Pair-Sharing Analysis of Object-Oriented Programs

Stefano Secci and Fausto Spoto

Dipartimento di Informatica, Università di Verona, Italy

Abstract. *Pair-sharing analysis* of object-oriented programs determines those pairs of program variables bound at run-time to overlapping data structures. This information is useful for program parallelisation and analysis. We follow a similar construction for logic programming and formalise the property, or abstract domain, Sh of *pair-sharing*. We prove that Sh induces a Galois *insertion w.r.t.* the concrete domain of program states. We define a compositional abstract semantics for the static analysis over Sh, and prove it correct.

1 Introduction

Static analysis determines, at compile-time, properties about the run-time behaviour of computer programs, in order to verify, debug and optimise the code. Abstract interpretation [7, 8] is a framework for defining static analyses from the property of interest (the *abstract domain*), and prove their correctness.

In object-oriented languages such as Java, program variables are bound to data structures, stored in a sharable memory, which might hence overlap. Consider for instance the method `clone` in Figure 1 which performs a shallow copy of a `StudentList`. Its Java-like syntax is defined in Section 3. Variables *out* and *ttail* are local to `clone`, and *out* holds its return value. If variables *sl1* and *sl2* have type `StudentList`, an assignment *sl1*:=*sl2*.`clone`() makes them *share* the `Students` of *sl2*, which become *reachable* from *sl1*. Without the line *out*.`head` :=*this*.`head` in Figure 1, variables *sl1* and *sl2* would not share anymore.

Possible sharing (or, equivalently, definite non-sharing) has many applications. Namely, assume that *sl1* and *sl2* do *not* share. Then

– We can execute the calls *sl1*.`tail`.`clone`(); and *sl2*.`clone`() on different processors with disjoint memories. Hence sharing analysis can be used for *automatic program parallelisation or distribution*;
– An assignment such as *sl1*.`head` := `new Person` does not affect the class of *sl2*.`head`. Hence sharing analysis improves a given *class analysis*, which determines at compile-time the run-time class of the objects bound to the expressions [17];
– If *sl2* is a non-cyclic list then an assignment *sl1*.`tail` :=*sl2* makes *sl1* non-cyclic. This is not necessarily true if *sl1* and *sl2* share: if *sl1* points to a node of *sl2*, the previous assignment builds a cycle. Hence sharing is useful for non-cyclicity analysis.

```
class Object {}
class Person extends Object { int age; }
class Student extends Person {}
class Car extends Object { int cost; }
class StudentList extends Object {
  Student head;    StudentList tail;
  StudentList clone() with ttail:StudentList is {
    out := new StudentList;
    out.head := this.head;
    ttail := this.tail;
    if (ttail = null) then {} else out.tail := ttail.clone()
  }
}
```

Fig. 1. Our running example: a method that performs a shallow copy of a list

In all examples above, alias information [5, 16] is not enough to reach the same conclusions. Namely, to express the sharing of *sl1* and *sl2* (of type `StudentList`) through aliasing, we must check if *sl1* and *sl2* are aliases, or *sl1*.`head` and *sl2*.`head`, or *sl1*.`tail` and *sl2*.`tail`, or *sl1*.`tail`.`head` and *sl2*.`tail`.`head` and so on. Thus sharing cannot be *finitely* computed from aliasing, which is a *special case* of sharing. Nevertheless, sharing is an abstraction of graph-based representations of the memory used by some alias analyses [5, 16]. Graphs are also used in the only sharing analysis for object-oriented programs we are aware of [13]. However, our goal is to follow previous constructions for logic programming [6, 10, 11, 12] and define a more abstract domain Sh for sharing analysis than graphs. Its elements contain the unordered pairs of program variables allowed to share. We prove that a Galois *insertion* exists between Sh and the concrete domain of program states *i.e.,* Sh is not redundant. This is not easy in a strongly-typed language such as Java, compared to untyped logic programming. We provide correct abstract operations over Sh in order to implement a static analysis. We use a denotational semantics, and abstract denotations are mappings over Sh which we can implement through efficient binary decision diagrams [3], by identifying each pair of program variables with a distinct binary variable. Moreover, denotational semantics yields a compositional analysis [18].

We preferred pair-sharing to full sharing [10], which determines the *sets* of variables which share a given data-structure. Our choice is motivated by the fact that abstract domains for pair-sharing should be simpler and smaller than abstract domains for full sharing [1]. There has been some discussion on the redundancy of sharing *w.r.t.* pair-sharing in logic programs [1, 4], whose conclusions, however, do not extend immediately beyond the logic programming realm. In any case, our construction can be easily rephrased for full sharing.

The rest of the paper is organised as follows. Section 2 contains the preliminaries. Section 3 shows our simple language. Section 4 defines the abstract domain Sh and proves the Galois insertion property. Section 5 defines an abstract semantics (analyser) over Sh and states its correctness. Section 6 concludes. Proofs are in [14].

2 Preliminaries

A total (partial) function f is denoted by $\mapsto$ ($\rightarrow$). The *domain* (*codomain*) of f is $\mathrm{dom}(f)$ ($\mathrm{rng}(f)$). 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 $\mathrm{dom}(f) \setminus s$). If $f(x) = x$ then x is a *fixpoint* of f. The composition $f \circ g$ of functions f and g is such that $(f \circ g)(x) = g(f(x))$ so that we often denote it as gf. The two components of a *pair* are separated by $\star$. A definition of S such as $S = a \star b$, with a and b meta-variables, silently defines the pair selectors $s.a$ and $s.b$ for $s \in S$.

A *poset* $S \star \leq$ is a set S with a reflexive, transitive and antisymmetric relation $\leq$. If $s \in S$ then $\downarrow s = \{s' \in S \mid s' \leq s\}$. An *upper (respectively, lower) bound* of $S' \subseteq S$ is an element $u \in S$ such that $u' \leq u$ (respectively, $u' \geq u$) for every $u' \in S'$. A *complete lattice* is a poset $C \star \leq$ where *least* upper bounds (*lub*, $\sqcup$) and *greatest* lower bounds (*glb*, $\sqcap$) always exist. If $C \star \leq$ and $A \star \preceq$ are posets, $f : C \mapsto A$ is *(co-)additive* if it preserves lub's (glb's).

Let $C \star \leq$ and $A \star \preceq$ be two posets (the concrete and the abstract domain). A *Galois connection* [7, 8] is a pair of monotonic maps $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ such that $\gamma\alpha$ is extensive and $\alpha\gamma$ is reductive. It is a *Galois insertion* when $\alpha\gamma$ is the identity map *i.e.,* when the abstract domain does not contain *useless* elements. This is equivalent to α being onto, or γ one-to-one. If C and A are complete lattices and α is additive (respectively, γ is co-additive), it is the abstraction map (respectively, concretisation map) of a Galois connection. An abstract operator $\hat{f} : A^n \mapsto A$ is *correct w.r.t.* $f : C^n \rightarrow C$ if $\alpha f \gamma \preceq \hat{f}$.

3 The Language

We describe here our simple Java-like object-oriented language.

Syntax. Variables have a type and contain values. We do not consider primitive types since their values cannot be shared but only copied.

Definition 1. *Each program in the language has a set of* variables *(or* identifiers*)* $\mathcal{V}$ *(including* res, out, $this$*) 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. A* type environment *describes a finite set of variables with associated class. It is any element of the set* $TypEnv = \{\tau : \mathcal{V} \rightarrow \mathcal{K} \mid \mathrm{dom}(\tau)$ *is finite*$\}$*. In the following,* τ *will stand for a type environment. Type environments describe the variables in scope in a given program point. Moreover, we write* $F(\kappa)$ *for the type environment that maps the fields of the class* $\kappa \in \mathcal{K}$ *to their type.*

Example 2. In Figure 1, $\mathcal{K} = \{\texttt{Object}, \texttt{Person}, \texttt{Student}, \texttt{Car}, \texttt{StudentList}\}$, where `Object` is the top of the hierarchy and $\texttt{Student} \leq \texttt{Person}$. Since we are not interested in primitive types, we have $F(\texttt{Object}) = F(\texttt{Student}) = F(\texttt{Person}) = F(\texttt{Car}) = []$ and $F(\texttt{StudentList}) = [\texttt{head} \mapsto \texttt{Student}, \texttt{tail} \mapsto \texttt{StudentList}]$.

Our expressions and commands are normalised versions of those of Java. For instance, only distinct variables can be the actual parameters of a method call; left-values in assignments can only be a variable or the field of a variable; conditional can only test for equality or nullness of variables; loops must be implemented through recursion. These simplifying assumptions can be relaxed without affecting subsequent results. Instead, it is significant that we allow downwards casts, since our notion of reachability (Definition 11) depends from their presence.

Definition 3. *Our simple language is made of* expressions[1] *and* commands

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

where $\kappa \in \mathcal{K}$ and $v, w, v_1, \ldots, v_n \in \mathcal{V}$ are distinct.

Each method $\kappa.\mathtt{m}$ is defined *inside class κ 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. Variables $w_1,\ldots,w_n$ are the* formal parameters *of the method, $w_{n+1},\ldots,w_{n+m}$ are its* local variables*. The method can also use a variable out of type κ_0 which holds its* return value*. We define $\mathit{body}(\kappa.\mathtt{m}) = \mathit{com}$, $\mathit{returnType}(\kappa.\mathtt{m}) = \kappa_0$, $\mathit{input}(\kappa.\mathtt{m}) = [\mathit{this} \mapsto \kappa, w_1 \mapsto \kappa_1,\ldots,w_n \mapsto \kappa_n]$, $\mathit{output}(\kappa.\mathtt{m}) = [\mathit{out} \mapsto \kappa_0]$, $\mathit{locals}(\kappa.\mathtt{m}) = [w_{n+1} \mapsto \kappa_{n+1},\ldots,w_{n+m} \mapsto w_{n+m}]$ and $\mathit{scope}(\kappa.\mathtt{m}) = \mathit{input}(\kappa.\mathtt{m}) \cup \mathit{output}(\kappa.\mathtt{m}) \cup \mathit{locals}(\kappa.\mathtt{m})$.*

Example 4. Consider `StudentList.clone` (just `clone` later) *i.e.,* the method `clone` of `StudentList` in Figure 1. Then $\mathit{input}(\mathtt{clone}) = [\mathit{this} \mapsto \mathtt{StudentList}]$, $\mathit{output}(\mathtt{clone}) = [\mathit{out} \mapsto \mathtt{StudentList}]$ and $\mathit{locals}(\mathtt{clone}) = [\mathit{ttail} \mapsto \mathtt{StudentList}]$.

Our language is strongly typed *i.e.,* expressions *exp* have a static (compile-time) type $\mathit{type}_\tau(\mathit{exp})$ in τ, consistent with their run-time values (see [14]).

Semantics. We describe here the *state* of the computation and how the language constructs modify it. We use a denotational semantics, hence compositional, in the style of [18]. However, we use a more complex notion of state, to account for dynamically-allocated and sharable data-structures. By using a denotational semantics, our states contain only a single frame, rather than an activation stack of frames. A method call is hence resolved by *plugging* the interpretation of the method (Definition 9) in its calling context. This is standard in denotational semantics and has been used for years in logic programming [2].

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.

[1] The `null` constant is decorated with the class κ induced by its context, as in $v{:=}\ \mathtt{null}\ \kappa$, where κ is the type of v. This way we avoid introducing a distinguished type for `null`. You can assume this decoration to be provided by the compiler.

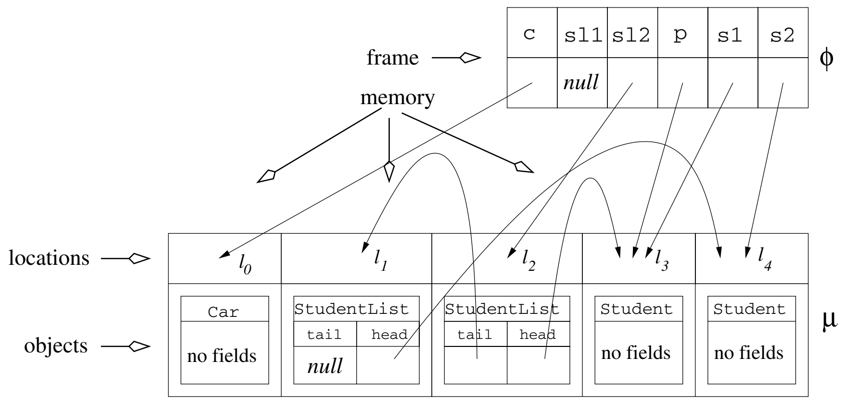

Fig. 2. A state (frame ϕ and memory μ) for $\tau = [c \mapsto \texttt{Car}, \mathit{sl1} \mapsto \texttt{StudentList}, \mathit{sl2} \mapsto \texttt{StudentList}, p \mapsto \texttt{Person}, \mathit{s1} \mapsto \texttt{Student}, \mathit{s2} \mapsto \texttt{Student}]$

Definition 5. *Let Loc be an infinite set of* locations*. We define* frames*,* objects *and* memories *as* $\mathit{Frame}_\tau = \{\phi \mid \phi \in \mathrm{dom}(\tau) \mapsto \mathit{Loc} \cup \{\mathit{null}\}\}$*,* $\mathit{Obj} = \{\kappa \star \phi \mid \kappa \in \mathcal{K},\ \phi \in \mathit{Frame}_{F(\kappa)}\}$ *and* $\mathit{Memory} = \{\mu \in \mathit{Loc} \to \mathit{Obj} \mid \mathrm{dom}(\mu) \text{ is finite}\}$*. A new object of class* κ *is* $\mathit{new}(\kappa) = \kappa \star \phi$*, with* $\phi(v) = \mathit{null}$ *for each* $v \in F(\kappa)$*.*

Example 6. Figure 2 shows a frame ϕ (with 6 variables) and a memory μ. Different occurrences of the same location are linked by arrows. For instance, *s1* is bound to a location l_3 and $\mu(l_3)$ is a `Student` object. Objects are represented as boxes in μ with a class tag and a local frame mapping fields to locations or *null*.

Type correctness $\phi \star \mu : \tau$ guarantees that in ϕ and in the objects in μ there are no dangling pointers and that variables and fields may only be bound to locations which contain objects allowed by τ or by the type environment for the fields of the objects (Definition 1). This is a sensible constraint for the memory allocated by strongly-typed languages, such as Java. For its formal definition, see [14]. We can now define the *states* as type correct pairs $\phi \star \mu$.

Definition 7. *Let* τ *be the type environment at a given program point p. The set of possible* states *at p is* $\Sigma_\tau = \{\phi \star \mu \mid \phi \in \mathit{Frame}_\tau,\ \mu \in \mathit{Memory},\ \phi \star \mu : \tau\}$*.*

Example 8. Consider Figure 2. The variables in ϕ are bound to *null* or to objects of a class allowed by τ. The `tail` fields of the objects in μ are bound to *null* or to a `StudentList`, consistently with $F(\texttt{StudentList})$ (Example 2). The `head` fields are bound to a `Student`, consistently with $F(\texttt{StudentList})$. Hence $\phi \star \mu : \tau$ and $\phi \star \mu \in \Sigma_\tau$.

Each method is *denoted* by a partial function from input to output states. A collection of such functions, one for each method, is an *interpretation*.

Definition 9. *An* interpretation *I maps methods to partial functions on states, such that* $I(\kappa.\mathtt{m}) : \Sigma_{\mathit{input}(\kappa.\mathtt{m})} \to \Sigma_{\mathit{output}(\kappa.\mathtt{m})}$ *for each method* $\kappa.\mathtt{m}$*.*

Definition 10 builds interpretations from the denotations of commands and expressions. These denotations are in [14]. Below, we discuss them informally.

Expressions in our language have side-effects and return a value. Hence their denotations are partial maps from an initial to a final state. The latter contains a distinguished variable *res* holding the value of the expression: $\mathcal{E}^I_\tau[\![\_]\!] : exp \mapsto (\Sigma_\tau \to \Sigma_{\tau+exp})$, where $\tau + exp = \tau[res \mapsto type_\tau(exp)]$. Namely, given an input state $\phi \star \mu$, the denotation of `null` κ binds *res* to *null* in ϕ. The denotation of `new` κ binds *res* to a new location bound to a new object of class κ. The denotation of v copies v into *res*. The denotation of v.`f` accesses the object $o = \mu(\phi(v))$ bound to v (provided $\phi(v) \neq null$) and then copies the field `f` of o (*i.e.,* $o.\phi(\mathtt{f})$) into *res*. The denotation of $(\kappa)v$ copies v into *res*, but only if the cast is satisfied. The denotation of method call uses the dynamic class of the receiver to fetch the denotation of the method from the current interpretation. It plugs that denotation in the calling context, by building a starting state $\sigma^\dagger$, whose formal parameters (including *this*) are bound to the actual parameters.

The denotation of a command is a partial map from an initial to a final state: $\mathcal{C}^I_\tau[\![\_]\!] : com \mapsto (\Sigma_\tau \to \Sigma_\tau)$. Given an initial state $\phi \star \mu$, the denotation of $v{:=}exp$ uses the denotation of *exp* to get a state whose variable *res* holds *exp*'s value. Then it copies *res* into v, and removes *res*. Similarly for $v.\mathtt{f} := exp$, but *res* is copied into the field `f` of the object $\mu\phi(v)$ bound to v, provided $\phi(v) \neq null$. The denotation of the conditionals checks their guard in $\phi \star \mu$ and then uses the denotation of `then` or the denotation of `else`. The denotation of a sequence of commands is the functional composition of their denotations.

By using $\mathcal{C}^I_\tau[\![\_]\!]$, we define a transformer on interpretations, which evaluates the bodies of the methods in I, by using an input state where local variables are bound to *null*. At the end, the final state is restricted to the variable *out*, so that Definition 9 is respected. This corresponds to the *immediate consequence operator* used in logic programming [2].

Definition 10. *The following* transformer on interpretations *tranforms an interpretation I into a new interpretation I' such that*

$$I'(\kappa.\mathtt{m}) = (\lambda\phi \star \mu \in \Sigma_{input(\kappa.\mathtt{m})}.\phi[out \mapsto null, w_{n+1} \mapsto null, \ldots, w_{n+m} \mapsto null] \star \mu) \\ \circ\, \mathcal{C}^I_{scope(\kappa.\mathtt{m})}[\![body(\kappa.\mathtt{m})]\!] \circ (\lambda\phi \star \mu \in \Sigma_{scope(\kappa.\mathtt{m})}.(\phi|_{out} \star \mu)).$$

The denotational semantics *of a program is the least fixpoint of this transformer on interpretations.*

4 An Abstract Domain for Pair-Sharing

We formalise here when two variables *share* and define our abstract domain Sh. We need a notion of *reachability* for locations. A location is reachable if it is bound to a variable or to a field of an object stored at a reachable location.

Definition 11. *Let $\phi \star \mu \in \Sigma_\tau$ and $v \in \mathrm{dom}(\tau)$. We define the set of locations* reachable *from v in $\phi \star \mu$ as $L_\tau(\phi \star \mu)(v) = \cup\{L^i_\tau(\phi \star \mu)(v) \mid i \geq 0\}$, where*

$L^0_\tau(\phi\star\mu)(v) = \{\phi(v)\} \cap Loc$ and $L^{i+1}_\tau(\phi\star\mu)(v) = \cup\{\mathrm{rng}(\mu(l).\phi) \cap Loc \mid l \in L^i_\tau(\phi\star\mu)(v)\}$. Two variables $v_1, v_2 \in \mathrm{dom}(\tau)$ share *in $\phi\star\mu$ if there is a location which is reachable from both i.e., if $L_\tau(\phi\star\mu)(v_1) \cap L_\tau(\phi\star\mu)(v_2) \neq \emptyset$.*

Note, in Definition 11, that if an object $o = \mu(l)$ is stored in a reachable location l, then also the locations $\mathrm{rng}(\mu(l).\phi) \cap Loc$ of *all* o's fields are reachable. This reflects the fact that we consider a language with (checked) casts (Section 3), which allow all fields of the objects to be accessed in a program.

Example 12. Consider the state $\sigma = \phi\star\mu$ in Figure 2. For every $i \geq 0$ we have

$$\begin{array}{llll}
L^0_\tau(\sigma)(c) = \{l_0\} & L^{i+1}_\tau(\sigma)(c) = \emptyset & L^i_\tau(\sigma)(sl1) = \emptyset & \\
L^0_\tau(\sigma)(sl2) = \{l_2\} & L^1_\tau(\sigma)(sl2) = \{l_1, l_3\} & L^2_\tau(\sigma)(sl2) = \{l_4\} & L^{i+3}_\tau(\sigma)(sl2) = \emptyset \\
L^0_\tau(\sigma)(p) = \{l_3\} & L^{i+1}_\tau(\sigma)(p) = \emptyset & L^0_\tau(\sigma)(s1) = \{l_3\} & L^{i+1}_\tau(\sigma)(s1) = \emptyset \\
L^0_\tau(\sigma)(s2) = \{l_4\} & L^{i+1}_\tau(\sigma)(s2) = \emptyset. & &
\end{array}$$

We conclude that $L_\tau(\sigma)(c) = \{l_0\}$, $L_\tau(\sigma)(sl1) = \emptyset$, $L_\tau(\sigma)(sl2) = \{l_1, l_2, l_3, l_4\}$, $L_\tau(\sigma)(p) = \{l_3\}$, $L_\tau(\sigma)(s1) = \{l_3\}$ and $L_\tau(\sigma)(s2) = \{l_4\}$. Hence, in σ, variable *sl2* shares with *sl2*, *p*, *s1*, *s2*; variable *p* does not share with *s2*; *c* shares only with *c*; *sl1* does not share with any variable, not even with itself.

By using reachability, we refine Definition 9 by requiring that a method does not write into the locations L of the input state which are *not* reachable from the formal parameters, nor read them, so that for instance no location in L is reachable from the method's return value. Programming languages such as Java and that of Section 3 satisfy these constraints. They let us prove the correctness of the abstract counterpart of method call that we define later (Figure 3).

Definition 13. *We refine Definition 9 by requiring that if $I(\kappa.\mathtt{m})(\phi\star\mu) = (\phi'\star\mu')$ and $L = \mathrm{dom}(\mu) \setminus (\cup\{L_{input(\kappa.\mathtt{m})}(\phi\star\mu)(v) \mid v \in \mathrm{dom}(input(\kappa.\mathtt{m}))\})$ then $\mu|_L = \mu'|_L$, $\phi'(out) \notin L$ and $\cup\{\mathrm{rng}(\mu'(l)) \cap L \mid l \in \mathrm{dom}(\mu'|_{-L})\} = \emptyset$.*

As a first attempt, our abstract domain is the powerset of the unordered pairs of variables in $\mathrm{dom}(\tau)$. The concretisation map says that if (v_1, v_2) belongs to an abstract domain element *sh*, then *sh* allows v_1 and v_2 to share.

Definition 14. *Let $sh \in \wp(\mathrm{dom}(\tau) \times \mathrm{dom}(\tau))$. We define*

$$\gamma_\tau(sh) = \left\{ \sigma \in \Sigma_\tau \;\middle|\; \begin{array}{l} \textit{for every } v_1, v_2 \in \mathrm{dom}(\tau) \\ \textit{if } L_\tau(\sigma)(v_1) \cap L_\tau(\sigma)(v_2) \neq \emptyset \textit{ then } (v_1, v_2) \in sh \end{array} \right\}.$$

It must be observed, however, that two variables might *never* be able to share if their static types do not let them be bound to overlapping data structures.

Example 15. In the state in Figure 2, variable *c* does not share with any of the other variables (Example 12). This is not specific to that state. There is no state in Σ_τ where *c* shares with anything but itself. This is because (Figure 1) a `Car` is not a `Person` nor a `Student` nor a `StudentList` nor vice versa. Moreover, it is not possible to reach a shared object from a `Car` and a `Person` (or a `Student` or a `StudentList`) because these classes have no field of the same type.

Example 15 must be taken into account if we are looking for a Galois *insertion*, rather than a Galois *connection*, between $\wp(\Sigma_\tau)$ and the abstract domain. The abstract domain must include only pairs of variables whose static types *share*. As in Definition 11, we first need a notion of *reachability* for classes.

Definition 16. *The set of classes* reachable *in τ from a variable v is $C_\tau(v) = \cup\{C^i_\tau(v) \mid i \geq 0\}$, where $C^0_\tau(v) = \downarrow\tau(v)$ and $C^{i+1}_\tau(v) = \downarrow(\cup\{\mathrm{rng}(F(\kappa)) \mid \kappa \in C^i_\tau(v)\})$. The set of pairs of variables in τ whose static types* share *is*

$$\mathcal{SV}_\tau = \{(v_1, v_2) \in \mathrm{dom}(\tau) \times \mathrm{dom}(\tau) \mid C_\tau(v_1) \cap C_\tau(v_2) \neq \emptyset\}.$$

In Definition 16, if a class κ is reachable, then all its subclasses $\downarrow\kappa$ are considered reachable. This reflects the fact that we consider a language with (checked) casts.

Example 17. Consider τ as in Figure 2. For every $i \geq 0$ we have $C^0_\tau(c) = \{\texttt{Car}\}$, $C^{i+1}_\tau(c) = \emptyset$, $C^0_\tau(\mathit{sl1}) = \{\texttt{StudentList}\}$, $C^{i+1}_\tau(\mathit{sl1}) = \{\texttt{StudentList}, \texttt{Student}\}$, $C^0_\tau(\mathit{sl2}) = \{\texttt{StudentList}\}$, $C^{i+1}_\tau(\mathit{sl2}) = \{\texttt{StudentList}, \texttt{Student}\}$, $C^0_\tau(p) = \{\texttt{Student}, \texttt{Person}\}$, $C^{i+1}_\tau(p) = \emptyset$, $C^0_\tau(\mathit{s1}) = \{\texttt{Student}\}$, $C^{i+1}_\tau(\mathit{s1}) = \emptyset$, $C^0_\tau(\mathit{s2}) = \{\texttt{Student}\}$ and $C^{i+1}_\tau(\mathit{s2}) = \emptyset$. Hence $C_\tau(c) = \{\texttt{Car}\}$, $C_\tau(\mathit{sl1}) = C_\tau(\mathit{sl2}) = \{\texttt{StudentList}, \texttt{Student}\}$, $C_\tau(p) = \{\texttt{Student}, \texttt{Person}\}$ and $C_\tau(\mathit{s1}) = C_\tau(\mathit{s2}) = \{\texttt{Student}\}$. So $\mathcal{SV}_\tau = (\mathrm{dom}(\tau) \times \mathrm{dom}(\tau)) \setminus \{(c, \mathit{sl1}), (c, \mathit{sl2}), (c, p), (c, \mathit{s1}), (c, \mathit{s2})\}$ *i.e.,* c can only share with c; all other variables can share with each other.

Abstract domain elements should only include pairs in $\mathcal{SV}_\tau$, since the others cannot share. A further observation shows that if v_1 and v_2 share, then they are not *null*. Thus v_1 shares with v_1 and v_2 shares with v_2. Also this constraint is needed to prove the Galois insertion property (Proposition 20).

Definition 18. *The* abstract domain for pair-sharing *is*

$$\mathsf{Sh}_\tau = \{sh \subseteq \mathcal{SV}_\tau \mid \textit{if } (v_1, v_2) \in sh \textit{ then } (v_1, v_1) \in sh \textit{ and } (v_2, v_2) \in sh\}$$

ordered by set-inclusion. From now on, by γ_τ we mean the restriction to Sh_τ of the map γ_τ of Definition 14.

Example 19. Let τ be as in Figure 2. Then $sh_1 = \{(c, \mathit{sl1}), (c, c), (\mathit{sl1}, \mathit{sl1})\} \notin \mathsf{Sh}_\tau$ since $(c, \mathit{sl1}) \notin \mathcal{SV}_\tau$ (Example 17); $sh_2 = \{(\mathit{sl1}, \mathit{sl2}), (\mathit{sl1}, \mathit{sl1})\} \notin \mathsf{Sh}_\tau$ since $(\mathit{sl1}, \mathit{sl2}) \in sh_2$ but $(\mathit{sl2}, \mathit{sl2}) \notin sh_2$; $sh_3 = \{(\mathit{sl1}, \mathit{sl2}), (\mathit{sl1}, \mathit{sl1}), (\mathit{sl2}, \mathit{sl2})\} \in \mathsf{Sh}_\tau$.

Proposition 20. *The map γ_τ of Definition 18 is the concretisation map of a Galois insertion from $\wp(\Sigma_\tau)$ to Sh_τ.*

In a Galois insertion, the concretisation map induces the abstraction map. Its explicit definition, below, states that the abstraction of a set of concrete states S is the set of pairs of variables which share in at least one $\sigma \in S$.

Proposition 21. *The abstraction map induced by the concretisation map of Definition 14 (restricted to Sh_τ) is such that, for every $S \subseteq \Sigma_\tau$,*

$$\alpha_\tau(S) = \left\{ (v_1, v_2) \in \mathrm{dom}(\tau) \times \mathrm{dom}(\tau) \;\middle|\; \begin{array}{l} \textit{there exists } \sigma \in S \textit{ such that} \\ L_\tau(\sigma)(v_1) \cap L_\tau(\sigma)(v_2) \neq \emptyset \end{array} \right\}.$$

Example 22. Consider the state $\phi \star \mu$ in Figure 2. Its reachability information is given in Example 12 so that (remember that pairs are unordered) $\alpha_\tau(\{\phi \star \mu\}) = \{(c, c), (sl2, sl2), (sl2, p), (sl2, s1), (sl2, s2), (p, p), (p, s1), (s1, s1), (s2, s2)\}$.

5 An Abstract Semantics on Sh.

The domain Sh_τ of Section 4 induces an abstract version of the semantics of Section 3, which we make explicit here. This semantics is an actual static analyser for pair-sharing which can be implemented inside generic engines such as our Julia analyser [15].

We start with the abstract counterpart of the interpretations of Definition 9. The idea is to map the approximation over Sh_τ of some input states into the approximation of the corresponding output states.

Definition 23. *A* sharing interpretation *I maps methods into total functions such that $I(\kappa.\mathtt{m}) : \mathsf{Sh}_{input(\kappa.\mathtt{m})} \mapsto \mathsf{Sh}_{output(\kappa.\mathtt{m})}$ for each method $\kappa.\mathtt{m}$.*

Example 24. Consider the method `clone` in Figure 1. We have $\mathsf{Sh}_{input(\mathtt{clone})} = \{\emptyset, \{(this, this)\}\}$ and $\mathsf{Sh}_{output(\mathtt{clone})} = \{\emptyset, \{(out, out)\}\}$. A sharing interpretation, consistent with the concrete semantics of the method, is $I = [\emptyset \mapsto \emptyset, \{(this, this)\} \mapsto \{(out, out)\}]$ *i.e.,* in the input, *this* shares with *this* if and only if, in the output, *out* shares with *out*.

Our goal now is to compute the interpretation of Example 24 automatically.

5.1 Abstract Denotation for the Expressions

The concrete semantics of Section 3 specifies how each expression *exp* transforms an initial state into a final state, where *res* holds the value of *exp*. To mimic this behaviour on the abstract domain, we specify how *exp* transforms input abstract states *sh* into final abstract states sh' where *res* refers to *exp*'s value. For correctness (Section 2), sh' must include the pairs of variables which share in the concrete states σ' obtained by evaluating *exp* from a concrete state $\sigma \in \gamma_\tau(sh)$.

The concrete semantics of *null* κ stores *null* in the variable *res* of σ', which otherwise coincides with σ. Hence, in σ', variable *res* does not share. The other variables share exactly as they do in σ. Consequently, we let $sh' = sh$.

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. Then we let $sh' = sh \cup \{(res, res)\}$.

The concrete semantics of v obtains σ' from σ by copying v into *res*. Hence, in σ', variable *res* shares with v and all those variable that v used to share with in σ. Since the other variables are unchanged, we let $sh' = sh \cup (sh[v \mapsto res]) \cup \{(v, res)\}$. By $sh[v \mapsto res]$ we mean *sh* where v is renamed into *res*. We improve this approximation for the case when $(v, v) \not\in sh$ *i.e.,* when v is definitely *null* so that variable v does not occur in *sh* (Definition 18) and $sh[v \mapsto res] = sh$.

Moreover, in such a case, v and *res* are *null* in σ' and do not share. Hence, in this case we let $sh' = sh$.

When it is defined, the cast $(\kappa)v$ stores in *res* the value of v. Hence the above approximation for v is also correct for $(\kappa)v$.

The concrete semantics of $v.\mathtt{f}$ stores in *res* the value of the field `f` of v, provided v is not *null*. When $(v, v) \notin sh$, variable v is *null* in σ, $v.\mathtt{f}$ *never* yields a final state and the best approximation of the resulting, empty set of final states is $\emptyset$. If instead $(v, v) \in sh$, variable *res* shares in σ' with a variable, say w, only if v shares in σ with w: from v one reaches $v.\mathtt{f}$ which is an alias of *res*. Moreover, v and *res* share in σ'. Thus we should let $sh' = sh \cup (sh[v \mapsto res]) \cup \{(v, res)\}$. However, Example 25 shows that sh' might contain pairs not in $\mathcal{SV}$ (Definition 16) and hence in general $sh' \notin \mathsf{Sh}$ (Definition 18).

Example 25. Let every `Student` be paired with its `Car` in the list:

```
class StudentCarList extends StudentList { Car car; }
```

Let $\tau = [v \mapsto \mathtt{StudentCarList}, w \mapsto \mathtt{Student}]$ and $sh = \{(v, w), (v, v), (w, w)\} \in \mathsf{Sh}_\tau$, so that $(res, w) \in sh'$. But a `Car` cannot share with a `Student` (Figure 1) *i.e.,* $(res, w) \notin \mathcal{SV}_{\tau+v.\mathtt{car}}$ and $sh' \notin \mathsf{Sh}_{\tau+v.\mathtt{car}}$.

We solve this problem by removing spurious pairs such as (res, w) in Example 25. Namely, we define $sh' = sh \cup [(sh[v \mapsto res]) \cap \mathcal{SV}_{\tau+v.\mathtt{f}}] \cup \{(v, res)\}$.

The concrete semantics of the method call $v.\mathtt{m}(v_1, \ldots, v_n)$ builds an input state $\sigma^\dagger = [this \mapsto \phi(v), w_1 \mapsto \phi(v_1), \ldots, w_n \mapsto \phi(v_n)] \star \mu$ for the callee *i.e.,* it restricts ϕ to $pars = \{v, v_1, \ldots, v_n\}$ and renames v into *this* and each v_i into w_i. We mimic this by restriction and renaming on the abstract domain.

Definition 26. *Let* $sh \in \mathsf{Sh}_\tau$ *and* $V \subseteq \mathrm{dom}(\tau)$. *We define* $sh|_V \in \mathsf{Sh}_\tau$ *as* $sh|_V = \{(v_1, v_2) \in sh \mid v_1 \in V \text{ and } v_2 \in V\}$. *Moreover, we define* $sh|_{-V} = sh|_{\mathrm{dom}(\tau)\setminus V}$.

Let hence $sh^\dagger = sh|_{pars}[v \mapsto this, v_1 \mapsto w_1, \ldots, v_n \mapsto w_n]$ approximate $\sigma^\dagger$. The abstract domain contains no information on the run-time class of v. Hence we conservatively assume that every method `m` in a subclass of the static type of v might be called [9] *i.e.,* we use $sh^\ddagger = \cup\{I(\kappa.\mathtt{m})(sh^\dagger) \mid \kappa \leq \tau(v)\}[out \mapsto res]$ as an approximation for the result of the call. We 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 must determine the effects of the call on the variables of the caller. We do it here in a relatively imprecise way. Subsection 5.4 shows how to improve this approximation. We use the fact that a method call can only modify (and access) input locations which are reachable from the actual arguments (Definition 13). Hence we let *res* share with *every* parameter which was not *null* at call-time. Formally, we build the approximation $sh^\flat = sh^\ddagger \cup \{(res, p) \mid (res, res) \in sh^\ddagger,\ p \in pars \text{ and } (p, p) \in sh\}$. Then we close transitively the sharing pairs *w.r.t.* the parameters, by computing the *star-closure* $(sh \cup sh^\flat)^*_{pars}$.

Definition 27. *Let* $sh \in \mathsf{Sh}_\tau$ *and* $V \subseteq \mathrm{dom}(\tau)$. *The* star-closure *of* sh *w.r.t.* V *is* $sh^*_V = sh \cup \big(\{(v_1, v_2) \mid v', v'' \in V,\ (v_1, v') \in sh \text{ and } (v_2, v'') \in sh\} \cap \mathcal{SV}_\tau\big)$.

In Definition 27 we use $\mathcal{SV}_\tau$ to discard pairs of variables which cannot share.

$$\mathcal{SE}^I_\tau[\![\texttt{null }\kappa]\!](sh) = sh \qquad \mathcal{SE}^I_\tau[\![\texttt{new }\kappa]\!](sh) = sh \cup \{(res, res)\}$$

$$\mathcal{SE}^I_\tau[\![v]\!](sh) = \mathcal{SE}^I_\tau[\![(\kappa)v]\!](sh) = \begin{cases} sh \cup (sh[v \mapsto res]) \cup \{(v, res)\} & \text{if } (v,v) \in sh \\ sh & \text{otherwise} \end{cases}$$

$$\mathcal{SE}^I_\tau[\![v.\texttt{f}]\!](sh) = \begin{cases} sh \cup \{(v, res)\} \cup (sh[v \mapsto res] \cap \mathcal{SV}_{\tau+v.\texttt{f}}) & \text{if } (v,v) \in sh \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathcal{SE}^I_\tau[\![v.\texttt{m}(v_1,\ldots,v_n)]\!](sh) = \begin{cases} (sh \cup sh^\flat)^*_{pars} & \text{if } (v,v) \in sh \\ \emptyset & \text{otherwise} \end{cases}$$

where $pars = \{v, v_1, \ldots, v_n\}$, $sh^\dagger = sh|_{pars}[v \mapsto this, v_1 \mapsto w_1, \ldots, v_n \mapsto w_n]$, $sh^\ddagger = \cup\{I(\kappa.\texttt{m})(sh^\dagger) \mid \kappa \le \tau(v)\}[out \mapsto res]$ and $sh^\flat = sh^\ddagger \cup \{(res, p) \mid (res, res) \in sh^\ddagger,\ p \in pars \text{ and } (p,p) \in sh\}$.

Fig. 3. The sharing interpretation for expressions

Definition 28. *Let τ describe the variables in scope and I be a sharing interpretation. Figure 3 defines the* sharing denotation $\mathcal{SE}^I_\tau[\![\_]\!] : exp \mapsto (\mathsf{Sh}_\tau \mapsto \mathsf{Sh}_{\tau+exp})$.

Example 29. Let $\tau = scope(\texttt{clone}) = [out \mapsto \texttt{StudentList}, this \mapsto \texttt{StudentList}, ttail \mapsto \texttt{StudentList}]$ describe the variables in scope in the `clone` method of Figure 1. Let I be the sharing interpretation of Example 24. Then

$$\mathcal{SE}^I_\tau[\![\texttt{new StudentList}]\!](\{(this, this)\}) = \{(res, res), (this, this)\}$$

$$\mathcal{SE}^I_\tau[\![this.\texttt{head}]\!]\left(\left\{\begin{array}{c}(out, out),\\(this, this)\end{array}\right\}\right) = \left\{\begin{array}{c}(out, out), (res, res),\\(this, res), (this, this)\end{array}\right\}$$

$$\mathcal{SE}^I_\tau[\![this.\texttt{tail}]\!]\left(\left\{\begin{array}{c}(out, out),\\(this, out),\\(this, this)\end{array}\right\}\right) = \left\{\begin{array}{c}(out, out), (res, out), (res, res),\\(res, this), (this, out), (this, this)\end{array}\right\}.$$

Consider now $sh = \{(out, out), (this, out), (this, this), (ttail, this), (ttail, out), (ttail, ttail)\}$. Let us compute $\mathcal{SE}^I_\tau[\![ttail.\texttt{clone()}]\!](sh)$. We have $pars = \{ttail\}$ and $sh^\dagger = \{(this, this)\}$. If we assume that `clone` is not overridden, then $sh^\ddagger = (I(\texttt{clone})(\{(this, this)\}))[out \mapsto res] = \{(res, res)\}$, $sh^\flat = \{(res, res), (res, ttail)\}$ and $(sh \cup sh^\flat)^*_{\{ttail\}} = (\{(out, out), (this, out), (this, this), (ttail, this), (ttail, out), (ttail, ttail)\} \cup \{(res, res), (res, ttail)\})^*_{\{ttail\}}$. This introduces the pairs (out, res) and $(res, this)$ yielding $\{(out, out), (out, res), (res, res), (res, this), (res, ttail), (this, out), (this, this), (ttail, out), (ttail, this), (ttail, ttail)\}$.

5.2 Abstract Denotation for the Commands

In the concrete semantics, each command c transforms an initial state into a final state. On the abstract domain, it transforms an initial state sh into an abstract state sh' which, for correctness (Section 2), includes the pairs of variables which share in the concrete states σ' obtained by evaluating c from each $\sigma \in \gamma_\tau(sh)$.

$$\mathcal{SC}^I_\tau[\![v{:=}exp]\!] = \mathcal{SE}^I_\tau[\![exp]\!] \circ setVar^v_{\tau+exp}$$
$$\text{where } setVar^v_\tau = \lambda sh \in \mathsf{Sh}_\tau . sh|_{-v}[res \mapsto v]$$
$$\mathcal{SC}^I_\tau[\![v.\mathtt{f} := exp]\!] = \mathcal{SE}^I_\tau[\![exp]\!] \circ setField^{v.\mathtt{f}}_{\tau+exp}$$
$$\text{where } setField^{v.\mathtt{f}}_\tau = \lambda sh \in \mathsf{Sh}_\tau . \begin{cases} (((sh \cup \{(v,res)\})^*_{res})|_{-res})^*_v & \text{if } (v,v) \in sh \\ \emptyset & \text{otherwise} \end{cases}$$
$$\mathcal{SC}^I_\tau\left[\!\!\left[\begin{array}{l}\mathtt{if}\ v = w\\ \mathtt{then}\ com_1\\ \mathtt{else}\ com_2\end{array}\right]\!\!\right](sh) = \mathcal{SC}^I_\tau[\![com_1]\!](sh) \cup \mathcal{SC}^I_\tau[\![com_2]\!](sh)$$
$$\mathcal{SC}^I_\tau\left[\!\!\left[\begin{array}{l}\mathtt{if}\ v = \mathtt{null}\\ \mathtt{then}\ com_1\\ \mathtt{else}\ com_2\end{array}\right]\!\!\right](sh) = \begin{cases} \mathcal{SC}^I_\tau[\![com_1]\!](sh|_{-v}) \cup \mathcal{SC}^I_\tau[\![com_2]\!](sh) & \text{if } (v,v) \in sh \\ \mathcal{SC}^I_\tau[\![com_1]\!](sh|_{-v}) & \text{otherwise} \end{cases}$$
$$\mathcal{SC}^I_\tau[\![\{com_1; \ldots; com_p\}]\!] = (\lambda sh \in \mathsf{Sh}_\tau . sh) \circ \mathcal{SC}^I_\tau[\![com_1]\!] \circ \cdots \circ \mathcal{SC}^I_\tau[\![com_p]\!].$$

The identity map $\lambda sh \in \mathsf{Sh}_\tau . sh$ for the sequence of commands is needed when $p = 0$.

Fig. 4. The sharing interpretation for commands

The concrete evaluation of $v{:=}exp$ evaluates exp and stores its result into v. Thus we define sh' as the functional composition of $\mathcal{SE}^I_\tau[\![exp]\!]$ with the map $setVar^v_\tau(sh) = sh|_{-v}[res \mapsto v]$ which renames res into v (v's original value is lost).

Similarly, for $v.\mathtt{f} := exp$ we use a *setField* map. Its definition has two cases. When $(v,v) \notin sh$, we know that v is *null* and hence there is no final state. The best approximation of the empty set of final states is $\emptyset$. Otherwise, its definition reflects the fact that after assigning exp to $v.\mathtt{f}$, variable v might share with every variable w which shares with the value of exp. This means that we must perform a star-closure *w.r.t.* res (Definition 27) and remove res. Moreover if, before this assignment, a variable v' shares with v, then the assignment might also affect v', so we conservatively assume that v' and w might share. This means that we must compute a star-closure *w.r.t.* v. In conclusion, in this second case we let $setField^{v.\mathtt{f}}_\tau(sh) = (((sh \cup \{(v,res)\})^*_{res})|_{-res})^*_v$.

A correct approximation of the conditionals of Definition 3 considers them non-deterministic, so that their denotation is $\mathcal{SC}^I_\tau[\![com_1]\!] \cup \mathcal{SC}^I_\tau[\![com_2]\!]$. But we can do better. Namely, if $(v,v) \notin sh$ then v is definitely *null* in σ, and the guard $v = \mathtt{null}$ is true. Vice versa, in the `then` branch we can assume that the guard is true. When the guard is $v = \mathtt{null}$, this means that v can be removed from the input approximation sh.

The composition of commands is denoted by functional composition over Sh.

Definition 30. *Let τ describe the variables in scope, I be a sharing interpretation. Figure 4 shows the* sharing denotation for commands $\mathcal{SC}^I_\tau[\![\_]\!] : com \mapsto (\mathsf{Sh}_\tau \mapsto \mathsf{Sh}_\tau)$.

Example 31. Let $\tau = scope(\mathtt{clone}) = [out \mapsto \mathtt{StudentList}, this \mapsto \mathtt{StudentList}, ttail \mapsto \mathtt{StudentList}]$ describe the variables in scope in the `clone` method of Figure 1. Let I be the sharing interpretation ofExample 24. We want to compute the abstract state sh_5 at the end of `clone` assuming that we run `clone` from $sh_1 = \{(\mathit{this}, \mathit{this})\}$. We use Definition 30 and we write $\{sh_i\}c\{sh_{i+1}\}$ for $\mathcal{SC}^I_\tau[\![c]\!](sh_i) = sh_{i+1}$ *i.e.,* we decorate each program point p with the abstract approximation at p. For the right-hand side of assignments, we use the denotations that we already computed in Example 29. We have

$$\begin{gathered}
sh_1 = \{(\mathit{this}, \mathit{this})\}\\
\mathit{out} := \texttt{new StudentList}\\
sh_2 = \{(\mathit{out}, \mathit{out}), (\mathit{this}, \mathit{this})\}\\
\mathit{out}.\texttt{head} := \mathit{this}.\texttt{head}\\
sh_3 = \{(\mathit{out}, \mathit{out}), (\mathit{this}, \mathit{out}), (\mathit{this}, \mathit{this})\}\\
\mathit{ttail} := \mathit{this}.\texttt{tail}\\
sh_4 = \{(\mathit{out}, \mathit{out}), (\mathit{this}, \mathit{out}), (\mathit{this}, \mathit{this}), (\mathit{ttail}, \mathit{out}), (\mathit{ttail}, \mathit{this}), (\mathit{ttail}, \mathit{ttail})\}\\
\texttt{if } \mathit{ttail} = \texttt{ null then } \{\} \texttt{ else } \mathit{out}.\texttt{tail} := \mathit{ttail}.\texttt{clone()}\\
sh_5 = sh_4.
\end{gathered}$$

Let us consider in detail how sh_5 is computed from sh_4. Since $(\mathit{ttail}, \mathit{ttail}) \in sh_4$,

$$\begin{aligned}
sh_5 &= \mathcal{SC}^I_\tau[\![\texttt{if} \ldots \mathit{ttail}.\texttt{clone()}]\!](sh_4)\\
&= \mathcal{SC}^I_\tau[\![\{\}]\!](sh_4|_{-\mathit{ttail}}) \cup \mathcal{SC}^I_\tau[\![\mathit{out}.\texttt{tail} := \mathit{ttail}.\texttt{clone()}]\!](sh_4)\\
&= sh_4|_{-\mathit{ttail}} \cup (\mathit{setField}^{\mathit{out}.\texttt{tail}}_{\tau+\mathit{ttail}.\texttt{clone()}}(\mathcal{SE}^I_\tau[\![\mathit{ttail}.\texttt{clone()}]\!](sh_4)))\\
\text{(Ex. 29)} &= sh_4|_{-\mathit{ttail}} \cup \left(\begin{array}{c} \mathit{setField}^{\mathit{out}.\texttt{tail}}_{\tau+\mathit{ttail}.\texttt{clone()}} \\ \underbrace{\left(\left\{ \begin{array}{c} (\mathit{out}, \mathit{res}), (\mathit{out}, \mathit{out}), (\mathit{res}, \mathit{this}), (\mathit{this}, \mathit{out}), \\ (\mathit{this}, \mathit{this}), (\mathit{ttail}, \mathit{this}), (\mathit{ttail}, \mathit{out}), (\mathit{ttail}, \mathit{ttail}), \\ (\mathit{res}, \mathit{res}), (\mathit{res}, \mathit{ttail}) \end{array} \right\} \right)}_{sh} \end{array} \right)\\
&= sh_4|_{-\mathit{ttail}} \cup (((sh \cup \{(\mathit{out}, \mathit{res})\})^*_{\mathit{res}})|_{-\mathit{res}})^*_{\mathit{out}}\\
&= sh_4|_{-\mathit{ttail}} \cup \left(\left\{ \begin{array}{c} (\mathit{out}, \mathit{out}), (\mathit{this}, \mathit{out}), (\mathit{this}, \mathit{this}), \\ (\mathit{ttail}, \mathit{this}), (\mathit{ttail}, \mathit{out}), (\mathit{ttail}, \mathit{ttail}) \end{array} \right\} \right)^*_{\mathit{out}}\\
&= sh_4|_{-\mathit{ttail}} \cup \left(\left\{ \begin{array}{c} (\mathit{out}, \mathit{out}), (\mathit{this}, \mathit{out}), (\mathit{this}, \mathit{this}), \\ (\mathit{ttail}, \mathit{this}), (\mathit{ttail}, \mathit{out}), (\mathit{ttail}, \mathit{ttail}) \end{array} \right\} \right) = sh_4.
\end{aligned}$$

The approximation sh_5 in Example 31 lets *out* (`clone`'s return value) share with itself (*i.e.,* it might be non-null), with *this* (`clone` performs a shallow clone of the `StudentList` *this*, by sharing the `Students`) and with *ttail* (because of the recursive call). You cannot drop any single pair from sh_5 without breaking the correctness of the analysis. Instead, $(\mathit{out}, \mathit{ttail})$ is redundant in sh_4. It is there since $\mathit{out}.\texttt{head} := \mathit{this}.\texttt{head}$ makes *out* share with *this* and $\mathit{ttail} := \mathit{this}.\texttt{tail}$ makes *ttail* share with *this* and hence, (too) conservatively, with *out*.

5.3 Correctness

The first result of correctness states that the abstract denotations are correct (Section 2) *w.r.t.* the concrete denotations.

Proposition 32. *The abstract denotations of Definitions 28 and 30 are correct.*

The concrete transformer on interpretations (Definition 10) induces an abstract transformer on sharing interpretations.

Definition 33. *Given a sharing interpretation I, we define a new sharing interpretation I' such that $I'(\kappa.\mathtt{m}) = \mathcal{SC}^{I}_{scope(\kappa.\mathtt{m})}[\![body(\kappa.\mathtt{m})]\!] \circ (\lambda sh \in \Sigma_{scope(\kappa.\mathtt{m})}.sh|_{out})$. The* sharing denotational semantics *of a program is the least fixpoint of this transformer on sharing interpretations.*

The following result follows from Proposition 32.

Proposition 34. *The transformer on sharing interpretations of Definition 33 is correct w.r.t. that on concrete interpretations of Definition 10. Hence, the sharing denotational semantics is a safe approximation of the denotational semantics.*

Example 35. Let us use, in Definition 33, the denotation of Example 31. We get an interpretation $I' = I$, hence a fixpoint of the transformer of Definition 33. We can actually *construct* I (Example 24) as the limit of a Kleene sequence of approximations, as usual in denotational abstract interpretation [7, 8]. Hence it is the *least* fixpoint *i.e.,* `clone`'s sharing denotational semantics.

5.4 Improving the Precision of Method Calls

The denotation for method calls of Definition 28 can be very imprecise.

Example 36. Let us remove the line *out*.`head := `*this*.`head` from Figure 1. The method `clone` builds now a `StudentList`, as long as *this*, but whose `Student`*s* are *null*. Hence, at the end of `clone`, variable *out* does not share with *this*. Let us verify if our analysis captures that, by re-executing what we did in Example 31.

$$
\begin{gathered}
sh_1 = \{(this, this)\}\\
\mathtt{out} := \mathtt{new\ StudentList}\\
sh_2 = \{(out, out), (this, this)\}\\
\mathtt{ttail} := \mathtt{this.tail}\\
sh_3 = \{(out, out), (this, this), (this, ttail), (ttail, ttail)\}\\
\mathtt{if\ ttail\ =\ null\ then}\ \{\}\ \mathtt{else\ out.tail\ :=\ ttail.clone()}\\
sh_4 = \{(out, out), (out, this), (this, this), (out, ttail), (this, ttail), (ttail, ttail)\}.
\end{gathered}
$$

Since $(out, this) \in sh_4$, our analysis is *not* able to guarantee that *this* does not share with the result of `clone`.

In Example 36, the problem is that, in order to approximate the recursive call *ttail*.`clone`(), we use a set $sh^\flat$ (Definition 28) which lets the parameters of the call share with its result, if they are not definitely *null*. In our example, $sh^\flat$ contains the spurious pair $(res, ttail)$, which by star-closure introduces further imprecisions, until $(out, this)$ is put in the approximation.

We can improve the precision of the analysis with explicit information on which actual parameters of a method call share with the return value. Hence we enlarge the set of the variables in the final states of the interpretations (Definitions 9 and 23). While $output(\kappa.\mathtt{m})$ provides information on *out* only (Definition 3), we use $output(\kappa.\mathtt{m}) \cup input'(\kappa.\mathtt{m})$ instead, where $input'(\kappa.\mathtt{m})$ are new local primed variables holding copies of the actual parameters of $\kappa.\mathtt{m}$. These variables are never modified, so that at the end they provide information on which actual parameters share with *out*, by renaming primed variables into unprimed ones: $sh^\flat = sh^\ddagger[\boldsymbol{v}' \mapsto \boldsymbol{v}]$ for the primed variables $\boldsymbol{v}'$.

Example 37. Let us re-execute the analysis of Example 36 with a primed variable $this'$. We use an interpretation I such that $I(\mathtt{clone})(\{(this, this)\}) = \{(out, out), (this', this')\}$ *i.e.,* at the end of `clone` the actual parameter passed for *this* does not share with the result of the method. We want to verify that this interpretation is a fixpoint of our semantics. We have

$$
\begin{gathered}
sh_1 = \{(this, this)\}\\
\mathtt{this}' := \mathtt{this} \quad // \ this' \text{ is initially aliased to } this\\
sh_2 = \{(this, this), (this, this'), (this', this')\}\\
\mathtt{out} := \mathtt{new}\ \mathtt{StudentList}\\
sh_3 = \{(out, out), (this, this), (this, this'), (this', this')\}\\
\mathtt{ttail} := \mathtt{this.tail}\\
sh_4 = \left\{\begin{matrix} (out, out), (this, this), (this, this'), (this', this'),\\ (ttail, this), (ttail, this'), (ttail, ttail)\end{matrix}\right\}\\
\mathtt{if}\ \mathtt{ttail} = \mathtt{null}\ \mathtt{then}\ \{\}\ \mathtt{else}\ \mathtt{out.tail} := \mathtt{ttail.clone}()\\
sh_5 = sh_4.
\end{gathered}
$$

We have $(out, this) \notin sh_5$ *i.e.,* our analysis guarantees now that *this* does not share with the result of `clone`. Note that $sh_5|_{\{out, this'\}} = \{(out, out), (this', this')\}$ *i.e., I* is a fixpoint of the transformer of Definition 33.

6 Conclusions

We have equipped our new abstract domain Sh for pair-sharing analysis with abstract operations which allow us to show a simple example of analysis (Example 31). We know that some of these operations are not optimal, so there is space for improvement. Moreover, we still miss an implementation and, hence, an actual evaluation. We plan to implement Sh as an abstract domain for the Julia analyser [15], for which we already implemented 8 other abstract domains. We will use binary decision diagrams [3] to represent the denotational transfer functions over Sh of Figures 3 and 4. Exceptions are automatically transformed by Julia into branches in the program's control-flow, so they can be easily embedded in our sharing analysis as we already did for other static analyses.

References

[1] R. Bagnara, P. M. Hill, and E. Zaffanella. Set-Sharing is Redundant for Pair-Sharing. *Theoretical Computer Science*, 277(1–2):3–46, April 2002.

[2] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-Semantics Approach: Theory and Applications. *Journal of Logic Programming*, 19/20:149–197, 1994.

[3] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[4] F. Bueno and M. J. García de la Banda. Set-Sharing Is Not Always Redundant for Pair-Sharing. In Y. Kameyama and P. J. Stuckey, editors, *Proc. of FLOPS'04*, volume 2998 of *Lecture Notes in Computer Science*, pages 117–131, Nara, Japan, April 2004. Springer-Verlag.

[5] J. D. Choi, M. Burke, and P. Carini. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. In *Proc. of the 20th Symposium on Principles of Programming Languages (POPL)*, pages 232–245, Charleston, South Carolina, January 1993. ACM.

[6] A. Cortesi and G. Filé. Abstract Interpretation of Logic Programs: An Abstract Domain for Groundness, Sharing, Freeness and Compoundness Analysis. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 52–61, Yale University, New Haven, Connecticut, USA, June 1991.

[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 the 4th ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[8] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[9] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In W. G. Olthoff, editor, *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, volume 952 of *LNCS*, pages 77–101, Århus, Denmark, August 1995. Springer.

[10] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent AND Parallelism. *Journal of Logic Programming*, 13(2 & 3):291–314, 1992.

[11] A. King. Pair-Sharing over Rational Trees. *Journal of Logic Programming*, 46(1–2):139–155, December 2000.

[12] V. Lagoon and P. J. Stuckey. Precise Pair-Sharing Analysis of Logic Programs. In *Proc. of the 4th international ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 99–108, Pittsburgh, PA, USA, October 2002. ACM.

[13] I. Pollet, B. Le Charlier, and A. Cortesi. Distinctness and Sharing Domains for Static Analysis of Java Programs. In *Proc. of the 25th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 77–98, Budapest, Hungary, June 2001.

[14] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. Available at `www.sci.univr.it/∼spoto/papers.html`, 2005.

[15] F. Spoto. The JULIA Static Analyser. `www.sci.univr.it/∼spoto/julia`, 2004.

[16] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proc. of the 23th ACM Symposium on Principles of Programming Languages (POPL)*, pages 32–41, St. Petersburg Beach, Florida, USA, January 1996.

[17] F. Tip and J. Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms. In *Proc. of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 35(10) of *SIGPLAN Notices*, pages 281–293, Minneapolis, Minnesota, USA, October 2000. ACM.

[18] G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, 1993.