Reachability-based acyclicity analysis by Abstract Interpretation

Samir Genaim a, Damiano Zanardini b,∗

a Complutense University of Madrid, Spain
b Technical University of Madrid, Spain

A R T I C L E   I N F O

Article history:
Received 7 June 2012
Received in revised form 14 November 2012
Accepted 14 December 2012
Communicated by D. Sannella

Keywords:
Abstract interpretation
Acyclicity analysis
Termination analysis
Object-oriented programming
Heap manipulation

A B S T R A C T

In programming languages with 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 provides an Abstract-Interpretation-based formalization of a 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, modeling the property that $v$ can point to a cyclic data structure. The analysis is proven to be sound and optimal with respect to the chosen abstraction.

© 2012 Elsevier B.V. All rights reserved.

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 [7,10,1,29], resource usage analysis [30,13,3], garbage collection [22], etc. As an example, 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) $x$. On the other hand, the possibility that $x$ points to a cyclic data structure forbids, in general, proving that the loop terminates.

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 $x > 0$, 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., they share a common part of the heap, then a cyclic structure might be created. This simple mechanism has been used in previous work [26] in order to declare x and y, among others, as (possibly) cyclic whenever they share before the update. In the following, we refer to this approach as the sharing-based approach to acyclic analysis.

The sharing-based approach to acyclicity is simple and efficient, however, there can be an important loss of precision in typical programming patterns. For example, 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. To this end, there are four possible scenarios: (1) x and y alias; (2) x reaches y; (3) y reaches x; (4) they both reach a common location. The field update x.f := y might create a cycle only in cases (1) and (3). An acyclicity analysis based on similar observations has been considered before in the context of C programs [17], where the analysis has been presented as a data-flow analysis; however, no formal justification for its correctness has been provided. In what follows, we refer to this approach as the reachability-based approach to acyclicity analysis.

1.1. Contributions

The main contribution of this paper is essentially theoretical. In particular, the paper formalizes an existing reachability-based acyclicity analysis [17] within the framework of Abstract Interpretation [11], and proves its soundness and optimality:

1. We define an abstract domain $A^r_x$, which captures the reachability information about program variables (i.e., whether there can be a path in the heap from the location $\ell_x$ bound to some variable $v$ to the location $\ell_w$ 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).

2. A provably sound and optimal abstract semantics $C^r_x(\_\_\_\_)$ of a simple object-oriented language is developed, which works on $A^r_x$ 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. With respect to the original analysis, the definition of the abstract semantics involves additional effort like dealing with specific features of object-oriented languages, and discussing some technical improvements.

As a proof of concept, the abstract semantics has been also implemented in the COSTA [2] COST and Termination Analyzer as a component whose result is an essential information for proving the termination or inferring the resource usage of programs written in Java bytecode. Focusing on full Java bytecode, the implementation has also to deal with advanced features of the language like exceptions and static fields.

The present paper is based on preliminary work by the same authors which was published as a short workshop version [15] and as a technical report [16].

1.2. Related work

A reachability-based acyclicity analysis for C programs was developed in [17]; however, that analysis was presented as a data-flow analysis, and it did not include any formal justification of its correctness. Our paper provides a formalization of a similar analysis in terms of Abstract Interpretation, and includes soundness proofs. Note that [17] uses the terms “direction” and “interference”, respectively, for reachability and sharing.

As far as Abstract-Interpretation-formalized cyclicity analyses are concerned, the one by Rossignoli and Spoto [26] is the most related work. This analysis is only based on sharing (not on reachability), and, as discussed in the paper, is less precise than the reachability-based approach.

The work on Shape Analysis [31] is related because it reasons about heap-manipulating programs in order to prove program properties. In most cases, safety properties are dealt with [6,27,25]. On the other hand, termination is a liveness property, and is typically the final property to be proven when analyzing acyclicity. Therefore, work on liveness properties will be considered more deeply. Most papers [24,4,7,10,9] use techniques based on Model Checking [23], Predicate Abstraction [20], Separation Logic [24] or Cyclic proofs [9] to prove properties for programs which work on single-linked heaps. This means that only one heap cell is directly reachable from another one, which is basically the same as having, in an object-oriented language, only one class with one field. This somehow restricts the structure of the heap and, in some cases, allows obtaining more precise results. On the contrary, the present paper deals with a technique which does not rely on such an assumption: as the language is object-oriented, every object can have multiple fields. Other works [5] deal with single-parent heaps, which are multi-linked but sharing-free; needless to say, the present paper handles heap structures where sharing is more than a possibility. There also exist other works [19] based on Separation Logic which efficiently prove program properties and deal with cyclic structures, but are specialized to a limited set of data structures like single-linked lists, double-linked lists or trees. Also, in most of these works, the heap size is bounded by some constant, which is also a minor limitation. On the contrary, the present paper deals with data structures which can have practically any shape, and tries to infer information about the shape on its own. It is convenient to point out that the acyclicity analysis under discussion does not focus on
class OrderedList {
    Node head, lastInserted;

    int insert(int i) {
        Node c, p, n;
        int pos;
        pos:=0;
        n:=new Node;
        n.value:=i;
        c:=this.head;
        while (c!=null && c.value<i) {
            pos:=pos+1;
            p:=c;
            c:=c.next;
            if (p=null) then this.head:=n;
            else p.next:=n;
            this.lastInserted:=n;
        }
        return pos+1;
    }
}

class Node {
    Node next;
    int value;
}

Fig. 1. The running example and the result of the analysis, put in comments.

Fig. 2. A graphical representation of the data structure on which the example works.

directly proving liveness properties; instead, it is supposed to provide useful information to a cost\(^1\) or termination analyzer which will perform the task.

1.3. Organization

The rest of the paper is organized as follows: Section 2 presents an example of reachability-based acyclicity analysis. Section 3 defines the syntax and semantics of a simple Java-like language. Section 4 introduces the abstract domains for reachability and cyclicity, and their reduced product, and Section 5 defines the abstract semantics, and proves some important properties. Finally, Section 6 concludes the paper. Proofs of the technical results are available in Appendix.

2. An example of reachability-based acyclicity analysis

This section describes the essentials of the reachability-based acyclicity analysis [17], and its advantages over the sharing-based one, by mean of an example. This example will also be used in the rest of the paper to illustrate the different technical parts of the analysis.

Consider the program depicted in Fig. 1. The class OrderedList implements an ordered linked list with two fields: head and lastInserted point to, respectively, the first element of the list and the last element which has been inserted. The class Node implements a linked list in the standard way, with two fields value and next. Fig. 2 shows a possible instance of OrderedList.

\(^1\) This analysis is actually implemented in COSTA, which handles both cost and termination of the Java bytecode programming language.
The method insert adds a new element to the ordered list: it takes an integer i, creates a new node n for i (lines 9–10), looks
for the position pos of n (lines 11–16), adds n to the list (lines 17–20), makes lastInserted point to the new node (lines 22),
and finally returns pos (lines 23). The goal is to infer that a call of the form “x.insert(i)” never makes x cyclic. This is important
since, when such a call is involved in a loop like following one

1. x:=new OrderedList;
2. 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 be traversing a cycle.

The challenge in this example is to prove that the instructions at lines 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 them as possibly cyclic, as done, for example, by
Rossignoli and Spoto [26].

3. 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 X of valid identifiers. A program consists of a set of classes ∑ ∈ X ordered by the subclass
relation ⊆. Following Java, a class declaration takes the form “class k1 [extends k2] { t1 f1; . . . ; tn fn; M1 . . . Mk}” where each “tif”
declares the field fi to have type ti ∈ ∑ ∪ {int}, and each Mi is a method definition. Similarly to Java, the optional statement
“extends k2” declares k1 to be a subclass of k2. A method definition takes the form “t m (t1 w1, . . . , tn wn) {n1+1 u w1+1; . . . ; n+p u w+p; com}”
where: t ∈ ∑ ∪ {int} is the type of the return value; w1, . . . , wn ∈ X are the formal parameters; wn+1, . . . , wn+p ∈ X
are local variables; and com is a sequence of instructions according to the following grammar:

\[ \text{exp ::= } n \mid \text{null} \mid v \mid f \mid \text{exp} \oplus \text{exp} \mid \text{new } \kappa \mid v.m(\bar{v}) \]

\[ \text{com ::= } v:=\text{exp} \mid v.f:=\text{exp} \mid \text{com}_1 \text{com}_2 \mid \text{if } \text{exp } \text{then } \text{com}_1 \text{ else } \text{com}_2 \mid \text{while } \text{exp } \text{do } \text{com } \mid \text{return } \text{exp} \]

where v, \bar{v}, m, f ∈ X; n ∈ Z; \kappa ∈ ∑; and \oplus is a binary operator (Boolean operators return 1 for true and 0 for false).
For simplicity, and without loss of generality, conditions in if and while statements are assumed not to create objects or
call methods. A method signature \kappa.m(t1, . . . , tn):t refers to a method m defined in class \kappa, taking n parameters of type
t1, . . . , tn ∈ ∑ ∪ {int}, and returning a value of type t. Given a method signature \kappa.m, let m be its code \kappa.m; m's set of input
tvariables {this, w1, . . . , wn}; \kappa.m's set of local variables \{wn1+1, . . . , wn+m\}; and \kappa.m = m' \sqcup m''

A type environment τ is a partial map from X to ∑ ∪ {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; i.e., the partial map
will be confused with its domain when the type of variables can be ignored. A state over τ is a pair consisting of a frame and
a heap. A heap µ is a partial mapping from an infinite and totally ordered set \mathcal{L} of memory locations to objects; µ(ℓ)
the object bound to \ell ∈ \mathcal{L} in the heap µ. An object o ∈ Θ is a pair consisting of a class tag o.tag ∈ ∑, and a frame o.frm which
maps its fields into \mathcal{V} = \mathbb{Z} ∪ \mathcal{L} ∪ \text{null}. Shorthand is used: o.f for o.frm.f(); µ(ℓ) → o to modify the heap µ such that a new
location \ell points to object o; and µ[ℓ.f → v] to modify the value of the field f of the object µ(ℓ) to v ∈ \mathcal{V}. A frame \phi maps
variables in dom (τ) to \mathcal{V}. For v ∈ dom (τ), (φ(v) refers to the value of v, and φ[v → v] is the frame where the value of v
has been set to v, or defined to be v if v \notin dom (φ). The set of possible states over τ is

\[ \Sigma_\tau = \{ \phi, \mu \mid \begin{array}{ll} 1. & \phi \text{ is a frame over } \tau, \mu \text{ is a heap, and both are well-typed} \\ 2. & \text{rng}(\phi) \cap \mathcal{L} \subseteq \text{dom } (\mu) \\ 3. & \forall \ell \in \text{dom } (\mu), \text{rng}(\mu(\ell).\text{frm}) \cap \mathcal{L} \subseteq \text{dom } (\mu) \end{array} \} \]

Given σ ∈ Σ, δ and σ refer to its frame and its heap, respectively. The complete lattice \mathcal{I}_\tau = (\phi(\Sigma_\tau), \Sigma_\tau, ∅, ∩, ∪) defines the concrete computation domain.

A denotation δ over two type environments τ1 and τ2 is a partial map from σ(τ1) to σ(τ2); it basically describes how the
state changes when a piece of code is executed. The set of denotations from τ1 to τ2 is \Delta(τ1, τ2). Interpretations are special
denotations which give a meaning to methods in terms of their input and output variables. An interpretation δ ∈ \Gamma maps
methods to denotations, and is such that \delta(m) ∈ \Delta(\mathbb{m}', \mathbb{out}) for each signature m in the program. Note that the variable
output is a special variable which will be used to denote the return value of a method.

Denotations for expressions and commands are depicted in Fig. 3. An expression denotation \mathcal{E}_j[\mathcal{E}] maps states from
\Sigma_j to states from \Sigma_j(\rho), where \rho is a special variable for storing the expression value. A command denotation \mathcal{C}_j[\mathcal{C}]
maps states to states, in presence of c ∈ \Gamma. The function newobj(κ) creates a new instance of the class κ with integer fields
initialized to 0 and reference fields initialized to null, while newloc(δ) returns the first free location, i.e., the first \ell \notin \text{dom } (δ)
according to the total ordering on locations. The function lkp resolves the method call and returns the signature of the method.
The acyclicity analysis discussed in this paper works on the reduced product [12] of two abstract domains, according to the theory of Abstract Interpretation [11]. 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 by accessing object fields.

Definition 4.1 (Reachable Heap Locations [26]). Given a heap $\mu$, the set of reachable locations from $\ell \in \text{dom}(\mu)$ is $R(\mu, \ell) = \bigcup \{ R^i(\mu, \ell) \mid i \geq 0 \}$, where $R^0(\mu, \ell) = \text{rng}(\mu(\ell).\text{frm}) \cap L$, and $R^{i+1}(\mu, \ell) = \bigcup \{ \text{rng}(\mu(\ell').\text{frm}) \cap L \mid \ell' \in R^i(\mu, \ell) \}$. The set of $\epsilon$-reachable locations from $\ell \in \text{dom}(\mu)$ is $R^\epsilon(\mu, \ell) = R(\mu, \ell) \cup \{ \ell \}$.

Note that $\epsilon$-reachable locations include the source location $\ell$ itself, while reachable locations do not (unless $\ell$ is reachable from itself through a cycle whose length is at least 1). The rest of this section is developed in the context of a given type environment $\tau$.

4.1. Reachability

Given a state $\sigma \in \Sigma$, a reference variable $v \in \tau$ is said to reach a reference variable $w \in \tau$ in $\sigma$ if $\hat{\sigma}(w) \in R(\hat{\sigma}, \hat{\sigma}(v))$. This means that, starting from $v$ and applying at least one dereference operation (i.e., going from the location pointed to by $v$ to the location pointed to by $v . f$ for some field $f$), it is possible to reach the object to which $w$ points. Due to strong typing, $\tau$ puts some restrictions on reachability; i.e., it might be impossible to have a heap where a variable of type $\kappa_2$ reaches one of type $\kappa_2$. Following Secci and Spoto [28], a class $\kappa_2 \in \mathcal{K}$ is said to be reachable from $\kappa_1 \in \mathcal{K}$ if there exist $\sigma \in \Sigma$, and two locations $\ell$, $\ell' \in \text{dom}(\hat{\sigma})$ such that (a) $\hat{\sigma}(\ell) . \text{tag} = \kappa_1$; (b) $\hat{\sigma}(\ell') . \text{tag} = \kappa_2$; and (c) $\ell' \in R(\hat{\sigma}, \ell)$. The use of this notion (as well as the notion of cyclic class introduced in Section 4.2 and used in Definition 4.5) in the definition of the reachability and cyclicity domains allows us to obtain the needed Galois insertions. It must be pointed out that both notions can be computed statically, so that they can be assumed to be pre-computed information.
Definition 4.2. (Reachability Domain). The reachability abstract domain is the complete lattice \( I^r = (\varnothing(\mathcal{R}^r), \subseteq, \emptyset, \mathcal{R}^r, \cap, \cup) \), where
\[
\mathcal{R}^r = \left\{ \langle v \leadsto w \rangle \mid v, w \in \text{dom}(\tau), \text{and there exist } \kappa_1 \leq \tau(v) \text{ and } \kappa_2 \leq \tau(w) \text{ such that } \kappa_2 \text{ is reachable from } \kappa_1 \right\}.
\]

Here and in the following, elements of the tuple \((A, \leq, \bot, \top, \land, \lor)\) denoting an abstract domain \(A\) represent, respectively, \((A)\) the set of abstract values, \((\leq)\) the partial order on them, \((\bot)\) the minimal (bottom) element of \(A\), \((\top)\) the maximal (top) element of \(A\), \((\land)\) the meet operator and \((\lor)\) the join operator on \(A\). This terminology is standard in Abstract Interpretation.

May-reach information is described by abstract values \( I_r \in \varnothing(\mathcal{R}^r) \). For example, \( \{x \leadsto y, y \leadsto z\} \) describes those states where \( x \) and \( y \) may reach \( z \). Note that a statement \( x \leadsto y \) 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 4.3. The following abstraction and concretization functions define a Galois insertion between \( I^r \) and \( I^r \):
\[
\alpha^r_i(l_i) = \{v \leadsto w \in \mathcal{R}^r \mid \exists \sigma \in l_i, v \text{ reaches } w \text{ in } \sigma\}
\]
\[
\gamma^r_i(l_i) = \{\sigma \in \Sigma_r \mid \forall v, w \in \tau. v \text{ reaches } w \text{ in } \sigma \Rightarrow v \leadsto w \in l_i\}.
\]
The top element \( \mathcal{R}^r \) is \( \alpha^r_i(\Sigma_r) \), and represents all states which are compatible with \( \tau \). This is because the presence of a reachability statement in an abstract value \( I \) does not require a reachability path to actually exist; rather, the concretization of \( I \) will include states where the path does exist, and states where it does not (this is the meaning of “may-information”). In other words, the absence of a reachability statement in the abstract state requires non-existence of a reachability path in its concretization.

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 \( w \). Note that, clearly, this set is not empty, and that the absence of a reachability statement actually rules out states where the reachability path exists.

Remark 4.4. Intuitively, reachability is a transitive property; i.e., if \( x \) reaches \( y \) and \( y \) reaches \( z \), then \( x \) also reaches \( z \). However, values in \( I^r \) are not closed by transitivity: e.g., it is possible to have \( l_r = \{x \leadsto y, y \leadsto z\} \) which contains \( x \leadsto y \) and \( y \leadsto z \), but not \( x \leadsto z \). Such an abstract value is a reasonable one, and approximates, for example, the execution of the following code

```python
1  x:=new C;
2  y:=new C;
3  if (w>0) then x:=y; else y:=z;
```

Moreover, this abstract value is consistent, i.e., it describes a set of concrete states which is not smaller (actually, it is greater) than \( \gamma^r_i(\emptyset) \). This happens because reachability is, actually, may-reach information, so that, for example, \( \gamma^r_i(\{x \leadsto y, y \leadsto z\}) \) 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 \( x \) and \( y \) do not reach \( z \). It is important to point out that \( \gamma^r_i(\{x \leadsto y, y \leadsto z\}) \) 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 by soundness since \( x \leadsto z \notin l_r \).

4.2. Cyclicity

Given a state \( \sigma \in \Sigma_r \), a variable \( v \in \text{dom}(\tau) \) is said to be cyclic in \( \sigma \) if there exists \( \ell \in \mathcal{R}(\hat{\sigma}, \hat{\sigma}(v)) \) such that \( \ell \in \mathcal{R}(\hat{\sigma}, \ell) \). In other words, \( v \) is cyclic if it reaches some memory location \( \ell \) (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 \( \kappa \). A class \( \kappa \in \mathcal{K} \) is said to be a cyclic class if there exist \( \sigma \in \Sigma_r \) and \( \ell, \ell' \in \text{dom}(\hat{\sigma}) \) such that \( \hat{\sigma}(\ell).\text{tag} = \kappa \), \( \ell' \in \mathcal{R}(\hat{\sigma}, \ell) \), and \( \ell' \in \mathcal{R}(\hat{\sigma}, \ell') \). The cyclicity domain is the dual of the non-cyclicity domain by Rossignoli and Spoto [26].

Definition 4.5. (Cyclicity Domain). The abstract domain for cyclicity is represented as the complete lattice \( I^c = (\varnothing(\mathcal{Y}^c), \subseteq, \emptyset, \mathcal{Y}^c, \cap, \cup) \) where
\[
\mathcal{Y}^c = \{\gamma^c | \gamma \in \mathcal{Y}^c, \text{and there exists a cyclic class } \kappa \leq \tau(\gamma)\}.
\]

Lemma 4.6. The following abstraction and concretization functions define a Galois insertion between \( I^c \) and \( I^c \):
\[
\alpha^c_i(l_i) = \{\gamma^c | \exists \sigma \in l_i, \exists \gamma \in \mathcal{Y}^c, \gamma \text{ is cyclic in } \sigma\}
\]
\[
\gamma^c_i(l_i) = \{\sigma | \sigma \in \Sigma_r \land \exists \gamma \in \mathcal{Y}^c, (\gamma \text{ is cyclic in } \sigma) \Rightarrow \gamma^c \in l_i\}.
\]

May-be-cyclic information is described by abstract values \( l_i \in \varnothing(\mathcal{Y}^c) \). For instance, \( \{\gamma^c\} \) represents states where no variable but \( x \) can be cyclic. The top element \( \mathcal{Y}^c \) is concretized to \( \Sigma_r \); i.e., all state are included since each variable can be either cyclic or acyclic. The bottom element \( \emptyset \) does not allow any variable to be cyclic, i.e., its concretization does not include any state with cyclic variables.

4.3. The reduced product

As it will be explained in Section 5, the abstract semantics uses reachability information in order to detect cycles, and cyclicity information in order to add, in some cases, reachability statements. Both kinds of information can be combined.
in the theory of Abstract Interpretation; this amounts to computing the reduced product [12] of the corresponding abstract domains. In the present context, the reduced product is obtained by reducing the Cartesian product $I^*_κ = I^*_κ \times I^*_γ$. Elements of $I^*_κ$ are pairs $(l_κ, l_γ)$, where $l_κ$ and $l_γ$ contain, respectively, the may-reach and the may-be-cyclic information. The abstraction and concretization functions are induced by those on $I^*_κ$ and $I^*_γ$:

$$γ^*_κ((l_κ, l_γ)) = γ^*_κ(l_κ) \cap γ^*_γ(l_γ) \quad α^*_κ(l) = (α^*_κ(I), α^*_γ(I)).$$

However, it can happen that two elements of $I^*_κ$ are mapped to the same set of concrete elements, which prevents having a Galois insertion between $I^*_κ$ and $I^*_γ$. The operation of reduction deals exactly with this problem. In order to compute it, an equivalence relation $≡$ has to be defined, which satisfies $I^*_κ \equiv I^*_γ$ is and only if $γ^*_κ(l^*_κ) = γ^*_γ(l^*_γ)$. Functions $γ^*_κ$ and $α^*_κ$ define a Galois insertion between $I^*_κ \equiv I^*_γ$, where $I^*_κ \equiv I^*_γ$ is $I^*_κ$ equipped (reduced) with the equivalence relation. The following lemma characterizes the equivalence relation on $I^*_κ$.

**Lemma 4.7.** For any abstract values $l^*_κ, l^*_κ' \in I^*_κ$ and $l^*_κ, l^*_κ' \in I^*_κ$, the concretization $γ^*_κ((l^*_κ, l^*_κ'))$ is equal to $γ^*_κ((l^*_κ', l^*_κ))$ if and only if both conditions hold: (a) $l^*_κ = l^*_κ'$ and (b) $l^*_κ' \setminus \{v \mapsto v' \mid ∅ \not\subseteq l^*_κ\} = l^*_κ \setminus \{v \mapsto v' \mid ∅ \not\subseteq l^*_κ\}$.

This 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 $v \mapsto v'$ to $l_κ$ results in representing a strictly larger set of states; and (b) adding a pair $v \mapsto v'$ to $l_κ$ in $I^*_κ$, 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 4.8.** As an example for case (a), consider two abstract values $l^*_κ = (l_κ, ∅)$ and $l^*_κ = (l_κ, \{x\})$ which result from adding $x \mapsto x$ to $l_κ$. Assuming that $x$ does not appear in $l_κ$, there is a state $σ$ which is compatible with $l_κ$ (for example, if $v$ reaches any $w$ in $σ$), and where $x$ is cyclic (note that this does not require $x$ to reach any other variable, not even itself, since the cycle does not need to go through $σ(x)$). This $σ$ belongs to $γ^*_κ(l^*_κ') \setminus γ^*_κ(l^*_κ)$ and is, therefore, an example of the difference between the abstract values.

As an example for (b), consider $l^*_κ = (∅, \{y\})$ and $l^*_κ = (\{x\} \setminus x, \{y\})$ which results from adding $x \mapsto x$ to $l_κ$. At a first glance, $l^*_κ$ describes a larger set of states, since it includes states (not belonging to $γ^*_κ(l^*_κ)$) where there is a path from $x$ to $x$. However, such states will neither belong to $γ^*_κ(l^*_κ')$, since such a path implies that $x$ is cyclic, which is not permitted by $\{y\}$, that only allows $y$ to be cyclic.

**Lemma 4.7** provides a way for computing the normal form of any $(l_κ, l_γ)$, which comes to be $(l_κ \setminus \{v \mapsto v \mid ∅ \not\subseteq l_κ\}, l_γ)$, i.e., the canonical form of its equivalence class. From now on, $I^*_κ$ will be a shorthand for $I^*_κ \equiv I^*_γ$, where $≡$ is left implicit.

5. Reachability-based acyclicity analysis

This section uses $I^*_κ$ to define an abstract semantics from which one can decide whether a variable $v$ is (or may not be) bounded to an acyclic data structure at a given program point. Informally, two variables $v$ and $w$ are said to share in a state $σ$ if they $e$-reach (i.e., in zero or more steps) a common location in the heap. The analysis is based on the observation that reachability information can tell how $v$ and $w$ share: this can happen because either (a) $v$ and $w$ alias; (b) $v$ reaches $w$; (c) $w$ reaches $v$; or (d) they both reach $ℓ ∈ \text{dom}(σ)$. Distinguishing among these four possibilities is crucial for a precise acyclicity analysis. Instead of trying to identify $v$ and $w$ as actually acyclic, they both become acyclic after executing $v, w = w$ if and only if, initially, $v$ either reaches $v$ or aliases with it. This is clearly more precise than declaring $v$ as cyclic whenever it was sharing with $w$ [26]. The presented analysis is an adaptation of the work by Ghiya and Hendren [17] to an object-oriented framework, where the chosen formalism is that of an abstract semantics on the domain described in Section 4.

Some optimizations w.r.t. the original analysis are also discussed.

The rest of this section formalizes the reachability-based analysis as an abstract semantics on $I^*_κ$, and proves some important results.

5.1 Preliminaries

May-share [28], may-alias [21] and purity [14] analyses are used as pre-existent components, i.e., programs are assumed to have been analyzed w.r.t. these properties by means of state-of-the-art tools. Two reference variables $v$ and $w$ share in $σ$ iff $R^e(σ, v) \cap R^e(σ, w) \neq ∅$; also, they alias in $σ$ if they point to the same location, namely, if $σ(v) = σ(w) ∈ \text{dom}(σ)$. Any non-null reference variable shares and aliases with itself; also, both are symmetric relations. 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 proposed by Genaim and Spoto [14] {base: previous work by Secci and Spoto [28]} can be applied: with it,
1. it is possible to know if \( v \) may share with \( w \) at any program point (denoted by the sharing statement \( \langle v \bullet w \rangle \)); and
2. for each method \( m \), a denotation \( \text{SP}_m \) is given: for a set of pairs \( I_\rho \) which safely describes the sharing between actual arguments in the input state, \( I_\rho = \text{SP}_m(I_\rho) \) is such that (i) if \( \langle v \bullet w \rangle \in I_\rho \), then \( v \) and \( w \) might share during the execution of \( m \); and (ii) \( v \in I_\rho \) means that the \( i \)-th argument might be non-pure.

According to the theory of Abstract Interpretation and to previous work, sharing and purity analysis can be defined as an abstract semantics over the abstract domain \( I_\rho \), whose elements \( I_\rho \) contain may-share statements \( \langle v \bullet w \rangle \) and may-be-non-pure statements \( \tilde{v} \). Abstraction and concretization functions \( \alpha_{\text{SP}} \) and \( \gamma_{\text{SP}} \) are defined in the standard way [14]: in particular, \( \gamma_{\text{SP}}(I_\rho) \) contains all the states where variables mentioned in sharing statements are the only ones which can possibly share between themselves, while variables mentioned in may-be-non-pure statements are the only ones which can possibly be non-pure.

As for aliasing, the abstract domain \( I_\rho \) contains sets of may-alias statements \( \langle v \bullet w \rangle \) if \( \langle v \bullet w \rangle \) is contained in \( I_\rho \), then its concretization \( \gamma_{\text{al}}(I_\rho) \) contains states where \( v \) and \( w \) actually alias and states where they do not. It is assumed that this information is available at each program point as a set of may-alias statements.

In the following, the domain \( I_\rho \) will be the reduced product between \( I_{\text{sp}} \) and \( I_{\text{al}} \), and combines sharing, aliasing, and purity information. As usual, \( \gamma(I_{\text{sp}}, I_{\text{al}}) \) is defined as \( \gamma_{\text{sp}}(I_{\text{sp}}) \cap \gamma_{\text{al}}(I_{\text{al}}) \), while \( \alpha_{\text{al}}(X) \) is defined as \( \alpha_{\text{al}}(X) \cup \alpha_{\text{al}}(X)_{\text{m}} \), where \( \equiv \) means that abstract elements with the same concretization have been unified (i.e., the product has been reduced).

Abusing notation, from now on \( I_\rho \) will be often used to denote an abstract value without specifying the abstract domain it belongs to. The use of \( I_\rho \) will be clear from the context: for example, writing \( \gamma_{\text{al}}(I_{\text{al}}) \) means applying \( \gamma_{\text{al}} \) to the part of \( I_{\text{al}} \) which represents aliasing information.

Moreover, an abstract element \( \langle i, I \rangle \in I_\rho \) will be represented by the set \( I = I \cup I \); therefore, \( v \sim w \in I \) and \( \sim \in I \) are shorthands for, respectively, \( v \sim w \in I \) and \( \sim \in I \). 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 \( v \sim w \) is not added to an abstract state if \( v \sim w \not\in R^\ast \), while a statement \( \sim \in I \) is not added if \( \sim \not\in \gamma^\ast \). It is important to point out that information about class-reachability and class-cyclicity (i.e., whether \( \kappa_1 \) reaches \( \kappa_2 \), or whether \( \kappa \) is cyclic) can be computed statically and before performing any acyclicity analysis. Therefore, it can be assumed that such information is available whenever it is necessary to decide whether a new reachability or cyclicity statement belongs or not to \( R^\ast \) or \( \gamma^\ast \).

5.2. The abstract semantics

An abstract denotation \( \xi \) from \( \tau_1 \) to \( \tau_2 \) is a partial map from \( I_{\text{rec}}^\tau_1 \) to \( I_{\text{rec}}^\tau_2 \). It describes how the abstract input state changes when a piece of code is executed. The set of all abstract denotations from \( \tau_1 \) to \( \tau_2 \) is denoted by \( \mathcal{E}(\tau_1, \tau_2) \). As in the concrete setting, interpretations provide abstract denotations for methods in terms of their input and output arguments. An interpretation \( \xi \) maps methods to abstract denotations, and is such that \( \xi(m) \in \mathcal{E}(m^\ast, \rho^\ast \cup \{\text{out}\}) \). Note that the range of such denotations is \( m^\ast \cup \{\text{out}\} \), instead of \{out\} (as in the concrete semantics): this point will get clarified below. Finally, \( \Psi \) denotes the set of all (abstract) interpretations.

Figs. 4 and 5 depict abstract denotations. An expression denotation \( \xi [\text{exp}] \) maps abstract states from \( I_{\text{rec}}^\tau \) to abstract states from \( I_{\text{rec}}^\gamma \) where \( \tau = \tau \cup \{\rho\} \), while a command denotation \( \xi [\text{com}] \) maps \( I_{\text{rec}}^\tau \) to \( I_{\text{rec}}^\gamma \).
In the definition, the abstract element $l_i$ contains the sharing, aliasing, and purity information precomputed by other analyses, and referring to the program point of interest.\footnotemark

\footnotetext{Note that $l_i$ could be represented explicitly as an input to the abstract semantics, next to $l$, but it is not written for better clarity.}

\subsection{5.2.1. Expressions}

An expression denotation $\xi_i^{\, \tau}[\exp]$ adds to an input state $l$ those reachability and cyclicity statements which result from evaluating $\exp$.

Nothing is added to $l$ in cases (1$_x$): $\xi_i^{\, \tau}[\Int{v}]$, (2$_x$): $\xi_i^{\, \tau}[\Null{v}]$, and (3$_x$): $\xi_i^{\, \tau}[\New{\kappa}]$ since the expression is evaluated without side effects, to, respectively, an integer value, null, or a newly allocated object which is not related to any other location with respect to reachability.

The same reasoning explains why the returned abstract value is also $l$ in case (4$_x$): $\xi_i^{\, \tau}[\Int{v}]$ when $\tau(v) = \Int$, and (5$_x$): $\xi_i^{\, \tau}[\Int{v}, \exp]$ when $f$ is an int field.

In case (4$_x$): $\xi_i^{\, \tau}[\Int{v}]$, when the type of $v$ is not int, the result variable $\rho$ has the same abstract behavior as $v$. Therefore, the semantics returns $l$, together with a cloned version $l[v/\rho]$ where statements about $v$ have been replaced by renamed statements about $\rho$.

In the case of (5$_x$): $\xi_i^{\, \tau}[\Int{v}, f]$, when $f$ is a reference field, the following information is added to $l$:

\begin{itemize}
  \item statements for \(v\) which are cloned for $\rho$
  \item \(w \rightarrow \rho\), if \(w\) might share with \(v\); note that \(v \rightarrow \rho\) is always added since \(v \rightarrow v\) (clearly, \(v\) cannot be null); if $v$ and $w$ reach a common location (which implies that they share), but do not reach each other, then, conservatively, the reachability statement $w \rightarrow \rho$ must be added because $v.f$ could be exactly the common location which is reached by both $v$ and $w$
  \item if $v$ might be cyclic, then, for soundness, $\rho \rightarrow \rho$; note that, in this case, $\rho^\rho$ is also guaranteed to have been previously added to the abstract state.
\end{itemize}

In case (6$_x$): $\xi_i^{\, \tau}[\Int{\exp_1 \oplus \exp_2}]$, the expression $\exp_1$ is first analyzed, then $\exp_2$ is analyzed on the resulting abstract state. Note that, in both cases, $\rho$ is removed since the return value has always type int.

Finally, method calls (7$_x$): $\xi_i^{\, \tau}[\Call{v_0, \dots, v_n}]$ will be explained later, after introducing denotations for commands.

\begin{example}
Consider $c := \Call{\next}$ at line 15 in Fig. 1. Evaluating the denotation $\xi_i^{\, \tau}[\Call{\next}]$ results in $\{\Call{\next} \rightarrow \Call{\next}, \Call{\next} \rightarrow c, c \rightarrow \Call{\next}, \Call{\next} \rightarrow c \rightarrow \Call{\next}, \Call{\next} \rightarrow \Call{\next}, c \rightarrow \Call{\next}, c \rightarrow c \rightarrow \Call{\next}\}$. The statement $c \rightarrow \Call{\next}$ is added since $c \rightarrow \Call{\next}$ and $\Call{\next} \rightarrow \Call{\next}$ are added because $(c \Call{\next})$ and $(\Call{\next} \Call{\next})$ hold after line 14.
\end{example}

\subsection{5.2.2. Variable assignment}

The denotation (1$_x$): $\xi_i^{\, \tau}[\Call{v} := \Call{\exp}]$ computes $\xi_i^{\, \tau}[\Call{\exp}]$, removes any statement about $v$ since it takes a new value, and finally renames $\rho$ to $v$. Note that it is safe to remove statements about $v$ since it is first cloned to $\rho$.

\begin{example}
Consider, again, line 15 in Fig. 1. Evaluating the denotation $\xi_i^{\, \tau}[\Call{c := \Call{\next}}]$ first computes $\xi_i^{\, \tau}[\Call{\next}]$ as in Example 5.1. Then, statements involving $c$ are removed, which results in $\{\Call{\next} \rightarrow \Call{\next}, \Call{\next} \rightarrow \Call{\next}, \Call{\next} \rightarrow \Call{\next}, c \rightarrow \Call{\next}, \Call{\next} \rightarrow \Call{\next}, c \rightarrow \Call{\next}, c \rightarrow c \rightarrow \Call{\next}\}$. Note that $c \rightarrow \Call{\next}$ is reinserted (by renaming $\Call{\next} \rightarrow \Call{\next}$) after being deleted by $3c$. Also, note that $\Call{\next} \rightarrow \Call{\next}$ has been removed by $3c$, so that, correctly, $c$ is not considered to reach itself after the assignment.
\end{example}
5.2.3. Field update

The denotation (2.7): \(E^+_x[v.f:=\text{exp}]\) accounts for field updates. The set \(I'_0\) results from computing \(E^+_x[\text{exp}](l)\), as usual. The following step is to apply an optimization (called the single-field optimization in the following) which allows removing statements after inspecting the declarations of the classes involved in the update. The abstract value \(I'\) is computed from \(I'_0\) by the function \(\text{condRemove}(I'_0, v, f)\), which is defined as follows. Let \(\kappa\) be the declared class of \(v\) (this means that the runtime type of \(v\) can be \(\kappa\) or any of its subclasses); then \(I'\) is obtained by \(I'_0\) by

- removing \(c^+\) if (1) \(f\) is the only reference field of any \(\kappa' \leq \kappa\); or (2) all the other reference fields of any \(\kappa' \leq \kappa\) have a declared class such that neither it nor any of its subclasses is a cyclic class;
- similarly, removing any statement \(v \rightarrow w\) such that \(f\) is the only field of any \(\kappa' \leq \kappa\) whose declared class \(\kappa_f\) (or any of its subclasses) reaches the declared class of \(w\) (or any of its subclasses);
- leaving all the statements in \(I'_0\) if these conditions do not hold.

Basically, this single-field optimization identifies cases where the only cycles or reachability paths starting from \(v\) must forcefully traverse \(f\), either because \(f\) is the only field, or because no other field makes such cycles or paths possible. It must be pointed out that this optimization relies on information about classes and fields which can be obtained statically by code inspection, and was not included in the original analysis of Ghiya and Hendren [17]. The sets \(I_1\) and \(I_c\) capture the effect of executing \(v.f:=\text{rho}\) on \(I'\). Moreover, \(I'_0\) contains sharing, purity and aliasing information after evaluating \(\text{exp}\).

The following reachability statements are added: for any \(w_1\) which might either alias with \(v\) or reach \(v\) (formalized as \((w_1.v) \in I'_0\) \lor \((w_1.v) \in I'')\), and any \(w_2\) aliasing with \(\rho\) or reachable from it from \((\rho.w_2) \in I'_0\) \lor \((\rho.w_2) \in I''\), the statement \(w_1 \rightarrow w_2\) is added since the new path created by the update implies that \(w_1\) can reach \(w_2\). This accounts for all possible paths which can be created by adding a direct link from \(v\) to \(\rho\) through \(f\).

New cyclicity statements are contained in \(I_c\). There are three possible scenarios where \(v\) might become cyclic:

- \(\rho\) reaches \(v\), so that a cycle from \(v\) to itself is created;
- \(\rho\) aliases with \(v\), so that \(v\) reaches itself with a path of length 1 (e.g., the command \(v.f:=\text{rho}\));
- \(\rho\) is cyclic, so that \(v\) becomes indirectly cyclic.

Whenever one of these scenarios occurs (formalized as \((\rho \rightarrow v) \in I'\) \lor \((\rho \rightarrow v) \in I''\)), any variable \(w\) aliasing with \(v\) or reaching it (formalized as \((w.v) \in I'_0\) \lor \((w.v) \in I''\)) has to be considered as possibly cyclic.

Example 5.3. Consider line 20 in Fig. 1. The abstract value before such a line, produced at line 17, is \(I_{17} = (\text{this} \rightarrow c, \text{this} \rightarrow p, p \rightarrow c, n \rightarrow c)\). The evaluation of \(E^+_x[p.\text{next}=n](l_{17})\) at line 20 adds a new statement \(p \rightarrow n\), as expected. Moreover, it also adds \(this \rightarrow n\) since this was reaching \(p\) and both \(p \rightarrow c\) and \(this \rightarrow c\) (which, however, were already contained in \(I_{17}\)) since \(n\) was reaching \(c\).

5.2.4. Conditions, loops, composition, and return command

Rules (3.,) \(E^+_x[\text{if exp then com}_1\ \text{else com}_2]\), (4.,) \(E^+_x[\text{while exp do com}]\), and (6.,) \(E^+_x[\text{com}_1;\ \text{com}_2]\) are quite straightforward and correspond, respectively, to the if conditional, the while loop, and command composition. Finally, rule (5.,) \(E^+_x[\text{return exp}]\) corresponds to the return command, and behaves, as expected, like the execution of \(\text{out}:=\text{exp}\).

5.2.5. Method calls

Rule (7.,) \(E^+_x[v_0.m(v_1,\ldots, v_n)]\) propagates the effect of a method call to the calling context, as follows:

1. the abstract state \(l\) is projected on the actual parameters \(\hat{v}\), thus obtaining \(l_0\); 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 possibly called at runtime is taken from the current interpretation, namely, \(\zeta(m)\), and applied to \(l_0[\hat{v}/m']\), which is the result of renaming the actual parameters \(\hat{v}\) to the formal parameters \(m'\) in \(l_0\);
3. formal parameters are renamed back to the actual parameters (plus \(\text{out}\) and \(\rho\)) in the resulting state \(\zeta(m)(l_0[\hat{v}/m'])\), and the states obtained from all possible signatures are merged into \(l_n\).

Step 2 takes more than one method into account because, in an Object-Oriented language with inheritance, it is in general not possible to decide, at compile-time, which method instance (among various method declarations whose signature is compatible with the type of the actual parameters and the expected return value) will be actually invoked after calling the function \(\text{meth}\) (Section 3). Therefore, the abstract semantics takes, conservatively, the union of all of them.

In the definition, \(I'\) is a safe approximation of the sharing among actual parameters, and \(I''\) safely approximates the sharing and purity information after the method call. The definitions of \(I_1, I_2, I_3, I_4\) account for the propagation of the effects of the method execution in the calling context:

- \(I_1\) states that, if the call creates reachability from \(v_1\) to \(v_2\), then any \(w_1\) sharing with \(v_1\) before the call might reach any \(w_2\) which is reachable from \(v_1\) or aliasing with \(v_1\). Note that adding these statements is necessary only if \(v_1\) is updated in the body of some \(m\) (this information is conservatively represented in \(I''\), so that the condition \(\hat{v}_1 \in I''\) must be checked): otherwise, it is guaranteed that no path from \(w_1\) to \(w_2\) will be created during the call.
The final result of processing a method call is the union of...

Example 5.4. Consider methods $t$ and $g$ of Fig. 6, and assume that both are defined in the class Node. Let $\xi$ be a denotation for $g$ such that $\xi(\emptyset) = \{\text{this} \leftarrow y, \text{out} \leftarrow y\}$. This example shows how an abstract state $\emptyset$ is transformed by executing the code of $t$. The first two commands in $t$ transform $\emptyset$ into $I = \{a \leftarrow b, c \leftarrow \text{this}\}$. Then, the denotation of $g$ is plugged into the calling context, as follows:

1. $I_1$ is projected on $\{b, c\}$, obtaining $I_0 = \emptyset$;
2. $\xi(\emptyset)$ is renamed such that $\text{this}, y, \text{and out}$ are renamed to, respectively, $b, c, \text{and } \rho$, and $I_m = \{b \leftarrow c, \rho \leftarrow c\}$ is obtained;
3. $a \leftarrow \text{this}$ is added to $I_1$ since $\{b \leftarrow c \in I_m\} \land \{(\langle b \leftarrow a \rangle \in I'_2) \land (c \leftarrow \text{this} \in I)\}$ is true; similarly, $b \leftarrow \text{this}, a \leftarrow c$ and $a \leftarrow \rho$ are also added to $I_1$;
4. no new statements have to be added because of $I_2$ or $I_3$;
5. $I_4$ is empty since nothing becomes cyclic in $g$;
6. finally, the denotation of return renames $\rho$ to $\text{out}$ in $I \cup I_m \cup I_1 \cup I_4$, and obtains $\{a \leftarrow b, c \leftarrow \text{this}, b \leftarrow c, \text{out} \leftarrow c, a \leftarrow \text{this}, a \leftarrow c, b \leftarrow \text{this}, a \leftarrow \text{out}\}$.

Next, the inference of a denotation for a method $m$ is shown, which uses the denotation $\mathcal{E}_{\tau}^c[m^p]$ of its code. Example 5.5 introduces the problems to be faced when trying to define a method denotation, and a solution is discussed below.

Example 5.5. In Example 5.4, when analyzing $b.g(c)$, the existence of a denotation $\xi$ for $g$ such that $\xi(\emptyset) = \{\text{this} \leftarrow y, \text{out} \leftarrow y\}$ was assumed. Intuitively, this $\xi(\emptyset)$ could be computed using $\mathcal{E}_{\tau}^c[m^p]$, as follows: the first command in $g$ adds $\text{this} \leftarrow y$, and the second one adds $\text{out} \leftarrow y$, which results in the desired abstract state $\{\text{this} \leftarrow y, \text{out} \leftarrow y\}$. After this result, one might think that $\mathcal{E}_{\tau}^c[m^b](l)$ is always the good way to compute $\xi(l)$, as just done. Yet, in general, this is not correct. For example, suppose the call $b.g(c)$ is replaced by $b.h(c)$ (which is defined in Fig. 6 also). 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 $\mathcal{E}_{\tau}^c[h^b](\emptyset)$ has a different result: the first instruction adds $\text{this} \leftarrow y$, but the second one removes it since the value of $y$ is overwritten, and the third does not add anything. Therefore, $\mathcal{E}_{\tau}^c[h^b](\emptyset) = \emptyset$, which is not sound to use as the result of $\xi(\emptyset)$.

The problem in Example 5.5 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 $t$ (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 or ghost variables, i.e., new auxiliary variables which are initialized when entering the method to the same values as the parameters, but are never modified in the body.

Example 5.6. Consider methods $h$ and $k$ in Fig. 6. Method $k$ is the result of instrumenting $h$ with a shallow variable $u$, mimicking $y$. It is easy to verify that $\mathcal{E}_{\tau}^c[k^b](\emptyset)$ comes to be $\{\text{this} \leftarrow u, \text{out} \leftarrow u\}$, 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 5.7. The abstract denotational semantics of a program $P$ is the lfp of the transformer

$$T_P(\xi) = \{ m \mapsto \lambda I \in I_{rc}^m (\exists X. C^I_I \xi (l \cup I[\bar{w}/\bar{u}]))[\bar{u}/\bar{w}] \mid m \in P \}$$

where $m^I = \{ \text{this}, w_1, \ldots, w_n \}$, and $\bar{u}$ is a variable set $\{ u_1, \ldots, u_n \}$ such that $\bar{u} \cap m^I = \emptyset$; moreover, $\text{dom} (\tau) = m^I \cup \bar{u}$, and $X = \text{dom} (\tau) \setminus (\bar{u} \cup \{ \text{this}, \text{out} \})$.

The definition is explained in the following. The operator $T_P$ transforms the interpretation $\xi$ by assigning a new denotation for each method $m \in P$, using those in $\xi$. The new denotation for $m$ maps a given input abstract state $I \in I_{rc}^m$ to an output state abstract from $I_{rc}^{m \cup \{ \text{out} \}}$, 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 $I_1 = C^I_I \xi (m^I)(I_0)$;
3. all variables but $\bar{u} \cup \{ \text{this}, \text{out} \}$ are eliminated from $I_1$ (using $\exists X$); and
4. shallow variables $\bar{u}$ are finally renamed back to $\bar{w}$.

Soundness is addressed in Section 5.3, next we see some examples.

Example 5.8. Consider the following method

```java
int mirror(Tree t) {
  Tree l, r;
  if (t == null) then {
    return 0;
  } else {
    l = t.left;
    r = t.right;
    t.left = r;
    t.right = l;
    return 1 + mirror(l) + mirror(r);
  }
}
```

and suppose that class $\text{Tree}$ implements binary trees in the standard way, with fields left and right. The call $\text{mirror}(t)$ 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 $\text{mirror}$ as follows. Suppose that the current interpretation $\xi$ is such that $\xi (\text{mirror}) = \xi$, 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 $\text{null}$, line 7 adds $t \rightarrow l$; line 8 adds $t \rightarrow r$; line 9 adds again $t \rightarrow r$; and line 10 adds again $t \rightarrow l$. Recursive calls $\text{mirror}(l)$ and $\text{mirror}(r)$ do not add any statement since $\xi (\emptyset) = \emptyset$. Finally, return adds nothing. Projecting $\{ t \rightarrow l, t \rightarrow r \}$ 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 5.9. Consider the following method

```java
Node connect() {
  Node curr;
  while (curr.next != null) {
    curr = curr.next;
  }
  curr.next = this;
  return curr;
}
```

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

5.3. Soundness

This section present the soundness theorem: the abstract state obtained by applying the abstract semantics to a method in a given input abstract state is a correct representation of (i.e., its concretization contains) the concrete state obtained by executing the method in any input concrete state which is correctly represented by such input abstract state. The proof of the theorem can be found in Appendix A.4.
Theorem 5.10 (Soundness). Let \( P \) be a program, and \( \iota \) and \( \zeta \) be, respectively, its concrete and abstract semantics according to Definitions 3.1 and 5.7. Moreover, let \( m \) be a method in \( P \), and let \( \delta = \iota(m) \) and \( \xi = \zeta(m) \). It holds that, for all \( \sigma_1 \in \Sigma_m \\
\)
\[
\sigma_2 = \delta(\sigma_1) \Rightarrow \langle \hat{\sigma}_1[\text{out} \mapsto \hat{\sigma}_2(\text{out})], \hat{\sigma}_2 \rangle \in \gamma^*_{\iota}(\xi(\alpha^*_{\iota}(\sigma_1))).
\]

5.4. Completeness and optimality

Completeness \[18\] is a well-known notion in Abstract Interpretation, and corresponds to require that no loss of precision is introduced by computing a semantic function on abstract states with respect to approximating the same (concrete) computation on concrete states. An abstract domain \( A \) (with abstract function \( \alpha \) and concretization function \( \gamma \)) and an abstract function \( f^A \) over it are backward-complete for the concrete function \( f \) if and only if, for every concrete input \( \sigma \), the abstraction \( \alpha(f(\sigma)) \) of a concrete computation is equal to the abstract computation \( f^A(\alpha(\sigma)) \). This property guarantees that \( \alpha(lfp(f)) = lfp(f^A) \).

By optimality we refer to the fact that the abstract function under study is the best correct approximation of the concrete function with respect to the associated abstraction: for every \( \iota \), \( f^A(\iota) \) must be equal to \( \alpha(f(\gamma(\iota))) \).

For the sake of the following discussion, the abstract semantics \( C^\iota_{\t}_1 \) (a similar discussion holds for \( \delta^\iota_{\t}_1 \)) is supposed to use, for collecting sharing, aliasing and purity information, the best correct approximation \( \delta^\iota_{\t}_1 \) of \( C^\iota_{\t}_1 \) with respect to \( I^\iota \); for every command \( \text{com} \) and abstract value \( I \in I^\iota \), \( \delta^\iota_{\t}_1(\text{com}(I)) \) is defined as \( \alpha(I(C^\iota_{\t}_1(\text{com}(\gamma(\iota))))). \) To introduce the abstract semantics over this domain is necessary in order to be able to properly talk about completeness and optimality of the reachability and cyclicity analysis, as it will be clear in the following.

Backward completeness. The present analysis is not backward-complete. In the following, the abstract domain under study will be \( I^\alpha_{\t} \cap I^\iota_{\t} \) (i.e., sharing, aliasing and purity are included). Consider the state \( \sigma \) obtained by executing the following statements, starting from a heap where all variables are \( \text{null} \): the final result of the execution is the heap shown in the picture.

```
1. y:=new C;
2. z:=new C;
3. y.f:=new C;
4. z.f:=y.f;
5. y.g:=z;
```

After this code fragment, \( y \) and \( z \) share because they reach a common location, and \( y \) is reaching \( z \). Then, the most precise approximation of the resulting concrete state \( \sigma \) is \( I = \{ y \cdot y \}, \{ z \cdot z \}, \{ y \cdot y \}, \{ z \cdot z \} \). Suppose that the statement

```
6. x:=y.f;
```

is executed afterward, giving the concrete state \( \sigma' \) in this case, the concrete function \( f \) under study is the semantic of this statement, namely, \( C^\iota_{\t}(x=y.f) \), and the state \( \sigma' \) corresponds to \( C^\iota_{\t}(x=y.f)(\sigma) \). Now, the abstraction of \( \sigma' \) with respect to \( I^\alpha_{\t} \cap I^\iota_{\t} \) is

\[
I' = \{ y \cdot x \}, \{ y \cdot y \}, \{ z \cdot z \}, \{ x \cdot y \}, \{ x \cdot x \}, \{ y \cdot y \}, \{ z \cdot z \}, \{ 1 \cdot x \}, \{ 1 \cdot z \}, \{ y \cdot z \}, \{ y \cdot y \}, \{ z \cdot z \}, \{ 1 \cdot x \}, \{ 1 \cdot z \}, \{ y \cdot z \}, \{ y \cdot y \}, \{ z \cdot z \}.
\]

which correctly represents the sharing between the fact that the three variables, and the fact that \( x \) points exactly to the location which is reached by both \( y \) and \( z \). On the other hand, computing the result of the abstract semantics \( f^A \) (i.e., the present analysis \( C^\iota_{\t}(x=y.f) \)) on the input abstract state \( I \) gives the state

\[
I'' = \{ x \cdot x \}, \{ y \cdot y \}, \{ z \cdot z \}, \{ x \cdot x \}, \{ x \cdot z \}, \{ y \cdot z \}, \{ y \cdot y \}, \{ z \cdot z \}, \{ y \cdot x \}, \{ y \cdot y \}, \{ 1 \cdot z \}, \{ 1 \cdot z \}, \{ y \cdot z \}, \{ y \cdot y \}, \{ z \cdot z \}.
\]

The reachability statement \( x \rightarrow z \) is added because the analysis admits that, since \( y \) is said to reach \( z \), the location pointed to by \( x \) could be exactly on the path from \( y \) to \( z \). Because of the difference between \( I' \) and \( I'' \), this counterexample is enough to prove the lack of backward completeness.

Optimality. This section argues that two important abstract state transformers included in the abstract semantics are optimal. The considered transformers are \( f^A \) and \( f^A \) for field access, which is optimal with respect to \( f_1 = E^\iota_{\t}(v.f) \), and \( f^A \) and \( f^A \) for field updates, which is optimal with respect to \( f_2 = C^\iota_{\t}(v.f:=\rho) \). The use of \( \rho \) means that the state transformers account for the field update after the expression \( exp \) has been evaluated. In other words, \( f_2 \) will be applied to the concrete state resulting from evaluating \( exp \), and \( f^A \) will be applied to the abstract state \( I' \) described in Fig. 5. In order to avoid confusion with names, let \( J \) be the abstract value which is given as input to the abstract state transformer, and let \( J \) the corresponding output; therefore, \( f' \) and similar names will play the same role as \( I' \) and similar names in Fig. 5.

Again, the abstract domain includes sharing, aliasing and purity, so that the concretization and abstraction functions \( \gamma \) and \( \alpha \) are the ones which are induced by the reduced product \( I^\alpha_{\t} \cap I^\iota_{\t} \) in the standard way. This means that optimality is

\[\text{The notation } (\_\_\_) \text{ and } (\_\_\_) \text{ is used in the beginning of Section 5.}\]
proven under the assumption that the abstract operators of sharing, aliasing and purity are also optimal. It is assumed that an abstract value contains sharing, aliasing and purity information, together with reachability and cycllicity, and that it will be clear from the context how to refer to each part.

By soundness, the non-strict inequalities $E^s_\ell[v.f](f) \supseteq \alpha(E^s_\ell[v.f](y(f)))$ and $E^s_\tau[v.f:=\rho](f) \supseteq \alpha(C^s_\tau[v.f:=\rho](y(f)))$ already hold, where set inclusion is the partial order on $I^s_\ell \cap I^s_\tau$. Therefore, to prove this claim amounts to demonstrate the other direction of the inclusion, i.e., that, for every reachability or cyclicity statement $st$ contained in $f$, there is a concrete state $\sigma \in y(f)$ such that $\sigma_1 = C^s_\ell[v.f:=\rho](\sigma)$ (the case of $E^s_\ell[v.f](\sigma)$ is similar) is a concrete state whose abstraction $\alpha(\sigma_1)$ contains $st$. In other words, $\sigma_1$ is a state where the may-information represented by $st$ is actually happening (for example, if $st$ is some $v \leftarrow w$, then there must actually be a path in the heap from $v$ to $w$ in $\sigma_1$, so that the abstraction of $\sigma_1$ will forcefully contain such a statement. In the proof, this idea of “a statement $st$ actually happening in a state $\sigma$” will be phrased as “$\sigma$ justifies $st$”.

- Case $J_1$: the output abstract state $J_1$ is basically the union of four sets: (a) $J$; (b) $[v/\rho]$; (c) $[w/\rho] \setminus \{ \text{any } v \} \in J$; and (d) $[\rho \cdot \rho] \ast \in J$. For every one of them it is necessary to prove that, for every statement $st$ contained in it, there exists at least one concrete input state $\sigma$ such that the corresponding output state $\sigma_1 = f_1(\sigma)$ justifies $st$.

  (a) Clearly, every statement $w$ which was already in $f$, and is therefore maintained in $f_1$, is justified by the fact that the structure of the heap does not change when evaluating the expression: by hypothesis, there was already a state $\sigma$ justifying $st$, and the corresponding output $\sigma_1$ still justifies such statement.

  (b) In this case, relevant statements in $f$ can be of four kinds (other statements which do not involve $v$ are not relevant), and we need to prove that the corresponding statements in $[v/\rho]$ (where $v$ is replaced by $\rho$) are justified.

  $v \leftarrow w$: In this case, there certainly exists $\sigma$ in the concretization of $f$ such that $v$ actually reaches $w$ at least two steps, and the first step goes through $\rho$; then, the location pointed to by the expression actually reaches $w$ (in fact, it is on the path from $v$ to $w$), so that $\sigma_1$ justifies the statement $\rho \leftarrow w$ contained in $[v/\rho]$, corresponding to $v \leftarrow w$.

  $w \leftarrow v$: This case is easy since there exists $\sigma$ such that $w$ actually reaches $v$, and it is straightforward to see that $\rho$ will be actually reached by $w$ in $\sigma_1$ (transitivity of reachability at the concrete level), thus justifying the corresponding statement $\rho \leftarrow w$ in $[v/\rho]$.

  $v \leftarrow v$: This case is also easy because there certainly exists $\sigma$ such that $v$ is cyclic, and the first step of the cycle when starting from $v$ goes through $\rho$; this means that $v.f$ is still in the cycle, and the location pointed to by the expression reaches itself, thus justifying the corresponding statement $\rho \leftarrow \rho$ in $[v/\rho]$;

  $\ast \in J$: This case is similar to the previous one.

  (c) In this case, every $w \leftarrow \rho$ must be justified, provided there is sharing (this is a case where it becomes clear that sharing must also be considered) between $v$ and $w$ in the input state. It is enough to take the same (up to variable renaming) concrete state used in the discussion about backward completeness, where $v$ and $w$ both reach (in one step, and through $f$) the same location in the heap: the location pointed to by $\rho$ in the output state comes to be actually reached by $w$, thus justifying the statement.

  (d) The last case is easy because it is enough to find some $\sigma$ where $v$ is cyclic (but not necessarily reaching itself), and the location pointed to by $v.f$ reaches itself.

- Case $J_2$: The first issue here is to note that optimality requires the single-field optimization discussed in Section 5.2.3, where $J_0$ is strictly smaller than $J_0'$ whenever it can be guaranteed that all the relevant reachability or cyclicity paths have been broken by updating $v.f$. In fact, consider the case where this optimization is not performed (i.e., $J' = J_0$). The following piece of code

1. $x := \text{new C}();$
2. $x.f := x;$
3. $x.f := \text{null};$

shows the lack of optimality under the condition that $i$ is the only field of $C$. In fact, let the abstract value $J$ before line 3 be $\{x\rightarrow x, \ast \}^X$ as it would be obtained by the analysis, so that $\gamma(f)$ contains all the states where $x$ is cyclic and reaches itself. However, the abstract semantics without the optimization would generate the same abstract value $\{x\rightarrow x, \ast \}^X$ as the final value. This is not optimal since any concrete state after executing this code would have $x$ acyclic and not self-reaching, so that its abstraction would be $\{\}$. (in other words, none of the states would be justified). On the other hand, the aforementioned optimization removes these statements from $J_0$, so that $J'$ is empty, thus achieving, in the end, optimality.

In the definition depicted in Fig. 5, the output abstract state $J_1$ consists of two more parts: (a) the one coming from $J_1 = \{w_1 \leftarrow w_2 \mid ((w_1.v)\in f_1) \lor (w_1 \leftarrow v \in f_1) \}$; (b) the one coming from $J_1 = \{(w \leftarrow v) \mid ((w \leftarrow v) \in f_1) \lor (w \leftarrow v \in f_1)\}$. (a) In order to justify a statement $w_1 \leftarrow w_2$, it is enough to take a concrete state $\sigma \in \gamma(f)$ (which clearly exists) where $w_1$ is actually reaching $v$, and the location pointed to by the result of the expression is actually reaching $w_2$. In this case, the field update will create a path from $w_1$ to $w_2$ in $\sigma_1$, so that the statement is justified.

(b) A statement $\ast \in$ can be easily justified by taking $\sigma$ such that the result of the expression points to an actually cyclic data structure, and $w$ actually reaches $v$. Then, the newly created path will make $w$ cyclic. The final elimination of $\ast$ is not problematic.
5.5. Note on an implementation

The present analysis has been implemented in the COSTA [2] COST and Termination Analyzer. The implementation works as a component of COSTA, and handles programs written in full sequential Java bytecode, which includes control flow that originates from the handling of exceptions. Static fields are accounted for as a kind of global variables: this means that, for every class κ and static field f, a global variable v_{κ.f} is added to the analysis (note that the set of such global variables is statically decidable by simply inspecting the program code). The acyclicity information is used by COSTA to prove the termination or infer the resource usage of programs.

It is worth mentioning that the implementation is a prototype, and that it can be optimized in many ways. In fact, the present paper focuses on the theoretical definition of an existing analysis, so that the implementation is not the most important issue. As a matter of fact, such implementation deals with a different language with respect to the original implementation; this implies, for example, having to account in a specific way for advanced features of Java and Java bytecode like objects, exceptions, and static fields. The single-field optimization discussed in Section 5.2.3 is not implemented.

6. Conclusions

This paper discusses an acyclicity analysis of a Java-like language with mutable data structures, based on reachability between variables. In particular, the main focus of the paper is on the formalization of an existing analysis within the framework of Abstract Interpretation. 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. It is also proven to be the best correct approximation of the concrete semantics with respect to the chosen abstraction. Moreover, it can be shown to obtain precise results in a number of non-trivial scenarios, where the sharing-based approach is less precise [26]. 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.

Appendix. Proofs

This appendix includes proofs for: Lemma 4.3 in Appendix A.1; Lemma 4.6 in Appendix A.2; Lemma 4.7 in Appendix A.3; and Theorem 5.10 in Appendix A.4.

A.1. Proof of Lemma 4.3

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

\[(a) \forall l_r \in I^r_f. \alpha^r_f(\gamma^r_f(l_r)) = l_r\]
and \[(b) \forall l_h \in I^h_f. \gamma^h_f(\alpha^h_f(l_h)) \supseteq l_h\]
hold, where \(\subseteq\) is the ordering on \(I^r_f\).

**Part (a).** We show that \(v \thicksim w \in l_r \iff v \thicksim w \in \gamma^r_f(\alpha^r_f(l_h))\). (\(\Rightarrow\)) Assume \(v \thicksim w \in l_r\); then, according to the definition of \(I^r_f\) and class reachability, there must be a concrete state \(\sigma \in \Sigma\), in which \(v\) reaches \(w\), since, otherwise, the statement \(v \thicksim w\) cannot be part of the domain \(I^r_f\). We construct a state \(\sigma'\) from \(\sigma\) by setting all reference variables but \(v\) and \(w\) to \texttt{null}. By the definition of \(\gamma^r_f\), this specific \(\sigma'\) must be in \(\gamma^r_f(l_r)\). This, according to the definition of \(\alpha^r_f\), implies that \(v \thicksim w \in \alpha^r_f(\gamma^r_f(l_h))\). (\(\Leftarrow\)) Assume \(v \thicksim w \in \alpha^r_f(\gamma^r_f(l_h))\). According to the definition of \(\alpha^r_f\), this means that there exists at least one \(\sigma \in \gamma^r_f(l_h)\) in which \(v\) reaches \(w\), and, according to the definition of \(\gamma^r_f\), this can only happen if \(v \thicksim w \in l_r\).

**Part (b).** We show that \(\sigma \in l_h \Rightarrow \sigma \in \gamma^h_f(\alpha^h_f(l_h))\). Let \(\sigma \in l_h\), and let \(l_r\) be the set of all reachability relations in \(\sigma\), i.e., \(v\) reaches \(w\) in \(\sigma\) iff \(v \thicksim w \in l_r\). Clearly, \(l_r \subseteq \alpha^r_f(l_h)\). Then, according to the definition of \(\gamma^h_f\), \(\sigma\) must be in \(\gamma^h_f(\alpha^h_f(l_h))\) since it satisfies \(\forall v, w \in \tau. v \thicksim w \in \sigma \Rightarrow v \thicksim w \in \alpha^h_f(l_h)\). □

A.2. Proof of Lemma 4.6

Very similar to the proof of Lemma 4.3. □
A.3. Proof of Lemma 4.7

(\Rightarrow). We show that:
\[
\gamma_r^c(I_1^1, I_1^2)(v) = \gamma_r^c(I_1^1, I_1^2) \Rightarrow I_1^1 = I_1^1 \land (I_1^1 \setminus \{v \rightarrow v \mid c \neq \emptyset \}) = (I_1^1 \setminus \{v \rightarrow v \mid c \neq \emptyset \}).
\]

First, note that the logical formula \(F \Rightarrow (G \land H)\) is equivalent to \((\neg G \Rightarrow \neg F) \land (\neg H \Rightarrow \neg F)\). The proof is by contradiction, and consists of two parts:

1. proving that \(I_1^1 \neq I_1^2\) implies \(\gamma((I_1^1, I_1^2)) \neq \gamma((I_1^2, I_1^2))\); and
2. proving that \((I_1^1 \setminus \{v \rightarrow v \mid c \neq \emptyset \}) \neq (I_1^2 \setminus \{v \rightarrow v \mid c \neq \emptyset \})\) implies \(\gamma((I_1^1, I_1^2)) \neq \gamma((I_1^2, I_1^2))\).

The proof goes as follows.

1. Suppose \(I_1^1 \neq I_1^2\), and let \(X_1 = \{v \mid c \neq \emptyset \in I_1^1 \}\), and \(X_2 = \{v \mid c \neq \emptyset \in I_1^2 \}\). Note that, by hypothesis, at least one of \(X_1\) and \(X_2\) must be non-empty. For \(i \in \{1, 2\}\), let \(\sigma_i\) 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., \(\sigma_i^\ast((\sigma_i)) = \emptyset\). Note that this requirement is consistent, since the cyclicity of some variables (in this case, those in \(X_i\)) does not necessarily imply the existence of a reachability path between variables.

   It is easy to see that \(\sigma_1\) and \(\sigma_2\) both belong to \(\gamma((I_1^1, I_1^1))\), but do not belong to \(\gamma((I_1^2, I_1^2))\). Clearly, in both cases above such state belongs to \(\gamma((I_1^1, I_1^2))\), but cannot be in \(\gamma((I_1^2, I_1^2))\) because: in (a), either \(v \rightarrow v \notin I_1^2\) (so that \(\sigma_1 \neq \gamma_1^c((I_1^2))\)), or \(I_1^2 \neq I_1^1\) (so that \(\sigma_1 \neq \gamma_1^c((I_1^2))\)), and; in (b), \(v \rightarrow v \notin I_1^2\), so that \(\sigma_1 \neq \gamma_1^c((I_1^2))\). Dually, if \(S_2\) is empty, then \(S_1\) cannot be empty, and, with a similar reasoning, a state \(\sigma_2\) can be found which belongs to \(\gamma((I_1^2, I_1^2))\), but not \(\gamma((I_1^1, I_1^2))\).

(\Leftarrow). We prove that:
\[
I_1^1 = I_1^1 \land (I_1^1 \setminus \{v \rightarrow v \mid c \neq \emptyset \}) = (I_1^1 \setminus \{v \rightarrow v \mid c \neq \emptyset \}) \Rightarrow \gamma_r^c((I_1^1, I_1^2)) = \gamma_r^c((I_1^2, I_1^2)).
\]

It follows easily from observing that, under the hypothesis of the above implication, the only difference between \((I_1^1, I_1^1)\) and \((I_1^2, I_1^2)\) is that \((I_1^1, I_1^2)\) may contain some statements \(v \rightarrow v\) for variables \(v\) such that \(c \neq \emptyset \in I_1^2\), and \((I_1^1, I_1^2)\) may contain some (different) statements \(v \rightarrow v\) for variables \(v\) such that \(c \neq \emptyset \notin I_1^2\). However, adding such statements to both abstract values does not change the set of abstract states they represent, since the possibility that \(v\) reaches itself in any abstract state is contradicted by the lack of the \(c \neq \emptyset\) statement. In other words, there is no concrete state which belongs either to \(\gamma_r^c((I_1^1, I_1^1))\) or \(\gamma_r^c((I_1^2, I_1^2))\), but not to \(\gamma_r^c((I_1^1 \setminus \{v \rightarrow v \mid c \neq \emptyset \}, I_1^1))\) and \(\gamma_r^c((I_1^2 \setminus \{v \rightarrow v \mid c \neq \emptyset \}, I_1^2))\) (which are equal by the hypothesis.) \(\square\)

A.4. Proof of Theorem 5.10

This proof of soundness amounts to proving the soundness of all abstract denotations for expressions and commands, assuming that a current interpretation \(x\) and a corresponding abstract one \(\zeta\) which correctly approximates \(x\) are available. Then, a simple induction can be applied to show that the abstract semantics of Definition 3.1 correctly approximates the concrete semantics of Definition 5.7 (the induction step basically applies the denotations on the elements of \(x\) and \(\zeta\)).

In the following, let \(\sigma\) be a concrete state, \(com\) be a command, \(exp\) be an expression, and \(\sigma^+\) be the state obtained by executing \(com\) or evaluating \(exp\) in \(\sigma\). The soundness of the abstract denotations for expressions and commands amounts to say that, if \(l \in \mathcal{E}_{\rho}^c\) approximately approximates \(\sigma\), i.e., \(\sigma \in \gamma_{\rho}^c(l)\), then the abstract state \(l^+ = \mathcal{E}_{\rho}^c[com](l)\) (or \(l^+ = \mathcal{E}_{\rho}^c[exp](l)\), in the case of expressions) correctly approximates \(\sigma^+\). Formally, we show that

1. \(\forall \sigma \in \Sigma_\tau, l \in \mathcal{E}_{\rho}^c, \sigma \in \gamma_{\rho}^{r \cup \rho} \Rightarrow E_r^{\rho}[\sigma] \in \gamma_{\rho}^{r \cup \rho}(\mathcal{E}_{\rho}^c[com](l))\)
2. \(\forall \sigma \in \Sigma_\tau, l \in \mathcal{E}_{\rho}^c, \sigma \in \gamma_{\rho}^{r \cup \rho} \Rightarrow C_r^{\rho}[\sigma] \in \gamma_{\rho}^{r \cup \rho}(\mathcal{E}_{\rho}^c[com](l))\).

Note that, if \(\sigma^+\) is obtained after evaluating an expression, then \(\rho \in dom(\sigma^+)\), while, if it is obtained after executing a command, then \(dom(\sigma^+) = dom(\sigma)\).

The soundness proof considers separately the rules of the abstract semantics \(\mathcal{E}_{\rho}^c[\_](\_\_\_\_)\) and \(\mathcal{E}_{\rho}^c[\_](\_\_\_\_)\). When some logical fact is said to hold by soundness, it means that it holds by the hypothesis on the input (i.e., that \(\sigma \in \gamma_{\rho}^c(l)\) holds), or by induction on sub-expressions or sub-commands. For example, the fact that \(v\) reaches \(w\) in \(\sigma\) implies \(v \rightarrow w \in l\) by soundness, since \(l\) is supposed to be a sound description of \(\sigma\).
Denotations (1a), (2a), and (3a). Suppose $\sigma^* \not\in \gamma^{tuc}_I(\rho)^*(I^*)$. Then, according to the definition of $\gamma^{tuc}_I(\rho)^*(I^*)$, it must be the case that (i) $w_1$ reaches $w_2$ in $\sigma^*$ but $w_1 \rightarrow w_2 \not\in I^*$; or (ii) $w$ is cyclic in $\sigma^*$ but $\sigma^w \not\in I^*$. This contradicts the soundness hypothesis $\sigma \in \gamma^{tuc}_I(I)$, since $I^* = I$ and $\sigma$ and $\sigma^*$ have the same reachability and cyclicity information.\(^5\)

Denotation (4a). Assume $\tau(v) \not\in \text{int}$, otherwise the reasoning we developed for case (1a) applies. Note that this case does not have any side effects, except defining the new variable $\rho$. If $\sigma^* \not\in \gamma^{tuc}_I(\rho)^*(I^*)$, then, according to the definition of $\gamma^{tuc}_I(\rho)^*$, it must be the case that (i) $w_1$ reaches $w_2$ in $\sigma^*$ but $w_1 \rightarrow w_2 \not\in I^*$; or (ii) $w$ is cyclic in $\sigma^*$ but $\sigma^w \not\in I^*$. Suppose we are in case (i):

- If $w_1 \neq \rho \wedge w_2 \neq \rho$, then $\hat{\sigma}(w_2) = \hat{\sigma}^*(w_2) \in R(\hat{\sigma}^*(w_1), \hat{\sigma}^*) = R(\hat{\sigma}(w_1), \hat{\sigma})$, i.e., $w_1$ reaches $w_2$ in $\sigma$. By the soundness hypothesis $\sigma \in \gamma^{tuc}_I(I)$ we have $w_1 \rightarrow w_2 \not\in I^*$, which contradicts $w_1 \rightarrow w_2 \not\in I^*$.
- If $w_1 = \rho \wedge w_2 \neq \rho$, then $\hat{\sigma}(w_2) = \hat{\sigma}^*(w) \in R(\hat{\sigma}^*(w_1), \hat{\sigma}^*) = R(\hat{\sigma}(w_1), \hat{\sigma})$, i.e., $v$ reaches $w_2$ in $\sigma$. By the soundness hypothesis $\sigma \in \gamma^{tuc}_I(I)$, we have $v \rightarrow w_2 \not\in I$ and thus $\rho \rightarrow w_2 \not\in I^*$, which contradicts $\rho \rightarrow w_2 \not\in I^*$.
- If $w_1 \neq \rho \wedge w_2 = \rho$, then $\hat{\sigma}(v) = \hat{\sigma}^*(\rho) \in R(\hat{\sigma}^*(w_1), \hat{\sigma}^*) = R(\hat{\sigma}(w_1), \hat{\sigma})$, i.e., $v$ reaches $w_2$ in $\sigma$. By the soundness hypothesis $\sigma \in \gamma^{tuc}_I(I)$, we have $v \rightarrow w_2 \not\in I$ and thus $\rho \rightarrow w_2 \not\in I^*$, which contradicts $\rho \rightarrow w_2 \not\in I^*$.

For case (ii), the reasoning is basically as (i), by considering cyclicity instead of reachability.

Denotation (5a). Assume $f$ is of reference type, otherwise the reasoning we have done for case (1a) applies. Note that this case does not have any side effects, except defining the new variable $\rho$. If $\sigma^* \not\in \gamma^{tuc}_I(\rho)^*(I^*)$, then, according to the definition of $\gamma^{tuc}_I(\rho)^*$, it must be the case that (i) $w_1$ reaches $w_2$ in $\sigma^*$ but $w_1 \rightarrow w_2 \not\in I^*$; or (ii) $w$ is cyclic in $\sigma^*$ but $\sigma^w \not\in I^*$. Suppose we are in case (i):

- If $w_1 \neq \rho \wedge w_2 \neq \rho$, then $\hat{\sigma}(w_2) = \hat{\sigma}^*(w_2) \in R(\hat{\sigma}^*(w_1), \hat{\sigma}^*) = R(\hat{\sigma}(w_1), \hat{\sigma})$, i.e., $w_1$ reaches $w_2$ in $\sigma$. By the soundness hypothesis $\sigma \in \gamma^{tuc}_I(I)$ we have $w_1 \rightarrow w_2 \not\in I^*$, which contradicts $w_1 \rightarrow w_2 \not\in I^*$.
- If $w_1 = \rho \wedge w_2 \neq \rho$, then $\hat{\sigma}(w_2) = \hat{\sigma}^*(w_2) \in R(\hat{\sigma}^*(\rho), \hat{\sigma}^*) \subseteq R(\hat{\sigma}(v), \hat{\sigma})$, i.e., $v$ reaches $w_2$ in $\sigma$. By the soundness hypothesis $\sigma \in \gamma^{tuc}_I(I)$, we have $v \rightarrow w_2 \not\in I$ and thus $\rho \rightarrow w_2 \not\in I^*$, which contradicts $\rho \rightarrow w_2 \not\in I^*$.
- If $w_1 \neq \rho \wedge w_2 = \rho$, then $\hat{\sigma}(v) = \hat{\sigma}^*(\rho) \in R(\hat{\sigma}^*(w_1), \hat{\sigma}^*) = R(\hat{\sigma}(w_1), \hat{\sigma})$, i.e., $v$ reaches $w_2$ in $\sigma$. By the soundness hypothesis $\sigma \in \gamma^{tuc}_I(I)$, we have $v \rightarrow w_2 \not\in I$ and thus $\rho \rightarrow w_2 \not\in I^*$, which contradicts $\rho \rightarrow w_2 \not\in I^*$.

For case (ii), the reasoning is basically as (i), by considering cyclicity instead of reachability.

Denotation (6a). The proof for this case is by structural induction on expressions, where the base-case include the non-compound expressions of cases (1a)–(5a) and (7a), for which we have seen already (case (7a) is done below) that the abstract denotations correctly approximate the concrete ones. Let $I_1 = E^\text{exp}_{\text{exp}}(l_1)$ and $I_1 = E^\text{exp}_{\text{exp}}(l_1)(\sigma)$. By the (structural) induction hypothesis, we have $\sigma_1 \in \gamma^{tuc}_I(l_1)$. Moreover, since the state $\langle \hat{\sigma}, \hat{\sigma}_1 \rangle$ is basically obtained by removing $\rho$ from $\sigma_1$, we also have $\langle \hat{\sigma}, \hat{\sigma}_1 \rangle \in \gamma^{tuc}_I(\exp(l_1))$. Now, let $I_2 = E^\text{exp}_{\text{exp}}(l_2)$ and $I_2 = E^\text{exp}_{\text{exp}}(l_2)(\hat{\sigma}, \hat{\sigma}_1)$; then, by the (structural) induction hypothesis, we have $\sigma_2 \in \gamma^{tuc}_I(l_2)$. Since $\sigma^*$ is obtained from $\sigma_2$ by setting $\rho$ to a number (i.e., there is no reachability or cyclicality relations in $\sigma^*$ that involve $\rho$), and since $I^* = \exp(l_2)$, we can conclude that $\sigma^* \in \gamma^{tuc}_I(l_1)$.

Denotation (7a). 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 some subsequent program point. This is similar to the way sharing information is dealt with in the present approach (following [28,14]).

Keeping track of cyclicality 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 $I$, 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 $I_n$ (which is introduced by the method for $v$), 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

---

\(^5\) 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.
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. A.7. In both cases, the creation of the path requires that an argument is modified in $m$ (condition $v_i \in sh'$), 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 $w_1 \rightarrow w_2$ 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.

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 which can be created in the body of $m$): the information about them is cloned for $\rho$.

**Denotation (1c).** Suppose $\sigma^* \notin \gamma^*_{rc}(\rho)$ (i.e.). Then, according to the definition of $\gamma^*_{rc}(\rho)$, it must be the case that (i) $w_1$ reaches $w_2$ in $\sigma^*$ but $w_1 \rightarrow w_2 \notin I^*$; or (ii) $w$ is cyclic in $\sigma^*$ but $\circ^w \notin I^*$. Suppose we are in case (i), and let $\sigma_e = E^*_e[[\exp](\sigma)]$ and $I_1 = E^*_e[[\exp]](I)$.

- If $w_1 \neq v \land w_2 \neq v$, then it must be the case that $w_1$ reaches $w_2$ in $\sigma_e$. By the soundness of the expressions denotations we must have $\sigma_e \in \gamma^*_{rc}(\rho)(I_1)$, which means that $w_1 \rightarrow w_2 \in I_1$; thus, $w_1 \rightarrow w_2 \in (\exists v.I_1)[\rho/v] = I^*$, which contradicts $w_1 \rightarrow w_2 \notin I^*$.
- If $w_1 = v \land w_2 \neq v$, then it must be the case that $\rho$ reaches $w_2$ in $\sigma_e$. By the soundness of the denotations for expressions, we must have $\sigma_e \in \gamma^*_{rc}(\rho)(I_1)$, which means that $\rho \rightarrow w_2 \in I_1$, and thus $v \rightarrow w_2 \in (\exists v.I_1)[\rho/v] = I^*$, which contradicts $v \rightarrow w_2 \notin I^*$.
- If $w_1 \neq v \land w_2 = v$, then it must be the case that $w_1$ reaches $\rho$ in $\sigma_e$. By the soundness of the denotations for expressions, we must have $\sigma_e \in \gamma^*_{rc}(\rho)(I_1)$, which means that $w_1 \rightarrow \rho \in I_1$, and thus $w_1 \rightarrow v \in (\exists v.I_1)[\rho/v] = I^*$, which contradicts $w_1 \rightarrow v \notin I^*$.
- If $w_1 = v \land w_2 = v$, then it must be the case that $\rho$ reaches $\rho$ in $\sigma_e$. By the soundness of the denotations for expressions, we must have $\sigma_e \in \gamma^*_{rc}(\rho)(I_1)$, which means that $\rho \rightarrow \rho \in I_1$, and thus $v \rightarrow v \in (\exists v.I_1)[\rho/v] = I^*$, which contradicts $v \rightarrow v \notin I^*$.

Case (ii) can be done with similar reasoning.

**Denotation (2c).** 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 $\sigma' = E^*_e[[\exp]](\sigma)$, and $\ell_e = \sigma'(\rho)$. If $v$ and $\ell_e$ are considered, then there are two main cases (Fig. A.8): (a) $\sigma(v) = \ell_e$; or (b) $\sigma'(v) \neq \ell_e$.

(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 $\sigma^*$ 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. A.8). In this case, $u$ reaches $v$ via $f$, and this is taken into account in the definition of $I_1$, where $u$ plays the role of $w_1$, and $v$ also plays the role of $w_2$. The result is that $I_1$ includes $u \rightarrow v$, as expected. The semantics correctly adds $v \rightarrow v$ as well (in fact, $v$ can play the role of both $w_1$ and $w_2$). As for cyclicity, the definition of $I_1$ guarantees that $\circ^v$ and $\circ^u$ will belong to $I^*$.

- $v$ reaches $u$ (case $u_3$ in the same figure). In this case, $u \rightarrow u \in I^*$ since, in the definition of $I_1$, $u$ plays the role of $w_2$ (note that $v$ and $\rho$ alias). $u$ will also be considered as cyclic by the definition of $I_1$;

- $v$ and $u$ both reach a common location $\ell$ (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).
\(f\) in this case, when considering \(u\), the number of possible scenarios for reachability is larger. Moreover, there are two scenarios where \(u\) would be cyclic after the update (1) \(\ell_e\) reaches \(u\), so that a cycle is created by the field update, and \(u\) becomes cyclic (if it was not already); or (ii) \(\ell_e\) does not reach \(u\), so that \(u\) is cyclic only if it was already cyclic in \(\sigma\), and the same applies to \(\ell_e\). In case (ii), it can be easily seen that the definition of \(\ell_e\) accounts for the cyclicity of \(u\) since \(\hat{\ell}^\sigma\) 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. A.8). In this case, it was also reaching \(v\) (or aliasing with it) in \(\sigma\), so that (in the case of reachability) \(u\sim v\in I\), which implies \(u\sim v\in I''\), 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 \(\ell_e\) or is reached by it (cases \(u_6\) and \(u_7\)). In this case, \(u\) plays the role of \(u_2\) in the definition of \(I_e\), and is correctly considered to be reached by \(v\). As for cyclicity, \(u\) will only become cyclic in case (i) if it points to \(\ell_e\), 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 \(I_e\)).
- \(w\) and \(\ell_e\) reach some common location \(\ell\) (case \(u_8\)). Also easy since nothing changes with respect to the reachability between \(u\) and \(v\).

Note that, due to the discussion in Section 5.2.3, the single-field optimization introduced by \texttt{condRemove} is not problematic for soundness, since the removal of statements is only applied if the required conditions about \(u\) and \(f\) are guaranteed to hold. In any case, the conservative choice of taking \texttt{condRemove}(\(I''_e\), \(v\), \(f\)) to be \(I''_e\) itself is also sound.

\textbf{Denotation (3.\textsuperscript{v}).} This case is quite straightforward, given the inductive hypothesis on \texttt{com} and \texttt{com2}, and the assumption that \texttt{exp} has no side effects and returns an \texttt{int}. Suppose \(\sigma^* = C_i[\texttt{com}](\sigma)\) for \(i \in \{1, 2\}\), then, by the induction hypothesis, 
\[
\sigma^* \in \gamma^*_\ell_i(C_i[\texttt{com}](\ell(I))) \subseteq \gamma^*_\ell_i(C_i[\texttt{com}](\ell(I))) \cup \gamma^*_\ell_i(C_i[\texttt{com}](\ell(I))) = I''.
\]

\textbf{Denotations (4.\textsuperscript{v}), (5.\textsuperscript{v}), and (6.\textsuperscript{v}).} Rules for loops and concatenation are easy, given the inductive hypothesis on the subcommands, and the definition of the fixpoint. The rule for the \texttt{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 5.7 and the definition of a denotational semantics, proves the theorem. \(\square\)

\textbf{References}

\begin{enumerate}
\end{enumerate}