Automatic Inference of Acyclicity
(long version with proofs⋆)

Samir Genaim[1] and Damiano Zanardini[2]

[1] DSIC, Complutense University of Madrid (UCM), Spain
[2] CLIP, DIA, Technical University of Madrid (UPM), Spain

Abstract. In programming languages with a dynamic use of memory, such as Java, knowing that a reference variable x points to an acyclic data structure is valuable for the analysis of *termination* and *resource usage* (e.g., execution time or memory consumption). For instance, this information guarantees that the *depth* of the data structure to which x points is greater than the depth of the data structure pointed to by x.f for any field f of x. This, in turn, allows bounding the number of iterations of a loop which traverses the structure by its depth, which is essential in order to prove the termination or infer the resource usage of the loop. The present paper develops an abstract-interpretation-based static analysis for inferring acyclicity, which works on the *reduced product* of two abstract domains: *reachability*, which models the property that the location pointed to by a variable w can be reached by dereferencing another variable v (in this case, v is said to reach w); and *cyclicity*, modelling the property that v can point to a cyclic data structure. Preliminary experiments within the COSTA analyzer show that the proposed analysis can prove termination and infer upper bounds on the resource usage of typical programs which could not be handled before.

1 Introduction

Programming languages with dynamic memory allocation, such as Java, allow creating and manipulating cyclic data structures. The presence of cyclic data structures in the program memory (the *heap*) is a challenging issue in the context of termination analysis [4,8,1,16], resource usage analysis [17,11,2], garbage collection [13], etc. Consider the loop "**while** (x!=**null**) **do** x:=x.next;". If x points to an acyclic data structure before the loop, then the depth of the data structure to which x points strictly decreases after each iteration; therefore, the number of iterations is bounded by the initial depth of (the structure pointed to by) x.

Automatic inference of such information is typically done by (1) *abstracting* the loop to a numeric loop "$while(x) \leftarrow \{x{>}0, x{>}x'\}, while(x')$"; and (2) bounding the number of iterations of the numeric loop. The numeric loop means that, if the loop entry is reached with x pointing to a data structure with depth $x > 0$,

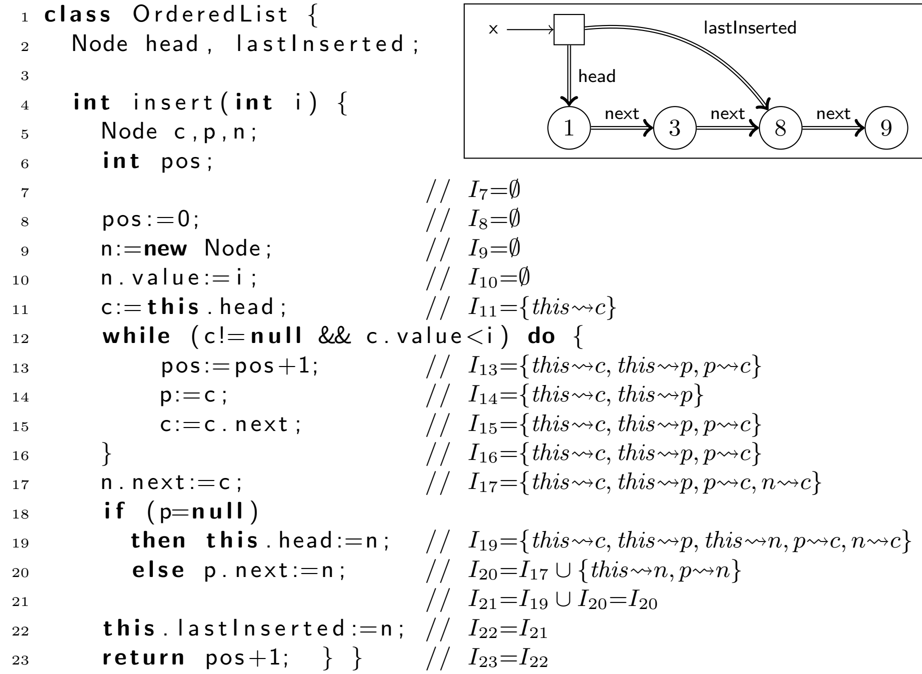
⋆ And a short NOTE TO THE REVIEWER: please see the proof of Theorem 1 in Section 4, and accept our apologies.

then it will eventually be reached again with x pointing to a structure with depth $x' < x$. The key point is that "x!=**null**" is abstracted to $x > 0$, meaning that the depth of a non-null variable cannot be 0; moreover, abstracting "x:=x.next" to $x > x'$ means that the depth decreases when accessing fields. While the former is valid for any structure, the latter holds only if x is acyclic. Therefore, acyclicity information is essential in order to apply such abstractions.

In mainstream programming languages with dynamic memory manipulation, data structures can only be modified by means of *field updates*. If, before x.f:=y, x and y are guaranteed to point to disjoint parts of the heap, then there is no possibility to create a cycle. On the other hand, if they are not disjoint, i.e., *share* a common part of the heap, then a cyclic structure might be created. This simple mechanism has been used in previous work [14] in order to declare x and y, among others, as cyclic whenever they share before the update. Such approach is simple and efficient. However there can be an important loss of precision in typical programming patterns. E.g., consider "y:=x.next.next;x.next:=y;" (which typically removes an element from a linked list), and let x be initially acyclic. After the first command, x and y clearly share, so that they should be declared as finally cyclic, even if, clearly, they are not. When considering x.f:=y, the precision of the acyclicity information can be improved if it is possible to know *how* x and y share: (1) x and y alias; (2) x reaches y; (3) y reaches x; (4) they both reach a common location. The field field update x:=y.f might create a cycle only in cases (1) and (3).

The present acyclicity analysis is based on the above observation. It defines an *abstract domain* $\mathcal{I}^{\tau}_{rc}$, which captures the reachability information about program variables (i.e., whether there can be a path in the heap from the location ℓ_v bound to some variable v and the location ℓ_w bound to some w), and the acyclicity of data structures (i.e., whether there can be a cyclic path starting from the location bound to some variable). A provably sound *abstract semantics* $\mathcal{C}^{\tau}_{\zeta}[\![_]\!](_)$ of a simple object-oriented language is developed, which works on $\mathcal{I}^{\tau}_{rc}$ and can often guarantee the acyclicity of *Directed Acyclic Graphs* (DAGs), which most likely will be considered as cyclic if only sharing, not reachability, is taken into account. The semantics has been implemented in the COSTA [3] COSt and Termination Analyzer as a component whose result is an essential information for proving the termination or inferring the resource usage of programs.

Related work. Sharing-based *cyclicity analysis* [14] is the most related work, and the source of inspiration for the present paper. As already discussed, in some cases, the sharing-based approach is less precise than the present analysis. *Shape analysis* [18] could also be used to infer acyclicity; however, it aims at inferring a much more general information about the heap. The present acyclicity analysis is simpler and tailored to a specific heap property, so that it can be remarkably simpler and more efficient. *Separation logic* has been used in recent work [6,7] to prove the termination of programs whose termination depends on the mutation of the heap. The main achievement of these works is that, in some cases, they prove the termination of programs which traverse cyclic data structures.




```
1  class OrderedList {
2    Node head, lastInserted;
3
4    int insert(int i) {
5      Node c,p,n;
6      int pos;
7                                   // I7=∅
8      pos:=0;                      // I8=∅
9      n:=new Node;                 // I9=∅
10     n.value:=i;                  // I10=∅
11     c:=this.head;                // I11={this⇝c}
12     while (c!=null && c.value<i) do {
13         pos:=pos+1;              // I13={this⇝c, this⇝p, p⇝c}
14         p:=c;                    // I14={this⇝c, this⇝p}
15         c:=c.next;               // I15={this⇝c, this⇝p, p⇝c}
16     }                            // I16={this⇝c, this⇝p, p⇝c}
17     n.next:=c;                   // I17={this⇝c, this⇝p, p⇝c, n⇝c}
18     if (p=null)
19       then this.head:=n;         // I19={this⇝c, this⇝p, this⇝n, p⇝c, n⇝c}
20       else p.next:=n;            // I20=I17 ∪ {this⇝n, p⇝n}
21                                  // I21=I19 ∪ I20=I20
22     this.lastInserted:=n;        // I22=I21
23     return pos+1;   } }          // I23=I22
```

Fig. 1. The running example and the analysis results (in comments).

A motivating example. Consider the program depicted in Fig. 1. The class OrderedList implements an *ordered linked list* with two fields: head and lastInserted point to, resp., the first element of the list and the last element which has been inserted. The class Node (not shown) implements a linked list in the standard way, with two fields value and next. The top-right corner of Fig. 1 shows a possible instance of OrderedList. The method insert adds a new element to the ordered list: it takes an integer i, creates a new node n for i, looks for the position pos of n, adds n to the list, makes lastInserted point to the new node, and finally returns pos. The goal is to infer that a call "x. insert (i)" never makes x cyclic. This is important since, when such call is involved in a loop like

```
x:=new OrderedList;
while (j>0) do { i:=read(); x.insert(i); j:=j−1; }
```

if x cannot be proven to be acyclic after insert, then it must be assumed to be cyclic from the second iteration on. This, in turn, prevents from proving termination of the loop at lines 12–16, since it might traverse a cycle.

The challenge in this example is to prove that the instructions at line 19 and 20 do not make any data structure cyclic. This is not trivial since **this**, p, and n share between each other at line 17; depending on how they share, the corresponding data structures might become cyclic or remain acyclic. Consider line 20: if there is a path (of length 0 or more) from n to p, then the data structures

bound to them become cyclic, while they remain acyclic in any other case. The present analysis is able to infer that n and p share before line 20, but n does not reach p, which, in turn, guarantees that no data structure ever becomes cyclic. It can be noted that reachability information is essential for proving acyclicity, since the mere information that p and n share, without knowing how they do, requires to consider then as possibly cyclic, as done, for example, in [14].

2 A simple object-oriented language

This section defines the syntax and the denotational semantics of a simplified version of Java. Class, method, field, and variable names are taken from a set $\mathcal{X}$ of valid *identifiers*. A *program* consists of a set of classes $\mathcal{K} \subseteq \mathcal{X}$ ordered by the *subclass* relation $\prec$. Following Java, a *class declaration* takes the form "**class** κ_1 [**extends** κ_2] { t_1 f_1;… t_n f_n; M_1 … M_k}" where each "t_i f_i" declares the field f_i to have type $t_i \in \mathcal{K} \cup \{\textbf{int}\}$, and each M_i is a method definition. A *method definition* takes the form "t $m(t_1\ w_1,\ldots,t_n\ w_n)$ $\{t_{n+1}\ w_{n+1};\ldots t_{n+p}\ w_{n+p};$ *com*}" where: $t \in \mathcal{K} \cup \{\textbf{int}\}$ is the type of the return value; $w_1, \ldots, w_n \in \mathcal{X}$ are the formal parameters; $w_{n+1}, \ldots, w_{n+p} \in \mathcal{X}$ are local variables; and *com* is a sequence of instructions according to the following grammar:

$$
\begin{aligned}
exp &::= n \mid \textbf{null} \mid v \mid v.f \mid exp_1 \oplus exp_2 \mid \textbf{new}\ \kappa \mid v.m(\bar{v}) \\
com &::= v{:=}exp \mid v.f{:=}exp \mid com_1\,;com_2 \mid \\
&\quad\ \textbf{if}\ exp\ \textbf{then}\ com_1\ \textbf{else}\ com_2 \mid \textbf{while}\ exp\ \textbf{do}\ com \mid \textbf{return}\ exp
\end{aligned}
$$

where $v, \bar{v}, m, f \in \mathcal{X}$; $n \in \mathbb{Z}$; $\kappa \in \mathcal{K}$; and $\oplus$ is a binary operator (boolean operators return 1 for **true** and 0 for **false**). For simplicity, *conditions* in **if** and **while** are assumed not to create objects or call methods. A *method signature* $\kappa.m(t_1, \ldots, t_n){:}t$ refers to a method m defined in class κ, taking n parameters of type $t_1, \ldots, t_n \in \mathcal{K} \cup \{\textbf{int}\}$, and returning a value of type t. Given a method signature m, let m^b be its code *com*; m^i its set of input variables $\{this, w_1, \ldots, w_n\}$; m^l its set of local variables $\{w_{n+1}, \ldots, w_{n+m}\}$; and $\mathsf{m}^s = \mathsf{m}^i \cup \mathsf{m}^l$.

A *type environment* τ is a partial map from $\mathcal{X}$ to $\mathcal{K} \cup \{\textbf{int}\}$ which associates types to variables at a given program point. Abusing notation, when the context is clear, type environments will be confused with sets of variables. A *state* over τ is a pair consisting of a frame and a heap. A *heap* μ is a partial mapping from an infinite set $\mathcal{L}$ of memory locations to objects; $\mu(\ell)$ is the object bound to $\ell \in \mathcal{L}$ in the heap μ. An *object* $o \in \mathcal{O}$ is a pair consisting of a class tag $o.\mathsf{tag} \in \mathcal{K}$, and a frame $o.\mathsf{frm}$ which maps its fields into $\mathcal{V} = \mathbb{Z} \cup \mathcal{L} \cup \{\textbf{null}\}$. Shorthands are used: $o.f$ for $o.\mathsf{frm}(f)$; $\mu[\ell \mapsto o]$ to modify the heap μ s.t. a new location ℓ points to object o; and $\mu[\ell.f \mapsto \mathrm{v}]$ to modify the value of the field f of the object $\mu(\ell)$ to $\mathrm{v} \in \mathcal{V}$. A *frame* ϕ maps variables in $\mathrm{dom}(\tau)$ to $\mathcal{V}$. For $v \in \mathrm{dom}(\tau)$, $\phi(v)$ refers to the value of v, and $\phi[v \mapsto \mathrm{v}]$ is the frame where the value of v has been set to v, or defined if $v \notin \mathrm{dom}(frm)$. The set of possible states over τ is

$$
\Sigma_\tau = \left\{ \langle \phi, \mu \rangle \,\middle|\, \begin{array}{l} 1.\ \phi \text{ is a frame over } \tau,\ \mu \text{ is a heap, and both are well-typed} \\ 2.\ \mathrm{rng}(\phi) \cap \mathcal{L} \subseteq \mathrm{dom}(\mu) \\ 3.\ \forall \ell \in \mathrm{dom}(\mu).\ \ \mathrm{rng}(\mu(\ell).\mathsf{frm}) \cap \mathcal{L} \subseteq \mathrm{dom}(\mu) \end{array} \right\}
$$

$$
\begin{array}{rl}
E^{\iota}_{\tau}[\![n]\!](\sigma) = & \langle\hat{\sigma}[\rho \mapsto n], \check{\sigma}\rangle \\
E^{\iota}_{\tau}[\![\textsf{null}]\!](\sigma) = & \langle\hat{\sigma}[\rho \mapsto \textsf{null}], \check{\sigma}\rangle \\
E^{\iota}_{\tau}[\![\textsf{new}\ \kappa]\!](\sigma) = & \langle\hat{\sigma}[\rho \mapsto \ell], \check{\sigma}[\ell \mapsto newobj(\kappa)]\rangle \text{ where } \ell \notin \mathrm{dom}(\check{\sigma}) \\
E^{\iota}_{\tau}[\![v]\!](\sigma) = & \langle\hat{\sigma}[\rho \mapsto \hat{\sigma}(v)], \check{\sigma}\rangle \\
E^{\iota}_{\tau}[\![v.f]\!](\sigma) = & \langle\hat{\sigma}[\rho \mapsto \check{\sigma}(\hat{\sigma}(v)).f], \check{\sigma}\rangle \\
E^{\iota}_{\tau}[\![exp_1 \oplus exp_2]\!](\sigma) = & \langle\hat{\sigma}[\rho \mapsto \hat{\sigma}_1(\rho) \oplus \hat{\sigma}_2(\rho)], \check{\sigma}_2\rangle \text{ where} \\
 & \quad \sigma_1 = E^{\iota}_{\tau}[\![exp_1]\!](\sigma) \text{ and } \sigma_2 = E^{\iota}_{\tau}[\![exp_1]\!](\langle\hat{\sigma}, \check{\sigma}_1\rangle) \\
E^{\iota}_{\tau}[\![v_0.m(v_1,\ldots,v_n)]\!](\sigma) = & \langle\hat{\sigma}[\rho \mapsto \hat{\sigma}_2(out)], \check{\sigma}_2\rangle \text{ where } \sigma_2 = \iota(\mathsf{m})(\sigma_1) \text{ s.t. } \sigma_1 \text{ is} \\
 & \check{\sigma}_1 = \check{\sigma}; \hat{\sigma}_1(this) = \hat{\sigma}(v_0); \forall 1 \le i \le n.\ \hat{\sigma}_1(w_i) = \hat{\sigma}(v_i); \\
 & \text{and } \mathsf{m} = lkp(\sigma, v_0.m(v_1,\ldots,v_n)); \\
\hline
C^{\iota}_{\tau}[\![v{:=}exp]\!](\sigma) = & \langle\hat{\sigma}[v \mapsto \hat{\sigma}_e(\rho)], \check{\sigma}_e\rangle \\
C^{\iota}_{\tau}[\![v.f{:=}exp]\!](\sigma) = & \langle\hat{\sigma}, \check{\sigma}[\ell.f \mapsto \hat{\sigma}_e(\rho)]\rangle \text{ where } \ell = \hat{\sigma}(v) \\
C^{\iota}_{\tau}\left[\!\!\left[\begin{array}{l}\textsf{if}\ exp\ \textsf{then}\ com_1 \\ \quad\textsf{else}\ com_2\end{array}\right]\!\!\right](\sigma) = & \text{if } \hat{\sigma}_e(\rho) \neq 0 \text{ then } C^{\iota}_{\tau}[\![com_1]\!](\sigma) \text{ else } C^{\iota}_{\tau}[\![com_2]\!](\sigma) \\
C^{\iota}_{\tau}[\![\textsf{while}\ exp\ \textsf{do}\ com]\!](\sigma) = & \delta(\sigma) \text{ where } \delta \text{ is the least fixpoint of} \\
 & \lambda w.\lambda\sigma.\ \text{if } \hat{\sigma}_e(\rho) \neq 0 \text{ then } w(C^{\iota}_{\tau}[\![com]\!](\sigma)) \text{ else } \sigma \\
C^{\iota}_{\tau}[\![\textsf{return}\ exp]\!](\sigma) = & \langle\hat{\sigma}[out \mapsto \hat{\sigma}_e(\rho)], \check{\sigma}_e\rangle \\
C^{\iota}_{\tau}[\![com_1; com_2]\!](\sigma) = & C^{\iota}_{\tau}[\![com_2]\!](C^{\iota}_{\tau}[\![com_1]\!](\sigma))
\end{array}
$$

Fig. 2. Denotations for expressions and commands. The state σ_e is $E^{\iota}_{\tau}[\![exp]\!](\sigma)$.

Given $\sigma \in \Sigma_\tau$, $\hat{\sigma}$ and $\check{\sigma}$ refer to its frame and its heap, respectively. The complete lattice $\mathcal{I}^{\tau}_{\flat} = \langle\wp(\Sigma_\tau), \Sigma_\tau, \emptyset, \cap, \cup\rangle$ defines the *concrete computation domain.*

A *denotation* δ over two type environments τ_1 and τ_2 is a partial map from Σ_{τ_1} to Σ_{τ_2}: it basically describes how the state changes when a piece of code is executed. The set of denotations from τ_1 to τ_2 is $\Delta(\tau_1,\tau_2)$. *Interpretations* are special denotations which give a meaning to methods in terms of their input and output variables. An interpretation $\iota \in \Gamma$ maps methods to denotations, and is such that $\iota(\mathsf{m}) \in \Delta(\mathsf{m}^i, \{out\})$ for each signature m in the program.

Denotations for expressions and commands are depicted in Fig 2. An expression denotation $E^{\iota}_{\tau}[\![exp]\!]$ maps states from Σ_τ to states from $\Sigma_{\tau\cup\{\rho\}}$, where ρ is a special variable for storing the expression value. A command denotation $C^{\iota}_{\tau}[\![com]\!]$ maps states to states, in presence of $\iota \in \Gamma$. The function $newobj(\kappa)$ creates a new instance of the class κ with integer (resp. reference) fields initialized to 0 (resp. **null**). The function *lkp* resolves the method call and returns the signature of the method to be called. The *concrete denotational semantics* of a program is defined as the least fixpoint of the following transformer of interpretations [5].

Definition 1. *The denotational semantics of a program P is the* lfp *of the operator* $T_P(\iota) = \{\mathsf{m} \mapsto \lambda\sigma{\in}\Sigma_{\tau_1}.\exists\tau\backslash out.C^{\iota}_{\tau_2}[\![\mathsf{m}^b]\!](extend(\sigma,\mathsf{m})) \mid \mathsf{m} \in P\}$ *where* $extend(\sigma,\mathsf{m}) = \langle\hat{\sigma}[\forall v \in \mathsf{m}^l \cup \{out\}.v \mapsto 0/\textsf{null}], \check{\sigma}\rangle$*,* $\tau_1 = \mathsf{m}^i$*, and* $\tau_2 = \mathsf{m}^s \cup \{out\}$*.*

The denotation for a method signature $\mathsf{m} \in P$ computed by the above operator as follows: (1) it extends (using $extend(\sigma,\mathsf{m})$) the input state $\sigma{\in}\Sigma_{\mathsf{m}^i}$ s.t. local variables are set to 0 or **null**, depending on their type; (2) it computes the denotation of the code of m (using $C^{\iota}_{\mathsf{m}^s}[\![\mathsf{m}^b]\!]$); and (3) it restricts the resulting denotation to the output variable *out* (using $\exists\tau\backslash out$).

3 The abstract domain

The acyclicity analysis discussed in this paper works on the reduced product [10] of two abstract domains, according to the theory of Abstract Interpretation [9]. The first domain captures *may-reachability*, while the second deals with the *may-be-cyclic* property of variables. Both are based on the notion of *reachable heap locations*, i.e., the part of the heap which can be reached from a location.

Definition 2 (reachable heap locations [14]). *Given a heap μ, the set of reachable locations from $\ell \in \mathrm{dom}(\mu)$ is $R(\mu,\ell){=}\cup\{R^i(\mu,\ell) \mid i \geq 0\}$, where $R^0(\mu,\ell){=}\mathrm{rng}(\mu(\ell).\mathsf{frm}){\cap}\mathcal{L}$, and $R^{i+1}(\mu,\ell){=}\cup\{\mathrm{rng}(\mu(\ell').\mathsf{frm}){\cap}\mathcal{L} \mid \ell' \in R^i(\mu,\ell)\}$. The set of ε-reachable locations from $\ell \in \mathrm{dom}(\mu)$ is $R^\varepsilon(\mu,\ell){=}R(\mu,\ell) \cup \{\ell\}$.*

Note that ε-reachable locations include the source location ℓ itself, while reachable locations do not (unless ℓ is reachable from itself through a cycle). The rest of this section is developed in the context of a type environment τ.

Reachability. Given a state $\sigma \in \Sigma_\tau$, a reference variable $v \in \tau$ is said to *reach* $w \in \tau$ in σ if $\hat{\sigma}(w) \in R(\check{\sigma}, \hat{\sigma}(v))$. This means that, starting from v and applying *at least one dereference operation*, it is possible to reach the object to which w points. Due to strong typing, τ puts some restrictions on reachability; i.e., it might not be possible to have a heap where a variable of type κ_1 reaches one of type κ_2. Following [15], a class $\kappa_2 \in \mathcal{K}$ is said to be reachable from $\kappa_1 \in \mathcal{K}$ if there exists $\sigma \in \Sigma_\tau$, and two locations $\ell, \ell' \in \mathrm{dom}(\check{\sigma})$ s.t. (a) $\check{\sigma}(\ell).\mathsf{tag} = \kappa_1$; (b) $\check{\sigma}(\ell').\mathsf{tag} = \kappa_2$; and (c) $\ell' \in R(\check{\sigma}, \ell)$.

Definition 3 (reachability domain). *The reachability abstract domain is the complete lattice $\mathcal{I}_r^\tau = \langle \wp(\mathcal{R}^\tau), \subseteq, \emptyset, \mathcal{R}^\tau, \cap, \cup \rangle$ where $\mathcal{R}^\tau = \{v{\rightsquigarrow}w \mid v, w \in \mathrm{dom}(\tau)$ the class $\tau(w)$ is reachable from the class $\tau(v)\}$.*

May-reach information is described by *abstract values* $I_r \in \wp(\mathcal{R}^\tau)$. For example, $\{\mathsf{x}{\rightsquigarrow}\mathsf{z}, \mathsf{y}{\rightsquigarrow}\mathsf{z}\}$ describes those states where x and y *may* reach z. Note that a statement $\mathsf{x}{\rightsquigarrow}\mathsf{y}$ does not prevent x and y from aliasing; instead, x can reach y and alias with it at the same time, e.g., when x, y, and x.f point to the same location.

Lemma 1. *The following abstraction and concretization functions define a Galois insertion between $\mathcal{I}_r^\tau$ and $\mathcal{I}_\flat^\tau$:*

$$\alpha_r^\tau(I_\flat) = \{v{\rightsquigarrow}w \in \mathcal{R}^\tau \mid \exists \sigma \in I_\flat . v \textit{ reaches } w \textit{ in } \sigma\}$$
$$\gamma_r^\tau(I_r) = \{\sigma \in \Sigma_\tau \mid \ \forall v, w \in \tau.\ v \textit{ reaches } w \textit{ in } \sigma \Rightarrow v{\rightsquigarrow}w \in I_r\}$$

Proof. Due to the definition of Galois insertion, the result to prove amounts to say that both

$$\begin{array}{rl} (a) & \forall I_r \in \mathcal{I}_r^\tau .\ \alpha_r^\tau(\gamma_r^\tau(I_r)) = I_r \\ \text{and } (b) & \forall I_\flat \in \mathcal{I}_\flat^\tau .\ \gamma_r^\tau(\alpha_r^\tau(I_\flat)) \supseteq I_\flat \end{array}$$

hold, where $\subseteq$ is the ordering on $\mathcal{I}_\flat^\tau$. Part (a) is first proven. By simply applying the definitions of α_r^τ and γ_r^τ, the abstract value $\alpha_r^\tau(\gamma_r^\tau(I_r))$ comes to be

$$\left\{ v{\rightsquigarrow}w \in \mathcal{R}^\tau \mid \exists \sigma \in \Sigma_\tau . \begin{pmatrix} \forall v', w' \in \tau.\ (v' \text{ reaches } w' \text{ in } \sigma \Rightarrow v'{\rightsquigarrow}w' \in I_r) \\ \wedge (v \text{ reaches } w \text{ in } \sigma) \end{pmatrix} \right\}$$

In order to prove the equality, it is enough to show that, for every variables v and w, the statement $v \rightsquigarrow w$ belongs to $\alpha_r^\tau(\gamma_r^\tau(I_r))$ if and only if it belongs to I_r. In fact, by definition of α_r^τ and γ_r^τ, the formula $v \rightsquigarrow w \in \alpha_r^\tau(\gamma_r^\tau(I_r))$ holds iff

$$\exists \sigma. \begin{pmatrix} \forall v', w' \in \tau.\ (v' \text{ reaches } w' \text{ in } \sigma \Rightarrow v' \rightsquigarrow w' \in I_r) \\ \wedge (v \text{ reaches } w \text{ in } \sigma) \end{pmatrix}$$

which, in turn, is logically equivalent to $v \rightsquigarrow w \in I_r$. The last equivalence follows from the observation that, (i) due to the notion of class reachability, for every v and w, there exists a state $\sigma \in \Sigma_\tau$ where v reaches w (otherwise, $v \rightsquigarrow w$ could never be contained in any abstract value); and (ii) such σ can be chosen such that, for any other pair of variables v' and w', v' does not reach w' in σ. Thus, the equality between I_r and $\alpha_r^\tau(\gamma_r^\tau(I_r))$ is proven.

The proof of part (b) goes as follows. The set of states $\gamma_r^\tau(\alpha_r^\tau(I_\flat))$ comes to be

$$\{\sigma \mid \forall v, w. (v \text{ reaches } w \text{ in } \sigma \ \Rightarrow\ \exists \sigma' \in I_\flat. v \text{ reaches } w \text{ in } \sigma')\}$$

The next step is to prove that any σ belonging to $I_\flat$ also belongs to $\gamma_r^\tau(\alpha_r^\tau(I_\flat))$. For every pair of variables v and w, there are two possibilities: (i) v reaches w in σ; or (ii) v does not reach w in σ. It is easy to see that any $\sigma \in I_\flat$ also belongs to $\gamma_r^\tau(\alpha_r^\tau(I_\flat))$ because, for every v and w, it satisfies the implication

$$v \text{ reaches } w \text{ in } \sigma \ \Rightarrow\ \exists \sigma' \in I_\flat. v \text{ reaches } w \text{ in } \sigma'$$

In fact, in case (i), the implication is true since σ is exactly the σ' required by the existential quantifier. In case (ii), the implication is true because the left-hand side is false. This proves the Lemma. □

The top element $\mathcal{R}^\tau$ is $\alpha_r^\tau(\Sigma_\tau)$, and represents all states which are compatible with τ. The bottom element $\emptyset$ models the set of all states where, for every two reference variables v and w (possibly the same variable), v does not reach w.

Remark 1. Intuitively, reachability is a transitive property; i.e., if x reaches y and y reaches z, then x also reaches z. However, values in $\mathcal{I}_r^\tau$ are *not* closed by transitivity: e.g., it is possible to have $I_r = \{\mathsf{x} \rightsquigarrow \mathsf{y}, \mathsf{y} \rightsquigarrow \mathsf{z}\}$ which contains $\mathsf{x} \rightsquigarrow \mathsf{y}$ and $\mathsf{y} \rightsquigarrow \mathsf{z}$, but not $\mathsf{x} \rightsquigarrow \mathsf{z}$. Such abstract value is a reasonable one, and approximates, for example, the execution of "x=**new** C; y=**new** C; **if** (w>0) **then** x.f=y; **else** y.f=z;". Moreover, this abstract value is consistent, i.e., describes a set of concrete states which is not smaller (actually, greater) than $\gamma_r^\tau(\emptyset)$. This happens because reachability is, actually, *may-reach* information, so that, for example, $\gamma_r^\tau(\{\mathsf{x} \rightsquigarrow \mathsf{y}, \mathsf{y} \rightsquigarrow \mathsf{z}\})$ includes (a) any state where x reaches y but y does not reach z; (b) any state where y reaches z but x does not reach y; and (c) any state where x does not reach y and y does not reach z. Importantly, $\gamma_r^\tau(\{\mathsf{x} \rightsquigarrow \mathsf{y}, \mathsf{y} \rightsquigarrow \mathsf{z}\})$ does not contain those states where both x reaches y and y reaches z, since, in this case, x would also reach z by transitivity, which is forbidden since $\mathsf{x} \rightsquigarrow \mathsf{z} \notin I_r$.

Cyclicity. Given a state $\sigma \in \Sigma_\tau$, a variable $v \in \mathrm{dom}(\tau)$ is said to be *cyclic* in σ if there exists $\ell \in R^\varepsilon(\check{\sigma}, \hat{\sigma}(v))$ such that $\ell \in R(\check{\sigma}, \ell)$. In other words, v is cyclic if it reaches some memory location ℓ (which can possibly be $\hat{\sigma}(v)$ itself) through which a cyclic path goes. Similarly to reachability, it might be impossible to generate a cyclic data structure starting from a variable of some type κ. Following [14], a class $\kappa \in \mathcal{K}$ is said to be a *cyclic class* if there exists $\sigma \in \Sigma_\tau$, and $\ell, \ell' \in \mathrm{dom}(\check{\sigma})$, such that $\check{\sigma}(\ell).\mathsf{tag} = \kappa$, $\ell' \in R^\varepsilon(\check{\sigma}, \ell)$, and $\ell' \in R(\check{\sigma}, \ell')$. The cyclicity domain is the dual of the non-cyclicity domain of [14].

Definition 4 (cyclicity domain). *The abstract domain for cyclicity is represented as the complete lattice* $\mathcal{I}_c^\tau = \langle \wp(\mathcal{Y}^\tau), \subseteq, \emptyset, \mathcal{Y}^\tau, \cap, \cup \rangle$ *where* $\mathcal{Y}^\tau = \{ \circlearrowleft^v \mid v \in \tau, \tau(v)$ *is a cyclic class*$\}$.

Lemma 2. *The following abstraction and concretization functions define a Galois insertion between* $\mathcal{I}_c^\tau$ *and* $\mathcal{I}_\flat^\tau$

$$\alpha_c^\tau(I_\flat) = \{ \circlearrowleft^v \mid \exists v \in \tau.\ \exists \sigma \in I_\flat.\ v \text{ is cyclic in } \sigma \}$$
$$\gamma_c^\tau(I_c) = \{ \sigma \mid \sigma \in \Sigma_\tau \wedge \forall v \in \tau.\ (v \text{ is cyclic in } \sigma) \Rightarrow \circlearrowleft^v \in I_c \}$$

Proof. Very similar to the proof of Lemma 1. □

May-be-cyclic information is described by *abstract values* $I_c \in \wp(\mathcal{Y}^\tau)$. E.g., $\{\circlearrowleft^{\mathsf{x}}\}$ represents states where no variable but x can be cyclic. The top element $\mathcal{Y}^\tau$ corresponds to Σ_τ; the bottom $\emptyset$ does not allow any variable to be cyclic.

The reduced product. As explained in Sec. 4, the abstract semantics uses reachability information in order to detect cycles, and cyclicity information in order to add, in some cases, reachability statements. Therefore, it makes sense to combine both kinds of information: in the theory of Abstract Interpretation, this amounts to computing the *reduced product* [10] of the corresponding abstract domains. In the present context, the reduced product can be computed by *reducing* the Cartesian product $\mathcal{I}_{rc}^\tau = \mathcal{I}_r^\tau \times \mathcal{I}_c^\tau$. Elements of $\mathcal{I}_{rc}^\tau$ are pairs $\langle I_r, I_c \rangle$, where I_r and I_c contain, respectively, the may-reach and the may-be-cyclic information. The abstraction and concretization functions are induced by those of $\mathcal{I}_c^\tau$ and $\mathcal{I}_r^\tau$:

$$\gamma_{rc}^\tau(\langle I_r, I_c \rangle) = \gamma_r^\tau(I_r) \cap \gamma_c^\tau(I_c) \qquad \alpha_{rc}^\tau(I) = \langle \alpha_r^\tau(I), \alpha_c^\tau(I) \rangle$$

However, it can happen that two elements of $\mathcal{I}_{rc}^\tau$ are mapped to the same concrete element, which prevents having a Galois insertion between $\mathcal{I}_{rc}^\tau$ and $\mathcal{I}_\flat^\tau$. Computing the reduced product deals exactly with this problem. In order to compute it, an equivalence relation $\equiv$ has to be defined, which satisfies $I_{rc}^1 \equiv I_{rc}^2$ iff $\gamma_{rc}^\tau(I_{rc}^1) = \gamma_{rc}^\tau(I_{rc}^2)$. Functions γ_{rc}^τ and α_{rc}^τ define a Galois insertion between $\mathcal{I}_{rc\equiv}^\tau$ and $\mathcal{I}_\flat^\tau$, where $\mathcal{I}_{rc\equiv}^\tau$ is $\mathcal{I}_{rc}^\tau$ equipped (reduced) with the equivalence relation. The following lemma characterizes the equivalence relation on $\mathcal{I}_{rc}^\tau$.

Lemma 3. *For every* $I_r^1, I_r^2 \in \mathcal{I}_r^\tau$ *and* $I_c^1, I_c^2 \in \mathcal{I}_c^\tau$, *the concretization* $\gamma_{rc}^\tau(\langle I_r^1, I_c^1 \rangle)$ *is equal to* $\gamma_{rc}^\tau(\langle I_r^2, I_c^2 \rangle)$ *if and only if both conditions hold: (a)* $I_c^1 = I_c^2$*; and (b)* $I_r^1 \setminus \{ v \rightsquigarrow v \mid \circlearrowleft^v \notin I_c^1 \} = I_r^2 \setminus \{ v \rightsquigarrow v \mid \circlearrowleft^v \notin I_c^2 \}$.

Proof. The proof only deals with the first direction of the "if and only if":

$$\begin{array}{c} \gamma^{\tau}_{rc}(\langle I^1_r, I^1_c\rangle){=}\gamma^{\tau}_{rc}(\langle I^2_r, I^2_c\rangle) \\ \Rightarrow \\ I^1_c{=}I^2_c \qquad \wedge \qquad (I^1_r \setminus \{v{\rightsquigarrow}v \mid \circlearrowleft^v \notin I^1_c\}){=}(I^2_r \setminus \{v{\rightsquigarrow}v \mid \circlearrowleft^v \notin I^2_c\}) \end{array}$$

The other direction is straightforward. By using the logical fact that $F \Rightarrow (G{\wedge}H)$ is equivalent to $(\neg G \Rightarrow \neg F) \wedge (\neg H \Rightarrow \neg F)$ (where F, G, and H are the three atoms appearing in the logical formula above, in the same order), the proof can go by contradiction, and consists of two parts:

1. proving that $I^1_c \neq I^2_c$ implies $\gamma(\langle I^1_r, I^1_c\rangle) \neq \gamma(\langle I^2_r, I^2_c\rangle)$;
2. proving that $(I^1_r \setminus \{v{\rightsquigarrow}v | \circlearrowleft^v{\notin}I^1_c\}) \neq (I^2_r \setminus \{v{\rightsquigarrow}v | \circlearrowleft^v{\notin}I^2_c\})$ implies $\gamma(\langle I^1_r, I^1_c\rangle) \neq \gamma(\langle I^2_r, I^2_c\rangle)$.

The proof goes as follows.

1. Suppose $I^1_c \neq I^2_c$, and let $X_1 = \{v \mid \circlearrowleft^v \in I^1_c \setminus I^2_c\}$, and $X_2 = \{v \mid \circlearrowleft^v \in I^2_c \setminus I^1_c\}$. Note that, by hypothesis, at least one between X_1 and X_2 must be non-empty. For $i \in \{1, 2\}$, let σ_i be a state where
   (a) every $v \in X_i$ is cyclic, but does not reach itself, and no other variable is cyclic; and
   (b) no variables reach any variables, i.e., $\alpha^{\tau}_r(\{\sigma_i\}) = \emptyset$. Note that this requirement is consistent, since the cyclicity of some variables (in this case, those in X_i) does not imply the existence of any reachability path between variables.

   It is easy to see that σ_1 and σ_2 both belong to $\gamma_r(I^1_r) \cap \gamma_r(I^2_r)$; therefore, if $X_1 \neq \emptyset$, then σ_1 belongs to $\gamma(\langle I^1_r, I^1_c\rangle)$, but not to $\gamma(\langle I^2_r, I^2_c\rangle)$, since $\langle I^2_r, I^2_c\rangle$ does not allow the cyclicity on variables from X_1. Dually, if $X_2 \neq \emptyset$, then σ_2 belongs to $\gamma(\langle I^2_r, I^2_c\rangle)$ but not to $\gamma(\langle I^1_r, I^1_c\rangle)$.
2. Suppose $R_1 = I^1_r \setminus \{v{\rightsquigarrow}v \mid \circlearrowleft^v{\notin}I^1_c\}$ is different from $R_2 = I^2_r \setminus \{v{\rightsquigarrow}v \mid \circlearrowleft^v{\notin}I^2_c\}$, and let $S_1 = R_1 \setminus R_2$ and $S_2 = R_2 \setminus R_1$. Note that at least one between S_1 and S_2 is non-empty. If S_1 is not empty, then let $p \in S_1$ be one of the statements which are not in R_2. A state σ_1 can be chosen such that
   (a) if $p = v{\rightsquigarrow}v$, then v is the only cyclic variable (note that the cyclicity of v must be allowed by I^1_c since, otherwise, p would not be included in X_1); otherwise, if p is some $v{\rightsquigarrow}w$ with $v \neq w$, then no variables can be cyclic; and
   (b) let $p = v{\rightsquigarrow}w$; then, v must reach w in σ_1, and no other variable reaches any other variable.

   Clearly, such state belongs to $\gamma(\langle I^1_r, I^1_c\rangle)$, but not to $\gamma(\langle I^2_r, I^2_c\rangle)$. Dually, if S_2 is empty, then S_1 cannot be empty, and, with a similar reasoning, a state σ_2 can be found which belongs to $\gamma(\langle I^2_r, I^2_c\rangle)$, but not to $\gamma(\langle I^1_r, I^1_c\rangle)$.

As for the other direction of the proof:

$$\begin{array}{c} I^1_c{=}I^2_c \qquad \wedge \qquad (I^1_r \setminus \{v{\rightsquigarrow}v \mid \circlearrowleft^v \notin I^1_c\}){=}(I^2_r \setminus \{v{\rightsquigarrow}v \mid \circlearrowleft^v \notin I^2_c\}) \\ \Rightarrow \\ \gamma^{\tau}_{rc}(\langle I^1_r, I^1_c\rangle){=}\gamma^{\tau}_{rc}(\langle I^2_r, I^2_c\rangle) \end{array}$$

it follows easily from observing that, under the hypothesis written in the first line of the formula, the only difference between $\langle I_r^1, I_c^1\rangle$ and $\langle I_r^2, I_c^2\rangle$ is that $\langle I_r^1, I_c^1\rangle$ may contain some statements $v{\leadsto}v$ for variables v such that $\circlearrowleft^v \notin I_c^1$, and $\langle I_r^2, I_c^2\rangle$ may contain some (different) statements $v{\leadsto}v$ for variables v such that or $\circlearrowleft^v \notin I_c^2$. However, adding such statements to both abstract values does not change the set of concrete states they represent, since the possibility that v reaches itself in a concrete state contradicts the lack of a $\circlearrowleft^v$ statement. In other words, there is no concrete state which belongs to $\gamma_{rc}^{\tau}(\langle I_r^1, I_c^1\rangle)$ or $\gamma_{rc}^{\tau}(\langle I_r^2, I_c^2\rangle)$ without also belonging to $\gamma_{rc}^{\tau}(\langle I_r^1 \setminus \{v{\leadsto}v|\circlearrowleft^v \notin I_c^1\},\ I_c^1\rangle)$ and $\gamma_{rc}^{\tau}(\langle I_r^2 \setminus \{v{\leadsto}v|\circlearrowleft^v \notin I_c^2\},\ I_c^2\rangle)$ (which, by hypothesis, are the same set). □

The above lemma means that: (a) may-be-cyclic information always makes a difference as regards the set of concrete states; that is, adding a new statement $\circlearrowleft^v$ to $I_{rc} \in \mathcal{I}_{rc}^{\tau}$ results in representing a larger set of states; and (b) adding a pair $v{\leadsto}v$ to $I_{rc} \in \mathcal{I}_{rc}^{\tau}$, when v cannot be cyclic, does not make it represent more concrete states, since the acyclicity of v excludes that it can reach itself.

Example 1. As an example for (a), consider $I_{rc}^1 = \langle I_r, \emptyset\rangle$ and $I_{rc}^2 = \langle I_r, \{\circlearrowleft^{\mathsf{x}}\}\rangle$ which results from adding $\circlearrowleft^{\mathsf{x}}$ to I_{rc}^1. Assuming that x does not appear in I_r, there is a state σ which is compatible with I_r (for example, if no v reaches w in σ), and where x is cyclic (note that this does not require x to reach any other variable). This σ belongs to $\gamma_{rc}^{\tau}(I_{rc}^2) \setminus \gamma_{rc}^{\tau}(I_{rc}^1)$. As an example for (b), consider $I_{rc}^1 = \langle \emptyset, \{\circlearrowleft^{\mathsf{y}}\}\rangle$ and $I_{rc}^2 = \langle \{\mathsf{x}{\leadsto}\mathsf{x}\}, \{\circlearrowleft^{\mathsf{y}}\}\rangle$ which results from adding $\mathsf{x}{\leadsto}\mathsf{x}$ to I_{rc}^1. At a first glance, I_{rc}^2 describes a larger set of states, since it includes states (not belonging to $\gamma_{rc}^{\tau}(I_{rc}^1)$) where there is a path from x to x. However, such states will neither belong to $\gamma_{rc}^{\tau}(I_{rc}^2)$, since such a path implies that x is cyclic, which is not permitted by $\{\circlearrowleft^{\mathsf{y}}\}$, that only allows y to be cyclic.

Lemma 3 provides a way for computing the *normal form* of any $\langle I_r, I_c\rangle$, which comes to be $\langle I_r \setminus \{v{\leadsto}v|\circlearrowleft^v \notin I_c\}, I_c\rangle$, i.e., the *canonical form* of its equivalence class. From now on, $\mathcal{I}_{rc}^{\tau}$ will be a shorthand for $\mathcal{I}_{rc\equiv}^{\tau}$, where $\equiv$ is left implicit.

4 Reachability-based acyclicity analysis

This section uses $\mathcal{I}_{rc}^{\tau}$ to define an abstract semantics from which one can decide if a variable v is bounded to an acyclic data structure at a given program point. The analysis is based on the observation that reachability information can tell *how* two variables v and w share: they are said to share in a state σ if they reach a common heap location. This can happen because either (a) v and w alias; (b) v reaches w; (c) w reaches v; or (d) they both reach $\ell \in \mathrm{dom}(\check{\sigma})$. Distinguishing among the above scenarios is crucial for a precise acyclicity analysis. In fact, assuming that v and w are initially acyclic, they become cyclic after executing $v.\mathsf{f}{:=}w$ *if and only if* w initially reaches v or aliases with it. This is clearly more precise than declaring v as cyclic whenever it was sharing with w [14]. The rest of this section defines and explains the analysis.

Possible sharing, *possible aliasing* and *purity* analysis are used as pre-existent components, i.e., programs are assumed to have been analyzed w.r.t. these properties. Two reference variables v and w *share* in σ iff $R^{\varepsilon}(\check{\sigma},\hat{\sigma}(v))\cap R^{\varepsilon}(\check{\sigma},\hat{\sigma}(w))\neq\emptyset$; also, they *alias* in σ if they point to the same location, namely, if $\hat{\sigma}(v)=\hat{\sigma}(w)\in\text{dom}(\check{\sigma})$. Importantly, any non-null reference variable shares and aliases with itself; also, both are symmetric relations (i.e., $\langle v{\bullet}w\rangle$ iff $\langle w{\bullet}v\rangle$, and $\langle v{\cdot}w\rangle$ iff $\langle w{\cdot}v\rangle$) The i-th argument of a method m is said to be *pure* if m does not update the data structure to which the argument initially points. For *sharing* and *purity*, the analysis described in [12] (based on [15]) is applied: with it, (1) it is possible to know if v might share with w at any program point (denoted by the sharing pair proposition $\langle v{\bullet}w\rangle$); and (2) for each method m, a denotation SH_{m} is given: for a set of pairs sh which safely describes the sharing between actual arguments in the input state, $sh'=\mathsf{SH}_{\mathsf{m}}(I)$ is such that (i) if $\langle v{\bullet}w\rangle\in sh'$, then v and w might share during the execution of m; and (ii) $\dot{v}_i\in sh'$ means that the i-th argument might be non-pure. As for *aliasing*, it is assumed that, at each program point, the pair $\langle v{\cdot}w\rangle$ tells if v and w can alias.

An abstract element $\langle I_r,I_c\rangle\in\mathcal{I}^{\tau}_{rc}$ will be represented by the set $I=I_r\cup I_c$; therefore, $v{\rightsquigarrow}w\in I$ and $\circlearrowleft^{v}\in I$ are shorthands for, resp., $v{\rightsquigarrow}w\in I_r$ and $\circlearrowleft^{v}\in I_c$. The operation $\exists v.I$ (*projection*) removes any statement about v from I, while $I[v/w]$ (*renaming*) v to w in I. For the sake of simplicity, class-reachability and class-cyclicity are taken into account *implicitly*: a new statement $v{\rightsquigarrow}w$ (resp., $\circlearrowleft^{v}$) is not added to an abstract state if $v{\rightsquigarrow}w\notin\mathcal{R}^{\tau}$ (resp., $\circlearrowleft^{v}\notin\mathcal{Y}^{\tau}$).

The abstract semantics. An *abstract denotation* ξ from τ_1 to τ_2 is a partial map from $\mathcal{I}^{\tau_1}_{rc}$ to $\mathcal{I}^{\tau_2}_{rc}$. It describes how the abstract input state changes when a piece of code is executed. The set of all abstract denotations from τ_1 to τ_2 is denoted by $\Xi(\tau_1,\tau_2)$. As in the concrete setting, interpretations provide abstract denotations for methods in terms of their input and output arguments. An *interpretation* ζ maps methods to abstract denotations, and is such that $\zeta(\mathsf{m})\in\Xi(\mathsf{m}^i,\mathsf{m}^i\cup\{out\})$. Note that the range of such denotations is $\mathsf{m}^i\cup\{out\}$, instead of $\{out\}$ (as in the concrete semantics): this point will get clarified below. Finally, Ψ denotes the set of all (abstract) interpretations.

Fig. 3 depicts abstract denotations. An *expression denotation* $\mathcal{E}^{\tau}_{\zeta}[\![exp]\!]$ maps abstract states from $\mathcal{I}^{\tau}_{rc}$ to abstract states from $\mathcal{I}^{\tau\cup\{\rho\}}_{rc}$, while a *command denotation* $\mathcal{C}^{\tau}_{\zeta}[\![com]\!]$ maps $\mathcal{I}^{\tau}_{rc}$ to $\mathcal{I}^{\tau}_{rc}$.

Expressions. An expression denotation $\mathcal{E}^{\tau}_{\zeta}[\![exp]\!]$ adds to an input state I those reachability and cyclicity statements which result from evaluating exp. Nothing is added to I in cases (1_e) (which consists of three similar cases), (2_e) when $\tau(v)$=**int**, and (3_e) when f has type **int**, since the expression is evaluated without side effects to an integer value, to **null** or to a newly allocated object. In (2_e), when $\tau(v)\neq$**int**, ρ has the same abstract behavior as v. Therefore, the semantics returns I, together with a *cloned* version $I[v/\rho]$ for ρ. In (3_e), when f is a reference field, the following information is added to I: (a) statements for v

$$
\begin{array}{ll}
(1_e) & \mathcal{E}^{\tau}_{\zeta}[\![n]\!](I) = \mathcal{E}^{\tau}_{\zeta}[\![\textbf{null}]\!](I) = \mathcal{E}^{\tau}_{\zeta}[\![\textbf{new}\ \kappa]\!](I) = I \\
(2_e) & \mathcal{E}^{\tau}_{\zeta}[\![v]\!](I) = \text{if } \tau(v){=}\textbf{int} \text{ then } I \text{ else } I \cup I[v/\rho] \\
(3_e) & \mathcal{E}^{\tau}_{\zeta}[\![v.f]\!](I) = \text{if } f \text{ has type } \textbf{int} \text{ then } I \text{ else } I \cup I' \text{ where} \\
& \quad I'{=}I[v/\rho] \cup \{w{\rightsquigarrow}\rho \mid \langle w{\bullet}v\rangle\} \cup \{\rho{\rightsquigarrow}\rho \mid \circlearrowleft^{v} \in I\} \\
(4_e) & \mathcal{E}^{\tau}_{\zeta}[\![exp_1 \oplus exp_2]\!](I) = \exists\rho.\mathcal{E}^{\tau}_{\zeta}[\![exp_2]\!](\exists\rho.\mathcal{E}^{\tau}_{\zeta}[\![exp_1]\!](I)) \\
(5_e) & \mathcal{E}^{\tau}_{\zeta}[\![v_0.m(v_1,..,v_n)]\!](I) = \cup\{I, I_m, I_3, I_4\} \text{ such that} \\
& \bar{v}{=}\{v_0,..,v_n\} \qquad I_0{=}\exists(\tau\backslash\bar{v}).I \\
& I_m = \cup\ \{\ (\zeta(\mathsf{m})(I_0[\bar{v}/\mathsf{m}^i]))[\mathsf{m}^i/\bar{v}, out/\rho]\ \mid \mathsf{m} \text{ might be called here }\} \\
& sh = \{\langle v_i{\bullet}v_j\rangle \mid v_i, v_j \in \bar{v} \text{ and } \langle v_i{\bullet}v_j\rangle\} \\
& sh' = \cup\{\mathsf{SH_m}(sh[\bar{v}/\mathsf{m}^i])[\mathsf{m}^i/\bar{v}, out/\rho] \mid \mathsf{m} \text{ might be called here}\} \\
& I_1 = \{w_1{\rightsquigarrow}w_2 \mid (v_i{\rightsquigarrow}v_j{\in}I_m) \wedge (\dot{v}_i{\in}sh') \wedge \langle w_1{\bullet}v_i\rangle \wedge ((v_j{\rightsquigarrow}w_2{\in}I){\vee}\langle w_2{\cdot}v_j\rangle)\} \\
& I_2 = \{w_1{\rightsquigarrow}w_2 \mid (\langle v_i{\bullet}v_j\rangle \in sh) \wedge (\dot{v}_i{\in}sh') \wedge \langle v_i{\bullet}w_1\rangle \wedge (v_j{\rightsquigarrow}w_2 \in I)\} \\
& I_3 = \cup\{(I_1{\cup}I_2)[v/\rho] \mid \langle v{\cdot}\rho\rangle \text{ after the call }\} \\
& I_4 = \{\circlearrowleft^{w} \mid \langle w{\bullet}v\rangle \wedge (\dot{v}{\in}sh') \wedge (\circlearrowleft^{v} \in I_m)\} \\
\hline
(1_c) & \mathcal{C}^{\tau}_{\zeta}[\![v{:=}exp]\!](I) = (\exists v.\mathcal{E}^{\tau}_{\zeta}[\![exp]\!](I))[\rho/v] \\
(2_c) & \mathcal{C}^{\tau}_{\zeta}[\![v.f{:=}exp]\!](I) = \exists\rho.(I' \cup I_r \cup I_c) \text{ where } I' = \mathcal{E}^{\tau}_{\zeta}[\![exp]\!](I) \text{ and} \\
& I_r{=}\{w_1{\rightsquigarrow}w_2 \mid (\langle w_1{\cdot}v\rangle \vee (w_1{\rightsquigarrow}v \in I')) \wedge (\langle\rho{\cdot}w_2\rangle \vee (\rho{\rightsquigarrow}w_2 \in I'))\} \\
& I_c{=}\{\circlearrowleft^{w} \mid ((\rho{\rightsquigarrow}v \in I') \vee \langle\rho{\cdot}v\rangle \vee (\circlearrowleft^{\rho} \in I')) \wedge (\langle w{\cdot}v\rangle \vee (w{\rightsquigarrow}v \in I'))\} \\
(3_c) & \mathcal{C}^{\tau}_{\zeta}\left[\!\!\left[\begin{array}{l}\textbf{if}\ \ exp\ \textbf{then}\ com_1 \\ \qquad\quad \textbf{else}\ com_2\end{array}\right]\!\!\right](I) = \mathcal{C}^{\tau}_{\zeta}[\![com_1]\!](I) \cup \mathcal{C}^{\tau}_{\zeta}[\![com_2]\!](I) \\
(4_c) & \mathcal{C}^{\tau}_{\zeta}[\![\textbf{while}\ exp\ \textbf{do}\ com]\!](I) = \xi(I) \text{ where } \xi = lfp(\lambda w.\lambda I.w(\mathcal{C}^{\tau}_{\zeta}[\![com]\!](I))) \\
(5_c) & \mathcal{C}^{\tau}_{\zeta}[\![\textbf{return}\ exp]\!](I) = \mathcal{E}^{\tau}_{\zeta}[\![exp]\!](I)[\rho/out] \\
(6_c) & \mathcal{C}^{\tau}_{\zeta}[\![com_1; com_2]\!](I) = \mathcal{C}^{\tau}_{\zeta}[\![com_2]\!](\mathcal{C}^{\tau}_{\zeta}[\![com_1]\!](I))
\end{array}
$$

Fig. 3. Abstract denotations for expressions and commands

which are cloned for ρ; (b) $w{\rightsquigarrow}\rho$, if w might share with v[3]; (c) if v might be cyclic, then, for soundness, $\rho{\rightsquigarrow}\rho$[4]. In (4_e), exp_1 is first analyzed, then exp_2 on the resulting state. ρ is removed since the return value is always an **int**. Case (5_e) will be explained later, after introducing command denotations.

Example 2. Consider c:=c.next at line 15 in Fig. 1. Evaluating $\mathcal{E}^{\tau}_{\zeta}[\![\mathsf{c.next}]\!](I_{14})$ results in $\{this{\rightsquigarrow}c, this{\rightsquigarrow}p, this{\rightsquigarrow}\rho, c{\rightsquigarrow}\rho, p{\rightsquigarrow}\rho\}$. The statement $this{\rightsquigarrow}\rho$ is added since $this{\rightsquigarrow}c \in I_{14}$; $c{\rightsquigarrow}\rho$ and $p{\rightsquigarrow}\rho$ are added because $\langle c{\bullet}c\rangle{\wedge}\langle c{\bullet}p\rangle$ after line 14.

Variable assignment. The denotation (1_c) computes $\mathcal{E}^{\tau}_{\zeta}[\![exp]\!](I)$, removes any statement about v since it takes a new value, and finally renames ρ to v. Note that it is safe to remove statements about v since it is first cloned to ρ.

Example 3. Consider, again, line 15 in Fig. 1. Evaluating $\mathcal{C}^{\tau}_{\zeta}[\![\mathsf{c{:=}c.next}]\!](I_{14})$ first computes $\mathcal{E}^{\tau}_{\zeta}[\![\mathsf{c.next}]\!](I_{14})$ as in Ex. 2. Then, statements involving c are removed, which results in $\{this{\rightsquigarrow}p, this{\rightsquigarrow}\rho, p{\rightsquigarrow}\rho\}$, and, finally, ρ is renamed to c, giving $\{this{\rightsquigarrow}p, this{\rightsquigarrow}c, p{\rightsquigarrow}c\}$. Note that $this{\rightsquigarrow}c$ is reinserted (by renaming $this{\rightsquigarrow}\rho$) after being deleted by $\exists c$. Also, note that $c{\rightsquigarrow}\rho$ has been removed by $\exists c$, so that, correctly, c is not considered to reach itself after the assignment.

[3] Note that $v{\rightsquigarrow}\rho$ is always added since $\langle v{\bullet}v\rangle$ (v cannot be **null**). Cases where v and w reach a common location, but not each other, are also handled.

[4] Note that, in this case, $\circlearrowleft^{\rho}$ has also been added.

Field update. The denotation (2_c) accounts for field updates. It adds reachability and cyclicity statements to I, and does not remove any statements from it, since, in general, there is no guarantee that any path in the heap will be broken. The set I' results from $\mathcal{E}^\tau_\zeta[\![exp]\!](I)$, while I_r and I_c capture the effect of executing $v.f{:=}\rho$ on I'. Reachability statements are added in I_r: any w_1 which might alias with v or reach v (i.e., $\langle w_1{\cdot}v\rangle \vee (w_1{\rightsquigarrow}v \in I')$) can also reach any w_2 aliasing with ρ or reachable from it (i.e., $\langle \rho{\cdot}w_2\rangle \vee (\rho{\rightsquigarrow}w_2 \in I')$). This covers all possible paths which can be created by adding a direct link from v to ρ through f. Cyclicity statements are added in I_c. There are three possible scenarios in which v might become cyclic: (a) ρ reaches v, so that a cycle from v to itself is created; (b) ρ aliases with v, so that v reaches itself with a path of length 1 (e.g., the command $\mathsf{y.f{:=}y}$); and (c) ρ is cyclic, so that v becomes indirectly cyclic. When one of these scenarios occurs (i.e., $(\rho{\rightsquigarrow}v \in I') \vee \langle \rho{\cdot}v\rangle \vee (\circlearrowleft^\rho \in I')$), any w aliasing with v or reach it (i.e., $\langle w{\cdot}v\rangle \vee (w{\rightsquigarrow}v \in I')$) has to be considered as possibly cyclic.

Example 4. Before line 20 in Fig. 1, $I_{17} = \{this{\rightsquigarrow}c, this{\rightsquigarrow}p, p{\rightsquigarrow}c, n{\rightsquigarrow}c\}$. Evaluating $\mathcal{C}^\tau_\zeta[\![\mathsf{p.next{:=}n}]\!](I_{17})$ adds a new statement $p{\rightsquigarrow}n$, and also (a) $this{\rightsquigarrow}n$ since **this** was reaching p; (b) $p{\rightsquigarrow}c$ and $this{\rightsquigarrow}c$ (already in I_{17}) since n was reaching c.

Conditions, loops, composition and return. Rules (3_s), (4_s), and (6_s) are quite straightforward and correspond to *if*, *while*, and *command composition.* Rule (5_s) corresponds to *return* and behaves, as expected, like $out{:=}exp$.

Method calls. Rule (5_e) propagates the effect of a method call to the calling context, as follows: (1) the abstract state I is projected on the actual parameters $\bar{v}$, thus obtaining I_0 (this is needed since the denotation of the callee is given in terms of its parameters); (2) the denotation of each method m which can be called is taken from the current interpretation, namely, $\zeta(\mathsf{m})$, and applied to $I_0[\bar{v}/\mathsf{m}^i]$, that is, I_0 after renaming the actual parameters $\bar{v}$ to the formal parameters m^i; (3) formal parameters are renamed back to the actual parameters (plus out and ρ) in the resulting state $\zeta(\mathsf{m})(I_0[\bar{v}/\mathsf{m}^i])$, and the states obtained from all possible signatures are merged into I_m. (4) sh is a safe approximation of the sharing between the actual parameters, and sh' is the sharing and purity information after the method call. Afterwards, the definitions of I_1, I_2, I_3, and I_4 account for the propagation of the effects of the method execution in the calling context: I_1 states that, if the call makes v_i reach v_j, then any w_1 sharing with v_i before the call might reach any w_2 which is reachable from v_j or aliasing with v_j. Note that adding these statements is necessary only if v_i is updated in the body of some m: otherwise, no path from w_1 to w_2 can be created in the called method. I_2 states that, if the call makes v_i share with v_j, then any w_1 sharing with v_i might reach any w_2 reachable from v_j. Again, this is required only if v_i is updated in the body of some m. I_3 contains the information on any v aliasing with ρ, cloned for ρ. I_4 includes the cyclicity of anything sharing with an argument which might become cyclic. The final result of processing a call is $I \cup I_m \cup I_3 \cup I_4$.

```
1 int mirror(Tree t) {
2   Tree l,r;
3
4   if (t=null) then {
5     return 0;
6   } else {
7     l:=t.left;
8     r:=t.right;
9     t.left:=r;
10    t.right:=l;
11    return 1+mirror(l)
12          +mirror(r); } }
```

```
1 Node connect() {
2   Node curr;
3
4   curr=this;
5   while (curr.next!=null) {
6      curr:=curr.next; }
7   curr.next:=this;
8   return curr; }
```

```
1 Node f(Node a,Node b,Node c) {
2   a.next:=b;
3   c.next:=this;
4   return b.g(c); }
```

```
1 Node k(Node y) {
2   u:=y;
3   this.next:=y;
4   y:=null;
5   return this; }
```

```
1 Node g(Node y) {
2   this.next:=y;
3   return this; }
```

```
1 Node h(Node y) {
2   this.next:=y;
3   y:=null;
4   return this; }
```

Fig. 4. Additional examples

Example 5. Consider methods f and g in Fig. 4, and assume that both are defined in the class Node. Let ξ be a denotation for g s.t. $\xi(\emptyset) = \{\mathit{this}{\leadsto}y, \mathit{out}{\leadsto}y\}$. This example shows how an abstract state $\emptyset$ is transformed by executing the code of f. The first two commands in f transform $\emptyset$ into $I = \{a{\leadsto}b, c{\leadsto}\mathit{this}\}$. Then, the denotation of g is plugged into the calling context as follow: (1) I is projected on $\{b, c\}$, obtaining $I_0 = \emptyset$; (2) $\xi(\emptyset)$ is renamed s.t. *this*, y, and *out* are renamed to, resp., b, c, and ρ, and $I_m = \{b{\leadsto}c, \rho{\leadsto}c\}$ is obtained; (3) $a{\leadsto}\mathit{this}$ is added to I_1 since $(b{\leadsto}c \in I_m) \wedge \langle b{\bullet}a\rangle \wedge (c{\leadsto}\mathit{this} \in I)$ is true; similarly, $b{\leadsto}\mathit{this}$, $a{\leadsto}c$ and $a{\leadsto}\rho$ are also added to I_1; (4)no new statements are added due to I_2 and I_3; (5) I_4 is empty since nothing becomes cyclic in g; (6) finally, the denotation of **return** renames ρ to *out* in $I \cup I_m \cup I_1 \cup I_4$, and obtains $\{a{\leadsto}b, c{\leadsto}\mathit{this}, b{\leadsto}c, \mathit{out}{\leadsto}c, a{\leadsto}\mathit{this}, a{\leadsto}c, b{\leadsto}\mathit{this}, a{\leadsto}\mathit{out}\}$.

Next, the inference of a denotation for a method m is shown, which uses the denotation $\mathcal{C}^{\tau}_{\zeta}[\![\mathsf{m}^b]\!]$ of its code. Ex. 6 introduces the problems to face when trying to define a method denotation, and a solution is discussed below.

Example 6. In Ex. 5, when analyzing b.g(c), the existence of a denotation ξ for g s.t. $\xi(\emptyset) = \{\mathit{this}{\leadsto}y, \mathit{out}{\leadsto}y\}$ was assumed. Intuitively, this $\xi(\emptyset)$ could be computed using $\mathcal{C}^{\tau}_{\zeta}[\![\mathsf{g}^b]\!]$, as follows: the first command in g adds $\mathit{this}{\leadsto}y$, and the second one adds $\mathit{out}{\leadsto}y$, which results in the desired abstract state $\{\mathit{this}{\leadsto}y, \mathit{out}{\leadsto}y\}$. After this result, one might think that $\mathcal{C}^{\tau}_{\zeta}[\![\mathsf{m}^b]\!](I)$ is always the good way to compute $\xi(I)$, as just done. Yet, in general, this is not correct. For example, suppose the call b.g(c) is replaced by b.h(c). The effect of this call should be the same as b.g(c), since both methods make b reach c and b reach the return value. However, computing $\mathcal{C}^{\tau}_{\zeta}[\![\mathsf{h}^b]\!](\emptyset)$ has a different result: the

first instruction adds $this{\rightsquigarrow}y$, the second one removes it since the value of y is overwritten, and the third does not add anything. Therefore, $\mathcal{C}^{\tau}_{\zeta}\,[\![\mathsf{h}^b]\!](\emptyset) = \emptyset$, which is not sound to use as the result of $\xi(\emptyset)$.

The problem in Ex. 6 comes from the call-by-value passing style for parameters, where, if the formal parameters are modified in the method, then the final abstract state does not describe the actual parameters anymore. This is why the expected reachability information is obtained for f (since it does not modify y), while it is not in the case of h (since y is modified in the body). A common solution to this problem is to mimic actual parameters by *shallow variables*, i.e., new variables which are initialized (when entering the method) to the same values as the parameters, but are never modified in the body. For example, the method k in Fig. 4 is the result of instrumenting h with a shallow variable u, mimicking y. It is easy to verify that $\mathcal{C}^{\tau}_{\zeta}\,[\![\mathsf{k}^b]\!](\emptyset)$ comes to be $\{this{\rightsquigarrow}u, out{\rightsquigarrow}u\}$, which includes the desired reachability information. The following definition defines the abstract denotational semantics of a program P as the least fixpoint of an (abstract) transformer of interpretations. Variables $\bar{u}$ play the role of shallow variables. Note that shallow variables appear at the level of the semantics, rather than by transforming the program.

Definition 5. *The abstract denotational semantics of a program P is the* lfp *of the transformer* $\mathcal{T}_P(\zeta) = \{\mathsf{m} \mapsto \lambda I{\in}\mathcal{I}^{\mathsf{m}^i}_{rc}(\exists X.\mathcal{C}^{\tau}_{\zeta}\,[\![\mathsf{m}^b]\!](I \cup I[\bar{w}/\bar{u}]))[\bar{u}/\bar{w}] \mid \mathsf{m} \in P\}$ *where* $\mathsf{m}^i = \{this, w_1, \ldots, w_n\}$*,* $\bar{u} = \{u_1, \ldots, u_n\}$ *s.t.* $\bar{u}{\cap}\mathsf{m}^s = \emptyset$*,* $\mathrm{dom}(\tau) = \mathsf{m}^l{\cup}\bar{u}$*, and* $X = \mathrm{dom}(\tau)\backslash(\bar{u} \cup \{this, out\})$*.*

The definition is explained in the following. The operator $\mathcal{T}_P$ transforms the interpretation ζ by assigning a new denotation for each method $\mathsf{m} \in P$, using those in ζ. The new denotation for m maps a given input abstract state $I \in \mathcal{I}^{\mathsf{m}^i}_{rc}$ to an output state abstract from $\mathcal{I}^{\mathsf{m}^i \cup \{out\}}_{rc}$, as follows: (1) it obtains an abstract state $I_0 = I \cup I[\bar{w}/\bar{u}]$ in which the parameters $\bar{w}$ are cloned into the shallow variables $\bar{u}$; (2) it applies the denotation of the code of m on I_0, obtaining $I_1 = \mathcal{C}^{\tau}_{\zeta}\,[\![\mathsf{m}^b]\!](I_0)$; (3) all variables but $\bar{u} \cup \{this, out\}$ are eliminated from I_1 (using $\exists X$); and (4) shallow variables $\bar{u}$ are finally renamed back to $\bar{w}$. Theorem 1 states that the abstract denotational semantics of Def. 1 is a safe approximation of the concrete denotational semantics of Def. 5.

Example 7. Consider the method mirror in Fig. 4, and suppose class Tree implements a binary tree in the standard way, with fields left and right. The call mirror(t) exchanges the values of left and right of each node in t, and returns the number of nodes in the tree. An initial state $\emptyset$ is transformed by mirror as follows. Suppose that the current interpretation ζ is such that $\zeta(\mathsf{mirror}) = \xi$, and $\xi(\emptyset) = \emptyset$. The first branch of the *if* (when t is **null**) does not change the initial denotation; on the other hand, when t is different from **null**, line 7 adds $t{\rightsquigarrow}l$; line 8 adds $t{\rightsquigarrow}r$; line 9 adds again $t{\rightsquigarrow}r$; and line 10 adds again $t{\rightsquigarrow}l$. Recursive calls mirror(l) and mirror(r) do not add any statement since $\xi(\emptyset) = \emptyset$. Finally, **return** adds nothing. Projecting $\{t{\rightsquigarrow}l, t{\rightsquigarrow}r\}$ on t and out results in $\emptyset$, so that $\xi(\emptyset)$ does

not change, and there is no need for another iteration. It can be concluded that, as expected, mirroring the tree does not make it cyclic.

Example 8. Consider the method connect in Fig. 4, defined in the class Node. A call l.connect() with l acyclic makes the last element of l point to l, so that it becomes cyclic. It also returns a reference to the last element in the list. An initial state $\emptyset$ is transformed by connect as follows. Line 4 does not add any statements, while line 6 in the loop adds $this \rightsquigarrow curr$. Another iteration of the loop does not change anything, so that the loop is exited with $\{this \rightsquigarrow curr\}$. Since **this** is now reaching curr, line 8 adds $\{curr \rightsquigarrow this,\ curr \rightsquigarrow curr, this \rightsquigarrow this\}$, and $\{\circlearrowleft^{curr}, \circlearrowleft^{this}\}$. Finally, line 9 clones $curr$ to out. In conclusion, the analysis correctly infers that l.connect() makes l and the return value cyclic.

Theorem 1. *Let P be a program, ι and ζ its concrete and abstract semantics as in Defs. 1 and 5,* m *a method in P, $\delta = \iota(\mathsf{m})$, $\xi = \zeta(\mathsf{m})$. It holds that, for all $\sigma_1 \in \Sigma_{m^i}$, if $\sigma_2 = \delta(\sigma_1)$ then $\langle \hat{\sigma}_1[out \mapsto \hat{\sigma}_2(out)], \check{\sigma}_2 \rangle \in \gamma^{\tau}_{rc}(\xi(\alpha^{\tau}_{rc}(\{\sigma_1\})))$.*

NOTE TO THE REVIEWER: *We noticed that Figure 3 in the submitted version contains a couple of typos, which have been fixed in the present long version: (a) in case (5_e), in the definition of I_1, the last reachability statements should be swapped, being $v_j \rightsquigarrow w_2$ instead of $w_2 \rightsquigarrow v_j$; and (b) still in case (5_e), in the definition of I_2 (resp., I_4), the condition $\dot{v}_i \in I_m$ (resp., $\dot{v} \in I_m$) should be replaced by $\dot{v}_i \in sh'$ (resp., $\dot{v} \in sh'$). We are sorry for these mistakes. Please accept our apologies.*

Proof. This proof of soundness starts by proving the soundness of all abstract denotations for expressions and commands, assuming that a current interpretation ι and a corresponding abstract one ζ which correctly approximates ι are available. Let σ be a concrete state, com be a command, exp be an expression, and σ^* be the state obtained by executing com or evaluating exp in σ. The soundness of the abstract denotations for expressions and commands amounts to say that, if $I \in \mathcal{I}^{\tau}_{rc}$ correctly approximates σ, i.e., $\sigma \in \gamma^{\tau}_{rc}(I)$, then the abstract state $I^* = \mathcal{C}^{\tau}_{\zeta}[\![com]\!](I)$ (or $I^* = \mathcal{E}^{\tau}_{\zeta}[\![exp]\!](I)$, in the case of expressions) correctly approximates σ^*. More formally,

1. $\forall \sigma \in \Sigma_\tau, I \in \mathcal{I}^{\tau}_{rc}.\ \sigma \in \gamma^{\tau}_{rc}(I) \ \Rightarrow\ E^{\iota}_{\tau}[\![exp]\!](\sigma) \in \gamma^{\tau}_{rc}(\mathcal{E}^{\tau}_{\zeta}[\![exp]\!](I))$
2. $\forall \sigma \in \Sigma_\tau, I \in \mathcal{I}^{\tau}_{rc}.\ \sigma \in \gamma^{\tau}_{rc}(I) \ \Rightarrow\ C^{\iota}_{\tau}[\![com]\!](\sigma) \in \gamma^{\tau}_{rc}(\mathcal{C}^{\tau}_{\zeta}[\![com]\!](I))$

The soundness proof considers separately the rules of the abstract semantics $\mathcal{E}^{\tau}_{\zeta}[\![_]\!](_)$ and $\mathcal{C}^{\tau}_{\zeta}[\![_]\!](_)$. When some logical fact is said to hold *by soundness*, it means that it holds by the hypothesis on the input (i.e., that $\sigma \in \gamma^{\tau}_{rc}(I)$ holds), or by induction on subexpressions or subcommands. For example, the fact that v reaches w in σ implies $v \rightsquigarrow w \in I$ *by soundness*, since I is supposed to be a sound description of σ.

Expressions. Consider $\mathcal{E}^{\tau}_{\zeta}[\![_]\!](_)$, and let ℓ_e denote the heap location obtained by evaluating the expression exp when it has reference type (if the tipe of exp is **int**, then the special variable ρ can be ignored). The goal is to prove that

– the special variable ρ correctly represents the abstract information about the result of evaluating the expression; that is, if there might exist a path from ℓ_e to $\hat{\sigma}^*(v)$ in σ^*, then the statement $\rho{\rightsquigarrow}v$ belongs to I^*; if there might exist a path from v to ℓ_e in σ^*, then $v{\rightsquigarrow}\rho \in I^*$; if ℓ_e may be cyclic in σ^*, then $\circlearrowleft^\rho \in I^*$; and
– side effects due to method calls are correctly dealt with.

Denotations (1_e)*.* This case is divided into three sub-cases which are all easy. In fact evaluating a number or the **null** value has no side effects, and does not introduce any reachability nor cyclicity, so that the output abstract value I^* is equal to I. This also holds for object creation, because the new object is not cyclic, and is not involved in any reachability paths[5].

Denotations (2_e) *when* v *has type* **int***, and* (3_e) *when the field* f *has type* **int***.* These cases are also straightforward. The variable ρ does not need to appear in I^*, since the value of the expression is not a memory location. Besides, there are no side effects which may involve either reachability or cyclicity.

Denotation (2_e) *when* v *has reference type.* The location ℓ_e comes to be, in this case, exactly $\hat{\sigma}(v)$, so that any reachability or cyclicity related to v can be simply copied into ρ, and this is obtained by letting I^* be $I \cup I[v/\rho]$, which, as expected, does not remove the information about v.

Denotation (3_e) *when the field* f *has reference type.* First, note that the new reachability and cyclicity information includes only statements about the newly-introduced ρ, since the heap is not modified. In addition, if w does not share with v, then it cannot have any reachability relation with ρ, since they correspond to disjoint regions of the heap. Thus, the proof only needs to consider a variable w sharing with v in σ.

– If v and w alias, then w reaches ℓ_e. By soundness, $\langle v{\cdot}w\rangle$ and $\langle v{\bullet}w\rangle$ (since aliasing implies sharing) hold in the input state. Then, $w{\rightsquigarrow}\rho$ is correctly added.
– If v reaches w, then $v{\rightsquigarrow}w \in I$ by soundness. In this case, it is possible that ℓ_e also reaches w, and this happens when the path from v to the location bound to w goes through ℓ_e. This requires adding $\rho{\rightsquigarrow}w$ to I^*, as the semantics does by adding $I[v/\rho]$.
– If w reaches v, then w reaches ρ, and this is accounted for by adding $w{\rightsquigarrow}\rho$ (note that reachability implies sharing, so that $\langle v{\bullet}w\rangle$ certainly holds by soundness).
– If w and v reach a common memory location, then w might reach ρ since the common location might be exactly ℓ_e. This case is also dealt with by adding $w{\rightsquigarrow}\rho$.

[5] Note, that, unlike in Java, the simple act of creating an object does not involve, in itself, any action on its content, i.e., there are no side effects due to the constructor.

– Finally, ρ might be cyclic in σ^* only if v is cyclic in σ. In such case, $\circlearrowleft^v \in I$ by soundness, so that the abstract semantics correctly adds $\circlearrowleft^\rho$ by means of the operation $I[v/\rho]$. Note that, since the cycle might go through ℓ_e, the statement $\rho \rightsquigarrow \rho$ is also added in order to account for this possibility.

Denotation (4_e). The result of $\oplus$ can only be of type **int**, so that I^* does not need to contain any statements about ρ. However, both exp_1 and exp_2 can possibly have side effects. This is handled in the abstract semantics by first evaluating $I' = \exists \rho.\mathcal{E}^\tau_\zeta[\![exp_1]\!](I)$, and then $\exists \rho.\mathcal{E}^\tau_\zeta[\![exp_2]\!](I')$, i.e., the evaluation of exp_2 considers the possible changes occurred while evaluating exp_1. Note that, even if $\oplus$ is $=$ or $\neq$, and works on references (as in $v \neq$ **null**), projecting away ρ from $\mathcal{E}^\tau_\zeta[\![exp_1]\!](I)$ is sound because the return value is not used at the abstract level. In any case, the result of applying $\oplus$ is still an **int**, so that the second removal of ρ (i.e., from $\mathcal{E}^\tau_\zeta[\![exp_2]\!](I')$) is also sound.

Denotation (5_e). Calling a method m consists of an abstract execution of its body on the actual parameters, followed by the propagation of the effects of m to the calling context (i.e., the input abstract state I). First, note that, in the abstract semantics, reachability and cyclicity statements are only removed when a variable is assigned. Due to the use of shallow variables for the parameters, statements about the formal parameters of m are never removed during an abstract execution of its body. Therefore, if, during the execution of m, the variable v reaches w, then, at the end of the method, v will be said to possibly reach w, even if this reachability is destroyed at somesubsequent program point. This is similar to the way sharing information is dealt with in the present approach (following [15,12]).

Keeping track of cyclicity is rather easy. In addition to keeping all cyclicity which is in I, a safe approximation is taken, which states that, if an argument v might become cyclic during the execution of m, then anything that shares with it before the execution might also become cyclic. This is accounted for in the definition of I_4, and is clearly safe. In fact, variables of the calling method which are not arguments of the call, and do not share with any argument v_i, cannot be affected by the execution of m.

The treatment of reachability is more complicated: in addition to I and I_m (which is introduced by the method for $\bar{v}$), it is necessary to take into account the effect of the method call on variables which are not arguments. This is done in the definition of I_1, I_2, and I_3, which model the effects of m on variables which share with its actual arguments. Consider two arguments v_i and v_j (where i can be equal to j): a path between two variables w_1 and w_2 (which can be arguments, or non-argument variables) can be created by m if (i) v_i and w_1 share before the call, v_j and w_2 alias before the call, v_i is modified in m, and v_i reaches v_j after the call; or (ii) v_i and w_1 share before the call, v_j reaches w_2 before the call, v_i is modified in m, and v_i and v_j share (without reaching each other) after the call. The two cases are accounted for in the definition of, resp., I_1 and I_2, and are depicted in Fig.5. In both cases, the creation of the path requires that an argument is modified in m (condition $\dot{v}_i \in sh'$), and that v_i and v_j do not point

to disjoint regions of the heap (i.e., either v_i reaches v_j, or they simply share). As a result, if these conditions are met, then the statement $w_1 \rightsquigarrow w_2$ is added. It can be seen that this accounts for all cases where some change in the arguments of m affects the reachability between non-argument variables.

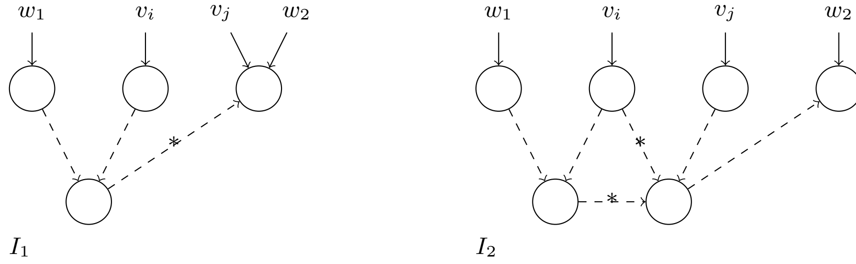


Fig. 5. Scenarios where a path from w_1 to w_2 can be created inside m. Dashed arrows represent reachability: they connect a variable to a reachable location (represented as a circle). Solid arrows connect a variable u to the location $\hat{\sigma}(u)$ directly bound to it. Arrows labeled with * are paths which are created inside m (strictly speaking, they could also exist before the call), while the others existed before the method call. In both cases, it can be seen that a reachability path from w_1 to w_2 is created, which contains a sub-path created inside m by modifying its arguments.

Finally, I_3 considers all variables v aliasing with the return value at the end of m (note that these are the only new aliasing statements involving arguments which can be created in the body of m) : the information about them is cloned for ρ.

Commands. As for commands, the goal of the proof is to guarantee that the output I^* is a correct description of σ^* whenever I correctly describes σ.

Denotation (1_c)*.* This instruction is interesting only if v has reference type. It is equivalent to first evaluating *exp*, then executing $v := \rho$. In the abstract setting, this amounts to: evaluating *exp*; removing any statements about v (since its value will be overwritten); and renaming ρ to v. This is exactly what the abstract semantic does, by means of, resp., the operations $\mathcal{E}^{\tau}_{\zeta}[\![exp]\!](I)$, $\exists v.$, and $[\rho/v]$. Note that $\mathcal{E}^{\tau}_{\zeta}[\![_]\!](_)$ works on I, but the substitution $[\rho/v]$ is performed after v has been projected away from $\mathcal{E}^{\tau}_{\zeta}[\![exp]\!](I)$; that is, initial statements about v are used while analyzing *exp*, but are removed afterwards.

Denotation (2_c)*.* This case is trivial when f has type **int**, since only side effects during the evaluation of *exp* have to be taken into account. If f has reference type, then this command is equivalent to first evaluating *exp*, and then executing $v.f := \rho$. Let $\sigma' = E^{\iota}_{\tau}[\![exp]\!](\sigma)$, and $\ell_e = \hat{\sigma}'(\rho)$. If v and ℓ_e are considered, then there are two main cases (Fig. 6): (a) $\hat{\sigma}(v) = \ell_e$; or (b) $\hat{\sigma}(v) \neq \ell_e$.

(a) In this case, a cycle on v is created, whose length is 1. If another variable u (possibly, v itself) sharing with v in σ^* is considered, then there are several

possible scenarios in the heap, and soundness has to be proven for each of them.

- u aliases with v or reaches v (cases u_1 and u_2 in the left-hand side of Fig. 6). In this case, u reaches v via f, and this is taken into account in the definition of I_r, where u plays the role of w_1, and v also plays the role of w_2. The result is that I_r includes $u{\rightsquigarrow}v$, as expected. The semantics correctly adds $v{\rightsquigarrow}v$ as well (in fact, v can play the role of both w_1 and w_2). As for cyclicity, the definition of I_c guarantees that $\circlearrowleft^v$ and $\circlearrowleft^u$ will belong to I^*.
- v reaches u (case u_3 in the same figure). In this case, $v{\rightsquigarrow}u \in I^*$ since, in the definition of I_r, u plays the role of w_2 (note that v and ρ alias). v will also be considered as cyclic by the definition of I_c;
- v and u both reach a common location ℓ (case u_4). If none of the previous cases happens, then v and u do not reach each other, so that I^* does not need to contain reachability statements between them. In general, only v will be considered as cyclic in this case (in the same way as the previous cases).

(b) In this case, when considering u, the number of possible scenarios for reachability is larger. Moreover, there are two scenarios where v would be cyclic *after* the update (i) ℓ_e reaches v, so that a cycle is created by the field update, and v becomes cyclic (if it was not already); or (ii) ℓ_e does not reach v, so that v is cyclic only if it was already cyclic in σ, and the same applies to ℓ_e. In case (ii), it can be easily seen that the definition of I_c accounts for the cyclicity of v since $\circlearrowleft^v$ belongs to I by soundness and will not be removed. Case (i) will be discussed in the following, for each scenario.

- u reaches v or aliases with it (cases u_1 and u_2 in the right-hand side of Fig. 6). In this case, it was also reaching v (or aliasing with it) in σ', so that (in the case of reachability) $u{\rightsquigarrow}v \in I'$, which implies $u{\rightsquigarrow}v \in I''$, as soundness requires. As for cyclicity, in case (i), the cyclicity of u is detected because it reaches v.
- Cases u_3, u_4, and u_5. These cases are easy, because nothing changes with respect to the reachability between u and v, and all the statements were already contained in I.
- u points to ℓ_e or is reached by it (cases u_6 and u_7). In this case, u plays the role of w_2 in the definition of I_r, and is correctly considered to be reached by v. As for cyclicity, u will only become cyclic in case (i) if it points to ℓ_e, or belongs to the cyclic path. In both cases, the semantics accounts for it since u would reach v, thus being considered as cyclic (definition of I_c).
- w and ℓ_e reach some common location ℓ (case u_8). Also easy since nothing changes with respect to the reachability between u and v.

Denotation (3_c). This case is quite straightforward, given the inductive hypothesis on com_1 and com_2, and the assumption that exp has no side effects and

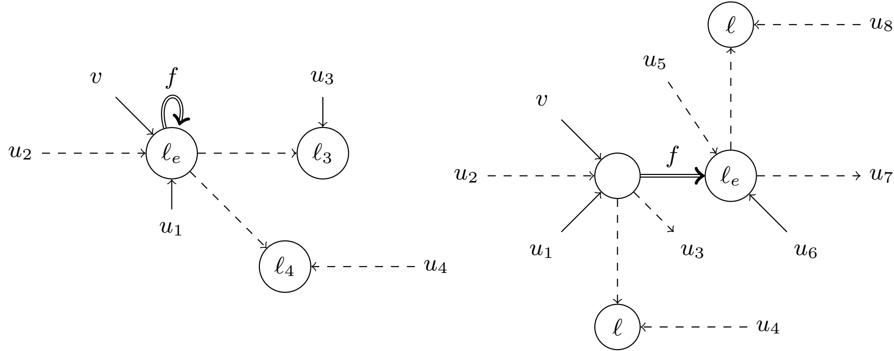


Fig. 6. The possible scenarios for case (2_c): (a) ℓ_e and $\hat{\sigma}(v)$ coincide (left); and (b) they do not coincide. Variables u_i represent the possible relations between the variable u used in the proof and the data structure modified by the field update. Double solid arrows stand for field dereferences, and are labeled with the name of the field. For the other kinds of arrows, see Fig. 5.

returns an **int**. The fact that the set union does not perform any transitive closure on reachability statements may be surprising, but is extensively discussed in Remark 1.

Denotations (4_c)*,* (5_c)*, and* (6_c)*.* Rules for loops and concatenation are easy, given the inductive hypothesis on the sub-commands, and the definition of the fixpoint. The rule for the **return** command is also easy, being basically similar to variable assignment.

Having proven that all abstract denotations are sound with respect to the concrete denotational semantics, together with Definition 5 and the definition of a denotational semantics, proves the theorem. □

Notes on the implementation. The present analysis has been implemented in the COSTA [3] COSt and Termination Analyzer. The implementation works as a component of COSTA, and deals directly with Java bytecode programs (not very different, in essence from the simple Java-like language). The acyclicity information is used by COSTA to prove the termination or infer the resource usage of programs. The implementation is still a prototype, but promising results have been obtained. In particular, it behaves as expected on the examples of the present paper, thus making COSTA able to deal with a larger class of programs (see, e.g., Fig 1). Due to lack of space, no experimental results are included, but it can be added, as a side note, that the prototype is reasonably efficient, and full integration into the analyzer is short-term future work.

5 Conclusions

The present paper studies the acyclicity of mutable data structures in the context of termination and resource usage analysis. A program P traversing a cyclic data structure d might not terminate since it might traverse an infinite path. Consequently, proving the acyclicity of d is crucial in order to guarantee the absence of such infinite paths through it.

The proposed acyclicity analysis is based on the observation that a field update `x.f=y` might create a new cycle iff `y` reaches `x` or aliases with it before the command. Two abstract domains are first defined, which capture the *may-reach* and *may-be-cyclic* properties. Then, an abstract semantics which works on their reduced product is introduced: it uses reachability information to improve the detection of cyclicity, and cyclicity to improve the tracking of reachability.

The analysis is proven to be sound; i.e., no cyclic data structure are ever considered acyclic. Moreover, it can be shown to obtain precise results in a number of non-trivial scenarios, where the sharing-based approach is less precise [14]. Indeed, since the existence of a directed path between the locations bound to two variables implies that such variables share, the proposed reachability-based analysis will never be less precise than the sharing-based approach. In particular, it is worth noticing that the reachability-based approach can often deal with directed acyclic graphs, whereas sharing-based techniques will consider, in general, any DAG as cyclic. A prototype implementation is available within COSTA [3].

References

1. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *Proc. of FMOODS'08*, volume 5051 of *LNCS*, pages 2–18. Springer, 2008.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proc. of ESOP'07*, volume 4421 of *LNCS*. Springer, 2007.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Proc. of FMCO'07*, volume 5382 of *LNCS*, pages 113–133. Springer, 2008.
4. J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proc. CAV*, 2006.
5. 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.
6. A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.
7. J. Brotherston, R. Bornat, and C. Calcagno. Cyclic proofs of program termination in separation logic. In *Proceedings of POPL-35*, Jan 2008.
8. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
9. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977.
10. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *POPL'79*, pages 269–282. ACM, 1979.
11. S. K. Debray and N. W. Lin. Cost analysis of logic programs. *ACM TOPLAS*, 15(5):826–875, November 1993.
12. S. Genaim and F. Spoto. Constancy Analysis. In M. Huisman, editor, *10th Workshop on Formal Techniques for Java-like Programs*, July 2008.
13. R. Jones and R. Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.
14. S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *Proc. of VMCAI'06*, volume 3855 of *LNCS*, pages 95–110. Springer, 2006.
15. S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *Proc. of SAS'05*, volume 3672 of *LNCS*, pages 320–335. Springer, 2005.
16. F. Spoto, F. Mesnard, and É. Payet. A Termination Analyser for Java Bytecode based on Path-Length. *Transactions on Programming Languages and Systems*, 32(3), 2010.
17. B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9), 1975.
18. R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *CC*, 2000.