Exploiting linearity in sharing analysis of object-oriented programs

Gianluca Amato Maria Chiara Meo Francesca Scozzari

Dipartimento di Economia, Università di Chieti-Pescara, Pescara, Italy

Abstract

We propose a new sharing analysis of object-oriented programs based on abstract interpretation. Two variables share when they are bound to data structures which overlap. We show that sharing analysis can greatly benefit from linearity analysis. We propose a combined domain including aliasing, linearity and sharing information. We use a graph-based representation of aliasing information which naturally encodes sharing and linearity information, and define all the necessary operators for the analysis of a Java-like language.

Keywords: Sharing analysis, linearity, aliasing, object-oriented programming.

1 Introduction

In object-oriented languages, program variables are often bound to complex data structures which may overlap. This is the case for Java programs, whose objects are stored in a shared memory called heap. Discovering whether two data structures may overlap is the scope of sharing analysis. This information is used in program parallelization and distribution: data structures which do not overlap allow the execution of methods on different processors, using disjoint memory. Moreover, it is very useful for improving other kind of analysis, like shape, pointer, class and cyclicity analysis. Sharing properties has been deeply studied for logic programs (e.g., [10,9,12,8,5,2]) and the large literature on this topic has been the starting point for designing our enhanced abstract domain for sharing analysis. In particular, the use of a linearity property [7,9,12,11,3,4] has proved to be very useful when dealing with sharing information (see [6] for a comparative evaluation). We show how the same idea can be rephrased to enhance sharing analysis of object-oriented programs. We propose a new combined analysis of sharing, aliasing and linearity properties for Java-like programs based on abstract interpretation, inspired by the corresponding domains on logic programs.

[1] Email: {gamato, cmeo, fscozzari}@unich.it

[2] The authors would like to thank Fausto Spoto for helpful suggestions.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

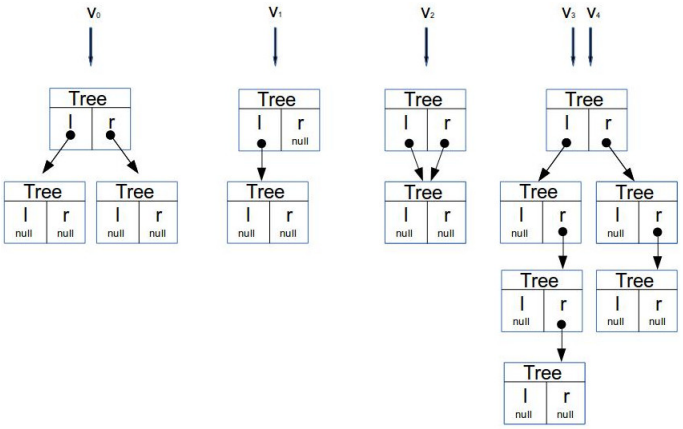


Fig. 1. A concrete state with variables v_0, v_1, v_2, v_3, v_4.

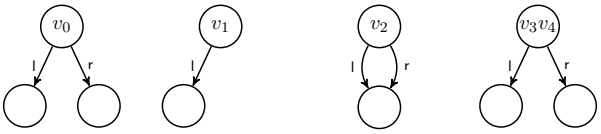


Fig. 2. Abstraction of the concrete state in Fig. 1.

A concrete state in a object-oriented program is usually described by a *frame*, which is a map from variables to memory locations (or `null`), and a *memory*, which is a map from locations to objects. Our idea is to abstract concrete states into a new kind of graphs we call ALPs graphs. For instance, given the class Tree with two fields l and r, the state in Fig. 1 is abstracted into the ALPs graph in Fig. 2. All ALPs graphs have at most two levels: nodes in the first level are labeled with one or more variables, while nodes in the second level are unlabeled. First-level nodes may have outgoing and incoming edges labeled with field names, while second-level nodes have no outgoing edges. Note that the data structures pointed by the variables v_0 and v_3 are abstracted in the same way, since we do not consider the whole data structure.

1.0.1 Aliasing and nullness.

ALPs graphs may encode *definite nullness* for variables and fields. A variable is definitively null if it does not appear as a label in the graph, while a field v_0.`f` is null when there is no edge labeled with `f` departing from the v_0 node. For example, v_1.`r` is definitively null according to Fig. 2.

The graph also encodes definite *weak aliasing*: two variables (or two fields) are weak aliased when they point to the same location (possibly `null`). In the ALPs graph, this means they are the same node. For instance, the variables v_3 and v_4 in Fig. 1 are in the same node. Moreover, the fields v_2.`l` and v_2.`r` are abstracted into a single node.

1.0.2 Sharing information.

We are interested in representing *possible sharing* information. We say that two locations share when it is possible to reach from them a common location. Consider the concrete state depicted in Fig. 3. Here v_7.`r` and v_8.`l` are bound to two different data structures which overlap, since v_7.`r.r.r` and v_8.`l.l.l` are bound to the same object: we say that v_7.`r` and v_8.`l` share. In the abstract graph, we represent this

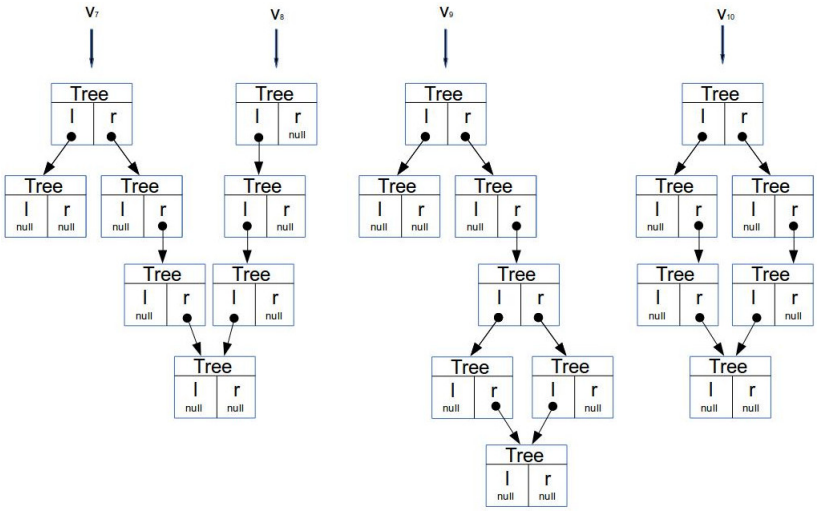


Fig. 3. A concrete state with variables v_7, v_8, v_9, v_{10}.

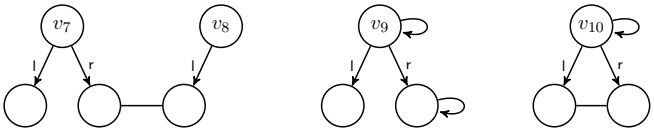


Fig. 4. Abstraction of the concrete state in Fig. 3.

information with an (undirected) edge between the two nodes. For instance, the sharing information in Fig. 3 is captured by the edge connecting v_7.r and v_8.l in Fig. 4. To keep the graph as simple as possible we omit sharing information which can be derived, such as the edge between v_7 and v_8.

1.0.3 Linearity information.
We say that a location is *non linear* when there are two different paths starting from it and reaching a common location. Consider for instance Fig. 3. Starting from v_9.r, we reach the same object by either v_9.r.r.l.r or v_9.r.r.r.l. Therefore we say that v_9.r is non linear. It is easy to note that also v_9 is non linear. In general, whenever a field v.f is non linear, the variable v is non linear too. We represent *possible non linearity* information by means of self loops. For instance, the concrete state for variables v_9 and v_{10} in Fig. 3 is abstracted as in Fig. 4.

1.1 An example program

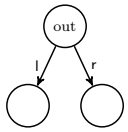
We show the relevance of linearity information with the help of the example program in Figure 5.

The makeTree method builds a complete tree of depth n, whose nodes are all different. Actually, with a bottom up static analysis using ALPs graphs, we can easily infer that, for any input $n \geq 2$, makeTree returns a data structure which may be described by the graph in Fig. 6a, where the label out denotes the return value of a method. Since there are no undirected edges between out.l and out.r, it means that out.l and out.r do not share. Moreover, since there are no self-loops, everything is guaranteed to be linear. In particular, out.l.l and out.l.r do not share.

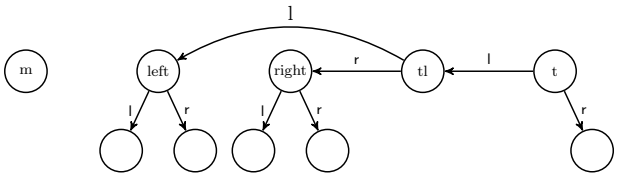
```
Tree makeTree(n:Integer)
 with m:Integer is {
   m = 0;
   if (n = m)
     out = null;
   else {
     out = new Tree;
     m = n-1;
     out.l = makeTree(m);
     out.r = makeTree(m);
 } }
```

```
void useTree()
 with m:Integer, t:Tree, tl: Tree,
      left:Tree,
      right:Tree is {
   m = 10;
   t = makeTree(m);
   tl  = t.l;
   right = tl.r;
   left  = tl.l;
 }
```

Fig. 5. The example program



(a) The ALPs graph returned by the makeTree method.



(b) ALPs graph representing the state at the end of the useTree method. All nodes in the first row are first level nodes, even if there are edges pointing to them.

Fig. 6. Two ALPs graphs for the example program.

The useTree method calls makeTree and extract two subtrees which do not share. In detail, in the useTree method, since we know that `t` is linear, we can infer that `tl` is linear too. Since `tl` is linear, its fields `tl.r` and `tl.l` do not share, and therefore `right` and `left` do not share. Note that linearity is not needed to prove that `t.l` and `t.r` do not share (sharing is enough for this). We need linearity when we want to go deeper and prove that `t.l.l` and `t.l.r` do not share. Linearity of `t` is essential here in proving that `left` and `right` do not share. The heap at the end of the useTree method may be described by the graph in Fig. 6b.

2 Preliminaries

We denote by $[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ the function f where $dom(f) = \{v_1, \ldots, v_n\}$ and $f(v_i) = t_i$ for $i = 1, \ldots, n$. Its *update* is $f[w_1 \mapsto d_1, \ldots, w_m \mapsto d_m]$, where the domain may be enlarged. By $f|_s$ ($f|_{-s}$) we denote the *restriction* of f to $s \subseteq \mathrm{dom}(f)$ (to $\mathrm{dom}(f) \setminus s$). The two components of a *pair* are separated by $\star$. A definition of a pair $S = a \star b$ silently defines the pair selectors $s.a$ and $s.b$.

2.1 The Language

We use the Java-like object-oriented language defined in [15], which is a normalized version of Java with downward casts, and which we extend with upper casts.

2.1.1 Syntax.

Each program has a set of *variables* (or *identifiers*) $\mathcal{V}$ and a finite set of *classes* (or *types*) $\mathcal{K}$ ordered by a *subclass relation* $\leq$ such that $\mathcal{K} \star \leq$ is a poset. The set $\mathcal{V}$ includes the special variables `this`, `res`, `out`. Since we do not allow multiple inheritance, for any class $\kappa \in \mathcal{K}$, the set $\{\kappa' \mid \kappa' \geq \kappa\}$ is a chain.

A *type environment* is a map from a finite set of variables to the associated class.

The set of type environments is

$$\mathit{TypEnv} = \{\tau : \mathcal{V} \to \mathcal{K} \mid \mathrm{dom}(\tau) \text{ is finite}\}.$$

Any class $\kappa \in \mathcal{K}$ defines a type environment $F(\kappa)$ that maps the fields of the class κ (including both the fields defined in κ and those inherited by the superclasses) to their types. For ease of notation, we require that fields cannot be redefined in subclasses. It means that if $\mathtt{f} \in \mathrm{dom}\, F(\kappa)$, and $\kappa' \leq \kappa$, then $\mathtt{f} \in \mathrm{dom}\, F(\kappa')$ and $F(\kappa')(\mathtt{f}) = F(\kappa)(\mathtt{f})$.

Example 2.1 For the example program in Sect. 1.1, $\mathcal{K} = \{\mathtt{Tree}, \mathtt{Integer}\}$ has a flat ordering. Moreover $F(\mathtt{Tree}) = [\mathtt{l} \mapsto \mathtt{Tree}, \mathtt{r} \mapsto \mathtt{Tree}]$ and $F(\mathtt{Integer}) = []$.

Expressions and commands are normalized versions of those of Java. We require that all casts are explicit, so that in any assignment $v := \mathit{exp}$ the types of v and *exp* coincide. The same is required for formal and actual parameters. This is not a limitation since we allow upward and downward casts. The syntax of expressions and commands is:

$$\begin{aligned} \mathit{exp} &::= \mathtt{null}\ \kappa \mid \mathtt{new}\ \kappa \mid v \mid v.\mathtt{f} \mid (\kappa)v \mid v.\mathtt{m}(v_1, \ldots, v_n) \\ \mathit{com} &::= v{:=}\mathit{exp} \mid v.\mathtt{f}{:=}\mathit{exp} \mid \{\mathit{com}; \cdots ; \mathit{com}\} \\ &\quad \mid \mathtt{if}\ v = w\ \mathtt{then}\ \mathit{com}\ \mathtt{else}\ \mathit{com} \mid \mathtt{if}\ v = \mathtt{null}\ \mathtt{then}\ \mathit{com}\ \mathtt{else}\ \mathit{com} \end{aligned}$$

where $\kappa \in \mathcal{K}$ and $v, w, v_1, \ldots, v_n \in \mathcal{V}$ are distinct when they appear in the same clause. Each method $\kappa.\mathtt{m}$ of a class κ is *defined* with a statement like

$$\kappa_0\ \mathtt{m}(w_1{:}\kappa_1, \ldots, w_n{:}\kappa_n)\ \mathtt{with}\ w_{n+1}{:}\kappa_{n+1}, \ldots, w_{n+m}{:}\kappa_{n+m}\ \mathtt{is}\ \mathit{com}$$

where $w_1, \ldots, w_n, w_{n+1}, \ldots, w_{n+m} \in \mathcal{V}$ are distinct and are not `res` nor `this` nor `out`. Their *declared types* are $\kappa_1, \ldots, \kappa_n, \kappa_{n+1}, \ldots, \kappa_{n+m} \in \mathcal{K}$, respectively. The method can also use a variable `out` of type κ_0 which holds its *return value*. We define $\mathit{body}(\kappa.\mathtt{m}) = \mathit{com}$ and $\mathit{returnType}(\kappa.\mathtt{m}) = \kappa_0$.

2.1.2 Semantics.
The semantics of the language is defined by means of *frames*, *objects* and *memories* defined as follows:

$$\begin{aligned} \mathit{Frame}_\tau &= \{\phi \mid \phi \in \mathrm{dom}(\tau) \mapsto \mathit{Loc} \cup \{\mathtt{null}\}\} \\ \mathit{Obj} &= \{\kappa \star \phi \mid \kappa \in \mathcal{K},\ \phi \in \mathit{Frame}_{F(\kappa)}\} \\ \mathit{Memory} &= \{\mu \in \mathit{Loc} \to \mathit{Obj} \mid \mathrm{dom}(\mu) \text{ is finite}\} \end{aligned}$$

where *Loc* is an infinite set of *locations*. A frame binds variables (identifiers) to locations or `null`. A memory binds such locations to objects, which contain a class tag and the frame for their fields. A new object of class κ is $\mathtt{new}(\kappa) = \kappa \star \phi$, with $\phi(v) = \mathtt{null}$ for each $v \in \mathrm{dom}(F(\kappa))$. The set of possible *states* with type environment τ is

$$\Sigma_\tau = \{\phi \star \mu \mid \phi \in \mathit{Frame}_\tau,\ \mu \in \mathit{Memory},\ \phi \star \mu : \tau\}$$

where $\phi \star \mu : \tau$ means that $\phi \star \mu$ is well typed w.r.t the type environment τ.

Example 2.2 Let $\tau = \{v_7 \mapsto \texttt{Tree}, v_8 \mapsto \texttt{Tree}\}$ and consider the state $\phi\star\mu$ depicted in Figure 3. We have that $\phi = \{v_7 \mapsto l_0, v_8 \mapsto l_1\}$ and

$$\begin{aligned}\mu = \{&l_0 \mapsto \texttt{Tree} \star \{\texttt{l} \mapsto l_2, \texttt{r} \mapsto l_3\}, l_1 \mapsto \texttt{Tree} \star \{\texttt{l} \mapsto l_4, \texttt{r} \mapsto \texttt{null}\},\\ &l_2 \mapsto \texttt{Tree} \star \{\texttt{l} \mapsto \texttt{null}, \texttt{r} \mapsto \texttt{null}\}, l_3 \mapsto \texttt{Tree} \star \{\texttt{l} \mapsto \texttt{null}, \texttt{r} \mapsto l_5\},\\ &l_4 \mapsto \texttt{Tree} \star \{\texttt{l} \mapsto l_6, \texttt{r} \mapsto \texttt{null}\}, l_5 \mapsto \texttt{Tree} \star \{\texttt{l} \mapsto \texttt{null}, \texttt{r} \mapsto l_7\},\\ &l_6 \mapsto \texttt{Tree} \star \{\texttt{l} \mapsto l_7, \texttt{r} \mapsto \texttt{null}\}, l_7 \mapsto \texttt{Tree} \star \{\texttt{l} \mapsto \texttt{null}, \texttt{r} \mapsto \texttt{null}\}\} \ .\end{aligned}$$

The denotation for an expression is a partial map $\mathcal{E}^I_\tau[\![\_]\!] : \mathit{exp} \mapsto (\Sigma_\tau \to \Sigma_{\tau+\mathit{exp}})$ from an initial to a final state, containing a distinguished variable `res` holding the value of the expression, where $\tau + \mathit{exp} = \tau[\texttt{res} \mapsto \mathit{type}_\tau(\mathit{exp})]$ and $\mathit{type}_\tau(\mathit{exp})$ is the static type of *exp*. The denotation of a command is a partial map from an initial to a final state: $\mathcal{C}^I_\tau[\![\_]\!] : \mathit{com} \mapsto (\Sigma_\tau \to \Sigma_\tau)$.

Each method $\kappa.\texttt{m}$ is *denoted* by a partial function from input to output states and an *interpretation* I maps methods to partial functions on states, such that $I(\kappa.\texttt{m}) : \Sigma_{\mathit{input}(\kappa.\texttt{m})} \to \Sigma_{\mathit{scope}(\kappa.\texttt{m})}$, with the type environments:

$$\begin{aligned}\mathit{input}(\kappa.\texttt{m}) &= [\texttt{this} \mapsto \kappa, w_1 \mapsto \kappa_1, \ldots, w_n \mapsto \kappa_n]\\ \mathit{scope}(\kappa.\texttt{m}) &= \mathit{input}(\kappa.\texttt{m}) \cup [\texttt{out} \mapsto \kappa_0, w'_1 \mapsto \kappa_1, \ldots, w'_n \mapsto \kappa_n,\\ &\qquad w_{n+1} \mapsto \kappa_{n+1}, \ldots, w_{n+m} \mapsto \kappa_{n+m}]\end{aligned}$$

where $w'_1, \ldots, w'_n$ are fresh variables used to keep track of the actual parameters. Each w'_i is assigned to the same value of the corresponding w_i at the beginning of the method execution, and it is never changed later.

Example 2.3 Consider the method `makeTree` in Section 1.1. We have that

$$\begin{aligned}\mathit{input}(\texttt{Tree.makeTree}) =&[\texttt{this} \mapsto \texttt{Tree}, n \mapsto \texttt{Integer}]\\ \mathit{scope}(\texttt{Tree.makeTree}) =&\mathit{input}(\texttt{Tree.makeTree}) \cup\\ &[\texttt{out} \mapsto \texttt{Tree}, n' \mapsto \texttt{Integer}, m \mapsto \texttt{Integer}] \ .\end{aligned}$$

3 Reachability, sharing, linearity and aliasing

We formalize here the concepts of reachability, sharing, linearity and aliasing for objects. In a later section we will use these concepts to introduce the new abstract domain ALPs. The following definition will simplify notation later.

Definition 3.1 (Fields of locations) *Given $\sigma = \phi \star \mu \in \Sigma_\tau$, $l \in \mathrm{dom}(\mu)$, $\texttt{f}$ an identifier and $\bar{\texttt{f}} = \texttt{f}_1, \ldots, \texttt{f}_n$ a possibly empty sequence of identifiers, when they exist we write:*

- *$l.\texttt{f}$ for $\mu(l).\phi(\texttt{f})$, which is the location reachable from l through the field $\texttt{f}$;*
- *$l.\bar{\texttt{f}}$ for $l.\texttt{f}_1.\ldots.\texttt{f}_n$; if $\bar{\texttt{f}}$ is empty, $l.\bar{\texttt{f}} = l$.*

Now we introduce some notation to treat variables and their fields as uniformly as possible.

Definition 3.2 (Qualified fields and identifiers) *Given a type environment τ, we call* qualified field *a pair $v.\mathtt{f}$ where $v \in \mathrm{dom}(\tau)$ and $\mathtt{f} \in \mathrm{dom}(F(\tau(v)))$ and we call* qualified identifier *either a variable in $\mathrm{dom}(\tau)$ or a qualified field. We denote by Q_τ and I_τ the set of qualified fields and identifiers respectively.*

It is worth noting that we only consider fields that are in the declared type of the variables, and we do not consider further fields that are in the actual type. This choice, although may decrease the precision of the analysis, simplifies a lot the correspondence between abstract and concrete semantics and may increase the speed of the analysis.

Example 3.3 In Example 2.2, the qualified fields are $Q_\tau = \{v_7.\mathtt{l}, v_7.\mathtt{r}, v_8.\mathtt{l}, v_8.\mathtt{r}\}$ and the qualified identifiers are $I_\tau = \{v_7, v_8\} \cup Q_\tau$.

Definition 3.4 (Notations for qualified fields) *If $\sigma = \phi \star \mu \in \Sigma_\tau$ and $v.\mathtt{f} \in Q_\tau$, for uniformity of notation with variables we define:*

- $\tau(v.\mathtt{f}) = F(\tau(v))(\mathtt{f})$*, i.e., the declared type of the field* `f` *for the variable v;*
- $\phi(v.\mathtt{f}) = \mathtt{null}$ *if* $\phi(v) = \mathtt{null}$*,* $\phi(v.\mathtt{f}) = \phi(v).\mathtt{f}$ *otherwise, which is the location pointed to by the field* `f` *in the variable v.*

Definition 3.5 (Sharing, linearity and aliasing) *Let $\sigma \in \Sigma_\tau$ and $i_1, i_2 \in I_\tau$. We say that:*

- *i_1 and i_2* share *in σ when $\phi(i_1) \neq \mathtt{null} \neq \phi(i_2)$ and there are $\bar{\mathtt{f}}_1$, $\bar{\mathtt{f}}_2$ such that $\phi(i_1).\bar{\mathtt{f}}_1 = \phi(i_2).\bar{\mathtt{f}}_2 \neq \mathtt{null}$;*
- *i_1 is* non-linear *in σ when $\phi(i_1) \neq \mathtt{null}$ and there are $\bar{\mathtt{f}}_1 \neq \bar{\mathtt{f}}_2$ such that $\phi(i_1).\bar{\mathtt{f}}_1 = \phi(i_1).\bar{\mathtt{f}}_2 \neq \mathtt{null}$; otherwise, i_1 is said to be* linear*.*
- *i_1 and i_2 are* (weakly) aliased *in σ when $\phi(i_1) = \phi(i_2)$.*

Example 3.6 Following Example 2.2, the field $v_7.\mathtt{r}$ shares with $v_8.\mathtt{l}$. As a consequence, v_7 shares with $v_8.\mathtt{l}$ and v_8. The fields $v_7.\mathtt{l}$ and $v_8.\mathtt{r}$ share only with themselves and the respective parents. All the identifiers are linear. In the example in Fig. 3, we have that $v_9.\mathtt{r}$ and v_{10} are not linear.

A qualified identifier $i \in I_\tau$ shares with itself if and only if it is not null. Moreover, each $i \in I_\tau$ such that $\phi(i) = \mathtt{null}$ is linear and does not share with any other identifier.

It must be observed that two qualified identifiers might *never* be able to share if their static types do not let them be bound to overlapping data structures. Analogously, certain qualified identifiers are forced to be linear.

Example 3.7 In the example in Section 1.1, we have that a `Tree` is not an `Integer`, an `Integer` is not a `Tree` and they do not have any field which can share. Therefore, any identifier of type `Tree` can never share with any identifier of type `Integer`. Moreover, any identifier of type `Integer` may only be linear.

We denote by *NL* the set of classes whose instances may be non-linear and by *SH* the set of pair of classes which may share. Both *NL* and *SH* may be computed by typing information only.

4 The new abstract domain

In this section we use the concepts of sharing, linearity and aliasing introduced before to define a new abstract domain, called ALPs (**A**liasing **L**inearity **P**air **s**haring), for the analysis of Java-like programs.

4.1 Aliasing graphs

We start by defining a basic domain encoding definite aliasing and definite nullness.

Definition 4.1 (Pre-Aliasing Graphs) *A* pre-aliasing graph *over the type environment τ is a directed graph $G = N \star E \star \ell$ such that:*

- *N is the set of* nodes*;*
- *$E \subseteq N \times \mathcal{V} \times N$ is the set of* directed edges*, each labeled by an identifier;*
- *$\ell : \mathrm{dom}(\tau) \twoheadrightarrow N$ is a partial map from variables to nodes;*

with the additional condition that

- *$\forall n \in N, \forall \mathtt{f} \in \mathcal{V}$, there is at most an outgoing edge from n labeled by $\mathtt{f}$ and*
- *$\forall n \in N, \forall \mathtt{f} \in \mathcal{V}$, if $\langle n, \mathtt{f}, n' \rangle \in E$ then there exists $v \in \mathrm{dom}(\tau)$ such that $\ell(v) = n$ and $v.\mathtt{f} \in Q_\tau$.*

We write $n_1 \xrightarrow{\mathtt{f}} n_2$ instead of $\langle n_1, \mathtt{f}, n_2 \rangle$. When it is clear from the context, we denote $G.N$, $G.E$ and $G.\ell$ just by N, E and ℓ. Moreover, we denote $G_i.N$, $G_i.E$ and $G_i.\ell$ by N_i, E_i, ℓ_i, and similarly for other typographical variants of G.

Definition 4.2 (Extension of ℓ) *Given a pre-aliasing graph $G = N \star E \star \ell$, we extend ℓ on qualified fields $v.\mathtt{f} \in Q_\tau$ by*

$$\ell(v.\mathtt{f}) = \begin{cases} n & \text{if } \ell(v) \neq \bot \wedge \ell(v) \xrightarrow{\mathtt{f}} n \in E \\ \bot & \text{otherwise} \end{cases}$$

The idea of a pre-aliasing graph is that, given an identifier $i \in I_\tau$, $\ell(i) = \bot$ means i is definitively `null`, while $\ell(i) = \ell(j)$ means that i and j are either both `null` or aliased. This suggest to define the following preorder.

Definition 4.3 (Preordering on pre-aliasing graphs) *Given $G_1, G_2 \in \mathcal{G}_\tau$, we say $G_1 \preceq G_2$ iff*

- *for each $i, i' \in I_\tau$, $\ell_2(i) = \ell_2(i') \Rightarrow \ell_1(i) = \ell_1(i')$;*
- *for each $i \in I_\tau$, $\ell_2(i) = \bot \Rightarrow \ell_1(i) = \bot$.*

Note that, given their intended meaning, some pre-aliasing graphs contain redundant information. For example, nodes which are not labeled by any qualified identifiers may be removed. On the converse, two identifiers i_1, i_2 of incomparable types may be (weak) aliased only if they are both `null`. We therefore restrict our attention to the pre-aliasing graphs which present some additional regularity conditions.

Definition 4.4 (Aliasing graph) *An aliasing graph is a pre-aliasing graph G such that, for all $n \in N$, $\{\tau(i) \mid i \in I_\tau \wedge \ell(i) = n\}$ is a non-empty chain. We denote by $\mathcal{G}_\tau$*

the set of aliasing graphs over the type environment τ, by $\tau_G(n) = \bigwedge\{\tau(i) \mid i \in I_\tau \wedge \ell(i) = n\}$ the type of the node n and by $\psi_G(n) = \bigwedge\{\tau(w) \mid w \in \mathrm{dom}(\tau) \wedge \ell(w) = n\}$ the type that may be inferred by variables only.

Given a concrete state $\sigma \in \Sigma_\tau$, we may abstract it into an aliasing graph which conveys the relevant information.

Definition 4.5 *Given $\sigma = \phi \star \mu \in \Sigma_\tau$, we define the abstraction of σ as an aliasing graph $\alpha_a(\sigma) = G \in \mathcal{G}_\tau$ where*

- $N = \{l \in Loc \mid \exists i \in I_\tau.\phi(i) = l\}$;
- *for each $v \in \mathrm{dom}(\tau)$, $\ell(v) = \phi(v)$ if $\phi(v) \neq \mathtt{null}$, $\ell(v) = \bot$ otherwise;*
- $l \xrightarrow{\mathtt{f}} l' \in E$ *iff there exists $v \in \mathrm{dom}(\tau)$ such that $\ell(v) = l$, $v.\mathtt{f} \in Q_\tau$ and $l.\mathtt{f} = l' \in N$.*

We say that $G \in \mathcal{G}_\tau$ is a correct abstraction of $\sigma \in \Sigma_\tau$ iff $\alpha_a(\sigma) \preceq G$. Note that if $\alpha_a(\sigma) \preceq G$ and $\ell(i) = \bot$, then $\phi(i) = \mathtt{null}$, hence ℓ may actually be used to represent definite nullness. Moreover, if $\ell(i_1) = \ell(i_2)$, then either $\phi(i_1) = \phi(i_2) \in Loc$ or $\phi(i_1) = \phi(i_2) = \mathtt{null}$. Hence ℓ actually encodes definite weak aliasing between variables.

4.2 ALPs *graphs*

Aliasing graphs are a very concrete representation of the part of the program state which is reachable from variables through a single field access. Pair-sharing and linearity, instead, summarize global properties of the state. We want to add possible sharing and possible non-linearity information to an aliasing graph.

Definition 4.6 (Pre-ALPs graph) *A pre-*ALPs *graph $\mathbb{G} = G \star sh \star nl$ is a pre-aliasing graph G with a set $sh \subseteq \{\{n, m\} \mid n, m \in N\}$ and a set $nl \subseteq N$.*

When it is clear from the context, we denote $\mathbb{G}.G, \mathbb{G}.sh, \mathbb{G}.nl$ by G, sh, nl and $\mathbb{G}_i.G, \mathbb{G}_i.sh, \mathbb{G}_i.nl$ by G_i, sh_i, nl_i. Similarly for other variants of $\mathbb{G}$.

The set sh in a pre-ALPs graph encodes possible pair sharing, while nl encodes possible non-linearity. In particular, two identifiers $i, j \in I_\tau$ may share when $\{\ell(i), \ell(j)\} \in sh$, while i may be non-linear when $\ell(i) \in nl$. This suggests to extend the preorder on aliasing graphs to ALPs graphs as follows:

Proposition 4.7 (Preordering on pre-ALPs graphs) *Pre-*ALPs *graphs are pre-ordered by*

$$\begin{aligned}\mathbb{G}_1 \preceq \mathbb{G}_2 \iff & G_1 \preceq G_2 \text{ and } \forall i \in I_\tau.\ \ell_1(i) \in nl_1 \Rightarrow \ell_2(i) \in nl_2 \text{ and} \\ & \forall i, j \in I_\tau.\ \{\ell_1(i), \ell_1(j)\} \in sh_1 \Rightarrow \{\ell_2(i), \ell_2(j)\} \in sh_2\ .\end{aligned}$$

Not all the pre-ALPs graphs make sense, due to the way aliasing, non-linearity and sharing interact. In particular, some sharing and non-linearity information can be derived by other information. For example, if n is a node in G, then $\{n\}$ should be in sh, otherwise any identifier i s.t. $\ell(i) = n$ is forced to be $\mathtt{null}$ and G could be simplified by removing the node n. Other non-linearity or sharing information is redundant, since it cannot happen in practice due to the class hierarchy under consideration: pairs $\{n, m\} \in sh$ such that classes $\tau_G(n)$ and $\tau_G(m)$ cannot share, or

variables $n \in nl$ such that $\tau_G(n) \notin NL$. We want to restrict our attention to those pre-ALPs graphs which explicitly show all implied and redundant information.

Definition 4.8 (ALPs graph) *An* ALPs *graph* $\mathbb{G}$ *is a pre-*ALPs *graph where*

- G *is an aliasing graph;*
- $\{\{n\} \mid n \in N\} \subseteq sh$*;*
- *if there is a non-empty loop in* G *involving* n*, then* $n \in nl$*;*
- *if* $\{n, m\} \in sh$ *then* $(\tau_G(n), \tau_G(m)) \in SH$*;*
- *if* $n \in nl$ *then* $\tau_G(n) \in NL$*;*
- $\mathrm{cl}_G(sh \star nl) = sh \star nl$*.*

Here $\mathrm{cl}_G(sh, nl)$ *is the smallest pair* $sh' \star nl'$*, under the component-wise ordering, such that*

- $\{n, m\} \in sh' \wedge n' \xrightarrow{\mathtt{f}} n \Rightarrow \{n', m\} \in sh'$*;*
- $n \xrightarrow{\mathtt{f}_1} m_1$*,* $n \xrightarrow{\mathtt{f}_2} m_2$*,* $\mathtt{f}_1 \neq \mathtt{f}_2$*,* $\{m_1, m_2\} \in sh' \Rightarrow n \in nl'$*;*
- $n \in nl' \wedge n' \xrightarrow{\mathtt{f}} n \Rightarrow n' \in nl'$*.*

We denote by ALPs_τ *the set of* ALPs *graphs over the type environment* τ*.*

Given a concrete state $\sigma \in \Sigma_\tau$, we may abstract it into an aliasing graph which conveys the relevant information.

Definition 4.9 (Abstraction map on ALPs graph) *Given* $\sigma = \phi \star \mu \in \Sigma_\tau$*, we define the abstraction of* σ *as* $\alpha(\sigma) = \alpha_a(\sigma) \star sh \star nl$ *where*

$$sh = \{\{l_1, l_2\} \subseteq N \mid l_1 \text{ and } l_2 \text{ share in } \sigma\} \ ,$$
$$nl = \{l \in N \mid l \text{ is not linear in } \sigma\} \ .$$

We say $\mathbb{G}$ *is a correct abstraction of* σ *when* $\alpha(\sigma) \preceq \mathbb{G}$*.*

Given $\sigma = \phi \star \mu$ and $\alpha(\sigma) \preceq \mathbb{G}$, if $i_1, i_2 \in I_\tau$ share in σ then $\{\ell(i_1), \ell(i_2)\} \in sh$. Moreover, if $i \in I_\tau$ is non-linear in σ, then $\ell(i) \in nl$. Hence $\mathbb{G}$ actually encodes possible sharing and non-linearity among variables.

Proposition 4.10 *The preordered set of* ALPs *graphs has least element* $\bot$*, greatest element* $\top$*, lub* $\curlyvee$*, and glb* $\curlywedge$*.*

We may define a concretization map $\gamma : \mathsf{ALPs}_\tau \to \wp(\Sigma_\tau)$ which maps aliasing graphs to the set of concrete states they represent as $\gamma(\mathbb{G}) = \{\sigma \in \Sigma_\tau \mid \alpha(\sigma) \preceq \mathbb{G}\}$. If we lift the map α in Def. 4.9 to an additive map $\alpha : \wp(\Sigma_\tau) \to \mathsf{ALPs}_\tau$ as $\alpha(S) = \curlyvee_{\sigma \in S} \alpha(\sigma)$, then α and γ form a Galois connection.

5 An Abstract Semantics on ALPs.

We present the abstract semantics on the domain ALPs_τ. We provide a correct abstract counterpart for each concrete operator in the standard semantics. The abstract counterpart of an interpretation is an ALPs interpretation, defined as follows.

Definition 5.1 *An* ALPs interpretation I *maps methods to total functions such that*

$$\mathcal{SE}^I_\tau[\![\texttt{null}\ \kappa]\!](\mathbb{G}) = \mathbb{G}$$
$$\mathcal{SE}^I_\tau[\![\texttt{new}\ \kappa]\!](\mathbb{G}) = N \cup \{n_{new}\} \star E \star \ell[\texttt{res} \mapsto n_{new}] \star sh \cup \{\{n_{new}\}\} \star nl$$
$$\mathcal{SE}^I_\tau[\![v]\!](\mathbb{G}) = N \star E \star \ell[\texttt{res} \mapsto \ell(v)] \star sh \star nl$$
$$\mathcal{SE}^I_\tau[\![(\kappa)v]\!](\mathbb{G}) = \begin{cases} \bot & \text{if } \tau(\ell(v)) \cup \{\kappa\} \text{ is not a chain} \\ \mathbb{G} & \text{if } \ell(v) = \bot \\ add(\mathbb{G}, \ell(v), \kappa) & \text{otherwise} \end{cases}$$
$$\mathcal{SE}^I_\tau[\![v.\texttt{f}]\!](\mathbb{G}) = \begin{cases} \bot & \text{if } \ell(v) = \bot \\ \mathbb{G} & \text{if } \ell(v.\texttt{f}) = \bot \\ add(\mathbb{G}, \ell(v.\texttt{f}), \tau(v.\texttt{f})) & \text{otherwise} \end{cases}$$

Fig. 7. The ALPs interpretation for expressions.

$I(\kappa.\texttt{m}) : \mathsf{ALPs}_{input(\kappa.\texttt{m})} \mapsto \mathsf{ALPs}_{scope(\kappa.\texttt{m})}$ *for each method* $\kappa.\texttt{m}$.

5.1 Abstract Denotation for the Expressions

Abstract denotations for expressions and commands are given compositionally on their syntax.

Definition 5.2 *Let τ describe the variables in scope and I be a* ALPs *interpretation. Figure 7 defines the* ALPs denotation *for expression (except method calls)* $\mathcal{SE}^I_\tau[\![\_]\!] : exp \mapsto (\mathsf{ALPs}_\tau \mapsto \mathsf{ALPs}_{\tau+exp})$.

We briefly explain the behavior of the abstract semantic operators with respect to the corresponding concrete ones. The concrete semantics of `null` κ stores `null` in the variable `res`. Therefore, in the abstract semantics, we only need to add the new variable `res` in the type environment, without modifying the abstract state.

The concrete semantics of `new` κ stores in `res` a reference to a new object o, whose fields are `null`. The other variables do not change. Since o is only reachable from `res`, variable `res` shares with itself only and is clearly linear. Therefore, we only need to add a new node labeled with `res`, without affecting sharing nor non-linearity information.

The concrete semantics of v simply makes `res` an alias for v. Since the type of v and `res` coincides, we only need to add the variable v to the same node of `res`. The other variables are unchanged.

When it is defined, the cast $(\kappa)v$ stores in `res` the value of v. When $\ell(v)$ is not null, we use an auxiliary operator $add(\mathbb{G}, n, \kappa)$ which adds the label `res` to the node $\ell(v)$, and possibly add new nodes for the fields of `res` which are not fields of v. In this case we can exploit the notion of linearity. In fact, when v is linear, we know that fields of `res` cannot share with each other and are linear.

The concrete semantics of $v.\texttt{f}$ stores in `res` the value of the field `f` of v, provided v is not `null`. When $v.\texttt{f}$ is not `null`, this essentially amounts to the same procedure of the previous case.

5.2 Abstract Denotation for the Commands

Definition 5.3 *Let τ describe the variables in scope, I be an* ALPs *interpretation. Figure 8 shows the* ALPs denotation for commands $\mathcal{SC}^I_\tau[\![\_]\!] : com \mapsto (\mathsf{ALPs}_\tau \mapsto \mathsf{ALPs}_\tau)$.

$$\mathcal{SC}^I_\tau[\![v{:=}exp]\!](\mathbb{G}) = \mathrm{prune}((N' \star E' \star \ell'[v \mapsto \ell'(\mathtt{res}), \mathtt{res} \mapsto \bot]) \star sh' \star nl')$$

$$\mathcal{SC}^I_\tau[\![v.\mathtt{f}{:=}exp]\!](\mathbb{G}) = \begin{cases} \bot & \text{if } \ell'(v) = \bot \\ \mathrm{cl}(\mathrm{prune}((N' \cup N_{new} \star E' \setminus E_{del} \cup E_{new} \star \ell'[\mathtt{res} \mapsto \bot]) \star \\ \quad sh' \cup sh_{new} \star nl' \cup \{n_{\ell'(x)} \mid n_{\ell'(x)} \in N_{new}, \ell'(x.\mathtt{f}) \in nl'\})) & \text{if } \ell'(v) \neq \bot \text{ and } \ell'(\mathtt{res}) = \bot \\ \mathrm{cl}(\mathrm{prune}((N' \cup N_{new} \star E' \setminus E_{del} \cup E'_{new} \star \ell'[\mathtt{res} \mapsto \bot]) \star \\ \quad sh' \cup sh'_{new} \star nl' \cup nl'_{new} \cup \{n_{\ell'(x)} \mid n_{\ell'(x)} \in N_{new}, \ell'(x.\mathtt{f}) \in nl'\})) & \text{otherwise} \end{cases}$$

$$\mathcal{SC}^I_\tau \left[\begin{array}{l} \mathtt{if}\ v = \mathtt{null} \\ \mathtt{then}\ com_1\ \mathtt{else}\ com_2 \end{array} \right] (\mathbb{G}) = \begin{cases} \mathcal{SC}^I_\tau[\![com_1]\!](\mathbb{G}) & \text{if } \ell(v) = \bot \\ \mathcal{SC}^I_\tau[\![com_1]\!](\mathbb{G}_{|v=\mathtt{null}}) \curlyvee \mathcal{SC}^I_\tau[\![com_2]\!](\mathbb{G}) & \text{otherwise} \end{cases}$$

$$\mathcal{SC}^I_\tau \left[\begin{array}{l} \mathtt{if}\ v = w \\ \mathtt{then}\ com_1\ \mathtt{else}\ com_2 \end{array} \right] (\mathbb{G}) = \begin{cases} \mathcal{SC}^I_\tau[\![com_1]\!](\mathbb{G}) & \text{if } \ell(v) = \ell(w) \\ \mathcal{SC}^I_\tau[\![com_1]\!](\mathbb{G}_{|v=w}) \curlyvee \mathcal{SC}^I_\tau[\![com_2]\!](\mathbb{G}) & \text{otherwise} \end{cases}$$

$$\mathcal{SC}^I_\tau[\![\{com_1; \ldots; com_p\}]\!] = (\lambda s \in ALPs_\tau.s) \circ \mathcal{SC}^I_\tau[\![com_p]\!] \circ \cdots \circ \mathcal{SC}^I_\tau[\![com_1]\!]$$

where $\mathcal{SE}^I_\tau[\![exp]\!](\mathbb{G}) = \mathbb{G}'$,

$V_{comp} = \{x \in \mathrm{dom}(\tau) \mid \ell'(x) \neq \ell'(v), \{x, v\} \in sh', \{\tau_{G'}(\ell'(x)), \tau_{G'}(\ell'(v)\} \text{ is a chain}\}$

$N_{new} = \{n_{\ell'(x)} \mid x \in V_{comp},\ \mathtt{f} \in \mathrm{dom}(F(\psi_{G'}(\ell'(x)))),\ \ell'(x.\mathtt{f}) \neq \ell'(\mathtt{res})\}$ new distinct nodes

$E_{del} = \{\ell'(v) \xrightarrow{\mathtt{f}} \ell'(v.\mathtt{f})\} \cup \{\ell'(x) \xrightarrow{\mathtt{f}} \ell'(x.\mathtt{f}) \in E' \mid x \in V_{comp}\}$

$E_{new} = \{\ell'(x) \xrightarrow{\mathtt{f}} n_{\ell'(x)} \mid x \in V_{comp},\ n_{\ell'(x)} \in N_{new}\}$

$sh_{new} = \{\{n_{\ell'(x)}, n'\} \mid n_{\ell'(x)} \in N_{new}, \{\ell'(x.\mathtt{f}), n'\} \in sh'\} \cup$
$\{\{n_{\ell'(x)}, n_{\ell'(y)}\} \mid n_{\ell'(x)}, n_{\ell'(y)} \in N_{new}, \{\ell'(x.\mathtt{f}), \ell'(y.\mathtt{f})\} \in sh\}$

$E'_{new} = E_{new} \cup \{\ell'(v) \xrightarrow{\mathtt{f}} \ell'(\mathtt{res})\}$

$sh'_{new} = \{\{n, n'\} \mid \{\ell'(v), n\} \in sh' \text{ and } \{\ell'(\mathtt{res}), n'\} \in sh'\} \cup$
$\{\{n_{\ell'(x)}, n'\} \mid n_{\ell'(x)} \in N_{new},\ \{\ell'(\mathtt{res}), n'\} \in sh'\} \cup$
$\{\{n_{\ell'(x)}, n'\} \mid n_{\ell'(x)} \in N_{new}, \{\ell'(x.\mathtt{f}), n'\} \in sh'\} \cup \{\{n_{\ell'(x)}, n_{\ell'(y)}\} \mid n_{\ell'(x)}, n_{\ell'(y)} \in N_{new}\}$

$nl'_{new} = \begin{cases} \{n \in N' \mid \{n, \ell'(v)\} \in sh'\} \cup \{n_{\ell'(x)} \mid n_{\ell'(x)} \in N_{new}\} & \text{if } \{\mathtt{res}, v\} \in sh' \text{ or } \mathtt{res} \in nl' \\ \{n \in N' \mid \{n, \ell'(v)\} \in sh', \{\mathtt{res}, n\} \in sh'\} & \text{otherwise} \end{cases}$

Fig. 8. The ALPs interpretation for commands.

The concrete semantics of $v{:=}exp$ evaluates exp and stores its result into v. Thus, the final abstract state is obtained by first computing $\mathcal{SE}^I_\tau[\![exp]\!]$ and then renaming `res` into v. Some of the nodes may become unlabeled and must be removed. This is accomplished by the auxiliary operation prune which removes unnecessary information, in particular unlabeled nodes and fields which are not in the declared type of the variables.

The evaluation of $v.\mathtt{f} := exp$ is the most complex operation of the abstract semantics, since we must take into account that v might be aliased to many different nodes. The candidates are those variables, denoted by V_{comp}, which share with $\ell(v)$ and have compatible types. For each node labeled by a variable in V_{comp}, we add a new fresh node in N_{new} pointed by an edge (labeled by the field `f`) in E_{new}. Finally, all possible sharing and non-linearity are added. A slightly different treatment is devoted to the special case when the result of the expression is definitively `null`. Note that a similar situation also arise in [14].

To determine a correct approximation of the conditional "`if` $v = $ `null`" we check whether $\ell(v) = \bot$. If this is the case, then we known that v is null and we evaluate com_1. Otherwise, we evaluate both branches and compute the lub. When evaluating the first branch, we may improve precision by using the auxiliary operator $\mathbb{G}_{|v=\mathtt{null}}$ which returns a correct approximation of the program states $\{\phi \star \mu \mid \alpha(\phi \star \mu) \preceq \mathbb{G} \wedge \phi(v) = \mathtt{null}\}$, i.e., the states correctly approximated by $\mathbb{G}$ where v is `null`. Similarly for the conditional "`if` $v = w$" where we use another auxiliary operator $\mathbb{G}_{|v=w}$ which returns a correct approximation of the set of program states $\{\phi \star \mu \mid$

$$\mathcal{SE}^I_\tau[\![v.\mathtt{m}(v_1,\ldots,v_n)]\!](\mathbb{G}) = \begin{cases} \bot & \text{if } \ell(v)=\bot \\ \curlyvee\{\text{match}_{v.\mathtt{m}}(\mathbb{G}, I(\kappa.\mathtt{m})(\mathbb{G}')) \mid \kappa \leq \tau_G(\ell(v))\} & \text{otherwise} \end{cases}$$

where $\ell^{input} = [\mathtt{this} \mapsto \ell(v), w_1 \mapsto \ell(v_1), \ldots, w_n \mapsto \ell(v_n)\}]$ and $\mathbb{G}' = \text{prune}(N \star E \star \ell^{input} \star sh \star nl)$.

Fig. 9. The ALPs interpretation for methods.

$\alpha(\phi\star\mu) \preceq \mathbb{G}\wedge\phi(v) = \phi(w)\}$. The composition of commands is denoted by functional composition over ALPs, where the identity map $\lambda s \in ALPs_\tau.s$ is needed when $p = 0$.

5.3 Abstract semantics of method call

When a method $v.\mathtt{m}$ is called, the class of v is inspected and the correct overloaded method for `m` is selected. The abstract domain contains only a partial information on the run-time class of v, since we only know that the class of v must be less than the class of any variable aliased with v, namely less than $\tau_G(\ell(v))$. We exploit this information in computing the abstract semantics of a method call in Figure 9. In practice, we conservatively assume that every method `m` in any subclass of $\tau_G(\ell(v))$ may be called. Note that methods defined only in superclasses of κ are already considered in κ.

When exiting from a method call, we need to rename `out` into `res` since, from the point of view of the caller, the returned value of the callee (`out`) is the value of the method call expression (`res`). We will use an auxiliary operation match which, given an initial and final state, updates the initial state trying to guess possible matching of variables in the abstract states.

Theorem 5.4 *The abstract denotations in Figures 7, 8 and 9 are correct. Moreover, the transformer on* ALPs *interpretations which, given a* ALPs *interpretation I, returns a new* ALPs *interpretation I' such that*

$$I'(\kappa.\mathtt{m}) = \mathcal{SC}^I_{scope(\kappa.\mathtt{m})}[\![body(\kappa.\mathtt{m})]\!] \circ (\lambda\mathbb{G} \in \mathsf{ALPs}_{input(\kappa.\mathtt{m})}.N \star E \star \ell' \star sh \star nl)$$

is correct, where $\ell' = \ell[w'_1 \mapsto \ell(w_1), \ldots, w'_n \mapsto \ell(w_n)]$.

The abstract denotational semantics is the least fixpoint of this transformer, and it is correct w.r.t. the concrete denotational semantics used in [15].

Example 5.5 Consider the method `Tree.makeTree` in Sect. 1.1, where

$$scope(\mathtt{Tree.makeTree}) = [\ \mathtt{this} \mapsto \mathtt{Tree}, n \mapsto \mathtt{Integer}, n' \mapsto \mathtt{Integer}, \\ m \mapsto \mathtt{Integer}, \mathtt{out} \mapsto \mathtt{Tree}]\ .$$

According to Theorem 5.4, we can compute a new ALPs interpretation from the least informative ALPs interpretation $I_\bot(\mathtt{Tree.makeTree}) = \lambda\mathbb{G}.\bot_{scope(\mathtt{Tree.makeTree})}$:

$$I^1(\mathtt{Tree.makeTree})(\mathbb{G}) = N \cup \{n_m, n_{out}\} \star E \star \\ \ell[n' \mapsto \ell(n), out \mapsto n_{out}, m \mapsto n_m] \star sh \cup sh' \star nl$$

where $sh' = \emptyset$ if $\ell(n) = \bot$, and $sh' = \{\{n_m, \ell(n)\}\}$ otherwise. Now, starting from

(a) First iteration. (b) Second iteration.

Fig. 10. ALPs interpretations for the makeTree method.

$I^1(\texttt{Tree.makeTree})$, we can compute a new interpretation as follows:

$$
\begin{aligned}
I^2\ &(\texttt{Tree.makeTree})(\mathbb{G}) = N \cup \{n_m, n_{out}, n_{out.l}, n_{out.r}, \} \star E \star \\
&\ell[n' \mapsto \ell(n), out \mapsto n_{out}, out.l \mapsto n_{out.l}, out.r \mapsto n_{out.r}, m \mapsto n_m] \\
&\star sh \cup sh' \cup \{\{n_{out}, n_{out.l}\}, \{n_{out}, n_{out.r}\}\} \star nl
\end{aligned}
$$

which is the least fixpoint. Relatively to the case $\ell(n) \neq \texttt{null}$, the abstract states $I^1(\texttt{Tree.makeTree})(\mathbb{G})$ and $I^2(\texttt{Tree.makeTree})(\mathbb{G})$ are depicted in Figure 10.

6 Conclusions

We propose a new abstract domain ALPs which combines aliasing, linearity and sharing analysis for an object-oriented language, and provide all the necessary abstract operations. We show in Sec. 1.1 a simple example where linearity plays a fundamental role in proving that the two subtrees do not share.

The property of sharing for object-oriented languages has been studied in a few work, while it is deeply studied in the context of logic programming, where it is commonly combined with linearity analysis. At the best of our knowledge, this is the first attempt to combine sharing with linearity for Java programs. We also plan to implement ALPs as an abstract domain for the Jandom static analyzer [1].

In [15] the authors propose a simple domain of (pair-)sharing. In [13] the authors extend this domain proposing a combined analysis of (set) sharing, nullness and classes. The main differences of our paper w.r.t. these proposals are:

- the ALPs abstract domain encodes linearity and aliasing information, in addition to pair-sharing;
- information is encoded at the level of the fields of the objects pointed to by the variables in the environment, not only at variable level.

In [14], the authors propose a framework for the analysis of object-oriented languages, and introduce two abstract domains for definite and possible aliasing respectively. The objects of these domains are similar to aliasing graphs, but without being restricted to two levels. Terminations is guaranteed by widening. These domains may be enriched by providing type information for the leaves of the graphs. However, they do not consider sharing or linearity properties.

The combination with linearity information allows us to improve the precision of the analysis in blocks of assignments, method calls, and thus on recursion. This is a fundamental issue that has not been considered in any previous analysis. In fact, an important point is that linearity information allows us to distinguish a tree from a

DAG (direct acyclic graph). For instance, the result of makeTree in Figure 5 is a tree. However, the abstract representation of a tree in any abstract domain containing only information about reachability, sharing, acyclicity, nullness, distinctness and aliasing cannot be distinguished from the abstraction of a DAG. For example, the data structure `t2`:

```
Tree t2 = new Tree()
t2.l = new Tree()
t2.r = new Tree()
t2.l.l = new Tree()
t2.l.r = t2.l.l
```

has the same sharing, acyclicity, nullness, etc etc... properties of a tree, but `t2` is not linear, while `t` in Figure 5 is linear. At the end of the method useTree, `left` is equal to `t.l.l`, `right` is `t.l.r`, but they do not share since `t` is a tree. However, `t2.l.l` and `t2.l.r` share. Therefore we need linearity to distinguish these two situations.

We believe that the ability to retrieve information after a call method is mandatory for an analysis to scale on real programs. Moreover, we also provide information on nullness and classes, which may help other analyses.

References

[1] Gianluca Amato, Simone Di Nardo Di Maio, and Francesca Scozzari. Numerical static analysis with Soot. In *Proc. of the ACM SIGPLAN SOAP '13*, New York, NY, USA, 2013. ACM.

[2] Gianluca Amato and Francesca Scozzari. Optimality in goal-dependent analysis of sharing. *Theory and Practice of Logic Programming*, 9(5):617–689, 2009.

[3] Gianluca Amato and Francesca Scozzari. On the interaction between sharing and linearity. *Theory and Practice of Logic Programming*, 10(1):49–112, 2010.

[4] Gianluca Amato and Francesca Scozzari. Optimal multibinding unification for sharing and linearity analysis. *Theory and Practice of Logic Programming*, 14:379–400, 2014.

[5] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Set-sharing is redundant for pair-sharing. *Theoretical Computer Science*, 277(1–2):3–46, 2002.

[6] Roberto Bagnara, Enea Zaffanella, and Patricia M. Hill. Enhanced sharing analysis techniques: A comprehensive evaluation. *Theory and Practice of Logic Programming*, 5(1–2):1–43, 2005.

[7] Michael Codish, Dennis Dams, and Eyal Yardeni. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In *Proc. of ICLP*, pages 79–93, 1991. The MIT Press.

[8] Michael Codish, Harald Søndergaard, and Peter J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.

[9] Werner Hans and Stephan Winkler. Aliasing and groundness analysis of logic programs through abstract interpretation and its safety. Technical Report 92–27, Technical University of Aachen, 1992. Available from `http://sunsite.informatik.rwth-aachen.de/Publications/AIB`. Last accessed June 5, 2015.

[10] Dean Jacobs and Anno Langen. Static analysis of logic programs for independent AND parallelism. *The Journal of Logic Programming*, 13(2–3):291–314, 1992.

[11] Andy King. A synergistic analysis for sharing and groundness which traces linearity. In *Proc. of ESOP '94*, volume 788 of *LNCS*, pages 363–378. Springer, Berlin Heidelberg, 1994.

[12] Kalyan Muthukumar and Manuel V. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *The Journal of Logic Programming*, 13(2–3):315–347, 1992.

[13] Mario Méndez-Lojo and ManuelV. Hermenegildo. Precise set sharing analysis for java-style programs. In *Proc. of VMCAI 2008*, volume 4905 of *LNCS*, pages 172–187. Springer Berlin Heidelberg, 2008.

[14] I. Pollet, B. Le Charlier, and A. Cortesi. Distinctness and Sharing Domains for Static Analysis of Java Programs. In *Proc. of ECOOP 2001*, volume 2072 of *LNCS*, pages 77–98, Budapest, Hungary, 2001.

[15] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *Proc. of SAS 2005*, volume 3672 of *LNCS*, pages 320–335, London, UK, 2005. Springer.