Precise Set Sharing Analysis
for Java-style Programs (and proofs)

Mario Méndez-Lojo\textsuperscript{1} and Manuel V. Hermenegildo\textsuperscript{1,2}

\textsuperscript{1} University of New Mexico (USA)
\textsuperscript{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 \cite{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 \cite{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., \cite{32,5,13,15}). Obtaining a safe (over-) approximation of which instances might share allows parallelizing segments of code, im-
proving 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 [31].

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 [28] 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 par-
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 [29, 18, 31] 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 {
    public void append(Vector v) {
        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);
    }
}

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, 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 → we refer to total functions; for partial ones we use →. The powerset of a set s is \( P(s) \); \( P^+(s) \) is an abbreviation for \( P(s) \setminus \{\emptyset\} \). The \textit{dom} function returns all the elements for which a function is defined; for the codomain we will use \textit{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 \textit{dom}(f). Conversely, \( f|_S \) restricts \textit{dom}(f) to \( S \). For tuples \( (f_1, \ldots, f_m)|_S \) and \( s = (x_1, \ldots, x_m) \), \( \text{renaming in the set } s \text{ of every variable in } S \text{ by the one in the same position in } T \) is written as \( s|_S^T \). This operator can also be applied for renaming single variables. We denote by \( B \) 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 \( \mathcal{V} \)

\[ \text{http://www.clip.dia.fis.upm.es/~mario} \]
also includes the elements this (instance where the current method is executed), and res (for the return value of the method). In the same way, $K$ represents the program-defined classes. We do not allow import declarations but assume as member of $K$ the predefined class Object.

$K$ forms a lattice implied by a subclass relation $\rightarrow: K \rightarrow \mathcal{P}(K)$ such that if $t_2 \leq_K t_1$ then $t_2 \leq_K t_1$. The semantics of the language implies $\text{Object} = K$.

Given $\text{def}: K \times M \rightarrow \mathcal{B}$, that determines whether a particular class provides its own implementation for a method, the Boolean function $\text{redef}: K \times K \times M. \rightarrow \mathcal{B}$ checks if a class $k_1$ redefines a method existing in the ancestor $k_2$:

$$\text{redef}(k_1, k_2, m) = \text{true} \text{ iff } \exists k \text{ s.t. } \text{def}(k, m), \ k_1 \leq_K k \leq_K k_2.$$ 

Static types are accessed by means of a function $\mu: \mathcal{V} \rightarrow \mathcal{K}$ that maps variables to their declared types. The purpose of an environment $\pi$ 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 $k \in K$ to their declared type), and $\text{type}_\pi(\text{expr})$, which maps expressions to types, according to $\pi$.

The description of the memory state is based on the formalization in [26, 12]. We define a frame as any element of $F_{\pi} = \{ \phi \mid \phi \in \text{dom}(\pi) \rightarrow \text{Loc} \cup \{ \text{null} \} \}$. Locations and objects are linked together through the memory $\text{Mem} = \{ \mu \mid \mu \in \text{Loc} \rightarrow \text{Obj} \}$. A new object of class $k$ is created as $\text{new}(k) = k \times \phi$ where $\phi(f) = \text{null}$ $\forall f \in F(k)$.

The object pointed to by $\phi$ in the frame $\phi$ and memory $\mu$ can be retrieved via the partial function $\theta_\phi(\phi \times \mu, \phi) = \phi(\phi(v))$. A valid heap configuration (concrete state $\phi \times \mu$) is any element of $\mathcal{S}_\phi = \{ (\phi \times \mu) \mid \phi \in F_{\pi}, \mu \in \text{Mem} \}$. We will sometimes refer to a pair $(\phi \times \mu)$ with $\delta$.

The set of locations $R_\delta(\phi \times \mu, v)$ reachable from $v \in \text{dom}(\pi)$ in the particular state $\phi \times \mu \in \Sigma_\pi$ is calculated as $R_\delta(\phi \times \mu, v) = \cup \{ R_\delta^i(\phi \times \mu, v) \mid i \geq 0 \}$, the base case being $R_\delta^0(\phi \times \mu, v) = \{ \text{obj}(\phi(v)) \} \subseteq \text{Loc}$ and the inductive one $R_\delta^{i+1}(\phi \times \mu, v) = \cup \{ \text{range}(\mu(l), \phi) \} \subseteq \text{Loc} \mid l \in R_\delta^i(\phi \times \mu, v) \}$. 

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 $\delta$ if there is at least one common location in their reachability sets, i.e., $\text{share}_\pi(\delta, V)$ is true iff $\bigcap_{i=1}^n R_\pi(\delta, v_i) \neq \emptyset$. A variable $v \in \text{dom}(\pi)$ is null in state $\delta$ if $R_\pi(\delta, v) = \emptyset$. Nullity is checked by means of $\text{null}_\pi: \Sigma_\pi \times \text{dom}(\pi) \rightarrow \mathcal{B}$, defined as $\text{null}_\pi(\phi \times \mu, v) = \text{true} \text{ iff } \phi(v) = \text{null}$.

The run-time type of a variable in scope is returned by $\psi_\pi: \Sigma_\pi \times \text{dom}(\pi) \rightarrow K$, which associates variables with their dynamic type, based on the information contained in the heap state: $\psi_\pi(\delta, v) = \text{obj}(\delta, v, k)$ if $\text{null}_\pi(\delta, v)$ and $\psi_\pi(\delta, v) = \pi(\delta, v)$ otherwise. In a type-safe language like Java runtime types are congruent with declared types, i.e., $\psi_\pi(\delta, v) \leq_K \pi(\delta, v)$ $\forall v \in \text{dom}(\pi), \forall \delta \in \Sigma_\pi$. Therefore, a correct approximation of $\psi_\pi$ can always be derived from $\pi$. Note that at the same program point we might have different run-time type states $\psi_\pi^1$ and $\psi_\pi^2$ 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 \( \text{expr} \rightarrow (\Sigma_{\pi} \rightarrow \Sigma_{\pi}) \) define the meaning of Java expressions, augmenting the actual scope \( \pi' = \pi[\text{res} \mapsto \text{type}(\text{expr})] \) with the temporal variable \( \text{res} \). Command functions \( \text{com} \rightarrow (\Sigma_{\pi} \rightarrow \Sigma_{\pi}) \) do the same for commands; semantics of a method \( m \) defined in class \( k \) is returned by the function \( I(k,m) : \Sigma_{\text{input}(k,m)} \rightarrow \Sigma_{\text{output}(k,m)} \). The definition of the respective environments, given a declaration in class \( k \) as \( \text{tret}(\text{this} : k, p_1 : t_1 ... p_n : t_n) \text{com} \), is \( \text{input}(k,m) = \{ \text{this} \mapsto k, p_1 \mapsto t_1, ..., p_n \mapsto t_n \} \) and \( \text{output}(k,m) = \text{input}(k,m)[\text{out} \mapsto \text{tret}] \).

Example 1. Assume that, in Figure 2, after entering in the method add of the class Vector we have an initial state \( \langle b_0 \mapsto x_0 \rangle \) s.t. \( \text{loc}_1 = \langle b_0(\text{el}) \neq \text{null} \rangle \). After executing \( \text{Vector v = new Vector}() \) the state is \( \langle b_1 \mapsto \text{loc}_2 \rangle \), with \( b_1(v) = \text{loc}_2 \), and \( \mu_1(\text{loc}_2).\phi(\text{first}) = \text{null} \). The field assignment \( \text{el.next = null} \) results in \( \langle b_2 \mapsto \text{loc}_2 \rangle \), verifying \( \phi(\text{first}) = \text{null} \). In the third line, \( \text{v.first = el} \) links \( \text{loc}_1 \) and \( \text{loc}_2 \) since now \( \mu_3(\text{loc}_2).\phi(\text{first}) = \text{loc}_1 \). Now \( v \) and \( e/ \) share, since their reachability sets intersect at least in \( \{ \text{loc}_1 \} \). Finally, assume that \( \text{append} \) attaches \( v \) to the end of the current instance \( \text{this} \) resulting in a memory layout \( \langle b_4 \mapsto \text{loc}_2 \rangle \). Given \( \text{loc}_4 = \text{obj(} \langle b_4 \mapsto \text{null}(\text{this}) \rangle \langle b_4 \mapsto \text{first} \rangle \langle b_4 \mapsto \text{first} \rangle \langle b_4 \mapsto \text{null} \rangle \) \( \phi(\text{next}) = \text{null} \). Now \( \text{this} \) shares with \( v \) and therefore with \( e/ \), because \( \text{loc}_1 \) is reachable from \( \text{loc}_2 \).

3 Abstract Semantics

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

The sharing abstract domain \( D_{\pi} = \{ \{ v_1, \ldots, v_n \} \mid \{ v_1, \ldots, v_n \} \in \mathcal{P}(\text{dom}(\pi)) \} \) is constrained by a class reachability function which retrieves those classes that are reachable from a particular variable: \( C_{\pi}(v) = \cup\{ C_{\pi}(v) \mid i \geq 0 \} \), given \( C_0(v) = \{ v \} \) and \( C_{\pi+1}(v) = \cup\{ \text{rng}(F(k)) \mid k \in C_{\pi}(v) \} \). 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 \( \leq_{D_{\pi}} \) is set inclusion.

We define several operators over sharing sets, standard in the sharing literature [14, 19]. The binary union \( \cup : D_{\pi} \times D_{\pi} \rightarrow D_{\pi} \), calculated as \( S_1 \cup S_2 = \{ Sh_1 \cup Sh_2 \mid Sh_1 \in S_1, Sh_2 \in S_2 \} \) and the closure under union \( \cup : D_{\pi} \rightarrow D_{\pi} \) 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
\( \mathcal{SE}_l[\text{null}] = (sh, nl, \tau) \)
\[ nl' = nl[\text{res} \mapsto \text{null}] \\
\tau' = \tau[\text{res} \mapsto \text{object}] \]
\( \mathcal{SE}_l[k](sh, nl, \tau) = (sh', nl', \tau') \)
\[ sh' = sh \cup \{ \{\text{res}\} \} \\
nl' = nl[\text{res} \mapsto \text{null}] \\
\tau' = \tau[\text{res} \mapsto \{\text{object}\}] \]
\( \mathcal{SE}_l[v](sh, nl, \tau) = (sh', nl', \tau') \)
\[ sh' = (\{\{\text{res}\}\} \cup s_h) \cup s_{-v} \\
nl' = nl[\text{res} \mapsto nl(v)] \\
\tau' = \tau[\text{res} \mapsto \tau(v)] \]
\( \mathcal{SE}_l[v.m(v_1, \ldots, v_n)](sh, nl, \tau) = \{ \bot \text{ if } nl(v) = \text{null} \}
\[ (sh', nl', \tau') \text{ otherwise} \]
\[ sh' = sh_{-v} \cup \bigcup \{ P^+(s_v \cup \{\text{res}\}) \} \text{ for } s \subseteq s_h \]
nl' = nl[\text{res} \mapsto \text{null}] \\
\tau' = \tau[\text{res} \mapsto \tau(v)] \\
\sigma' = \mathcal{SE}_l[c\text{all}(v, m(v_1, \ldots, v_n))](sh, nl', \tau) \\
nl' = nl[v \mapsto \text{null}] \]

Fig. 3. Abstract semantics for the expressions.

\( v \) is \( s_{-v} = \{ s \in s_h | v \notin s \} \), which we overloaded for sets. Similarly, \( s_{-v} = \{ s \in s | v \notin s \} \). The projection \( s_{h \upharpoonright v} \) is equivalent to \( \{ s | s' \subseteq s \cap V, s'' \in s_h \} \).

The nullity domain is \( \mathcal{DN}_\tau = \mathcal{P}(\text{dom}(\pi) \mapsto \mathcal{N}_\tau) \), where \( \mathcal{N}_\tau = \{ \text{null}, \text{null}, \text{unk} \} \). The order \( \leq \mathcal{N}_\tau \) of the nullity values (null \( \leq \mathcal{N}_\tau \) unk, unk \( \leq \mathcal{N}_\tau \) unk) induces a partial order in \( \mathcal{DN}_\tau \) s.t. \( nl_1 \leq \mathcal{DN}_\tau nl_2 \) if \( nl_1(v) \leq \mathcal{N}_\tau nl_2(v) \) \( \forall v \in \text{dom}(\pi) \). Finally, the domain of types maps variables to sets of types congruent with \( \pi \): \( \mathcal{DT}_\pi = \{ (v, \{ t_1, \ldots, t_n \}) \in \text{dom}(\pi) \mapsto \mathcal{P}(\{ t \}) | \{ t_1, \ldots, t_n \} \subseteq \{ \pi(t) \} \} \).

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 \( \gamma_{\pi}(sh, nl, \tau) = \{ \delta \in \Sigma_{\pi} | \forall V \subseteq \text{dom}(\pi), \text{share}_{\pi}(\delta, V) \text{ and } \exists W, V \subseteq W \subseteq \text{dom}(\pi) \} \) s.t. \( \text{share}_{\pi}(\delta, V) \Rightarrow V \subseteq s_h \) and \( \text{share}_{\pi}(\delta, v) = \emptyset \text{ if } nl(v) = \text{null} \), and \( \text{share}_{\pi}(\delta, v) \neq \emptyset \) if \( nl(v) = \text{null} \), and \( \psi_{\pi}(\delta, v) \in \tau(v) \), \( \forall v \in \text{dom}(\pi) \).

The abstract semantics of expressions and commands is listed in Figs. 3 and 4. They correctly approximate the standard semantics, as proved in Sect. C [16] of the appendix. As their concrete counterparts, they take an expression or command and map an input state \( \sigma \in D_\pi \) to an output state \( \sigma' \in D_\pi \) where \( \pi = \pi' \) in commands and \( \pi' = \tau[\text{res} \mapsto \text{type}_\pi(\text{expr})] \) in expression \( \text{expr} \). 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 \texttt{add} method (Fig. 2), assume that \( \sigma = \{ \{\text{this, el}, \{v\}\}, \{\text{this}/\text{null}, \text{el}/\text{null}, v/\text{null}\} \} \) right before evaluating \texttt{el} in the third line (we
skip type information for simplicity). The expression \texttt{el} binds to \texttt{res} the location of \texttt{el}, i.e., forces \texttt{el} and \texttt{res} to share. Since \texttt{nl(el)} \neq \texttt{null} the new sharing is \(sh' = (\{(\texttt{res})\} \cup sh \texttt{el}) \cup sh \texttt{el} = (\{(\texttt{res})\} \cup \{\texttt{this, el}\}) \cup \{\texttt{v}\} = \{(\texttt{res, this, el}) , \{\texttt{v}\}\}.

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

Example 3. Our multivariant analysis keeps two different call contexts for the \texttt{append} method in the \texttt{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: \(\sigma_1 = (\{(\texttt{this})\}, \{(\texttt{this, v})\}, \{\texttt{this/null, v/unk}\}, \{\texttt{this/ vector}, v/\{\texttt{vector}\}\}). \) The second corresponds to an internal (within the class) call, for which the analysis infers that \(\texttt{this} \) and \(\texttt{v} \) do not share: \(\sigma_2 = (\{(\texttt{this})\}, \{\texttt{v}\}, \{\texttt{this/null, v/unk}\}, \{\texttt{this/ vector}, v/\{\texttt{vector}\}\}). \) Inside \texttt{append}, we avoid creating a circular list by checking that \(\texttt{this} \neq \texttt{v}\). Only then is the last element of \texttt{this} linked to the first one of \texttt{v}. We use \texttt{com} to represent the series of commands \texttt{Element e = first; if (e=null)...else...}.
Algorithm 1: Extend operation

input: state before the call \( \sigma \), result of analyzing the call \( \sigma_\lambda \) and actual parameters \( A \)
output: resulting state \( \sigma_f \)

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

\[ \text{star} = (sh_\lambda \cup \{\{res\}\})^* \]
\[ \text{sh}_{\text{ext}} = \{ s \mid s \in \text{star}, s |_{AR} \in sh_\lambda \} \]
\[ sh_f = sh_{\text{ext}} \cup sh_{-A} \]
\[ nl_f = nl[\text{res} \mapsto nl_\lambda(\text{res})] \]
\[ \tau_f = \tau[\text{res} \mapsto \tau_\lambda(\text{res})] \]
\[ \sigma_f = (sh_f, nl_f, \tau_f) \]
end

and bdy for the whole body of the method. Independently of whether the input state is \( \sigma_1 \) or \( \sigma_2 \) our analysis infers that \( \mathcal{SC}^f_\pi[\text{com}] \sigma_1 = \mathcal{SC}^f_\pi[\text{com}] \sigma_2 = (\{(\text{this}, v)\}, \{(\text{this}/null, v/null)\}, \{(\text{this}/\{\text{vector}\}, v/\{\text{vector}\})\}) = \sigma_3 \). However, the more precise sharing information in \( \sigma_2 \) results in a more precise analysis of bdy, because of the guard (\texttt{this!} = v). In the case of the external calls, \( \mathcal{SC}^f_\pi[\text{bdy}] \sigma_1 = \mathcal{SC}^f_\pi[\text{com}] \sigma_1 \cup \mathcal{SC}^f_\pi[\text{skip}] \sigma_1 = \sigma_1 \cup \sigma_3 = \sigma_1 \). When the entry state is \( \sigma_2 \), the semantics at the same program point is \( \mathcal{SC}^f_\pi[\text{bdy}] \sigma_2 = \mathcal{SC}^f_\pi[\text{com}] \sigma_2 = \sigma_3 \). So while the internal call requires \( v \neq \text{null} \) 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 \( A = \{ v_1, \ldots, v_n \} \) and \( F = \text{dom}(\text{input}(k.m)) \) be ordered lists. We first calculate the projection \( \sigma_y = \sigma|_A \) and an entry state \( \sigma_y = \sigma_y|_F \). The abstract execution of the call takes place only in the set of classes \( K = \tau(v) \), resulting in an exit state \( \sigma_x = \bigcup \{ \mathcal{SC}^f_\pi[k'.m]\sigma_y[k'] = \text{lookup}(k, m), k \in K \} \), where \text{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 \( K \subseteq \downarrow \tau(v) \) increases analysis precision and is the ultimate purpose of tracking run-time types in our abstraction. We now remove the local variables \( \sigma_h = \sigma_x|_{\downarrow \tau(v)} \) and rename back to the scope of the caller: \( \sigma_\lambda = \sigma_h|_{\downarrow \tau(v)} \); the final state \( \sigma_f \) is calculated as \( \sigma_f = \text{extend}(\sigma, \sigma_\lambda, A) \). The \text{extend} \( : D_\pi \times D_\pi \times \mathcal{P}(\text{dom}(\pi)) \rightarrow D_\pi \) 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 $\sigma_f$. For example, in a method $\text{foo}(\text{Vector } v)\{\text{if } v! = \text{null skip else throw null}\}$, it is clear that we can only finish normally if $n\ell_\sigma(v) = \text{nnull}$, 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 $\text{foo}(\text{Vector } v)\{v = \text{new Vector}\}$, which also finishes with $n\ell_\sigma(v) = \text{nnull}$. 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 $\text{Vector}$ class of the form $\text{if } (v2! = \text{null}) \text{v1.append(v2) else com}$, which is analyzed in state $\sigma = (\{v_1\}, \{v_1/\text{null}, v_2/\text{nonnull}\}, \{v_1/\{\text{vector}\}, v_2/\{\text{vector}\}\})$. Since we have nullity information, it is possible to identify the block $\text{com}$ 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 $F = \{\text{this, e}\}$, then $\sigma_f = \sigma|_A = \sigma$ and the entry state $\sigma_A = (\{(\text{this}), \{v\}\}, \{\text{this/\text{nonnull}, v/\text{nonnull}\}, \{\text{this/\{vector\}}, v/\{\text{vector}\}\})$. The only class where $\text{append}$ can be executed is $\text{Vector}$ and results (see Example 3) in an exit state for the formal parameters and the return variable $\sigma' = (\{(\text{this, v})\}, \{\text{this/\text{nonnull}, v/\text{nonnull}, out/\text{null}}\}, \{\text{this/\{vector\}}, v/\{\text{vector}\}, \text{res/\{void\}}\})$. Since the method returns a $\text{void}$ type we can treat $\text{res}$ as a primitive (null) variable so $\sigma_f = \text{extend}(\sigma, \sigma_A, \{v_1, v_2\}) = (\{(v_1, v_2)\}, \{v_1/\text{nonnull}, v_2/\text{nonnull, res/\text{null}}\}, \{v_1/\{\text{vector}\}, v_2/\{\text{vector}\}, \text{res/\{void\}}\})$.

**Example 5.** The $\text{extend}$ 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 $\text{append}(\text{v1, v2})$ in sharing state $sh = (\{v_0, v_1\}, \{v_1\}, \{v_2\})$ results in $sh_A = \{\{v_1, v_2\}\}$. Let $A$ and $AR$ be the sets $\{v_1, v_2\}$ and $\{v_1, v_2, res\}$, respectively. The $\text{extend}$ operation proceeds as follows: first we calculate $\star$ as $(sh_A \cup \{\text{res}\})^* = (sh \cup \{\text{res}\})^* = (\{(v_0, v_1), \{v_1\}, \{v_2\}, \{\text{res}\}\})^* = (\{v_0, v_1\}, \{v_0, v_1, v_2\}, \{v_0, v_1, v_2, res\}, \{v_0, v_1, res\}, \{v_1, v_2\}, \{v_1, v_2, res\}, \{v_1, res\}, \{v_2\}, \{v_2, res\}, \{\text{res}\})$, from which we delete those elements whose projection over $AR$ is not included in $sh_A$, obtaining $sh_{ext} = (\{v_0, v_1, v_2\}$.
The resulting sharing component is the union of that \(sh_{\text{ext}}\) with \(\text{sh}_{-A} = \emptyset\), so \(sh_{f1} = sh_{\text{ext}} = \{\{v_0, v_1, v_2\}, \{v_1, v_2\}\}\).

When the same \(sh\) and \(sh_A\) are represented in their pair sharing versions \(sh^p = \{\{v_0, v_1\}, \{v_0, v_3\}, \{v_1, v_1\}, \{v_2, v_2\}\}\) and \(sh^p_A = \{\{v_1, v_2\}, \{v_1, v_1\}, \{v_2, v_2\}\}\), the extend operation in [26] introduces spurious sharings in \(sh_f\) because of the lower precision of the pair-sharing representation. In this case, \(sh^{f2}_{\text{f2}} = (sh \cup sh^p_A)^*_{\text{f2}} = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_1, v_2\}, \{v_0, v_0\}, \{v_1, v_1\}, \{v_2, v_1\}\}\). This information, expressed in terms of set sharing, results in \(sh^{f2}_{\text{f2}} = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_0, v_0\}, \{v_1, v_2\}, \{v_1, v_1\}, \{v_2, v_1\}\}\), which is much less precise than \(sh_{f1}\).

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 (SSNITau) 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 SSNITau 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 SSNI. In these versions without dynamic type inference only declared types can affect \(\tau\) 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.
Fig. 5. Analysis times, number of program points, and number of abstract states.

The benchmarks used have been adapted from previous literature on either abstract interpretation for Java or points-to analysis [26, 24, 23, 30]. 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 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 (Ni) 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 #σ provides the total number of these abstract states inferred by the analysis. The level of multivariance is the ratio #σ/#rp. It can be observed that the simple set sharing analysis (SS) creates more abstract states for the same number of reachable points. In general, such a larger number for #σ tends to indicate more precise results (as we will see later). On the other hand, the fact that addition of Ni and Tau reduces the number of reachable program points interacts with precision to obtain the final #σ 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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#tp</th>
<th>#rp</th>
<th>#up</th>
<th>#σ</th>
<th>t</th>
<th>#rp/#up</th>
<th>%Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>dyndisp (*)</td>
<td>71</td>
<td>68</td>
<td>3</td>
<td>114</td>
<td>30</td>
<td>3</td>
<td>29</td>
</tr>
<tr>
<td>clone</td>
<td>41</td>
<td>38</td>
<td>3</td>
<td>42</td>
<td>52</td>
<td>3</td>
<td>50</td>
</tr>
<tr>
<td>dfs</td>
<td>102</td>
<td>98</td>
<td>4</td>
<td>103</td>
<td>68</td>
<td>4</td>
<td>108</td>
</tr>
<tr>
<td>passau (*)</td>
<td>167</td>
<td>164</td>
<td>3</td>
<td>296</td>
<td>97</td>
<td>3</td>
<td>304</td>
</tr>
<tr>
<td>qsort</td>
<td>185</td>
<td>142</td>
<td>43</td>
<td>182</td>
<td>125</td>
<td>142</td>
<td>204</td>
</tr>
<tr>
<td>integerqsort</td>
<td>191</td>
<td>148</td>
<td>43</td>
<td>159</td>
<td>110</td>
<td>148</td>
<td>197</td>
</tr>
<tr>
<td>pollet01 (*)</td>
<td>154</td>
<td>126</td>
<td>28</td>
<td>276</td>
<td>196</td>
<td>126</td>
<td>423</td>
</tr>
<tr>
<td>zipvector (*)</td>
<td>272</td>
<td>269</td>
<td>3</td>
<td>513</td>
<td>388</td>
<td>269</td>
<td>712</td>
</tr>
<tr>
<td>cleanness (*)</td>
<td>314</td>
<td>277</td>
<td>37</td>
<td>360</td>
<td>233</td>
<td>277</td>
<td>385</td>
</tr>
<tr>
<td>overall</td>
<td>1497</td>
<td>1330</td>
<td>167</td>
<td>2045</td>
<td>1299</td>
<td>1330</td>
<td>2374</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>SSNI</th>
<th>SSNITau</th>
</tr>
</thead>
<tbody>
<tr>
<td>#tp</td>
<td>#rp</td>
<td>#up</td>
</tr>
<tr>
<td>dyndisp (*)</td>
<td>71</td>
<td>61</td>
</tr>
<tr>
<td>clone</td>
<td>41</td>
<td>31</td>
</tr>
<tr>
<td>dfs</td>
<td>102</td>
<td>91</td>
</tr>
<tr>
<td>passau (*)</td>
<td>167</td>
<td>157</td>
</tr>
<tr>
<td>qsort</td>
<td>185</td>
<td>142</td>
</tr>
<tr>
<td>integerqsort</td>
<td>191</td>
<td>148</td>
</tr>
<tr>
<td>poUetOl (*)</td>
<td>154</td>
<td>119</td>
</tr>
<tr>
<td>zipvector (*)</td>
<td>272</td>
<td>269</td>
</tr>
<tr>
<td>cleanness (*)</td>
<td>314</td>
<td>276</td>
</tr>
<tr>
<td><strong>overall</strong></td>
<td><strong>1497</strong></td>
<td><strong>1294</strong></td>
</tr>
</tbody>
</table>

as dead code by the analysis. Thus, the #σ values for SSNI and SSNITau 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 %Δt columns show the percentage variation in the analysis time with respect to the reference pair-sharing (PS) analysis, calculated as $\Delta_{\text{dom}}%t = 100 \times (t_{\text{dom}} - t_{PS})/t_{PS}$. 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 $\mathcal{T}$ is the power set). This is of course assuming $\sigma$ 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 \textit{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 $\%_{\text{Max}} = 100 \times \left(1 - \frac{\# \text{sh}}{2^\# \text{vo} - 1}\right)$ which gives $\# \text{sh}$ as a percentage of its maximum possible value, where $\# \text{vo}$ is the total number of object variables in all the states. The results are given in column $\% \text{sh}$. In this metric 0% means all abstract states are T (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.


\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|c|}
\hline
 & PS & & SS & \\
 & $\# \text{sh}$ & $\% \text{sh}$ & $\# \text{sh}$ & $\% \text{sh}$ \\
\hline
\text{dyndisp} (*) & 640 & 60.37 & 435 & 73.07 \\
\text{clone} & 174 & 53.10 & 151 & 60.16 \\
\text{dfs} & 1573 & 96.46 & 1109 & 97.51 \\
\text{passau} (*) & 5828 & 94.56 & 3492 & 96.74 \\
\text{qsort} & 1481 & 67.41 & 1082 & 76.34 \\
\text{integerqsort} & 2413 & 66.47 & 1874 & 75.65 \\
\text{pollet01} (*) & 793 & 89.81 & 1043 & 91.81 \\
\text{zipvector} (*) & 6161 & 68.71 & 5064 & 80.28 \\
\text{cleanness} (*) & 1300 & 63.63 & 1189 & 70.61 \\
\hline
\text{overall} & 20363 & 73.39 & 15439 & 80.24 \\
\hline
\end{tabular}
\caption{Sharing precision results.}
\end{table}

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.
References

A Concrete semantics

We essentially analyze the same language as in [26]; there is a technical report available [27] containing the standard semantics of that subset of Java.

B Other semantics

\[ C^I_r[\text{return } \text{expr}](\phi \ast \mu) = C^I_r[\text{out=expr}](\phi \ast \mu) \]
\[ C^I_r[v : t](\phi \ast \mu) = \phi[v \mapsto \text{def}_r(t)] \ast \mu \]
\[ C^I_r[\text{skip}](\phi \ast \mu) = (\phi \ast \mu) \]

\[ \text{SC}^I_r[\text{return } \text{expr}](\phi \ast \mu) = \text{SC}^I_r[\text{out=expr}](\phi \ast \mu) \]
\[ \text{SC}^I_r[v : t](\phi \ast \mu) = \phi[\text{null}] \ast \mu \]
\[ \text{SC}^I_r[\text{skip}](\phi \ast \mu) = \phi \]

C Proofs

We have to prove that \( \alpha_\pi(C^I_r[\text{expr}](\gamma_r(\sigma)) \leq \text{SC}^I_r[\text{expr}](\sigma) \) (in the case of commands, \( \alpha_\pi(C^I_r[\text{com}](\gamma_r(\sigma)) \leq \text{SC}^I_r[\text{com}](\sigma) \). We denote by LHS the left-hand side of the equation, which will be further rewritten until showing that it is approximated by the right-hand side (RHS), the semantics described in Fig. 3 and 4. The abstraction function for the sharing component is \( \alpha_\pi(S) = \{ V \subseteq \text{dom}(\pi) \mid \exists \delta \in S \text{ s.t. } R_\pi(\delta, v_i) \neq \emptyset \text{ and } \exists W \subseteq \text{dom}(\pi) \text{ s.t. } V \subset W \text{ and } \bigcap_{v_i \in V} R_\pi(\delta, w_i) \neq \emptyset \} \). For the nullity component the abstraction is \( \alpha_\pi(S) = \{ v_i/\text{null} \in \text{dom}(\pi) \times \text{DN}_r \mid \forall \delta \in S, R_\pi(\delta, v_i) = \emptyset \} \cup \{ y_i/\text{null} \in \text{dom}(\pi) \times \text{DN}_r \mid \forall \delta \in S, R_\pi(\delta, w_i) = \emptyset \} \cup \{ y_i/\text{unk} \in \text{dom}(\pi) \times \text{DN}_r \mid y_i \notin V, y_i \notin W \} \}. Finally, types in the set of states \( S \) are abstracted as \( \alpha_\pi(S) = \{ v/T \in \text{dom}(\pi) \times \mathcal{P}(\mathcal{K}) \mid \forall \delta \in S, \psi_\pi(\delta, v) \in T \} \).

null

LHS = \( \alpha_\pi(\{ \phi[\text{res} \mapsto \text{null}] \ast \mu \mid \phi \ast \mu \in \gamma_r(\sigma) \}) \). However, the addition of null variables cannot affect the sharing (from the definition of \( \alpha_\pi \)) but only the nullity component. Therefore, LHS = \( \alpha_\pi(\{ \phi \ast \mu \mid \phi \ast \mu \in \gamma_r(\sigma) \}) \ast \text{null} = \alpha_\pi(\gamma_r(\sigma)) \ast \text{null} = (\text{null} \ast \text{null}) \leq \text{SC}^I_r[\text{null}]. \) The nullity value for \( \text{res} \) is trivially correct; the rest of the variables are unaffected. The type value of \( \text{res} \) is the most general one and therefore correct.

new k

LHS = \( \alpha_\pi(\{ \phi[\text{res} \mapsto l] \ast \mu[l \mapsto o] \mid \phi \ast \mu \in \gamma_r(\sigma) \}) \). Since \( l \) is a fresh location, \( \text{res} \) cannot reach any location already pointed to by another variable, so we can separate the memory state after the expression in two independent parts. By semantics of the language, \( l \) is a non-null location and therefore the nullity value for \( \text{res} \) correctly approximates the standard semantics; the type
value for \( res \) is just the one of the class constructor invoked; the rest of the variables see no changes and their current values for \( nl \) and \( \tau \) remain correct. LHS = \( \alpha_\pi([\phi \times \mu \mid \phi \times \mu \in \gamma_\pi(\sigma)]) \cup ([\{res\}] , [\{res/null\}] , [\{res/\{k\}\}]) = \alpha_\pi(\gamma_\pi(sh_h) \cup ([\{res\}] , [\{res/null\}] , [\{res/\{k\}\}]) = SE_{\pi}[new:\{k\}][\sigma]

\[ LHS = \alpha_\pi([\phi\{res \mapsto \phi(v)\} \times \mu \mid \phi \times \mu \in \gamma_\pi(\sigma)]) \]. We will call the new frame \( \phi' \). Since \( res \) is removed after evaluating an expression, we only have to check whether its addition to the frame is properly approximated. The new nullity and type values correctly approximate the effect of evaluating the expression, since \( v \) was correctly approximated by \( nl \) and \( \tau \) and now \( res \) is a synonym of \( v \); the rest of the variables remain unchanged so \( (nl[\{res \mapsto nl(v)\}], \tau[\{res \mapsto \tau(v)\}]) \) is a correct approximation for them.

If \( nl(v) = null \) the semantics is the same as in \texttt{null}; if not, in the new state \( \phi' \times \mu \) there is a subset of variables which did not reach any location reachable from \( v \). Those variables are unaffected and their previous approximation \( sh_{-v} \) is correct. For the rest of the variables, if \( sh_h \), approximated their reachability then \( sh_v \sqcup \{\{res\}\} \) is the minimal approximation for \( \phi' \times \mu \), since \( R_\pi(\phi' \times \mu, v) = R_\pi(\phi' \times \mu, res) \) and therefore there cannot be any sharing in which \( v \) is included but \( res \) is not.

\[ v.f \]
\[ LHS = \alpha_\pi([\phi\{res \mapsto l\} \times \mu \mid l = (obj(\phi\times\mu, v).f), \phi \times \mu \in \gamma_\pi(\sigma)] = \alpha_\pi([\phi' \times \mu]). \]

In a normal execution all those variables which did not reach a location reachable from \( v \) cannot be reached from \( res \), and therefore they are correctly approximated by \( sh_{-v} \). Variables \( \{w_1, \ldots, w_n\} \) in any \( \phi \times \mu \) verifying \( R_\pi(\phi \times \mu, w_1) \cap R_\pi(\phi \times \mu, v) \neq \emptyset \) might reach the \( l \) location or be reached from it. However, the only definitive information is that \( R_\pi(\phi' \times \mu, v) \cap R_\pi(\phi' \times \mu, res) \neq \emptyset \); information captured by applying the \( \omega \) operator between \{\{v\}\} and any sharing set where \( res \) appears. The remaining possibilities (including those already existing in \( \phi' \times \mu \)) are correctly abstracted by \( \{\{v\}\} \cup P(s_{-v} \cup \{res\}) \mid s \in sh_h \}, since we create a set for every possibility in a sharing set of \( sh_h \) but without introducing impossible sharings; for example, if \{\{v, a\}, \{v, b\}\} was the starting state, the expression \( v.f \) cannot introduce sharing between \( a \) and \( b \) and the result is \{\{v, a\}, \{v, a, res\}, \{v, b\}, \{v, b, res\}, \{v, res\}\}. The nullity value for \( res \) is correct since it is the most general one; type information is also trivially correct.

\texttt{call}(v, m(v_1, \ldots, v_n))

(We provide here an informal proof; the reader interested in the how the fixpoint is calculated in the presence of method calls can refer to \[21\].)

In Java method calls cannot alter the caller frame \( \{Fr_s\} \), but just subsequent levels of indirection: fields of variables in the scope of the caller. The only exception to this is the returned value \( res \). Hence, an analysis of the call which strictly computes the most general sharing for the actual parameters and \( res \)
starting at the caller state, and that assumes the most general nullity and type values for \(res\), is always correct.

An initial approximation for \(\sigma_f\) is therefore \(\text{star} = (\langle sh_{\lambda} \cup \{\{res\}\}\rangle)^* \cup \langle sh_{\lambda}, n|\text{res} \mapsto \text{unk}, \tau|\text{res} \mapsto (\langle \pi(\text{res})\rangle)\rangle\). We can improve precision by using the semantics of the callees \(\sigma_\lambda\). That semantics is correct since it approximates the call in all the class hierarchy of \(\pi(v)\), including \(\psi_\lambda(v)\).

The nullity and type value are trivial, since they cannot change during the call (but we can possibly find a more precise value for them, see Sect. 3.1) except for \(res\): we just copy the new values for that variable to \(nl_f\) and \(\tau_f\), which for the rest are clones of \(nl\) and \(\tau\), respectively. The sharing case is more complicated. On one hand we have \(sh_{\text{star}}\), which (provably) is an over-approximation for \(sh_f\), and on the other \(sh_{\lambda}\), which describes the final state exclusively about the actual parameters. We now filter out from the former those elements such that their information about the actual parameters and \(res\) is incompatible with \(sh_{\lambda}\), regardless of the other elements in the set, obtaining \(sh_{\text{ext}} = \{s \mid s \in sh_{\text{star}}, s|_{AR} \in sh_{\lambda}\}\). All the sharings not related to the actual parameters are preserved, resulting in \(sh_f = sh_{\text{ext}} \cup sh_{\lambda}\).

\[
v = \text{expr}
\]

LHS = \(\alpha_\gamma(\{\phi|v \mapsto \phi'(\text{res})\})\|_{-\text{res}} \times \mu|\phi \times \mu \in \gamma_\tau(\sigma))\}.\) The proof is analogous to the one of the \(v\) expression. Assume that the semantics \(\mathcal{C}_\gamma[\text{expr}]\) is correct, the concrete semantics of the assignment is identical to that of expression evaluation, just exchanging the \(res\) and \(v\) variables. In the case of nullity and types, the resulting state just replaces \(res\) by \(v\), which is the result of overwriting \(v\) values with those of \(res\) and then removing any occurrence of \(res\).

The sharing component is more complex. First, all previous sharings of \(v\) are deleted (\(sh' = sh|\_{-v}\)) and it now appears in all sharing groups where \(res\) was, approximated by (\(sh'_{\text{res}} \cup \langle sh_{\text{res}} \| \{v\}\rangle\)|\_{-\text{res}} = \(sh'_{\text{res}} \cup \langle sh'_{\text{res}}\rangle_{\|\text{res}}\)) = \(sh'|_{\|\text{res}} = (\mathcal{S}C_{\gamma}[v = \text{expr}|\sigma])_{\|\text{sh}}\).

\[v.f = \text{expr}\]

Analogous to the \(v.f\) proof, but taking into account that \(res\) might share with other variables (and has to be removed after the assignment). In this case, we propagate the created sharing sets through the star operation \([14,19]\).

\[
\text{if } v = w \text{ com}_1 \text{ else com}_2
\]

If \(sh|_{v,w} = \emptyset\), then \(\emptyset \in \gamma_\sigma(\sigma)\) s.t. \(\phi(v) = \phi(w)\) by definition of \(\gamma_\sigma(\sigma)\). Therefore, LHS = \(\alpha_\sigma(C_{\gamma}[\text{if} \ldots] \{\phi \times \mu \in \gamma_\sigma(\sigma) \mid \phi(v) = \phi(w)\}) = \alpha_{\sigma}(C_{\gamma}[\text{com}_2] \{\phi \times \mu \in \gamma_\sigma(\sigma) \mid \phi(v) = \phi(w)\}) \leq \mathcal{S}C_{\gamma}[\text{com}_2]_{\|\sigma} = \text{RHS}\).

If \(sh|_{v,w} \neq \emptyset\), then we might have \(\phi(v) = \phi(w)\) and LHS = \(\alpha_\sigma(C_{\gamma}[\text{com}_1] \{\phi \times \mu \in \gamma_\sigma(\sigma) \mid \phi(v) = \phi(w)\}) \cup \alpha_\sigma(C_{\gamma}[\text{com}_2] \{\phi \times \mu \in \gamma_\sigma(\sigma) \mid \phi(v) = \phi(w)\}) \leq \mathcal{S}C_{\gamma}[\text{com}_1]_{\|\sigma} \cup \mathcal{S}C_{\gamma}[\text{com}_2]_{\|\sigma} = \text{RHS}\).
if \( v==\text{null} \) \( \text{com}_1 \) else \( \text{com}_2 \)

If \( \sigma,\text{null}[v] = \text{null} \), the concretization function ensures \( \phi(v) = \text{null} \) \( \forall \phi \star \mu \in \gamma_\pi(\sigma) \)

thus \( \text{LHS} = \alpha_\pi(\mathcal{C}_\pi^I[\text{if...}]({\{ \phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = \text{null} \}}) - \alpha_\pi(\mathcal{C}_\pi^I[\text{com}_1]({\{ \phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = \text{null} \}})) \leq \mathcal{S}_\pi^I[\text{com}_1]_\pi = \text{RHS} \).

A similar reasoning can be applied for the case where \( \text{null}[v] = \text{null} \).

If \( \text{null}[v] = \text{unk} \), \( \text{LHS} = \alpha_\pi(\mathcal{C}_\pi^I[\text{if...}]({\{ \phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = \text{null} \}}) \cup \alpha_\pi(\mathcal{C}_\pi^I[\text{if...}]({\{ \phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = \text{null} \}})). \)

The first term is equivalent to \( \alpha_\pi(\mathcal{C}_\pi^I[\text{com}_1]({\{ \phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = \text{null} \}})) = \alpha_\pi(\mathcal{C}_\pi^I[\text{com}_1]({\{ \phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = \text{null} \}})) \text{ (by definition of } \gamma_\pi\text{), which is } \leq \mathcal{S}_\pi^I[\text{com}_1]_\pi \text{ (by definition of } \gamma_\pi\text{).} \)

In an analogous way, the second term is \( \alpha_\pi(\mathcal{C}_\pi^I[\text{com}_2]({\{ \phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = \text{null} \}})) = \alpha_\pi(\mathcal{C}_\pi^I[\text{com}_2]({\{ \phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = \text{null} \}})) \text{ (by definition of } \gamma_\pi\text{).} \)

Therefore, the left-hand side of the equation is approximated by the semantics given.

\( \text{com}_1 ; \text{com}_2 \)

True by correctness of the composition of correct operations.