \mid \texttt{skip} \\
 & & \mid \texttt{return}\ \mathit{expr} \mid \mathit{com};\mathit{com} \\
 & & \mid \texttt{if}\ \mathrm{v}\ (==|!=)\ (\texttt{null}|\mathrm{w})\ \mathit{com}\ \texttt{else}\ \mathit{com} \\
\mathit{decl} & ::= & \mathrm{v{:}t} \\
\mathit{var\_lit} & ::= & \mathrm{v} \mid \mathrm{a} \\
\mathit{expr} & ::= & \texttt{null} \mid \texttt{new}\ \mathrm{k} \mid \mathrm{v}.f \mid \mathrm{v.m}(\mathrm{v}_1, \ldots \mathrm{v}_n) \mid \mathit{var\_lit}
\end{array}
$$

Fig. 1. Grammar for the language

particularly useful in the presence of multivariance. In addition, we deal directly with a larger set of object features such as inheritance and visibility. Finally, we have implemented our domains (as well as the pair sharing domain of [26]), integrated them in our multivariant analysis and verification framework [17], and benchmarked the system. Our experimental results are encouraging in the sense that they seem to support that our contributions improve the analysis precision at reasonable cost.

In [23,24], the authors use a *distinctness* domain in the context of an abstract interpretation framework that resembles our sharing domain: if two variables point to different abstract locations, they do not share at the concrete level. Their approach is closer to shape analysis [25] than to sharing analysis, which can be inferred from the former. Although information retrieved in this way is generally more precise, it is also more computationally demanding and the abstract operations are more difficult to design. We also support some language constructs (e.g., visibility of methods) and provide detailed experimental results, which are not provided in their work.

Most recent work [28,18,30] has focused on context-sensitive approaches to the points-to problem for Java. These solutions are quite scalable, but flow-insensitive and overly conservative. Therefore, a verification tool based on the results of those algorithms may raise spurious warnings. In our case, we are able to express sharing information in a safe manner, as invariants that all program executions verify at the given program point.

2 Standard Semantics

The source language used is defined as a subset of Java which includes most of its object-oriented (inheritance, polymorphism, object creation) and specific (e.g., access control) features, but at the same time simplifies the syntax, and does not deal with interfaces, concurrency, packages, and static methods or variables. Although we support primitive types in our semantics and implementation, they will be omitted from the paper for simplicity.

```
class Element {
   int value;
   Element next;}

class Vector {
   Element first;

    public void add(Element el) {
        Vector v  = new Vector();
        el.next  = null;
        v.first = el;
        append(v);
    }
}
```

```
public void append(Vector v) {

  if (this != v) {
      Element e = first;
      if (e == null)
          first = v.first;
      else {
         while (e.next != null)
            e = e.next;
         e.next = v.first;
      }
  }
}
```

Fig. 2. Vector example

The rules for the grammar of this language are listed in Fig. 1. The `skip` statement, not present in the Java standard specification [11], has the expected semantics. Fig. 2 shows an example program in the supported language, an alternative implementation for the `java.util.Vector` class of the JDK in which vectors are represented as linked lists. Space constraints prevent us from showing the full code here,[1] although the figure does include the relevant parts.

2.1 Basic Notation

We first introduce some notation and auxiliary functions used in the rest of the paper. By $\mapsto$ we refer to total functions; for partial ones we use $\rightarrow$. The powerset of a set s is $\mathcal{P}(s)$; $\mathcal{P}^+(s)$ is an abbreviation for $\mathcal{P}(s) \setminus \{\emptyset\}$. The *dom* function returns all the elements for which a function is defined; for the codomain we will use *rng*. A substitution $f[k_1 \mapsto v_1, \ldots, k_n, \mapsto v_n]$ is equivalent to $f(k_1) = v_1, \ldots, f(k_n) = v_n$. We will overload the operator for lists so that $f[K \mapsto V]$ assigns $f(k_i) = v_i$, $i = 1, \ldots, m$, assuming $|K| = |V| = m$. By $f|_{\_S}$ we denote removing S from $dom(f)$. Conversely, $f|_S$ restricts $dom(f)$ to S. For tuples $(f_1, \ldots, f_m)|_S = (f_1|_S, \ldots, f_m|_S)$. Renaming in the set s of every variable in S by the one in the same position in T ($|S| = |T|$) is written as $s|_S^T$. This operator can also be applied for renaming single variables. We denote by $\mathcal{B}$ the set of Booleans.

2.2 Program State and Sharing

With $\mathcal{M}$ we designate the set of all method names defined in the program. For the set of distinct identifiers (variables and fields) we use $\mathcal{V}$. We assume that $\mathcal{V}$ also includes the elements *this* (instance where the current method is executed),

[1] Full source code for the example can be found in
`http://www.clip.dia.fi.upm.es/~mario`

and *res* (for the return value of the method). In the same way, $\mathcal{K}$ represents the program-defined classes. We do not allow `import` declarations but assume as member of $\mathcal{K}$ the predefined class `Object`.

$\mathcal{K}$ forms a lattice implied by a subclass relation $\downarrow\colon \mathcal{K} \to \mathcal{P}(\mathcal{K})$ such that if $t_2 \in \downarrow t_1$ then $t_2 \leq_{\mathcal{K}} t_1$. The semantics of the language implies $\downarrow\texttt{Object} = \mathcal{K}$. Given $def : \mathcal{K} \times \mathcal{M} \mapsto \mathcal{B}$, that determines whether a particular class provides its own implementation for a method, the Boolean function $redef : \mathcal{K} \times \mathcal{K} \times \mathcal{M} \mapsto \mathcal{B}$ checks if a class k_1 redefines a method existing in the ancestor k_2: $redef(k_1, k_2, \mathrm{m}) = true$ iff $\exists k$ s.t. $def(k, \mathrm{m})$, $k_1 \leq_{\mathcal{K}} k <_{\mathcal{K}} k_2$.

Static types are accessed by means of a function $\pi : \mathcal{V} \mapsto \mathcal{K}$ that maps variables to their declared types. The purpose of an *environment* π is twofold: it indicates the set of variables accessible at a given program point and stores their declared types. Additionally, we will use the auxiliary functions $F(k)$ (which maps the fields of $k \in \mathcal{K}$ to their declared type), and $type_\pi(expr)$, which maps expressions to types, according to π.

The description of the memory state is based on the formalization in [26,12]. We define a frame as any element of $Fr_\pi = \{\, \phi \mid \phi \in dom(\pi) \mapsto Loc \cup \{\mathrm{null}\}\}$, where $Loc = \mathbb{I}^+$ is the set of memory locations. A frame represents the first level of indirection and maps variable names to locations except if they are null. The set of all objects is $Obj = \left\{\, k \star \phi \mid\ k \in \mathcal{K}, \phi \in Fr_{F(k)} \right\}$. Locations and objects are linked together through the memory $Mem = \{\, \mu \mid\ \mu \in Loc \mapsto Obj\}$. A new object of class k is created as $new(k) = k \star \phi$ where $\phi(f) = null\ \ \forall f \in F(k)$. The object pointed to by v in the frame ϕ and memory μ can be retrieved via the partial function $obj(\phi \star \mu, v) = \mu(\phi(v))$. A valid heap configuration (concrete state $\phi \star \mu$) is any element of $\Sigma_\pi = \{\, (\phi \star \mu) \mid\ \phi \in Fr_\pi, \mu \in Mem\}$. We will sometimes refer to a pair $(\phi \star \mu)$ with δ.

The set of locations $R_\pi(\phi \star \mu, v)$ reachable from $v \in dom(\pi)$ in the particular state $\phi \star \mu \in \Sigma_\pi$ is calculated as $R_\pi(\phi \star \mu, v) = \cup \left\{\, R^i_\pi(\phi \star \mu, v) \;\middle|\; i \geq 0 \right\}$, the base case being $R^0_\pi(\phi \star \mu, v) = \{(\phi(v))|_{Loc}\}$ and the inductive one $R^{i+1}_\pi(\phi \star \mu, v) = \cup \left\{\, rng(\mu(l).\phi))|_{Loc} \mid\ l \in R^i_\pi(\phi \star \mu, v) \right\}$. Reachability is the basis of two fundamental concepts: sharing and nullity. Distinct variables $V = \{v_1, \ldots, v_n\}$ *share* in the actual memory configuration δ if there is at least one common location in their reachability sets, i.e., $share_\pi(\delta, V)$ is true iff $\cap_{i=1}^n R_\pi(\delta, v_i) \neq \emptyset$. A variable $v \in dom(\pi)$ is *null* in state δ if $R_\pi(\delta, v) = \emptyset$. Nullity is checked by means of $nil_\pi : \Sigma_\pi \times dom(\pi) \mapsto \mathcal{B}$, defined as $nil_\pi(\phi \star \mu, v) = true$ iff $\phi(v) = null$.

The *run-time type* of a variable in scope is returned by $\psi_\pi : \Sigma_\pi \times dom(\pi) \mapsto \mathcal{K}$, which associates variables with their dynamic type, based on the information contained in the heap state: $\psi_\pi(\delta, v) = obj(\delta, v).k$ if $\overline{nil_\pi}(\delta, v)$ and $\psi_\pi(\delta, v) = \pi(v)$ otherwise. In a type-safe language like Java runtime types are congruent with declared types, i.e., $\psi_\pi(\delta, v) \leq_{\mathcal{K}} \pi(v)\ \ \forall v \in dom(\pi), \forall \delta \in \Sigma_\pi$. Therefore, a correct approximation of ψ_π can always be derived from π. Note that at the same program point we might have different run-time type states ψ^1_π and ψ^2_π depending on the particular program path executed, but the static type state is unique.

Denotational (compositional) semantics of sequential Java has been the subject of previous work (e.g., [2]). In our case we define a simpler version of that semantics for the subset defined in Sect. 2, described as transformations in the frame-memory state. The descriptions are similar to [26]. Expression functions $\mathcal{E}^I_\pi[\![]\!] : expr \mapsto (\Sigma_\pi \mapsto \Sigma_{\pi'})$ define the meaning of Java expressions, augmenting the actual scope $\pi' = \pi[res \mapsto type_\pi(exp)]$ with the temporal variable res. Command functions $\mathcal{C}^I_\pi[\![]\!] : com \mapsto (\Sigma_\pi \mapsto \Sigma_\pi)$ do the same for commands; semantics of a method `m` defined in class k is returned by the function $I(k.\mathtt{m}) : \Sigma_{input(k.m)} \to \Sigma_{output(k.m)}$. The definition of the respective environments, given a declaration in class k as t_{ret} `m`$(this : k, p_1 : t_1 \ldots p_n : t_n)$ `com`, is $input(k.m) = \{this \mapsto k, p_1 \mapsto t_1, \ldots, p_n \mapsto t_n\}$ and $output(k.m) = input(k.m)[out \mapsto t_{ret}]$.

Example 1. Assume that, in Figure 2, after entering in the method `add` of the class `Vector` we have an initial state $(\phi_0 \star \mu_0)$ s.t. $loc_1 = \phi_0(el) \neq null$. After executing `Vector v = new Vector()` the state is $(\phi_1 \star \mu_1)$, with $\phi_1(v) = loc_2$, and $\mu_1(loc_2).\phi(first) = null$. The field assignment `el.next = null` results in $(\phi_2 \star \mu_2)$, verifying $\mu_2(loc_1).\phi(next) = null$. In the third line, `v.first = el` links loc_1 and loc_2 since now $\mu_3(loc_2).\phi(first) = loc_1$. Now v and el share, since their reachability sets intersect *at least* in $\{loc_1\}$. Finally, assume that `append` attaches v to the end of the current instance $this$ resulting in a memory layout $(\phi_4 \star \mu_4)$. Given $loc_3 = obj((\phi_4 \star \mu_4)(this)).\phi(first)$, it should hold that $\mu_4(\ldots \mu_4(loc_3).\phi(next) \ldots).\phi(next) = loc_2$. Now $this$ shares with v and therefore with el, because loc_1 is reachable from loc_2.

3 Abstract Semantics

An abstract state $\sigma \in D_\pi$ in an environment π approximates the sharing, nullity, and run-time type characteristics (as described in Sect. 2.2) of set of concrete states in Σ_π. Every abstract state combines three abstractions: a sharing set $sh \in \mathcal{DS}_\pi$, a nullity set $nl \in \mathcal{DN}_\pi$, and a type member $\tau \in \mathcal{DT}_\pi$, i.e., $D_\pi = \mathcal{DS}_\pi \times \mathcal{DN}_\pi \times \mathcal{DT}_\pi$.

The sharing abstract domain $\mathcal{DS}_\pi = \{\{v_1, \ldots, v_n\} \mid \{v_1, \ldots, v_n\} \in \mathcal{P}(dom(\pi)), \cap_{i=1}^n C_\pi(v_i) \neq \emptyset\}$ is constrained by a class reachability function which retrieves those classes that are reachable from a particular variable: $C_\pi(v) = \cup\{C^i_\pi(v) \mid i \geq 0\}$, given $C^0_\pi(v) = \downarrow\pi(v)$ and $C^{i+1}_\pi(v) = \cup\{rng(F(k)) \mid k \in C^i_\pi(v)\}$. By using class reachability, we avoid including in the sharing domain sets of variables which cannot share in practice because of the language semantics. The partial order $\leq_{\mathcal{DS}_\pi}$ is set inclusion.

We define several operators over sharing sets, standard in the sharing literature [14,19]. The binary union $\uplus : \mathcal{DS}_\pi \times \mathcal{DS}_\pi \mapsto \mathcal{DS}_\pi$, calculated as $S_1 \uplus S_2 = \{Sh_1 \cup Sh_2 \mid Sh_1 \in S_1, Sh_2 \in S_2\}$ and the closure under union $* : \mathcal{DS}_\pi \mapsto \mathcal{DS}_\pi$ operators, defined as $S^* = \{\cup SSh \mid SSh \in \mathcal{P}^+(S)\}$; we later filter their results using class reachability. The relevant sharing with respect to v is $sh_v = \{s \in sh \mid v \in s\}$, which we overloaded for sets. Similarly, $sh_{-v} = \{s \in sh \mid v \notin s\}$. The projection $sh|_V$ is equivalent to $\{S \mid S = S' \cap V, S' \in sh\}$.

$$\mathcal{SE}^I_\pi[\![\texttt{null}]\!](sh, nl, \tau) = (sh, nl', \tau')$$
$$nl' = nl[res \mapsto null]$$
$$\tau' = \tau[res \mapsto \downarrow object]$$

$$\mathcal{SE}^I_\pi[\![\texttt{new } k]\!](sh, nl, \tau) = (sh', nl', \tau')$$
$$sh' = sh \cup \{\{res\}\}$$
$$nl' = nl[res \mapsto nnull]$$
$$\tau' = \tau[res \mapsto \{\kappa\}]$$

$$\mathcal{SE}^I_\pi[\![v]\!](sh, nl, \tau) = (sh', nl', \tau')$$
$$sh' = (\{\{res\}\} \uplus sh_v) \cup sh_{-v}$$
$$nl' = nl[res \mapsto nl(v)]$$
$$\tau' = \tau[res \mapsto \tau(v)]$$

$$\mathcal{SE}^I_\pi[\![v.\texttt{f}]\!](sh, nl, \tau) = \begin{cases} \bot \text{ if } nl(v) = null \\ (sh', nl', \tau') \text{ otherwise} \end{cases}$$
$$sh' = sh_{-v} \cup \bigcup\{\mathcal{P}^+(s|_{-v} \cup \{res\}) \uplus \{\{v\}\} \mid s \in sh_v\}$$
$$nl' = nl[res \mapsto unk, v \mapsto nnull]$$
$$\tau' = \tau[res \mapsto\downarrow F(\pi(v)(\texttt{f}))]$$

$$\mathcal{SE}^I_\pi[\![v.\texttt{m}(v_1, \ldots, v_n)]\!](sh, nl, \tau) = \begin{cases} \bot \text{ if } nl(v) = null \\ \sigma' \text{ otherwise} \end{cases}$$
$$\sigma' = \mathcal{SE}^I_\pi[\![\texttt{call}(v, m(v_1, \ldots, v_n))]\!](sh, nl', \tau)$$
$$nl' = nl[v \mapsto nnull]$$

Fig. 3. Abstract semantics for the expressions

The nullity domain is $\mathcal{DN}_\pi = \mathcal{P}(dom(\pi) \mapsto \mathcal{NV})$, where $\mathcal{NV} = \{null, nnull, unk\}$. The order $\leq_{\mathcal{NV}}$ of the nullity values ($null \leq_{\mathcal{NV}} unk$, $nnull \leq_{\mathcal{NV}} unk$) induces a partial order in $\mathcal{DN}_\pi$ s.t. $nl_1 \leq_{\mathcal{DN}_\pi} nl_2$ if $nl_1(v) \leq_{\mathcal{NV}} nl_2(v)$ $\forall v \in dom(\pi)$. Finally, the domain of types maps variables to sets of types congruent with π: $\mathcal{DT}_\pi = \{(v, \{t_1, \ldots, t_n\}) \in dom(\pi) \mapsto \mathcal{P}(\mathcal{K}) \mid \{t_1, \ldots, t_n\} \subseteq \downarrow\pi(v)\}$.

We assume the standard framework of abstract interpretation as defined in [8] in terms of Galois insertions. The concretization function $\gamma_\pi : D_\pi \mapsto \mathcal{P}(\Sigma_\pi)$ is $\gamma_\pi(sh, nl, \tau) = \{\delta \in \Sigma_\pi | \ \forall V \subseteq dom(\pi), share_\pi(\delta, V)$ and $\nexists W, V \subset W \subseteq dom(\pi)$ s.t. $share_\pi(\delta, W) \Rightarrow V \in sh$, and $R_\pi(\delta, v) = \emptyset$ if $nl(v) = null$, and $R_\pi(\delta, v) \neq \emptyset$ if $nl(v) = nnull$, and $\psi_\pi(\delta, v) \in \tau(v)$, $\forall v \in dom(\pi)\}$.

The abstract semantics of expressions and commands is listed in Figs. 3 and 4. They correctly approximate the standard semantics, as proved in [16]. As their concrete counterparts, they take an expression or command and map an input state $\sigma \in D_\pi$ to an output state $\sigma' \in D^{\sigma}_{\pi'}$ where $\pi = \pi'$ in commands and $\pi' = \pi[res \mapsto type_\pi(expr)]$ in expression $expr$. The semantics of a method call is explained in Sect. 3.1. The use of set sharing (rather than pair sharing) in the semantics prevents possible losses of precision, as shown in Example 2.

Example 2. In the `add` method (Fig. 2), assume that $\sigma = (\{\{this, el\}, \{v\}\}, \{this/nnull, el/nnull, v/nnull\})$ right before evaluating `el` in the third line (we skip type information for simplicity). The expression `el` binds to res the location of el, i.e., forces el and res to share. Since $nl(el) \neq null$ the new sharing is $sh' = (\{\{res\}\} \uplus sh_{el}) \cup sh_{-el} = (\{\{res\}\} \uplus \{\{this, el\}\}) \cup \{\{v\}\} = \{\{res, this, el\}, \{v\}\}$.

$$\mathcal{SC}^I_\pi[\![v\texttt{=}expr]\!]\sigma = ((sh'|_{-v})|^v_{res}, nl'|^v_{res}, \tau^{''}|_{-res})$$
$$\tau^{''} = \tau'[v \mapsto (\tau'(v) \cap \tau'(res))]$$
$$(sh', nl', \tau') = \mathcal{SE}^I_\pi[\![expr]\!]\sigma$$

$$\mathcal{SC}^I_\pi[\![v\texttt{.f=}expr]\!]\sigma = (sh^{''}, nl^{''}, \tau')|_{-res}$$
$$sh^{''} = \begin{cases} \bot & \text{if } nl'(v) = null \\ sh' & \text{if } nl'(res) = null \\ sh^y \cup sh'_{-\{v,res\}} & \text{otherwise} \end{cases}$$
$$nl^{''} = nl'[v \mapsto nnull]$$
$$sh^y = (\bigcup\{\mathcal{P}(s|_{-v} \cup \{res\}) \uplus \{\{v\}\} \mid s \in sh'_v\} \cup \bigcup\{\mathcal{P}(s|_{-res} \cup \{v\}) \uplus \{\{res\}\} \mid s \in sh'_{res}\})^*$$
$$(sh', nl', \tau') = \mathcal{SE}^I_\pi[\![expr]\!]\sigma$$

$$\mathcal{SC}^I_\pi[\![\texttt{if } v\texttt{==null } com_1 \texttt{ else } com_2]\!]\sigma = \begin{cases} \sigma'_1 & \text{if } nl(v) = null \\ \sigma'_2 & \text{if } nl(v) = nnull \\ \sigma_1 \sqcup \sigma_2 & \text{if } nl(v) = unk \end{cases}$$
$$\sigma'_i = \mathcal{SC}^I_\pi[\![com_i]\!]\sigma$$
$$\sigma_1 = \mathcal{SC}^I_\pi[\![com_1]\!](sh|_{-v}, nl[v \mapsto null], \tau[v \mapsto \downarrow\pi(v)])$$
$$\sigma_2 = \mathcal{SC}^I_\pi[\![com_2]\!](sh, nl[v \mapsto nnull], \tau)$$

$$\mathcal{SC}^I_\pi[\![\texttt{if } v\texttt{==}w\ com_1 \texttt{ else } com_2]\!](sh, nl, \tau) = \begin{cases} \sigma'_1 & \text{if } nl(v) = nl(w) = null \\ \sigma'_2 & \text{if } sh|_{\{v,w\}} = \emptyset \\ \sigma'_1 \sqcup \sigma'_2 & \text{otherwise} \end{cases}$$
$$\sigma'_i = \mathcal{SC}^I_\pi[\![com_i]\!](sh, nl, \tau)$$

$$\mathcal{SC}^I_\pi[\![com_1\texttt{;}com_2]\!]\sigma = \mathcal{SC}^I_\pi[\![com_2]\!](\mathcal{SC}^I_\pi[\![com_1]\!]\sigma)$$

Fig. 4. Abstract semantics for the commands

In the case of pair-sharing, the transfer function [26] for the same initial state $sh = \{\{this, el\}, \{v, v\}\}$ returns $sh'_p = \{\{res, el\}, \{res, this\}, \{this, el\}, \{v, v\}\}$, which translated to set sharing results in $sh'' = \{\{res, el\}, \{res, this\}, \{res, this, el\}, \{this, el\}, \{v\}\}$, a less precise representation (in terms of $\leq_{\mathcal{DS}_\pi}$) than sh'.

Example 3. Our multivariant analysis keeps two different call contexts for the `append` method in the `Vector` class (Fig. 2). Their different sharing information shows how sharing can improve nullity results. The first context corresponds to external calls (invocation from other classes), because of the `public` visibility of the method: $\sigma_1 = (\{\{this\}, \{this, v\}, \{v\}\}, \{this/nnull, v/unk\}, \{this/\{vector\}, v/\{vector\}\})$. The second corresponds to an internal (within the class) call, for which the analysis infers that *this* and v do not share: $\sigma_2 = (\{\{this\}, \{v\}\}, \{this/nnull, v/unk\}, \{this/\{vector\}, v/\{vector\}\})$. Inside `append`, we avoid creating a circular list by checking that $this \neq v$. Only then is the last element of *this* linked to the first one of v. We use `com` to represent the series of commands `Element e = first; if (e==null)...else..` and `bdy` for the whole body of the method. Independently of whether the input state is σ_1 or σ_2 our analysis infers that $\mathcal{SC}^I_\pi[\![\texttt{com}]\!]\sigma_1 = \mathcal{SC}^I_\pi[\![\texttt{com}]\!]\sigma_2 = (\{\{this, v\}\}, \{this/nnull, v/nnull\}, \{this/\{vector\}, v/\{vector\}\}) = \sigma_3$. However, the more precise sharing information in σ_2 results in a more precise analysis

Algorithm 1. Extend operation

input : state before the call σ, result of analyzing the call σ_λ and actual parameters A
output: resulting state σ_f

if $\sigma_\lambda = \bot$ **then**
  $\sigma_f = \bot$
else
  let $\sigma = (sh, nl, \tau)$, and $\sigma_\lambda = (sh_\lambda, nl_\lambda, \tau_\lambda)$, and $AR = A \cup \{res\}$

  $star = (sh_A \cup \{\{res\}\})^*$
  $sh_{ext} = \{s \mid s \in star, s|_{AR} \in sh_\lambda\}$
  $sh_f = sh_{ext} \cup sh_{-A}$

  $nl_f = nl[res \mapsto nl_\lambda(res)]$
  $\tau_f = \tau[res \mapsto \tau_\lambda(res)]$
  $\sigma_f = (sh_f, nl_f, \tau_f)$
end

of `bdy`, because of the guard `(this!=v)`. In the case of the external calls, $\mathcal{SC}^I_\pi[\![\texttt{bdy}]\!]\sigma_1 = \mathcal{SC}^I_\pi[\![\texttt{com}]\!]\sigma_1 \sqcup \mathcal{SC}^I_\pi[\![\texttt{skip}]\!]\sigma_1 = \sigma_1 \sqcup \sigma_3 = \sigma_1$. When the entry state is σ_2, the semantics at the same program point is $\mathcal{SC}^I_\pi[\![\texttt{bdy}]\!]\sigma_2 = \mathcal{SC}^I_\pi[\![\texttt{com}]\!]\sigma_2 = \sigma_3 < \sigma_1$. So while the internal call requires $v \neq null$ to terminate, we cannot infer the final nullity of that parameter in a public invocation, which might finish even if v is null.

3.1 Method Calls

The semantics of the expression $\texttt{call}(v, m(v_1, \ldots, v_n))$ in state $\sigma = (sh, nl, \tau)$ is calculated by implementing the top-down methodology described in [21]. We will assume that the formal parameters follow the naming convention F in all the implementations of the method; let $A = \{v, v_1, \ldots, v_n\}$ and $F = dom(input(k.m))$ be ordered lists. We first calculate the projection $\sigma_p = \sigma|_A$ and an entry state $\sigma_y = \sigma_p|^F_A$. The abstract execution of the call takes place only in the set of classes $K = \tau(v)$, resulting in an exit state $\sigma_x = \bigsqcup\{\mathcal{SC}^I_\pi[\![k'.m]\!]\sigma_y | k' = lookup(k, m), k \in K\}$, where *lookup* returns the body of k's implementation of m, which can be defined in k or inherited from one of its ancestors. The abstract execution of the method in a subset $K \subseteq \downarrow\pi(v)$ increases analysis precision and is the ultimate purpose of tracking run-time types in our abstraction. We now remove the local variables $\sigma_b = \sigma_x|_{F \cup \{out\}}$ and rename back to the scope of the caller: $\sigma_\lambda = \sigma_b|^{A \cup \{res\}}_{F \cup \{out\}}$; the final state σ_f is calculated as $\sigma_f = extend(\sigma, \sigma_\lambda, A)$. The $extend : D_\pi \times D_\pi \times \mathcal{P}(dom(\pi)) \mapsto D_\pi$ function is described in Algorithm 1.

In Java references to objects are passed by value in a method call. Therefore, they cannot be modified. However, the call might introduce new sharing between actual parameters through assignments to their fields, given that the formal parameters they correspond to have not been reassigned. We keep the original information by copying all the formal parameters at the beginning of each call,

as suggested in [23]. Those copies cannot be modified during the execution of the call, so a meaningful correspondence can be established between A and F.

We can do better by realizing that analysis might refine the information about the actual parameters within a method and propagating the new values discovered back to σ_f. For example, in a method `foo(Vector v){if v!=null skip else throw_null}`, it is clear that we can only finish normally if $nl_x(v) = nnull$, but in the actual semantics we do not change the nullity value for the corresponding argument in the call, which can only be more imprecise. Note that the example is different from `foo(Vector v){v = new Vector}`, which also finishes with $nl_x(v) = nnull$. The distinction over whether new attributes are preserved or not relies on keeping track of those variables which have been assigned inside the method, and then applying the propagation only for the unset variables.

Example 4. Assume an extra snippet of code in the `Vector` class of the form `if (v2!=null) v1.append(v2) else com`, which is analyzed in state $\sigma = (\{\{v_1\}, \{v_2\}\}, \{v_1/nnull,\ v_2/nnull\}, \{v_1/\{vector\}, v_2/\{vector\}\})$. Since we have nullity information, it is possible to identify the block `com` as dead code. In contrast, sharing-only analyses can only tell if a variable is definitely null, but never if it is definitely non-null. The call is analyzed as follows. Let $A = \{v_1, v_2\}$ and $F = \{this, v\}$, then $\sigma_p = \sigma|_A = \sigma$ and the entry state σ_y is $\sigma|_A^F = (\{\{this\}, \{v\}\}, \{this/nnull, v/nnull\}, \{this/\{vector\}, v/\{vector\}\})$. The only class where `append` can be executed is `Vector` and results (see Example 3) in an exit state for the formal parameters and the return variable $\sigma_b = (\{\{this, v\}\}, \{this/nnull, v/nnull, out/null\}, \{this/\{vector\}, v/\{vector\}, out/\{void\}\})$, which is further renamed to the scope of the caller obtaining $\sigma_\lambda = (\{\{v_1, v_2\}\}, \{v_1/nnull, v_2/nnull, res/null\}, \{v_1/\{vector\}, v_2/\{vector\}, res/\{void\}\})$. Since the method returns a `void` type we can treat res as a primitive (null) variable so $\sigma_f = extend(\sigma, \sigma_\lambda, \{v_1, v_2\}) = (\{\{v_1, v_2\}\}, \{v_1/nnull, v_2/nnull, res/null\}, \{v_1/\{vector\}, v_2/\{vector\}, res/\{void\}\})$.

Example 5. The *extend* operation used during interprocedural analysis is a point where there can be significant loss of precision and where set sharing shows its strengths. For simplicity, we will describe the example only for the sharing component; nullity and type information updates are trivial. Assume a scenario where a call to `append(v1,v2)` in sharing state $sh = \{\{v_0, v_1\}, \{v_1\}, \{v_2\}\}$ results in $sh_\lambda = \{\{v_1, v_2\}\}$. Let A and AR be the sets $\{v_1, v_2\}$ and $\{v_1, v_2, res\}$ respectively. The *extend* operation proceeds as follows: first we calculate *star* as $(sh_A \cup \{\{res\}\})^* = (sh \cup \{\{res\}\})^* = (\{\{v_0, v_1\}, \{v_1\}, \{v_2\}, \{res\}\})^* = \{\{v_0, v_1\}, \{v_0, v_1, v_2\}, \{v_0, v_1, v_2, res\}, \{v_0, v_1, res\}, \{v_1\}, \{v_1, v_2\}, \{v_1, v_2, res\}, \{v_1, res\}, \{v_2\}, \{v_2, res\}, \{res\}\}$, from which we delete those elements whose projection over AR is not included in sh_λ, obtaining $sh_{ext} = \{\{v_0, v_1, v_2\}, \{v_1, v_2\}\}$. The resulting sharing component is the union of that sh_{ext} with $sh_{-A} = \emptyset$, so $sh_{f1} = sh_{ext} = \{\{v_0, v_1, v_2\}, \{v_1, v_2\}\}$.

When the same sh and sh_λ are represented in their pair sharing versions $sh^p = \{\{v_0, v_1\}, \{v_o, v_0\}, \{v_1, v_1\}, \{v_2, v_2\}\}$ and $sh^p_\lambda = \{\{v_1, v_2\}, \{v_1, v_1\}, \{v_2, v_2\}\}$, the *extend* operation in [26] introduces spurious sharings in sh_f because of the lower precision of the pair-sharing representation. In this case, $sh^p_{f2} = (sh \cup$

$sh^p_\lambda)^*_A = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_1, v_2\}, \{v_0, v_0\}, \{v_1, v_1\}, \{v_2, v_1\}\}$. This information, expressed in terms of set sharing, results in $sh_{f2} = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_0, v_1, v_2\}, \{v_1, v_2\}, \{v_0\}, \{v_1\}, \{v_2\}\}$, which is much less precise that sh_{f1}.

4 Experimental Results

In our analyzer the abstract semantics presented in the previous section is evaluated by a highly optimized fixpoint algorithm, based on that of [21]. The algorithm traverses the program dependency graph, dynamically computing the strongly-connected components and keeping detailed dependencies on which parts of the graph need to be recomputed when some abstract value changes during the analysis of iterative code (loops and recursions). This reduces the number of steps and iterations required to reach the fixpoint, which is specially important since the algorithm implements *multivariance*, i.e., it keeps different abstract values at each program point for every calling context, and it computes (a superset of) all the calling contexts that occur in the program. The dependencies kept also allow relating these values along execution paths (this is particularly useful for example during error diagnosis or for program specialization).

We now provide some precision and cost results obtained from the implementation in the framework described in [17] of our set-sharing, nullity, and class ($SSNlTau$) analysis. In order to be able to provide a comparison with the closest previous work, we also implemented the pair sharing (PS) analysis proposed in [26]. We have extended the operations described in [26], enabling them to handle some additional cases required by our benchmark programs such as primitive variables, visibility of methods, etc. Also, to allow direct comparison, we implemented a version of our $SSNlTau$ analysis, which is referred to simply as SS, that tracks set sharing using only declared type information and does not utilize the (non-)nullity component. In order to study the influence of tracking run-time types we have implemented a version of our analysis with set sharing and (non-)nullity, but again using only the static types, which we will refer to as $SSNl$. In these versions without dynamic type inference only declared types can affect τ and thus the dynamic typing information that can be propagated from initializations, assignments, or correspondence between arguments and formal parameters on method calls is not used. Note however that the version that includes tracking of dynamic typing can of course only improve analysis results in the presence of polymorphism in the program: the results should be identical (except perhaps for the analysis time) in the rest of the cases. The polymorphic programs are marked with an asterisk in the tables.

The benchmarks used have been adapted from previous literature on either abstract interpretation for Java or points-to analysis [26,24,23,29]. We added two different versions of the `Vector` example of Fig. 2. Our experimental results are summarized in Tables 5, 6, and 7.

The first column ($\#tp$) in Tables 5 and 6 shows the total number of program points (commands or expressions) for each program. Column $\#rp$ then provides, for each analysis, the total number of *reachable* program points, i.e., the

		PS				SS				
	$\#tp$	$\#rp$	$\#up$	$\#\sigma$	t	$\#rp$	$\#up$	$\#\sigma$	t	$\%\Delta t$
dyndisp (*)	71	68	3	114	30	68	3	114	29	-2
clone	41	38	3	42	52	38	3	50	81	55
dfs	102	98	4	103	68	98	4	108	68	0
passau (*)	167	164	3	296	97	164	3	304	120	23
qsort	185	142	43	182	125	142	43	204	165	32
integerqsort	191	148	43	159	110	148	43	197	122	10
pollet01 (*)	154	126	28	276	196	126	28	423	256	30
zipvector (*)	272	269	3	513	388	269	3	712	1029	164
cleanness (*)	314	277	37	360	233	277	37	385	504	116
overall	1497	1330	167	2045	1299	1330	167	2497	2374	82.75

Fig. 5. Analysis times, number of program points, and number of abstract states

		SSNl					SSNlTau				
	$\#tp$	$\#rp$	$\#up$	$\#\sigma$	t	$\%\Delta t$	$\#rp$	$\#up$	$\#\sigma$	t	$\%\Delta t$
dyndisp (*)	71	61	10	103	53	77	61	10	77	20	-33
clone	41	31	10	34	100	92	31	10	34	90	74
dfs	102	91	11	91	129	89	91	11	91	181	166
passau (*)	167	157	10	288	117	18	157	10	270	114	17
qsort	185	142	43	196	283	125	142	43	196	275	119
integerqsort	191	148	43	202	228	107	148	43	202	356	224
pollet01 (*)	154	119	35	364	388	98	98	56	296	264	35
zipvector (*)	272	269	3	791	530	36	245	27	676	921	136
cleanness (*)	314	276	38	383	276	38	266	48	385	413	77
overall	1497	1294	203	2452	2104	61.97	1239	258	2227	2634	102.77

Fig. 6. Analysis times, number of program points, and number of abstract states

number of program points that the analysis explores, while $\#up$ represents the $(\#tp - \#rp)$ points that are not analyzed because the analysis determines that they are unreachable. It can be observed that tracking (non-)nullity (Nl) reduces the number of reachable program points (and increases conversely the number of unreachable points) because certain parts of the code can be discarded as dead code (and not analyzed) when variables are known to be non-null. Tracking dynamic types (Tau) also reduces the number of reachable points, but, as expected, only for (some of) the programs that are polymorphic. This is due to the fact that the class analysis allows considering fewer implementations of methods, but obviously only in the presence of polymorphism.

Since our framework is multivariant and thus tracks many different *contexts* at each program point, at the end of analysis there may be more than one abstract state associated with each program point. Thus, the number of abstract states inferred is typically larger than the number of reachable program points. Column $\#\sigma$ provides the total number of these abstract states inferred by the analysis. The level of multivariance is the ratio $\#\sigma/\#rp$. It can be observed that the simple

	PS		SS	
	$\#sh$	$\%sh$	$\#sh$	$\%sh$
dyndisp (*)	640	60.37	435	73.07
clone	174	53.10	151	60.16
dfs	1573	96.46	1109	97.51
passau (*)	5828	94.56	3492	96.74
qsort	1481	67.41	1082	76.34
integerqsort	2413	66.47	1874	75.65
pollet01 (*)	793	89.81	1043	91.81
zipvector (*)	6161	68.71	5064	80.28
cleanness (*)	1300	63.63	1189	70.61
overall	20363	73.39	15439	80.24

Fig. 7. Sharing precision results

set sharing analysis (SS) creates more abstract states for the same number of reachable points. In general, such a larger number for $\#\sigma$ tends to indicate more precise results (as we will see later). On the other hand, the fact that addition of Nl and Tau reduces the number of reachable program points interacts with precision to obtain the final $\#\sigma$ value, so that while there may be an increase in the number of abstract states because of increased precision, on the other hand there may be a decrease because more program points are detected as dead code by the analysis. Thus, the $\#\sigma$ values for $SSNl$ and $SSNlTau$ in some cases actually decrease with respect to those of PS and SS.

The t column in Tables 5 and 6 provides the running times for the different analyses, in milliseconds, on a Pentium M 1.73Ghz, 1Gb of RAM, running Fedora Core 4.0, and averaging several runs after eliminating the best and worst values. The $\%\Delta t$ columns show the percentage variation in the analysis time with respect to the reference pair-sharing (PS) analysis, calculated as $\Delta_{dom}\%t = 100*(t_{dom} - t_{PS})/t_{PS}$. The more complex analyses tend to take longer times, while in any case remaining reasonable. However, sometimes more complex analyses actually take less time, again because the increased precision and the ensuing dead code detection reduces the amount of program that must be analyzed.

Table 7 shows precision results in terms of sharing, concentrating on the SP and SS domains, which allow direct comparison. A more usage-oriented way of measuring precision would be to study the effect of the increased precision in an application that is known to be sensitive to sharing information, such as, for example, program parallelization [4]. On the other hand this also complicates matters in the sense that then many other factors come into play (such as, for example, the level of intrinsic parallelism in the benchmarks and the parallelization algorithms) so that it is then also harder to observe the precision of the analysis itself. Such a client-level comparison is beyond the scope of this paper, and we concentrate here instead on measuring sharing precision directly.

Following [6], and in order to be able to compare precision directly in terms of sharing, column $\#sh$ provides the sum over all abstract states in all reachable program points of the cardinality of the sharing *sets* calculated by the analysis.

For the case of pair sharing, we converted the pairs into their equivalent set representation (as in [6]) for comparison. Since the results are always correct, a smaller number of sharing sets indicates more precision (recall that $\top$ is the power set). This is of course assuming σ is constant, which as we have seen is not the case for all of our analyses. On the other hand, if we compare PS and SS, we see that SS has consistently more abstract states than PS and consistently lower numbers of sharing sets, and the trend is thus clear that it indeed brings in more precision. The only apparent exception is *pollet01* but we can see that the number of sharing sets is similar for a significantly larger number of abstract states.

An arguably better metric for measuring the relative precision of sharing is the ratio $\%_{Max} = 100 * (1 - \#sh/(2^{\#vo} - 1))$ which gives $\#sh$ as a percentage of its maximum possible value, where $\#vo$ is the total number of object variables in all the states. The results are given in column $\%sh$. In this metric 0% means all abstract states are $\top$ (i.e., contain no useful information) and 100% means all variables in all abstract states are detected not to share. Thus, larger values in this column indicate more precision, since analysis has been able to infer smaller sharing sets. This relative measure shows an average improvement of 7% for SS over PS.

5 Conclusions

We have proposed an analysis based on abstract interpretation for deriving precise sharing information for a Java-like language. Our analysis is multivariant, which allows separating different contexts, and combines Set Sharing, Nullity, and Classes: the domain captures which instances definitely do not share or are definitively null, and uses the classes to refine the static information when inheritance is present. We have implemented the analysis, as well as previously proposed analyses based on Pair Sharing, and obtained encouraging results: for all the examples the set sharing domains (even without combining with Nullity or Classes) offer more precision than the pair sharing counterparts while the increase in analysis times appears reasonable. In fact the additional precision (also when combined with nullity and classes) brings in some cases analysis time reductions. This seems to support that our contributions bring more precision at reasonable cost.

Acknowledgments

The authors would like to thank Samir Genaim for many useful comments to previous drafts of this document. Manuel Hermenegildo and Mario Méndez-Lojo are supported in part by the Prince of Asturias Chair in Information Science and Technology at UNM. This work was also funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of

Education under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the *PROMESAS* project.

References

1. Agesen, O.: The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 2–26. Springer, Heidelberg (1995)
2. Alves-Foss, J. (ed.): Formal Syntax and Semantics of Java. LNCS, vol. 1523. Springer, Heidelberg (1999)
3. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: Proc. of OOPSLA 1996, SIGPLAN Notices, October 1996, vol. 31(10), pp. 324–341 (1996)
4. Bueno, F., García de la Banda, M., Hermenegildo, M.: Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. ACM Transactions on Programming Languages and Systems 21(2), 189–238 (1999)
5. Burke, M.G., et al.: Carini, Jong-Deok Choi, and Michael Hind. In: Pingali, K.K., et al. (eds.) LCPC 1994. LNCS, vol. 892, pp. 234–250. Springer, Heidelberg (1995)
6. Codish, M., et al.: Improving Abstract Interpretations by Combining Domains. ACM Transactions on Programming Languages and Systems 17(1), 28–44 (1995)
7. Cortesi, A., et al.: Complementation in abstract interpretation. ACM Trans. Program. Lang. Syst. 19(1), 7–47 (1997)
8. Cousot, P., Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of POPL 1977, pp. 238–252 (1977)
9. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Sixth ACM Symposium on Principles of Programming Languages, San Antonio, Texas, pp. 269–282 (1979)
10. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995)
11. Gosling, J., et al.: Java(TM) Language Specification, 3rd edn. Addison-Wesley Professional, Reading (2005)
12. Hill, P.M., Payet, E., Spoto, F.: Path-length analysis of object-oriented programs. In: Proc. EAAI 2006 (2006)
13. Hind, M., et al.: Interprocedural pointer alias analysis. ACM Trans. Program. Lang. Syst. 21(4), 848–894 (1999)
14. Jacobs, D., Langen, A.: Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In: 1989 North American Conference on Logic Programming, MIT Press, Cambridge (1989)
15. Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural pointer aliasing (with retrospective). In: McKinley, K.S. (ed.) Best of PLDI, pp. 473–489. ACM Press, New York (1992)
16. Méndez-Lojo, M., Hermenegildo, M.: Precise Set Sharing for Java-style Programs (and proofs). Technical Report CLIP2/2007.1, Technical University of Madrid (UPM), School of Computer Science, UPM (November 2007)
17. Méndez-Lojo, M., Navas, J., Hermenegildo, M.: A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In: 17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007) (August 2007)

18. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In: ISSTA, pp. 1–11 (2002)
19. Muthukumar, K., Hermenegildo, M.: Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In: 1989 North American Conference on Logic Programming, October 1989, pp. 166–189. MIT Press, Cambridge (1989)
20. Muthukumar, K., Hermenegildo, M.: Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In: 1991 International Conference on Logic Programming, June 1991, pp. 49–63. MIT Press, Cambridge (1991)
21. Navas, J., Méndez-Lojo, M., Hermenegildo, M.: An Efficient, Context and Path Sensitive Analysis Framework for Java Programs. In: 9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007 (July 2007)
22. Palsberg, J., Schwartzbach, M.I.: Object-oriented type inference. In: OOPSLA, pp. 146–161 (1991)
23. Pollet, I.: Towards a generic framework for the abstract interpretation of Java. PhD thesis, Catholic University of Louvain, Dept. of Computer Science (2004)
24. Pollet, I., Le Charlier, B., Cortesi, A.: Distinctness and sharing domains for static analysis of java programs. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, Springer, Heidelberg (2001)
25. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL 1999 (1999)
26. Secci, S., Spoto, F.: Pair-sharing analysis of object-oriented programs. In: SAS, pp. 320–335 (2005)
27. Søndergaard, H.: An application of abstract interpretation of logic programs: occur check reduction. In: Duijvestijn, A.J.W., Lockemann, P.C. (eds.) Trends in Information Processing Systems. LNCS, vol. 123, pp. 327–338. Springer, Heidelberg (1981)
28. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: PLDI, pp. 387–400 (2006)
29. Streckenbach, M., Snelting, G.: Points-to for java: A general framework and an empirical comparison. Technical report, University Passau (November 2000)
30. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI, pp. 131–144. ACM Press, New York (2004)
31. Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for C programs. In: PLDI, pp. 1–12 (1995)