Precise Set Sharing and Nullity Analysis for Java-style Programs

Mario Méndez Lojo
University of New Mexico (USA)
mario@cs.unm.edu

Manuel V. Hermenegildo
Technical University of Madrid (Spain)
University of New Mexico (USA)
herme@unm.edu

ABSTRACT

Finding useful *sharing* information between instances in object-oriented programs has been recently the focus of much research. The applications of such static analysis are multiple: by knowing which variables share in memory we can apply conventional compiler optimizations, find coarse-grained parallelism opportunities, or, more importantly,erify 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. The analysis is multivariant, which allows separating different contexts. We propose a combined *Set Sharing + Nullity + Classes* domain which captures which instances share and which ones do not or are definitively null, and which uses the classes to refine the static information when inheritance is present. Carrying the domains in a combined way facilitates the interaction among the domains in the presence of mutivariance in the analysis. We show 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.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*

General Terms

Languages, Verification, Reliability, Experimentation

Keywords

Pair sharing, set sharing, context sensitivity, class analysis.

1. INTRODUCTION

The technique of Abstract Interpretation [7] 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 [12, 19, 24] aims to detect which variables share in memory, i.e., 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., [30, 4, 11, 13]). Obtaining a safe approximation of which instances might share allows parallelizing segments of code, 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, 9, 20], (sometimes called *type* analysis) focuses on determining, in the presence of polymorphic calls, which particular implementation of a given method will be executed at run-time, i.e., what is the specific class of the called object in the hierarchy. Multiple compilation optimizations can be derived 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 however most of its important features: inheritance, polymorphism, visibility of methods, etc. The analysis is multivariant, based on the algorithm of [15], which allows separating different contexts and 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 [29].

The objective of using a reduced cardinal product [8] 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 [5, 8]. 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 [5, 6]. The final objectives of our analysis include verification, static debugging, and optimization.

The closest related work is that of [24] which develops a *pair*-sharing [26] 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 can result in increased accuracy in some situations and can be more appropriate for certain applications, such as detecting independence in program parallelization. Also, our domain and abstract semantics track additionally nullity and classes in a combined fashion which, as we have argued above, is specially useful in the presence of multivariance. In addition, we also deal directly with a larger set of object features such as inheritance or visibility. Finally, we have implemented our domains (as well as the pair sharing domain of [24]), integrated them in our multivariant analysis and verification framework, and benchmarked the results. Our experimental results are encouraging in the sense that they seem to support that our contributions bring more precision at reasonable cost. In [22], 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 [23] 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 (the examples in [21] do not exceed one hundred lines) and the abstract operations are more difficult to design. We also support some language constructions (e.g., visibility of methods) and provide detailed benchmarks omitted in their work.

Most recent work [27, 17, 29] has focused on context-sensitive approaches to the points-to problem for Java. These solutions are quite scalable and precise, but flow-insensitive and unsound. 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

```
prog        ::=  class_decl*
class_decl  ::=  class k1 [extends k2] decl* meth_decl*
meth_decl   ::=  vsly (t_ret|void) meth decl* com
vsly        ::=  public | private
com         ::=      v = expr          |   v.f = expr
                 |   decl              |   skip
                 |   return expr       |   com;com
                 |   if v (== |! =) (null|w) com else com
decl      ::=  v:t
var_lit   ::=  v | a
expr      ::=  null | new k | v.f | v.m(v1, ... vn) | var_lit
```

Figure 1: Grammar for the language.

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.

The rules for the grammar of this language are listed in Fig. 1. The `skip` statement, not present in the Java standard specification [10], 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 interesting parts that we will be referring to.

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 $\rightharpoonup$. The powerset of a set s is $\mathcal{P}(s)$. We use $f : D_1 \star D_2$ to designate functions, i.e., possibly infinite tables of pairs such that there are no repeated elements of D_1. The $dom : D_1 \star D_2 \mapsto D_1$ function returns all the elements in D_1 for which an f is defined; for the codomain we will use $rng : D_1 \star D_2 \mapsto D_2$. 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 sets 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 from f all pairs (d_1, d_2) s.t. $d_1 \in S$. Conversely, $f|_S$ restricts $dom(f)$ to S. In both *projections* we require $S \subseteq D_1$. 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), 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 members of $\mathcal{K}$ the predefined classes $Pr = \{$`Object`,`null`$\}$.

[1]Full source code for the example can be found in
http://www.cliplab.org/Users/mario/research/oo_shnltau/examples/

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

class Vector{
   Element first;

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

Figure 2: Vector example.

$\mathcal{K}$ forms a semi-lattice implied by a subclass relation $\downarrow : \mathcal{K} \rightarrow \mathcal{P}(\mathcal{K})$ such that if $t_2 \in \downarrow t_1$ then $t_2 \leq_{\mathcal{K}} t_1$. The semantics of the language implies $\downarrow$`Object` $= \mathcal{K}$ and `null`$\leq_{\mathcal{K}} k \quad \forall k \in K$. Given $def : K \star \mathcal{M} \mapsto \mathcal{B}$, that determines whether a particular class provides its own implementation for a method, the boolean function $redef : K \star K \star \mathcal{M} \mapsto \mathcal{B}$ 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 $\pi : dom(\pi) \mapsto \mathcal{K}$ 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 retrieves the fields of $k \in \mathcal{K}$) $field\_type : \mathcal{K} \star \mathcal{V} \mapsto \mathcal{K}$ (which returns the class of a particular field of another class assuming $f \in F(k)$), and $type_\pi(expr)$ (which maps expressions to types, according to π).

The description of the memory state is based on the formalization in [24, 16]. We define a frame as any element of $Fr_\pi = \{\phi \mid \phi \in dom(\pi) \mapsto Loc \cup \{\text{null}\}\}$. A frame represents the first level of indirection and maps variable names to locations (where $Loc \cap \{\text{null}\} = \emptyset$) 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 $\Sigma_\pi = \{(\phi \star \mu) \mid \phi \in Fr_\pi, \mu \in Mem, (\phi \star \mu) : \pi\}$. 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 $R_\pi(\phi \star \mu, v) = lfp(\cup \left\{ R^i_\pi(\phi \star \mu, v) \mid i \geq 0 \right\})$, 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 $\{v_1, \ldots, v_n\}$ *share* in the actual memory configuration $\phi \star \mu$ if there is at least one common location in their reachability sets, i.e., $\cap^n_{i=1} R_\pi(\phi \star \mu, v_i) \neq \emptyset$. A variable $v \in dom(\pi)$ is *null* in state $\phi \star \mu$ if $R_\pi(\phi \star \mu, v) = \emptyset$. Nullity is checked by means of $nil_\pi : \Sigma_\pi \star dom(\pi) \mapsto \mathcal{B}$, 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 $\psi_\pi : \Sigma_\pi \star dom(\pi) \mapsto \mathcal{K}$, 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 [24]. 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 $I(k.\mathtt{m}) : \Sigma_{input(k.m)} \rightarrow \Sigma_{output(k.m)}$. The definition of the respective environments, given a declaration in class k as t_{ret} $\mathtt{m}(this : k, p_1 : t_1 \ldots p_n : t_n)$ `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. $loc_1 = \phi_0(element) \neq null$. After executing `Element e = new Element()` the state is $(\phi_1 \star \mu_1)$, with $\phi_1(e) = loc_2$, $\mu_1(loc_2).\phi(next) = null$, and $\mu_2(loc_2).\phi(value) = 0$. The second line in the method manipulates *primitive* values, which are different from locations, producing $\mu_2(loc_2)\ .\phi(value) = i \notin Loc$ so $R_\pi((\phi_2 \star \mu_2), e) \cap R_\pi((\phi_2 \star \mu_2), element) = \emptyset$. The creation of `v` sets $loc_3 = \phi_3(v)$ and `v.first = e` links loc_2 and loc_3 since now $\mu_4(loc_3).\phi(first) = loc_2$. Now v and e share, since their reachability sets intersect *at least* in $\{loc_2\}$. Finally, assume that `append` attaches v to the end of the current instance *this* resulting in a memory layout $(\phi_5 \star \mu_5)$. Given $loc_4 = obj((\phi_5 \star \mu_5)(this)).\phi(first)$, it should hold that $\mu_5(.\overset{*}{.}.\mu_5(loc_4).\phi(next)\ .\overset{*}{.}.).\phi(next) = loc_3$. Now *this* shares with v and therefore with e, because loc_2 is reachable from loc_3.

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 $\mathcal{DS}_\pi = \{\{v_1, \ldots, v_n\} \in \mathcal{P}(dom(\pi)) \mid \cap^n_{i=1} C_\pi(v_i) \neq \emptyset\}$ is constrained by a class reachability function which retrieves those classes that are reachable from a particular variable: $C_\pi(v) = lfp(\cup\{C^i_\pi(v) \mid i \geq 0\})$,

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

$$\mathcal{SE}^I_\pi\llbracket\texttt{new}\ k\rrbracket(sh,nl,\tau)=(sh',nl',\tau')$$
$$sh'=sh\cup\{\{res\}\}\qquad nl'=nl[res\mapsto nnull]$$
$$\tau'=\tau[res\mapsto\{\kappa\}]$$
$$\mathcal{SE}^I_\pi\llbracket v\rrbracket(sh,nl,\tau)=(sh',nl',\tau')$$
$$sh'=(\{\{res\}\}\uplus sh_v)\cup\{\{v,res\}\}\cup sh_{-v}$$
$$nl'=nl[res\mapsto nl(v)]\qquad \tau'=\tau[res\mapsto\tau(v)]$$

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

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

Figure 3: Abstract semantics for the expressions.

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.

The binary union $\uplus:\mathcal{DS}_\pi\times\mathcal{DS}_\pi\mapsto\mathcal{DS}_\pi$ and closure under union $*:\mathcal{DS}_\pi\mapsto\mathcal{DS}_\pi$ operators are standard in the sharing literature [12, 18]; we just 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 $\{S\mid S=S'\cap V, S'\in sh\}$.

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 [7] in terms of Galois insertions. The concretization function $\gamma_\pi:D_\pi\mapsto\mathcal{P}(\Sigma_\pi)$ is $\gamma_\pi(sh,nl,\tau)=\{\delta\in\Sigma_\pi\mid\forall V\subseteq dom(\pi)$ if $\bigcap_{v_i\in V}R_\pi(\delta,v_i)\neq\emptyset$ then $V\in sh$ 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 Sect. C. 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 allows preventing a loss of precision,

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

$$\mathcal{SC}^I_\pi\llbracket v\texttt{.f=}expr\rrbracket\sigma=(sh'',nl',\tau')|_{-res}$$
$$sh''=\begin{cases}\bot \quad \text{if } nl'(v)=null\\ sh' \quad \text{if } nl'(res)=null\\ sh^y\cup sh'_{-\{v,res\}} \quad \text{otherwise}\end{cases}$$
$$sh^y=(\{\{\{v\}\}\uplus\mathcal{P}(s|_{-v}\cup\{res\})\mid s\in sh'_v\}\cup$$
$$\{\{\{res\}\}\uplus\mathcal{P}(s|_{-res}\cup\{v\})\mid s\in sh'_{res}\})^*$$
$$(sh',nl',\tau')=\mathcal{SE}^I_\pi\llbracket expr\rrbracket\sigma$$
$$\mathcal{SC}^I_\pi\left\llbracket\begin{array}{l}\texttt{if}\ v\texttt{==null}\\ com_1\\ \texttt{else}\ com_2\end{array}\right\rrbracket\sigma=\begin{cases}\sigma'_1 \quad \text{if } nl(v)=null\\ \sigma_1\sqcup\sigma_2 \quad \text{if } nl_n(v)=unk\\ \quad \sigma_1=\mathcal{SC}^I_\pi\llbracket com_1\rrbracket(sh|_{-v},nl[v\mapsto null],\tau)\\ \quad \sigma_2=\mathcal{SC}^I_\pi\llbracket com_2\rrbracket(sh,nl[v\mapsto nnull],\tau)\\ \sigma'_2 \quad \text{if } nl(v)=nnull\end{cases}$$
$$\sigma'_i=\mathcal{SC}^I_\pi\llbracket com_i\rrbracket\sigma$$
$$\mathcal{SC}^I_\pi\left\llbracket\begin{array}{l}\texttt{if}\ v\texttt{==}w\ com_1\\ \texttt{else}\ com_2\end{array}\right\rrbracket\sigma=\begin{cases}\sigma'_2 \quad \text{if } sh|_{\{v,w\}}=\emptyset\\ \sigma'_1\sqcup\sigma'_2 \quad \text{otherwise}\end{cases}$$
$$\mathcal{SC}^I_\pi\llbracket com_1;com_2\rrbracket\sigma=\mathcal{SC}^I_\pi\llbracket com_2\rrbracket(\mathcal{SC}^I_\pi\llbracket com_1\rrbracket\sigma)$$

Figure 4: Abstract semantics for the commands.

as shown in the following example.

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 $sh'=(\{\{res\}\}\uplus sh_{el})\cup\{\{res,this\}\}\cup sh_{-el}=\{\{res\}\}\uplus\{\{this,el\}\}\cup\{\{v\}\}=\{\{res,this\},\{res,this,el\},\{v\}\}$. Note that, for expressing the same information in pair-sharing, we can only use $\{\{res,this\},\{res,el\},\{this,el\},\{v,v\}\}$, which is also the pair-sharing representation of the more imprecise set sharing $\{\{res,this\},\{res,this,el\},\{res,el\},\{this,el\},\{v\}\}$.

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 the last element of *this* is 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 $\mathcal{SC}^I_\pi\llbracket com\rrbracket\sigma_1=\mathcal{SC}^I_\pi\llbracket com\rrbracket\sigma_2=(\{\{this,v\}\},\{this/nnull,v/nnull\},\{this/\{\{vector\}\},v/\{\{vector\}\}\})=\sigma_3$. However, the more precise sharing information in σ_2 results on a more precise analysis of `bdy`, because of the guard `(this!=v)`. In the case of the external calls, $\mathcal{SC}^I_\pi\llbracket\texttt{bdy}\rrbracket\sigma_1=\mathcal{SC}^I_\pi\llbracket com\rrbracket\sigma_1\sqcup\mathcal{SC}^I_\pi\llbracket skip\rrbracket\sigma_1=\sigma_1\sqcup\sigma_3=\sigma_1$. When the entry state is σ_2, the semantics at the same program point is $\mathcal{SC}^I_\pi\llbracket\texttt{bdy}\rrbracket\sigma_2=\mathcal{SC}^I_\pi\llbracket com\rrbracket\sigma_2=\sigma_3$. 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 `call`$(v,\ m(v_1,\ldots,v_n))$ in state $\sigma = (sh, nl, \tau)$ is calculated by implementing the top-down methodology described in [15]. 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 $\sigma_y = \sigma_p|_A^F$. The abstract execution of the call takes place only in the set of classes $K = \tau(v)$, resulting in an exit state $\sigma_x = \bigsqcup\{\mathcal{SC}_\pi^I[\![k^{'}.m]\!]\sigma_y \mid k^{'} = lookup(k, m), k \in K\}$, where *lookup* returns the body of k's implementation of m, which can be defined in k or inherited from one of its ancestors; we assumed that the formal parameters follow the naming convention F in all the implementations. 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: $\sigma_\lambda = \sigma_b|_{F\cup\{out\}}^{A\cup\{res\}}$; the final state σ_f is calculated as $sh_f = ext(sh_\lambda, sh)$, $nl_f = nl[res \mapsto nl_\lambda(res)]$, and $\tau_f = \tau[res \mapsto \tau_\lambda(res)]$. The $ext : \mathcal{DS}_\pi \times \mathcal{DS}_\pi \mapsto \mathcal{DS}_\pi$ function for sharing sets is essentially the same as the one described in [18], but taking into account that *res* is null before the call but it might be not null after it.

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 (and *this*) at the beginning of each call, as suggested in [22]. 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 A 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 like [24] can only tell if a variable is definitely null, but never if it is definitely not 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 $\sigma_b = (\{\{this, v\}\}, \{this/nnull, v/nnull\}, \{this/\{vector\}, v/\{vector\}\})$, which is further renamed to the scope of the caller obtaining $\sigma_\lambda = (\{\{v_1, v_2\}\}, \{v_1/nnull, v_2/nnull\}, \{v_1/\{vector\}, v_2/\{vector\}\})$. Since the method returns a `void` type we can treat *res* as a primitive (null) variable so $sh_f = ext(\{\{v_1, v_2\}\}, \{\{v_1\}\{v_2\}\}) = \{\{v_1, v_2\}\}, nl_f = nl[res \mapsto null], \tau_f = \tau[res \mapsto \{void\}]$.

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 [15]. It traverses the program dependency graph computing dynamically the strongly connected components and keeping detailed dependencies which track which parts of the graph need to be recomputed when some abstract value changes during analysis of iterative code (loops and recursions). This reduces the number of steps and iterations required to reach fixpoint. This 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 of our set-sharing, nullity, and class (*SSNlTau*) analysis. In order to be able to provide a comparison with previous work, we also implemented the pair sharing (*PS*) analysis proposed in [24]. We have extended somewhat the operations described in [24] extending it in order to handle some additional cases required by our benchmark programs such as primitive variables, visibility of methods, etc. Also, to allow direct comparison, we have implemented a version of our *SSNlTau* analysis, which we will refer to simply as *SS*, that tracks simply set sharing using only declared type information and also without the (non-)nullity component. Also, in order to study the influence of tracking run-time types we have implemented as well 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. In order to keep track of this, 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 [24, 22, 21, 28]. We added two different versions of the `Vector` example of Fig. 2. Our experimental results are summarized in Tables 5 and 6.

The first column (*#tp*) in Table 5 shows the total number

		PS				SS				SSNl				SSNlTau			
	$\#tp$	$\#rp$	$\#up$	$\#\sigma$	t	$\#rp$	$\#up$	$\#\sigma$	$\%\Delta t$	$\#rp$	$\#up$	$\#\sigma$	$\%\Delta t$	$\#rp$	$\#up$	$\#\sigma$	$\%\Delta t$
dyndisp*	71	68	3	114	30	68	3	114	-2	61	10	103	77	61	10	77	-33
clone	41	38	3	42	52	38	3	50	55	31	10	34	92	31	10	34	74
dfs	102	98	4	103	68	98	4	108	0	91	11	91	89	91	11	91	166
passau*	167	164	3	296	97	164	3	304	23	157	10	288	18	157	10	270	17
qsort	185	142	43	182	125	142	43	204	32	142	43	196	125	142	43	196	119
intgrqsort	191	148	43	159	110	148	43	197	10	148	43	202	107	148	43	202	224
pollet01*	154	126	28	276	196	126	28	423	30	119	35	364	98	98	56	296	35
zipvector*	272	269	3	513	388	269	3	712	164	269	3	791	36	245	27	676	136
cleanness*	314	277	37	360	233	277	37	385	116	276	38	383	38	266	48	385	77

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

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 (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 can thus keep track of 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 analysis. The level of multivariance is the ratio $\#\sigma/\#rp$. 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 $\#\sigma$ 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 Table 5 provides the running times for the different analyses, in milliseconds, on a Pentium III 2.0Ghz, 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 6 shows precision results in terms of sharing, concentrating on the SP and SS domains, which allow direct comparison. Following [5], and in order to be able to compare precision 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 [5])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 encouraging improvements 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 sep-

	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

Figure 6: Sharing precision results.

arating different contexts, and combines Set Sharing, Nullity, and Classes: the domain captures which instances share and which ones do not 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.

6. REFERENCES

[1] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP*, pages 2–26, 1995.

[2] Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer, 1999.

[3] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA*, pages 324–341, 1996.

[4] Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *LCPC*, pages 234–250, 1994.

[5] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems*, 17(1):28–44, January 1995.

[6] Agostino Cortesi, Gilberto File, Francesco Ranzato, Roberto Giacobazzi, and Catuscia Palamidessi. Complementation in abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 19(1):7–47, 1997.

[7] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.

[8] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Sixth ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979.

[9] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP*, pages 77–101, 1995.

[10] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[11] Michael Hind, Michael G. Burke, Paul R. Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, 1999.

[12] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[13] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing (with retrospective). In Kathryn S. McKinley, editor, *Best of PLDI*, pages 473–489. ACM, 1992.

[14] M. Méndez and M. Hermenegildo. Precise Set Sharing and Nullity Analysis for Java-style Programs. Technical Report CLIP2/2007.0, Technical University of Madrid (UPM), School of Computer Science, UPM, February 2007.

[15] M. Méndez, J. Navas, and M. Hermenegildo. An Efficient, Parametric Fixpoint Algorithm for Analysis of Java Bytecode. In *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*, 2007. To appear.

[16] Patricia M.Hill, Etienne Payet, and Fausto Spoto. Path-length analysis of object-oriented programs. In *Proc. EAAI*, 2006.

[17] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ISSTA*, pages 1–11, 2002.

[18] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[19] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[20] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *OOPSLA*, pages 146–161, 1991.

[21] Isabelle Pollet. *Towards a generic framework for the abstract interpretation of Java*. PhD thesis, Catholic University of Louvain, 2004. Dept. of Computer Science.

[22] Isabelle Pollet, Baudouin Le Charlier, and Agostino Cortesi. Distinctness and sharing domains for static analysis of java programs. In *ECOOP '01: 15th European Conference on Object-Oriented Programming*, London, UK, 2001.

[23] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.

[24] Stefano Secci and Fausto Spoto. Pair-sharing analysis of object-oriented programs. In *SAS*, pages 320–335, 2005.

[25] Stefano Secci and Fausto Spoto. Pair-sharing analysis of object-oriented programs. Technical report, Dpt. of Computer Science, Universitity of Verona, Italy, 2005.

[26] H. Sondergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[27] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *PLDI*, pages 387–400, 2006.

[28] M. Streckenbach and G. Snelting. Points-to for java: A general framework and an empirical comparison. Technical report, University Passau, November 2000.

[29] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144. ACM, 2004.

[30] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *PLDI*, pages 1–12, 1995.

APPENDIX

A. CONCRETE SEMANTICS

We essentially analyze the same language as in [24]; there is a technical report available [25] containing the standard semantics of that subset of Java.

B. OTHER SEMANTICS

$\mathcal{C}^I_\pi[\![\texttt{return}\ expr]\!](\phi \star \mu) = \mathcal{C}^I_\pi[\![out{=}expr]\!](\phi \star \mu)$
$\mathcal{C}^I_\pi[\![v : t]\!](\phi \star \mu) = \phi[v \mapsto def\_val(t)] \star \mu$
$\mathcal{C}^I_\pi[\![\texttt{skip}]\!](\phi \star \mu) = (\phi \star \mu)$

$\mathcal{SC}^I_\pi[\![\texttt{return}\ expr]\!]\sigma = \mathcal{SC}^I_\pi[\![out{=}expr]\!]\sigma$
$\mathcal{SC}^I_\pi[\![v : t]\!](sh, nl, \tau) = (sh, nl[v \mapsto null], \tau[v \mapsto \downarrow t])$
$\mathcal{SC}^I_\pi[\![\texttt{skip}]\!]\sigma = \sigma$

C. PROOFS

He have to prove that $\alpha_\pi(\mathcal{E}^I_\pi[\![expr]\!](\gamma_\pi(\sigma)) \leq \mathcal{SE}^I_\pi[\![expr]\!]\sigma$ (alternatively, $\alpha_\pi(\mathcal{C}^I_\pi[\![com]\!](\gamma_\pi(\sigma)) \leq \mathcal{SC}^I_\pi[\![com]\!]\sigma$). 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 $\alpha_\pi(S) = \{V \subseteq dom(\pi) \mid \exists\delta \in S \text{ s.t. } \bigcap_{v_i \in V} R_\pi(\delta, v_i) \neq \emptyset \text{ and } \nexists W \subseteq dom(\pi) \text{ s.t. } V \subset W \text{ and } \bigcap_{w_i \in W} R_\pi(\delta, w_i) \neq \emptyset\}$. For the nullity component the abstraction is $\alpha_\pi(S) = \{v_i/null \in dom(\pi) \times \mathcal{DN}_\pi \mid \forall\delta \in S,\ R_\pi(\delta, v_i) = \emptyset\} \cup \{w_i/nnull \in dom(\pi) \times \mathcal{DN}_\pi \mid \forall\delta \in S,\ R_\pi(\delta, w_i) \neq \emptyset\} \cup \{y_i/unk \in dom(\pi) \times \mathcal{DN}_\pi \mid y_i \notin V, y_i \notin W\}$. Finally, types in the set of states S are abstracted as $\alpha_\pi(S) = \{v/T \in dom(\pi) \times \mathcal{P}(\mathcal{K}) \mid \forall\delta \in S,\ \psi_\pi(\delta, v) \in T\}$.

`null`
LHS $= \alpha_\pi(\{\phi[res \mapsto null] \star \mu \mid \phi \star \mu \in \gamma_\pi(\sigma)\})$. However, the addition of null variables cannot affect the sharing (from the definition of α_π) but only the nullity component. Therefore, LHS $=\alpha_\pi(\{\phi\star\mu \mid \phi\star\mu \in \gamma_\pi(\sigma)\}).nl[res \mapsto null] = \alpha_\pi(\gamma_\pi(\sigma)).nl[res \mapsto null] = (sh, nl[res \mapsto null], \tau) \leq \mathcal{SE}^I_\pi[\![\texttt{null}]\!]\sigma$. The nullity value for res is trivially correct (same applies for types); the rest of variables are unaffected. The type value of res is the most general one and therefore correct.

`new` k
LHS $= \alpha_\pi(\{\phi[res \mapsto l] \star \mu[l \mapsto o] \mid \phi\star\mu \in \gamma_\pi(\sigma)\})$. Since l is a fresh location, res cannot reach any location already pointed to by another variable. LHS$= \alpha_\pi(\{\phi \star \mu \mid \phi \star \mu \in \gamma_\pi(\sigma)\}) \sqcup (\{\{res\}\}, \{nl \mapsto null\}, \tau) = \alpha_\pi(\gamma_\pi(sh)) \sqcup (\{\{res\}\}, \{nl \mapsto nnull\}, \tau) = \mathcal{SE}^I_\pi[\![\texttt{new}\ k]\!]\sigma$. By semantics of the language, l is a not null location and therefore the nullity value for res correctly approximates the standard semantics; the type value for res is just the one of the class constructor invoked; the rest of variables see no changes and their current values for nl and τ remain correct.

v
LHS $= \alpha_\pi(\{\phi[res \mapsto \phi(v)] \star \mu \mid \phi \star \mu \in \gamma_\pi(\sigma)\})$. We will call the new frame ϕ'. 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 τ and now res is a synonym of v; the rest of 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 `null`; if not, in the new state $\phi' \star \mu$ 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 variables, if sh_v approximated their reachabilities then $sh_v \uplus \{\{res\}\}$ is the minimal approximation for $(\phi' \star \mu') = \phi[res \mapsto \phi(v)] \star \mu$, since $R_\pi((\phi' \star \mu'), v) = R_\pi((\phi' \star \mu'), res)$ and therefore there cannot be any sharing in which v is included but res is not.

`v.f`
LHS $= \alpha_\pi(\{\phi[res \mapsto l] \star \mu \mid l = (obj(\phi \star \mu, v).\phi)(f)\ , \phi \star \mu \in \gamma_\pi(\sigma)\}) = \alpha_\pi(\{\phi' \star \mu\})$. 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 $(\phi \star \mu)$ verifying $R_\pi(\phi \star \mu, w_i) \cap R_\pi(\phi \star \mu, v) \neq \emptyset$ might reach the l location or be reached from it. Therefore, the only sure information is that $R_\pi(\phi' \star \mu, v) \cap R_\pi(\phi' \star \mu, res) \neq \emptyset$, information captured by $\{\{v\}\} \uplus \{\{res\}\}$. The remaining possibilities (including those already existing in $\phi \star \mu$) are correctly abstracted by $\{\{\{v\}\} \uplus \mathcal{P}(s|_{-v} \cup \{res\}) \mid s \in sh_v\}$, since we create a set for every possibility in a sharing set of sh_v 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 $\{\{V, A\}, \{V, A, Res\}, \{V, B\}, \{V, B, Res\}\}$. The nullity value for res is correct since it is the most general one.

`call`$(v, m(v_1, \ldots, v_n))$
See the description of the fixpoint algorithm in [14].

$v = expr$
LHS $= (\alpha_\pi(\{\phi'[v \mapsto \phi'(res)])|_{-res}) \star \mu' \mid \phi \star \mu \in \gamma_\pi(\sigma)\})$. The proof is analogous to the one of the v expression. Assume that the semantics $\mathcal{E}^I_\pi[\![expr]\!]$ 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 remove any appereance 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 $(sh'_{-res} \cup (sh_{res} \uplus \{\{v\}\}))|_{-res} = sh'_{-res} \cup (sh'_{res}|^v_{res}) = sh'|^v_{res} = (\mathcal{SC}^I_\pi[\![v\text{=}expr]\!]\sigma).sh.$

`v.f`$=$ $expr$
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 [12, 18].

`if` v==w com_1 `else` com_2
If $sh|_{\{v,w\}} = \emptyset$, then $\nexists \delta \in \gamma_\pi(\sigma)$ s.t. $\phi(v) = \phi(w)$ by definition of $\gamma_\pi(\sigma)$. Therefore, LHS$=\alpha_\pi(\mathcal{C}^I_\pi[\![\texttt{if}\ldots]\!](\{\phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) \neq \phi(w)\}))=\alpha_\pi(\mathcal{C}^I_\pi[\![com_2]\!](\{\phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) \neq \phi(w)\}))=\alpha_\pi(\mathcal{C}^I_\pi[\![com_2]\!](\{\phi \star \mu \in \gamma_\pi(\sigma)\}))\leq \mathcal{SC}^I_\pi[\![com_2]\!]\sigma =$RHS.

If $sh|_{\{v,w\}} \neq \emptyset$, then we might have $\phi(v) = \phi(w)$ and LHS$=$ $\alpha_\pi(\mathcal{C}^I_\pi[\![com_1]\!](\{\phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = \phi(w)\})) \sqcup \alpha_\pi(\mathcal{C}^I_\pi[\![com_2]\!](\{\phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) \neq \phi(w)\})) \leq \alpha_\pi(\mathcal{C}^I_\pi[\![com_1]\!](\{\phi \star \mu \in \gamma_\pi(\sigma)\})) \sqcup \alpha_\pi(\mathcal{C}^I_\pi[\![com_2]\!](\{\phi \star \mu \in \gamma_\pi(\sigma)\})) \leq \mathcal{SC}^I_\pi[\![com_1]\!]\sigma \sqcup \mathcal{SC}^I_\pi[\![com_2]\!]\sigma$=RHS.

`if` v==`null` com_1 `else` com_2
If $\sigma.nl[v] = null$, the concretization function ensures $\phi(v) = null\ \forall \phi \star \mu \in \gamma_\pi(\sigma)$ thus LHS$=\alpha_\pi(\mathcal{C}^I_\pi[\![\texttt{if}\ldots]\!](\{\phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = null\}))=\alpha_\pi(\mathcal{C}^I_\pi[\![com_1]\!](\{\phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = null\}))=\ \alpha_\pi(\mathcal{C}^I_\pi[\![com_1]\!](\{\phi \star \mu \in \gamma_\pi(\sigma)\})) \leq \mathcal{SC}^I_\pi[\![com_1]\!]\sigma$ $=$RHS. A similar reasoning can be applied for the case where $nl[v] = nnull$.

If $nl[v] = unk$, LHS$=\alpha_\pi(\mathcal{C}^I_\pi[\![\texttt{if}\ldots]\!](\{\phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = null\})) \sqcup \alpha_\pi(\mathcal{C}^I_\pi[\![if...]\!](\{\phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = nnull\}))$. The first term is equivalent to $\alpha_\pi(\mathcal{C}^I_\pi[\![com_1]\!])(\{\phi \star \mu \in \gamma_\pi(\sigma) | \phi(v) = null\})) = \alpha_\pi(\mathcal{C}^I_\pi[\![com_1]\!])(\{\phi \star \mu \in \gamma_\pi(sh|_{-v}, nl[v \mapsto null])\})$ (by definition of γ_π), which is $\leq \mathcal{SC}^I_\pi[\![com_1]\!](sh|_{-v}, nl[v \mapsto null])$. In an analogous way, the second term is $\alpha_\pi(\mathcal{C}^I_\pi[\![com_2]\!](\{\phi \star \mu \in \gamma_\pi(\sigma) | \phi(v) \neq null\})) = \alpha_\pi(\mathcal{C}^I_\pi[\![com_2]\!])(\{\phi \star \mu \in \gamma_\pi(sh, nl[v \mapsto nnull])\}) \leq \mathcal{SC}^I_\pi[\![com_2]\!](sh, nl[v \mapsto nnull])$. Therefore, the left-hand side of the equation is approximated by the semantics given.

$com_1; com_2$
True by correctness of the composition of correct operations.