Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

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

ARTICLE INFO

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

ABSTRACT

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.

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

[*] Corresponding author. Tel.: +34 600317288.
E-mail address: damiano.zanardini@gmail.com (D. Zanardini).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.12.018

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 acyclicity 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* $\mathcal{I}^{\tau}_{rc}$, which captures the reachability information about program variables (i.e., whether there can be a path in the heap from the location ℓ_v bound to some variable v to the location ℓ_w bound to some w), and the acyclicity of data structures (i.e., whether there can be a cyclic path starting from the location bound to some variable).
2. A provably sound and optimal *abstract semantics* $\mathcal{C}^{\tau}_{\zeta}[\![\_]\!](\_)$ of a simple object-oriented language is developed, which works on $\mathcal{I}^{\tau}_{rc}$ and can often guarantee the acyclicity of *Directed Acyclic Graphs* (DAGs), which most likely will be considered as cyclic if only sharing, not reachability, is taken into account. 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

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

```
 1 class OrderedList {
 2   Node head, lastInserted;
 3
 4   int insert(int  i)  {
 5     Node c,p,n;
 6     int pos;
 7                                  // $I_7 = \emptyset$
 8     pos:=0;                      // $I_8 = \emptyset$
 9     n:=new Node;                 // $I_9 = \emptyset$
10     n.value:=i;                  // $I_{10} = \emptyset$
11     c:=this.head;                // $I_{11} = \{this \rightsquigarrow c\}$
12     while (c!=null && c.value<i) do {
13         pos:=pos+1;              // $I_{13} = \{this \rightsquigarrow c, this \rightsquigarrow p, p \rightsquigarrow c\}$
14         p:=c;                    // $I_{14} = \{this \rightsquigarrow c, this \rightsquigarrow p\}$
15         c:=c.next;               // $I_{15} = \{this \rightsquigarrow c, this \rightsquigarrow p, p \rightsquigarrow c\}$
16     }                            // $I_{16} = \{this \rightsquigarrow c, this \rightsquigarrow p, p \rightsquigarrow c\}$
17     n.next:=c;                   // $I_{17} = \{this \rightsquigarrow c, this \rightsquigarrow p, p \rightsquigarrow c, n \rightsquigarrow c\}$
18     if  (p=null)
19       then this.head:=n;         // $I_{19} = \{this \rightsquigarrow c, this \rightsquigarrow p, this \rightsquigarrow n, p \rightsquigarrow c, n \rightsquigarrow c\}$
20       else p.next:=n;            // $I_{20} = I_{17} \cup \{this \rightsquigarrow n, p \rightsquigarrow n\}$
21                                  // $I_{21} = I_{19} \cup I_{20} = I_{20}$
22     this.lastInserted:=n;        // $I_{22} = I_{21}$
23     return pos+1;                // $I_{23} = I_{22}$
24   }
25 }
26
27 class Node {
28   Node next;
29   int value;
30 }
```

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

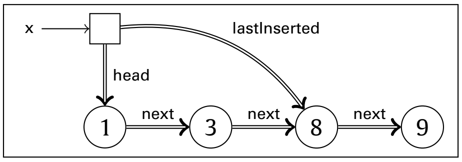


Fig. 2. A graphical representation of the data structure on which the example works.

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.

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

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

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

if x cannot be proven to be acyclic after insert, then it must be assumed to be cyclic from the second iteration on. This, in turn, prevents from proving termination of the loop at lines 12–16, since it might 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 $\mathcal{X}$ of valid *identifiers*. A *program* consists of a set of classes $\mathcal{K} \subseteq \mathcal{X}$ ordered by the *subclass* relation $\prec$. Following Java, a *class declaration* takes the form “**class** κ_1 [**extends** κ_2] $\{\, t_1\, f_1; \ldots\ t_n\, f_n; M_1 \ldots M_k\}$” where each “$t_i\, f_i$” declares the field f_i to have type $t_i \in \mathcal{K} \cup \{\textbf{int}\}$, and each M_i is a method definition. Similarly to Java, the optional statement “**extends** κ_2” declares κ_1 to be a subclass of κ_2. A *method definition* takes the form “$t\ m\ (t_1\ w_1, \ldots, t_n\ w_n)\ \{t_{n+1}\ w_{n+1}; \ldots\ t_{n+p}\ w_{n+p};\ com\}$” where: $t \in \mathcal{K} \cup \{\textbf{int}\}$ is the type of the return value; $w_1, \ldots, w_n \in \mathcal{X}$ are the formal parameters; $w_{n+1}, \ldots, w_{n+p} \in \mathcal{X}$ are local variables; and *com* is a sequence of instructions according to the following grammar:

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

where $v, \bar{v}, m, f \in \mathcal{X}$; $n \in \mathbb{Z}$; $\kappa \in \mathcal{K}$; and $\oplus$ is a binary operator (Boolean operators return 1 for **true** and 0 for **false**). For simplicity, and without loss of generality, *conditions* in **if** and **while** statements are assumed not to create objects or call methods. A *method signature* $\kappa.m(t_1, \ldots, t_n){:}t$ refers to a method m defined in class κ, taking n parameters of type $t_1, \ldots, t_n \in \mathcal{K} \cup \{\textbf{int}\}$, and returning a value of type t. Given a method signature m, let m^b be its code *com*; m^i its set of input variables $\{this, w_1, \ldots, w_n\}$; m^l its set of local variables $\{w_{n+1}, \ldots, w_{n+m}\}$; and $\mathsf{m}^s = \mathsf{m}^i \cup \mathsf{m}^l$.

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

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

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

A *denotation* δ over two type environments τ_1 and τ_2 is a partial map from Σ_{τ_1} to Σ_{τ_2}: it basically describes how the state changes when a piece of code is executed. The set of denotations from τ_1 to τ_2 is $\Delta(\tau_1,\tau_2)$. *Interpretations* are special denotations which give a meaning to methods in terms of their input and output variables. An interpretation $\iota \in \Gamma$ maps methods to denotations, and is such that $\iota(\mathsf{m}) \in \Delta(\mathsf{m}^i,\{out\})$ for each signature m in the program. Note that the variable *out* 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 $E^\iota_\tau[\![exp]\!]$ maps states from Σ_τ to states from $\Sigma_{\tau\cup\{\rho\}}$, where ρ is a special variable for storing the expression value. A command denotation $C^\iota_\tau[\![com]\!]$ maps states to states, in presence of $\iota \in \Gamma$. The function $newobj(\kappa)$ creates a new instance of the class κ with integer fields initialized to 0 and reference fields initialized to **null**, while $newloc(\check{\sigma})$ returns the first free location, i.e., the first $\ell \notin \mathrm{dom}\,(\check{\sigma})$ according to the total ordering on locations. The function *lkp* resolves the method call and returns the signature of the method

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

$$
\begin{aligned}
E^{\iota}_{\tau}\llbracket n\rrbracket(\sigma) =& \langle\hat{\sigma}[\rho\mapsto n],\check{\sigma}\rangle\\
E^{\iota}_{\tau}\llbracket \textbf{null}\rrbracket(\sigma) =& \langle\hat{\sigma}[\rho\mapsto \textbf{null}],\check{\sigma}\rangle\\
E^{\iota}_{\tau}\llbracket \textbf{new}\ \kappa\rrbracket(\sigma) =& \langle\hat{\sigma}[\rho\mapsto \ell],\check{\sigma}[\ell\mapsto newobj(\kappa)]\rangle \text{ where } \ell = newloc(\check{\sigma})\\
E^{\iota}_{\tau}\llbracket v\rrbracket(\sigma) =& \langle\hat{\sigma}[\rho\mapsto \hat{\sigma}(v)],\check{\sigma}\rangle\\
E^{\iota}_{\tau}\llbracket v.f\rrbracket(\sigma) =& \langle\hat{\sigma}[\rho\mapsto \check{\sigma}(\hat{\sigma}(v)).f],\check{\sigma}\rangle\\
E^{\iota}_{\tau}\llbracket exp_1\oplus exp_2\rrbracket(\sigma) =& \langle\hat{\sigma}[\rho\mapsto \hat{\sigma}_1(\rho)\oplus\hat{\sigma}_2(\rho)],\check{\sigma}_2\rangle \text{ where}\\
& \sigma_1 = E^{\iota}_{\tau}\llbracket exp_1\rrbracket(\sigma) \text{ and } \sigma_2 = E^{\iota}_{\tau}\llbracket exp_1\rrbracket(\langle\hat{\sigma},\check{\sigma}_1\rangle)\\
E^{\iota}_{\tau}\llbracket v_0.m(v_1,\ldots,v_n)\rrbracket(\sigma) =& \langle\hat{\sigma}[\rho\mapsto \hat{\sigma}_2(out)],\check{\sigma}_2\rangle \text{ where}\\
& \sigma_2 = \iota(\mathsf{m})(\sigma_1) \text{ and } \sigma_1 \text{ is such that}\\
& \ \text{1. } \check{\sigma}_1 = \check{\sigma};\\
& \ \text{2. } \hat{\sigma}_1(this) = \hat{\sigma}(v_0);\\
& \ \text{3. } \forall 1{\le}i{\le}n.\ \hat{\sigma}_1(w_i) = \hat{\sigma}(v_i);\ \text{ and}\\
& \ \text{4. } \mathsf{m} = lkp(\sigma, v_0.m(v_1,\ldots,v_n));
\end{aligned}
$$

$$
\begin{aligned}
C^{\iota}_{\tau}\llbracket v{:=}exp\rrbracket(\sigma) =& \langle\hat{\sigma}[v\mapsto \hat{\sigma}_e(\rho)],\check{\sigma}_e\rangle\\
C^{\iota}_{\tau}\llbracket v.f{:=}exp\rrbracket(\sigma) =& \langle\hat{\sigma},\check{\sigma}[\ell.f\mapsto \hat{\sigma}_e(\rho)]\rangle \text{ where } \ell = \hat{\sigma}(v)\\
C^{\iota}_{\tau}\left\llbracket\begin{array}{r}\textbf{if}\ exp\ \textbf{then}\ com_1\\ \textbf{else}\ com_2\end{array}\right\rrbracket(\sigma) =& \text{if } \hat{\sigma}_e(\rho)\neq 0 \text{ then } C^{\iota}_{\tau}\llbracket com_1\rrbracket(\sigma) \text{ else } C^{\iota}_{\tau}\llbracket com_2\rrbracket(\sigma)\\
C^{\iota}_{\tau}\llbracket \textbf{while}\ exp\ \textbf{do}\ com\rrbracket(\sigma) =& \delta(\sigma) \text{ where } \delta \text{ is the least fixpoint of}\\
& \lambda w.\lambda\sigma.\ \text{if } \hat{\sigma}_e(\rho)\neq 0 \text{ then } w(C^{\iota}_{\tau}\llbracket com\rrbracket(\sigma)) \text{ else } \sigma\\
C^{\iota}_{\tau}\llbracket \textbf{return}\ exp\rrbracket(\sigma) =& \langle\hat{\sigma}[out\mapsto \hat{\sigma}_e(\rho)],\check{\sigma}_e\rangle\\
C^{\iota}_{\tau}\llbracket com_1;\ com_2\rrbracket(\sigma) =& C^{\iota}_{\tau}\llbracket com_2\rrbracket(C^{\iota}_{\tau}\llbracket com_1\rrbracket(\sigma))
\end{aligned}
$$

Fig. 3. Denotations for expressions and commands. The state σ_e is $E^{\iota}_{\tau}\llbracket exp\rrbracket(\sigma)$.

to be called. The *concrete denotational semantics* of a program is defined as the *least fixpoint* of the following transformer of interpretations [8].

Definition 3.1. The denotational semantics of a program P is the least fixpoint (*lfp*) of the following operator:

$$T_P(\iota) = \{\mathsf{m}\mapsto \lambda\sigma\in\Sigma_{\mathsf{m}^i}.\exists\tau\backslash out.C^{\iota}_{\mathsf{m}^s\cup\{out\}}\left\llbracket \mathsf{m}^b\right\rrbracket(extend(\sigma,\mathsf{m}))\mid \mathsf{m}\in P\}$$

where $extend(\sigma,\mathsf{m}) = \langle\hat{\sigma}[\forall v\in\mathsf{m}^l\cup\{out\}.v\mapsto 0/\textbf{null}],\check{\sigma}\rangle$.

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

4. The abstract domain

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 μ, the set of reachable locations from $\ell\in\text{dom}(\mu)$ is $R(\mu,\ell) = \cup\{R^i(\mu,\ell)\mid i\geq 0\}$, where $R^0(\mu,\ell) = \text{rng}(\mu(\ell).\mathsf{frm})\cap\mathcal{L}$, and $R^{i+1}(\mu,\ell) = \cup\{\text{rng}(\mu(\ell').\mathsf{frm})\cap\mathcal{L}\mid \ell'\in R^i(\mu,\ell)\}$. The set of ε-reachable locations from $\ell\in\text{dom}(\mu)$ is $R^{\varepsilon}(\mu,\ell) = R(\mu,\ell)\cup\{\ell\}$.

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

4.1. Reachability

Given a state $\sigma\in\Sigma_\tau$, a reference variable $v\in\tau$ is said to *reach* a reference variable $w\in\tau$ in σ if $\hat{\sigma}(w)\in R(\check{\sigma},\hat{\sigma}(v))$. This means that, starting from v and applying *at least one dereference operation* (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, τ puts some restrictions on reachability; i.e., it might be impossible to have a heap where a variable of type κ_1 reaches one of type κ_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 $\sigma\in\Sigma_\tau$, and two locations $\ell,\ell'\in\text{dom}(\check{\sigma})$ such that (a) $\check{\sigma}(\ell).\mathsf{tag} = \kappa_1$; (b) $\check{\sigma}(\ell').\mathsf{tag} = \kappa_2$; and (c) $\ell'\in R(\check{\sigma},\ell)$. 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.

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

Definition 4.2 (*Reachability Domain*). The reachability abstract domain is the complete lattice $\mathcal{I}_r^\tau = \langle \wp(\mathcal{R}^\tau), \subseteq, \emptyset, \mathcal{R}^\tau, \cap, \cup \rangle$, where

$$\mathcal{R}^\tau = \left\{ v \rightsquigarrow w \;\middle|\; \begin{array}{l} v, w \in \mathrm{dom}\,(\tau), \text{and there exist } \kappa_1 \preceq \tau(v) \text{ and } \kappa_2 \preceq \tau(w) \\ \text{such that } \kappa_2 \text{ is reachable from } \kappa_1 \end{array} \right\}.$$

Here and in the following, elements of the tuple $\langle A, \leq, \bot, \top, \wedge, \vee \rangle$ denoting an abstract domain $\mathcal{A}$ 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, ($\wedge$) the meet operator and ($\vee$) the join operator on A. This terminology is standard in Abstract Interpretation.

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

Lemma 4.3. *The following abstraction and concretization functions define a* Galois insertion *between $\mathcal{I}_r^\tau$ and $\mathcal{I}_\flat^\tau$:*

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

The top element $\mathcal{R}^\tau$ is $\alpha_r^\tau(\Sigma_\tau)$, and represents all states which are compatible with τ. 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 $\mathcal{I}_r^\tau$ are *not* closed by transitivity: e.g., it is possible to have $I_r = \{\mathsf{x}\rightsquigarrow\mathsf{y}, \mathsf{y}\rightsquigarrow\mathsf{z}\}$ which contains $\mathsf{x}\rightsquigarrow\mathsf{y}$ and $\mathsf{y}\rightsquigarrow\mathsf{z}$, but not $\mathsf{x}\rightsquigarrow\mathsf{z}$. Such an abstract value is a reasonable one, and approximates, for example, the execution of the following code

```
x:=new C;
y:=new C;
if (w>0) then x.f:=y; else y.f:=z;
```

Moreover, this abstract value is consistent, i.e., it describes a set of concrete states which is not smaller (actually, it is greater) than $\gamma_r^\tau(\emptyset)$. This happens because reachability is, actually, *may-reach* information, so that, for example, $\gamma_r^\tau(\{\mathsf{x}\rightsquigarrow\mathsf{y}, \mathsf{y}\rightsquigarrow\mathsf{z}\})$ includes (a) any state where x reaches y but y does not reach z; (b) any state where y reaches z but x does not reach y; and (c) any state where x does not reach y and y does not reach z. It is important to point out that $\gamma_r^\tau(\{\mathsf{x}\rightsquigarrow\mathsf{y}, \mathsf{y}\rightsquigarrow\mathsf{z}\})$ does not contain those states where both x reaches y and y reaches z, since, in this case, x would also reach z by transitivity, which is forbidden by soundness since $\mathsf{x}\rightsquigarrow\mathsf{z} \notin I_r$.

4.2. Cyclicity

Given a state $\sigma \in \Sigma_\tau$, a variable $v \in \mathrm{dom}\,(\tau)$ is said to be *cyclic* in σ if there exists $\ell \in R^\varepsilon(\check{\sigma}, \hat{\sigma}(v))$ such that $\ell \in R(\check{\sigma}, \ell)$. In other words, v is cyclic if it reaches some memory location ℓ (which can possibly be $\hat{\sigma}(v)$ itself) through which a cyclic path goes. Similarly to reachability, it might be impossible to generate a cyclic data structure starting from a variable of some type κ. A class $\kappa \in \mathcal{K}$ is said to be a *cyclic class* if there exist $\sigma \in \Sigma_\tau$ and $\ell, \ell' \in \mathrm{dom}\,(\check{\sigma})$ such that $\check{\sigma}(\ell).\mathsf{tag} = \kappa$, $\ell' \in R^\varepsilon(\check{\sigma}, \ell)$, and $\ell' \in R(\check{\sigma}, \ell')$. The cyclicity domain is the dual of the non-cyclicity domain by Rossignoli and Spoto [26].

Definition 4.5 (*Cyclicity Domain*). The abstract domain for cyclicity is represented as the complete lattice $\mathcal{I}_c^\tau = \langle \wp(\mathcal{Y}^\tau), \subseteq, \emptyset, \mathcal{Y}^\tau, \cap, \cup \rangle$ where

$$\mathcal{Y}^\tau = \{\circlearrowleft^v \mid v \in \tau, \text{and there exists a cyclic class } \kappa \preceq \tau(v)\}.$$

Lemma 4.6. *The following abstraction and concretization functions define a Galois insertion between $\mathcal{I}_c^\tau$ and $\mathcal{I}_\flat^\tau$*

$$\begin{array}{l} \alpha_c^\tau(I_\flat) = \{\circlearrowleft^v \mid \exists v \in \tau .\, \exists \sigma \in I_\flat .\, v \textit{ is cyclic in } \sigma\} \\ \gamma_c^\tau(I_c) = \{\sigma \mid \sigma \in \Sigma_\tau \wedge \forall v \in \tau .\, (v \textit{ is cyclic in } \sigma) \Rightarrow \circlearrowleft^v \in I_c\}. \end{array}$$

May-be-cyclic information is described by *abstract values* $I_c \in \wp(\mathcal{Y}^\tau)$. For instance, $\{\circlearrowleft^{\mathsf{x}}\}$ represents states where no variable but x can be cyclic. The top element $\mathcal{Y}^\tau$ is concretized to Σ_τ; 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:

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

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 $\mathcal{I}^{\tau}_{rc} = \mathcal{I}^{\tau}_{r} \times \mathcal{I}^{\tau}_{c}$. Elements of $\mathcal{I}^{\tau}_{rc}$ are pairs $\langle I_r, I_c \rangle$, where I_r and I_c contain, respectively, the may-reach and the may-be-cyclic information. The abstraction and concretization functions are induced by those on $\mathcal{I}^{\tau}_{c}$ and $\mathcal{I}^{\tau}_{r}$:

$$\gamma^{\tau}_{rc}(\langle I_r, I_c \rangle) = \gamma^{\tau}_{r}(I_r) \cap \gamma^{\tau}_{c}(I_c) \qquad \alpha^{\tau}_{rc}(I) = \langle \alpha^{\tau}_{r}(I), \alpha^{\tau}_{c}(I) \rangle.$$

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

Lemma 4.7. *For any abstract values $I^1_r, I^2_r \in \mathcal{I}^{\tau}_{r}$ and $I^1_c, I^2_c \in \mathcal{I}^{\tau}_{c}$, the concretization $\gamma^{\tau}_{rc}(\langle I^1_r, I^1_c \rangle)$ is equal to $\gamma^{\tau}_{rc}(\langle I^2_r, I^2_c \rangle)$ if and only if both conditions hold:* (a) $I^1_c = I^2_c$*; and* (b) $I^1_r \setminus \{v \rightsquigarrow v \mid \circlearrowleft^v \notin I^1_c\} = I^2_r \setminus \{v \rightsquigarrow v \mid \circlearrowleft^v \notin I^2_c\}$.

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 $\circlearrowleft^v$ to $I_{rc} \in \mathcal{I}^{\tau}_{rc}$ results in representing a strictly larger set of states; and (b) adding a pair $v \rightsquigarrow v$ to $I_{rc} \in \mathcal{I}^{\tau}_{rc}$, 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 $I^1_{rc} = \langle I_r, \emptyset \rangle$ and $I^2_{rc} = \langle I_r, \{\circlearrowleft^{\mathsf{x}}\} \rangle$ which result from adding $\circlearrowleft^{\mathsf{x}}$ to I^1_{rc}. Assuming that x does not appear in I_r, there is a state σ which is compatible with I_r (for example, if no v reaches 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 $\hat{\sigma}(\mathsf{x})$). This σ belongs to $\gamma^{\tau}_{rc}(I^2_{rc}) \setminus \gamma^{\tau}_{rc}(I^1_{rc})$ and is, therefore, an example of the difference between the abstract values.

As an example for (b), consider $I^1_{rc} = \langle \emptyset, \{\circlearrowleft^{\mathsf{y}}\} \rangle$ and $I^2_{rc} = \langle \{\mathsf{x} \rightsquigarrow \mathsf{x}\}, \{\circlearrowleft^{\mathsf{y}}\} \rangle$ which results from adding $\mathsf{x} \rightsquigarrow \mathsf{x}$ to I^1_{rc}. At a first glance, I^2_{rc} describes a larger set of states, since it includes states (not belonging to $\gamma^{\tau}_{rc}(I^1_{rc})$) where there is a path from x to x. However, such states will neither belong to $\gamma^{\tau}_{rc}(I^2_{rc})$, since such a path implies that x is cyclic, which is not permitted by $\{\circlearrowleft^{\mathsf{y}}\}$, that only allows y to be cyclic.

Lemma 4.7 provides a way for computing the *normal form* of any $\langle I_r, I_c \rangle$, which comes to be $\langle I_r \setminus \{v \rightsquigarrow v | \circlearrowleft^v \notin I_c\}, I_c \rangle$, i.e., the *canonical form* of its equivalence class. From now on, $\mathcal{I}^{\tau}_{rc}$ will be a shorthand for $\mathcal{I}^{\tau}_{rc\equiv}$, where $\equiv$ is left implicit.

5. Reachability-based acyclicity analysis

This section uses $\mathcal{I}^{\tau}_{rc}$ 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 ε-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 $\ell \in \mathrm{dom}(\check{\sigma})$. Distinguishing among these four possibilities is crucial for a precise acyclicity analysis. In fact, assuming that v and w are initially acyclic, they both become cyclic after executing $v.\mathsf{f}{:=}w$ *if and only if*, initially, w 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 $\mathcal{I}^{\tau}_{rc}$, 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.[2] Two reference variables v and w *share* in σ iff $R^{\varepsilon}(\check{\sigma}, \hat{\sigma}(v)) \cap R^{\varepsilon}(\check{\sigma}, \hat{\sigma}(w)) \neq \emptyset$; also, they *alias* in σ if they point to the same location, namely, if $\hat{\sigma}(v) = \hat{\sigma}(w) \in \mathrm{dom}(\check{\sigma})$. 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] (based on previous work by Secci and Spoto [28]) can be applied: with it,

[2] One could argue that aliasing and sharing analyses benefit from reachability information, so that all the components should better work "in parallel"; however, for the sake of this presentation, the three components (sharing, aliasing, and reachability-cyclicity) are supposed to be independent. See Section 5.4 for a further discussion about the interplay between all the analyses.

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

$$
\begin{array}{lrl}
(1_e) & \mathcal{E}^{\tau}_{\zeta}\llbracket n \rrbracket(I) = & I \\
(2_e) & \mathcal{E}^{\tau}_{\zeta}\llbracket \textbf{null} \rrbracket(I) = & I \\
(3_e) & \mathcal{E}^{\tau}_{\zeta}\llbracket \textbf{new}\ \kappa \rrbracket(I) = & I \\
(4_e) & \mathcal{E}^{\tau}_{\zeta}\llbracket v \rrbracket(I) = & \text{if } \tau(v){=}\textbf{int} \text{ then } I \text{ else } I \cup I[v/\rho] \\
(5_e) & \mathcal{E}^{\tau}_{\zeta}\llbracket v.f \rrbracket(I) = & \text{if } f \text{ has type } \textbf{int} \text{ then } I \text{ else } I \cup I' \text{ where} \\
 & & I'{=}I[v/\rho] \cup \{w{\rightsquigarrow}\rho | \langle w{\bullet}v\rangle{\in}I_s\} \cup \{\rho{\rightsquigarrow}\rho \mid \circlearrowleft^{v} \in I\} \\
(6_e) & \mathcal{E}^{\tau}_{\zeta}\llbracket exp_1 \oplus exp_2 \rrbracket(I) = & \exists\rho.\mathcal{E}^{\tau}_{\zeta}\llbracket exp_2 \rrbracket(\exists\rho.\mathcal{E}^{\tau}_{\zeta}\llbracket exp_1 \rrbracket(I)) \\
(7_e) & \mathcal{E}^{\tau}_{\zeta}\llbracket v_0.m(v_1,..,v_n) \rrbracket(I) = & I \cup I_m \cup I_3 \cup I_4 \text{ where}
\end{array}
$$

$$
\begin{array}{l}
\bar{v}{=}\{v_0,..,v_n\} \\
I_0{=}\exists(\tau\backslash\bar{v}).I \\
I_m = \cup\{\ (\zeta(\mathsf{m})(I_0[\bar{v}/\mathsf{m}^i]))[\mathsf{m}^i/\bar{v}, out/\rho]\ \mid \mathsf{m} \text{ might be called here}^{a}\ \} \\
I'_s = \{\langle v_i{\bullet}v_j\rangle \mid v_i, v_j \in \bar{v} \text{ and } \langle v_i{\bullet}v_j\rangle \in I_s\} \cup \{\dot{v}|v \in \bar{v} \text{ and } \dot{v} \in I_s\} \\
I''_s = \cup\{\mathsf{SP}_{\mathsf{m}}(I'_s[\bar{v}/\mathsf{m}^i])[\mathsf{m}^i/\bar{v}, out/\rho] \mid \mathsf{m} \text{ might be called here}\} \\
I_1 = \{w_1{\rightsquigarrow}w_2 \mid (v_i{\rightsquigarrow}v_j{\in}I_m) \wedge (\dot{v}_i{\in}I''_s) \wedge (\langle w_1{\bullet}v_i\rangle{\in}I'_s) \wedge \\
\quad ((v_j{\rightsquigarrow}w_2{\in}I) \vee \langle w_2{\cdot}v_j\rangle{\in}I'_s)\} \\
I_2 = \{w_1{\rightsquigarrow}w_2 \mid (\langle v_i{\bullet}v_j\rangle \in I'_s) \wedge (\dot{v}_i \in I''_s) \wedge (\langle v_i{\bullet}w_1\rangle \in I'_s) \wedge (v_j{\rightsquigarrow}w_2 \in I)\} \\
I_3 = \cup\{(I_1{\cup}I_2)[v/\rho] \mid \langle v{\cdot}\rho\rangle \text{ after the call }\} \\
I_4 = \{\circlearrowleft^{w} \mid (\langle w{\bullet}v\rangle \in I'_s) \wedge (\dot{v} \in I''_s) \wedge (\circlearrowleft^{v} \in I_m)\}
\end{array}
$$

[a] See Section 5.2.5

Fig. 4. Abstract denotations for expressions.

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 SP_{m} is given: for a set of pairs I_{sp} which safely describes the sharing between actual arguments in the input state, $I'_{sp} = \mathsf{SP}_{\mathsf{m}}(I_{sp})$ is such that (i) if $\langle v{\bullet}w\rangle \in I'_{sp}$, then v and w might share during the execution of m; and (ii) $\dot{v}_i \in I'_{sp}$ 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 $\mathcal{I}^{\tau}_{sp}$, whose elements I_{sp} contain may-share statements $\langle v{\bullet}w\rangle$ and may-be-non-pure statements $\dot{v}$. Abstraction and concretization functions α_{sp} and γ_{sp} are defined in the standard way [14]: in particular, $\gamma_{sp}(I_{sp})$ 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 possible be non-pure.

As for *aliasing*, the abstract domain $\mathcal{I}^{\tau}_{al}$ contains sets of may-alias statements $\langle v{\cdot}w\rangle$: if $\langle v{\cdot}w\rangle$ is contained in I_{al}, then its concretization $\gamma_{al}(I_{al})$ 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 $\mathcal{I}^{\tau}_{s}$ will be the reduced product between $\mathcal{I}^{\tau}_{sp}$ and $\mathcal{I}^{\tau}_{al}$, and combines sharing, aliasing, and purity information. As usual, $\gamma_s((I_{sp}, I_{al}))$ is defined as $\gamma_{sp}(I_{sp}) \cap \gamma_{al}(I_{al})$, while $\alpha_s(X)$ is defined as $(\alpha_{sh}(X) \cup \alpha_{al}(X))_{\equiv}$, 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_s will be often used to denote an abstract value without specifying the abstract domain it belongs to. The use of I_s will be clear from the context: for example, writing $\gamma_{al}(I_s)$ means applying γ_{al} to the part of I_s which represents aliasing information.

Moreover, an abstract element $\langle I_r, I_c\rangle \in \mathcal{I}^{\tau}_{rc}$ will be represented by the set $I = I_r \cup I_c$; therefore, $v{\rightsquigarrow}w \in I$ and $\circlearrowleft^{v} \in I$ are shorthands for, respectively, $v{\rightsquigarrow}w \in I_r$ and $\circlearrowleft^{v} \in I_c$. The operation $\exists v.I$ (*projection*) removes any statement about v from I, while $I[v/w]$ (*renaming*) v to w in I. For the sake of simplicity, class-reachability and class-cyclicity are taken into account *implicitly*: a new statement $v{\rightsquigarrow}w$ is not added to an abstract state if $v{\rightsquigarrow}w \notin \mathcal{R}^{\tau}$, while a statement $\circlearrowleft^{v}$ is not added if $\circlearrowleft^{v} \notin \mathcal{Y}^{\tau}$. It is important to point out that information about class-reachability and class-cyclicity (i.e., whether κ_1 reaches κ_2, or whether κ 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 $\mathcal{R}^{\tau}$ or $\mathcal{Y}^{\tau}$.

5.2. The abstract semantics

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

Figs. 4 and 5 depict abstract denotations. An *expression denotation* $\mathcal{E}^{\tau}_{\zeta}\llbracket exp \rrbracket$ maps abstract states from $\mathcal{I}^{\tau}_{rc}$ to abstract states from $\mathcal{I}^{\tau_1}_{rc}$ where $\tau_1 = \tau \cup \{\rho\}$, while a *command denotation* $\mathcal{C}^{\tau}_{\zeta}\llbracket com \rrbracket$ maps $\mathcal{I}^{\tau}_{rc}$ to $\mathcal{I}^{\tau}_{rc}$.

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

$$
\begin{array}{lrl}
(1_c) & \mathcal{C}_\zeta^\tau\llbracket v{:=}exp\rrbracket(I) = & (\exists v.\mathcal{E}_\zeta^\tau\llbracket exp\rrbracket(I))[\rho/v] \\
(2_c) & \mathcal{C}_\zeta^\tau\llbracket v.f{:=}exp\rrbracket(I) = & \exists\rho.(I'\cup I_r\cup I_c)\text{ where} \\
 & I_0' = & \mathcal{E}_\zeta^\tau\llbracket exp\rrbracket(I) \\
 & I' = & condRemove(I_0',v,f) \\
 & I_r = & \big\{w_1{\rightsquigarrow}w_2 \mid (((\langle w_1{\cdot}v\rangle{\in}I_s') \vee (w_1{\rightsquigarrow}v{\in}I')) \wedge \\
 & & (((\langle\rho{\cdot}w_2\rangle{\in}I_s') \vee (\rho{\rightsquigarrow}w_2{\in}I'))\big\} \\
 & I_c = & \big\{\circlearrowleft^w \mid ((\rho{\rightsquigarrow}v\in I') \vee (\langle\rho{\cdot}v\rangle{\in}I_s') \vee (\circlearrowleft^\rho{\in}I')) \wedge \\
 & & (((\langle w{\cdot}v\rangle{\in}I_s') \vee (w{\rightsquigarrow}v{\in}I'))\big\} \\
(3_c) & \mathcal{C}_\zeta^\tau\left\llbracket\begin{array}{ll}\textbf{if } exp & \textbf{then } com_1\\ & \textbf{else } com_2\end{array}\right\rrbracket(I) = & \mathcal{C}_\zeta^\tau\llbracket com_1\rrbracket(I)\ \cup\ \mathcal{C}_\zeta^\tau\llbracket com_2\rrbracket(I) \\
(4_c) & \mathcal{C}_\zeta^\tau\llbracket\textbf{while } exp \textbf{ do } com\rrbracket(I) = & \xi(I)\text{ where }\xi = lfp(\lambda w.\lambda I.w(\mathcal{C}_\zeta^\tau\llbracket com\rrbracket(I))) \\
(5_c) & \mathcal{C}_\zeta^\tau\llbracket\textbf{return } exp\rrbracket(I) = & \mathcal{E}_\zeta^\tau\llbracket exp\rrbracket(I)[\rho/out] \\
(6_c) & \mathcal{C}_\zeta^\tau\llbracket com_1;\, com_2\rrbracket(I) = & \mathcal{C}_\zeta^\tau\llbracket com_2\rrbracket(\mathcal{C}_\zeta^\tau\llbracket com_1\rrbracket(I))
\end{array}
$$

Fig. 5. Abstract denotations for commands.

In the definition, the abstract element I_s contains the sharing, aliasing, and purity information pre-computed by other analyses, and referring to the program point of interest.[3]

5.2.1. Expressions

An expression denotation $\mathcal{E}_\zeta^\tau\llbracket exp\rrbracket$ adds to an input state I those reachability and cyclicity statements which result from evaluating *exp*.

Nothing is added to I in cases (1_e): $\mathcal{E}_\zeta^\tau\llbracket n\rrbracket$, (2_e): $\mathcal{E}_\zeta^\tau\llbracket\textbf{null}\rrbracket$, and (3_e): $\mathcal{E}_\zeta^\tau\llbracket\textbf{new }\kappa\rrbracket$ 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 I in case (4_e): $\mathcal{E}_\zeta^\tau\llbracket v\rrbracket$ when $\tau(v){=}\textbf{int}$, and (5_e): $\mathcal{E}_\zeta^\tau\llbracket v.f\rrbracket$ when f is an **int** field.

In case (4_e): $\mathcal{E}_\zeta^\tau\llbracket v\rrbracket$, when the type of v is not **int**, the result variable ρ has the same abstract behavior as v. Therefore, the semantics returns I, together with a *cloned* version $I[v/\rho]$ where statements about v have been replaced by renamed statements about ρ.

In the case of (5_e): $\mathcal{E}_\zeta^\tau\llbracket v.f\rrbracket$, when f is a reference field, the following information is added to I:

- statements for v which are cloned for ρ;
- $w{\rightsquigarrow}\rho$, if w might share with v; note that $v{\rightsquigarrow}\rho$ is always added since $\langle v{\bullet}v\rangle\in I_s$ (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 $w{\rightsquigarrow}\rho$ must be added because $v.f$ could be exactly the common location which is reached by both v and w;
- if v might be cyclic, then, for soundness, $\rho{\rightsquigarrow}\rho$; note that, in this case, $\circlearrowleft^\rho$ is also guaranteed to have been previously added to the abstract state.

In case (6_e): $\mathcal{E}_\zeta^\tau\llbracket exp_1\oplus exp_2\rrbracket$, the expression exp_1 is first analyzed, then exp_2 is analyzed on the resulting abstract state. Note that, in both cases, ρ is removed since the return value has always type **int**.

Finally, method calls (7_e): $\mathcal{E}_\zeta^\tau\llbracket v_0.m(v_1,..,v_n)\rrbracket$ will be explained later, after introducing denotations for commands.

Example 5.1. Consider c:=c.next at line 15 in Fig. 1. Evaluating the denotation $\mathcal{E}_\zeta^\tau\llbracket\text{c.next}\rrbracket(I_{14})$ results in $\{this{\rightsquigarrow}c,\ this{\rightsquigarrow}p,\ this{\rightsquigarrow}\rho,\ c{\rightsquigarrow}\rho,\ p{\rightsquigarrow}\rho\}$. The statement $this{\rightsquigarrow}\rho$ is added since $this{\rightsquigarrow}c\in I_{14}$; $c{\rightsquigarrow}\rho$ and $p{\rightsquigarrow}\rho$ are added because $\langle c{\bullet}c\rangle$ and $\langle c{\bullet}p\rangle$ hold after line 14.

5.2.2. Variable assignment

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

Example 5.2. Consider, again, line 15 in Fig. 1. Evaluating the denotation $\mathcal{C}_\zeta^\tau\llbracket\text{c:=c.next}\rrbracket(I_{14})$ first computes $\mathcal{E}_\zeta^\tau\llbracket\text{c.next}\rrbracket(I_{14})$ as in Example 5.1. Then, statements involving c are removed, which results in $\{this{\rightsquigarrow}p,\ this{\rightsquigarrow}\rho,\ p{\rightsquigarrow}\rho\}$, and, finally, ρ is renamed to c, giving $\{this{\rightsquigarrow}p,\ this{\rightsquigarrow}c,\ p{\rightsquigarrow}c\}$. Note that $this{\rightsquigarrow}c$ is reinserted (by renaming $this{\rightsquigarrow}\rho$) after being deleted by $\exists c$. Also, note that $c{\rightsquigarrow}\rho$ has been removed by $\exists c$, so that, correctly, c is not considered to reach itself after the assignment.

[3] Note that I_s could be represented explicitly as an input to the abstract semantics, next to I, but it is not written for better clarity.

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

5.2.3. Field update

The denotation (2_c): $\mathcal{C}^{\tau}_{\zeta}[\![v.f{:=}exp]\!]$ accounts for field updates. The set I'_0 results from computing $\mathcal{E}^{\tau}_{\zeta}[\![exp]\!](I)$, 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 $condRemove(I'_0, v, f)$, which is defined as follows. Let κ be the declared class of v (this means that the runtime type of v can be κ or any of its subclasses); then I' is obtained by I'_0 by

- removing $\circlearrowleft^{v}$ if (1) f is the only reference field of any $\kappa' \preceq \kappa$; or (2) all the other reference fields of any $\kappa' \preceq \kappa$ have a declared class such that neither it nor any of its subclasses are a cyclic class;
- similarly, removing any statement $v{\rightsquigarrow}w$ such that f is the only field of any $\kappa' \preceq \kappa$ whose declared class κ_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_r and I_c capture the effect of executing $v.f{:=}\rho$ on I'. Moreover, I'_s contains sharing, purity and aliasing information after evaluating *exp*.

The following reachability statements are added: for any w_1 which might either alias with v or reach v (formalized as $(\langle w_1{\cdot}v\rangle \in I'_s) \vee (w_1{\rightsquigarrow}v \in I')$), and any w_2 aliasing with ρ or reachable from it (formalized as $(\langle \rho{\cdot}w_2\rangle \in I'_s) \vee (\rho{\rightsquigarrow}w_2 \in I')$), the statement $w_1{\rightsquigarrow}w_2$ 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 ρ through f.

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

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

Whenever one of these scenarios occurs (formalized as $(\rho{\rightsquigarrow}v \in I') \vee (\langle \rho{\cdot}v\rangle \in I'_s) \vee (\circlearrowleft^{\rho} \in I')$), any variable w aliasing with v or reaching it (formalized as $(\langle w{\cdot}v\rangle \in I'_s) \vee (w{\rightsquigarrow}v \in I')$) 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 $I_{17} = \{this{\rightsquigarrow}c, this{\rightsquigarrow}p, p{\rightsquigarrow}c, n{\rightsquigarrow}c\}$. The evaluation of $\mathcal{C}^{\tau}_{\zeta}[\![\mathsf{p.next{:=}n}]\!](I_{17})$ at line 20 adds a new statement $p{\rightsquigarrow}n$, as expected. Moreover, it also adds $this{\rightsquigarrow}n$ since **this** was reaching p, and both $p{\rightsquigarrow}c$ and $this{\rightsquigarrow}c$ (which, however, were already contained in I_{17}) since n was reaching c.

5.2.4. Conditions, loops, composition, and return command

Rules (3_c): $\mathcal{C}^{\tau}_{\zeta}[\![\textbf{if}\ exp\ \textbf{then}\ com_1\ \textbf{else}\ com_2]\!]$, (4_c): $\mathcal{C}^{\tau}_{\zeta}[\![\textbf{while}\ exp\ \textbf{do}\ com]\!]$, and (6_c): $\mathcal{C}^{\tau}_{\zeta}[\![com_1;\ com_2]\!]$ are quite straightforward and correspond, respectively, to the *if* conditional, the *while* loop, and *command composition*. Finally, rule (5_c): $\mathcal{C}^{\tau}_{\zeta}[\![\textbf{return}\ exp]\!]$ corresponds to the *return* command, and behaves, as expected, like the execution of *out*:=*exp*.

5.2.5. Method calls

Rule (7_e): $\mathcal{E}^{\tau}_{\zeta}[\![v_0.m(v_1, .., v_n)]\!]$ propagates the effect of a method call to the calling context, as follows:

1. the abstract state I is projected on the actual parameters $\bar{v}$, thus obtaining I_0; this is needed since the denotation of the callee is given in terms of its parameters;
2. the denotation of each method m which can be possibly called at runtime is taken from the current interpretation, namely, $\zeta(\mathsf{m})$, and applied to $I_0[\bar{v}/\mathsf{m}^i]$, which is the result of renaming the actual parameters $\bar{v}$ to the formal parameters m^i in I_0;
3. formal parameters are renamed back to the actual parameters (plus *out* and ρ) in the resulting state $\zeta(\mathsf{m})(I_0[\bar{v}/\mathsf{m}^i])$, and the states obtained from all possible signatures are merged into I_m.

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 *lkp* (Section 3). Therefore, the abstract semantics takes, conservatively, the union of all of them.

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

- I_1 states that, if the call creates reachability from v_i to v_j, then any w_1 sharing with v_i before the call might reach any w_2 which is reachable from v_j or aliasing with v_j. Note that adding these statements is necessary only if v_i is updated in the body of some m (this information is conservatively represented in I''_s, so that the condition $\dot{v}_i \in I''_s$ must be checked): otherwise, it is guaranteed that no path from w_1 to w_2 will be created during the call.

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

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

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

```
Node k(Node y) {
  u:=y;
  this.next:=y;
  y:=null;
  return this;
}
```

Fig. 6. Some more examples.

- I_2 states that, if the call makes v_i share with v_j, then any w_1 sharing with v_i might reach any w_2 reachable from v_j. Again, this is required only if v_i is updated in the body of any m.
- I_3 contains the information about any variable v aliasing with ρ, which is cloned for ρ.
- I_4 will include the possible cyclicity of anything sharing with an argument which might become cyclic.

The final result of processing a method call is the union $I \cup I_m \cup I_3 \cup I_4$.

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

1. I is projected on $\{b, c\}$, obtaining $I_0 = \emptyset$;
2. $\xi(\emptyset)$ is renamed such that *this*, y, and *out* are renamed to, respectively, b, c, and ρ, and $I_m = \{b{\rightsquigarrow}c, \rho{\rightsquigarrow}c\}$ is obtained;
3. $a{\rightsquigarrow}this$ is added to I_1 since $(b{\rightsquigarrow}c \in I_m) \wedge (\langle b{\bullet}a\rangle \in I'_s) \wedge (c{\rightsquigarrow}this \in I)$ is true; similarly, $b{\rightsquigarrow}this$, $a{\rightsquigarrow}c$ and $a{\rightsquigarrow}\rho$ 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 ρ to *out* in $I \cup I_m \cup I_1 \cup I_4$, and obtains $\{a{\rightsquigarrow}b, c{\rightsquigarrow}this, b{\rightsquigarrow}c, out{\rightsquigarrow}c, a{\rightsquigarrow}this, a{\rightsquigarrow}c, b{\rightsquigarrow}this, a{\rightsquigarrow}out\}$.

Next, the inference of a denotation for a method m is shown, which uses the denotation $\mathcal{C}^{\tau}_{\varsigma}\left[\!\left[\mathsf{m}^b\right]\!\right]$ 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 ξ for g such that $\xi(\emptyset) = \{this{\rightsquigarrow}y, out{\rightsquigarrow}y\}$ was assumed. Intuitively, this $\xi(\emptyset)$ could be computed using $\mathcal{C}^{\tau}_{\varsigma}\left[\!\left[\mathsf{g}^b\right]\!\right]$, as follows: the first command in g adds $this{\rightsquigarrow}y$, and the second one adds $out{\rightsquigarrow}y$, which results in the desired abstract state $\{this{\rightsquigarrow}y, out{\rightsquigarrow}y\}$. After this result, one might think that $\mathcal{C}^{\tau}_{\varsigma}\left[\!\left[\mathsf{m}^b\right]\!\right](I)$ is always the good way to compute $\xi(I)$, as just done. Yet, in general, this is not correct. For example, suppose the call b.g(c) is replaced by b.h(c) (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 $\mathcal{C}^{\tau}_{\varsigma}\left[\!\left[\mathsf{h}^b\right]\!\right](\emptyset)$ has a different result: the first instruction adds $this{\rightsquigarrow}y$, but the second one removes it since the value of y is overwritten, and the third does not add anything. Therefore, $\mathcal{C}^{\tau}_{\varsigma}\left[\!\left[\mathsf{h}^b\right]\!\right](\emptyset) = \emptyset$, 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 f (since it does not modify y), while it is not in the case of h (since y is modified in the body). A common solution to this problem is to mimic actual parameters by *shallow variables* 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 $\mathcal{C}^{\tau}_{\varsigma}\left[\!\left[\mathsf{k}^b\right]\!\right](\emptyset)$ comes to be $\{this{\rightsquigarrow}u, out{\rightsquigarrow}u\}$, which includes the desired reachability information.

The following definition defines the abstract denotational semantics of a program P as the least fixpoint of an (abstract) transformer of interpretations. Variables $\bar{u}$ play the role of shallow variables. Note that shallow variables appear at the level of the semantics, rather than by transforming the program.

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

Definition 5.7. The *abstract denotational semantics* of a program P is the *lfp* of the transformer

$$\mathcal{T}_P(\zeta) = \{\mathsf{m} \mapsto \lambda I \in \mathcal{I}_{rc}^{\mathsf{m}^i} (\exists X.\mathcal{C}_\zeta^\tau \left[\!\left[\mathsf{m}^b\right]\!\right] (I \cup I[\bar{w}/\bar{u}]))[\bar{u}/\bar{w}] \mid \quad \mathsf{m} \in P \quad \}$$

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

The definition is explained in the following. The operator $\mathcal{T}_P$ transforms the interpretation ζ by assigning a new denotation for each method $\mathsf{m} \in P$, using those in ζ. The new denotation for m maps a given input abstract state $I \in \mathcal{I}_{rc}^{\mathsf{m}^i}$ to an output state abstract from $\mathcal{I}_{rc}^{\mathsf{m}^i \cup \{out\}}$, 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 $I_1 = \mathcal{C}_\zeta^\tau \left[\!\left[\mathsf{m}^b\right]\!\right] (I_0)$;
3. all variables but $\bar{u} \cup \{this, out\}$ are eliminated from I_1 (using $\exists X$); and
4. shallow variables $\bar{u}$ are finally renamed back to $\bar{w}$.

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

Example 5.8. Consider the following method

```
 1int mirror(Tree t) {
 2  Tree l,r;
 3
 4  if  (t=null)  then {
 5    return 0;
 6  } else {
 7    l:=t.left;
 8    r:=t.right;
 9    t.left:=r;
10    t.right:=l;
11    return 1+mirror(l)+mirror(r);
12  }
13}
```

and suppose that class Tree implements binary trees in the standard way, with fields left and right. The call mirror(t) exchanges the values of left and right of each node in t, and returns the number of nodes in the tree. An initial state $\emptyset$ is transformed by mirror as follows. Suppose that the current interpretation ζ is such that $\zeta(\text{mirror}) = \xi$, 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 **null**, line 7 adds $t \rightsquigarrow l$; line 8 adds $t \rightsquigarrow r$; line 9 adds again $t \rightsquigarrow r$; and line 10 adds again $t \rightsquigarrow l$. Recursive calls mirror(l) and mirror(r) do not add any statement since $\xi(\emptyset) = \emptyset$. Finally, **return** adds nothing. Projecting $\{t \rightsquigarrow l, t \rightsquigarrow r\}$ on t and out results in $\emptyset$, so that $\xi(\emptyset)$ does not change, and there is no need for another iteration. It can be concluded that, as expected, mirroring the tree does not make it cyclic.

Example 5.9. Consider the following method

```
 1Node connect() {
 2  Node curr;
 3
 4  curr=this;
 5  while (curr.next!=null) {
 6    curr:=curr.next;
 7  }
 8  curr.next:=this;
 9  return curr;
10}
```

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

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 ι and ζ 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 $\delta = \iota(\mathsf{m})$ and $\xi = \zeta(\mathsf{m})$. It holds that, for all $\sigma_1 \in \Sigma_{m^i}$,*

$$\sigma_2 = \delta(\sigma_1) \qquad \Rightarrow \qquad \langle \hat{\sigma}_1[out \mapsto \hat{\sigma}_2(out)], \check{\sigma}_2\rangle \in \gamma^\tau_{rc}(\xi(\alpha^\tau_{rc}(\{\sigma_1\}))).$$

5.4. Completeness and optimality

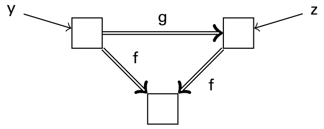
Completeness [18] is a well-known notion in Abstract Interpretation, and corresponds to require that no loss of precision is introduced by computing an abstract semantic function on abstract states with respect to approximating the same (concrete) computation on concrete states. An abstract domain A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

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 I, $f^\#(I)$ must be equal to $\alpha(f(\gamma(I)))$.

For the sake of the following discussion, the abstract semantics $\mathcal{C}^\tau_\zeta[\![\_]\!]$ (a similar discussion holds for $\mathcal{E}^\tau_\zeta[\![\_]\!]$) is supposed to use, for collecting sharing, aliasing and purity information, the *best correct approximation* $\mathcal{S}^\zeta_\tau[\![\_]\!]$ of $C^\iota_\tau[\![\_]\!]$ with respect to $\mathcal{I}^\tau_s$: for every command *com* and abstract value $I \in \mathcal{I}^\tau_s$, $\mathcal{S}^\zeta_\tau[\![com]\!](I)$ is defined as $\alpha_s(C^\iota_\tau[\![com]\!](\gamma_s(I)))$. 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 $\mathcal{I}^\tau_{rc} \sqcap \mathcal{I}^\tau_s$ (i.e., sharing, aliasing and purity are included). Consider the state σ obtained by executing the following statements, starting from a heap where all variables are **null**: the final result of the execution is the heap shown in the picture.

```
y:=new C;
z:=new C;
y.f:=new C;
z.f:=y.f;
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 σ is $I = \{\langle y{\bullet}y\rangle, \langle z{\bullet}z\rangle, \langle y{\bullet}z\rangle, \langle y{\cdot}y\rangle, \langle z{\cdot}z\rangle, y{\rightsquigarrow}z\}$.[4] Suppose that the statement

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

is executed afterward, giving the concrete state σ': in this case, the concrete function f under study is the semantic of this statement, namely, $C^\zeta_\tau[\![\mathsf{x{:=}y.f}]\!]$, and the state σ' corresponds to $C^\zeta_\tau[\![\mathsf{x{:=}y.f}]\!](\sigma)$. Now, the abstraction of σ' with respect to $\mathcal{I}^\tau_{rc} \sqcap \mathcal{I}^\tau_s$ is

$$I' = \{\langle x{\bullet}x\rangle, \langle y{\bullet}y\rangle, \langle z{\bullet}z\rangle, \langle x{\bullet}y\rangle, \langle x{\bullet}z\rangle, \langle y{\bullet}z\rangle, \langle x{\cdot}x\rangle, \langle y{\cdot}y\rangle, \langle z{\cdot}z\rangle, y{\rightsquigarrow}x, y{\rightsquigarrow}z, z{\rightsquigarrow}x\}$$

which correctly represents the sharing between 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 $f^\#$ (i.e., the present analysis $\mathcal{C}^\tau_\zeta[\![\mathsf{x{:=}y.f}]\!]$) on the input abstract state I gives the state

$$\begin{aligned} I'' = \{&\langle x{\bullet}x\rangle, \langle y{\bullet}y\rangle, \langle z{\bullet}z\rangle, \langle x{\bullet}y\rangle, \langle x{\bullet}z\rangle, \langle y{\bullet}z\rangle, \\ &\langle x{\cdot}x\rangle, \langle y{\cdot}y\rangle, \langle z{\cdot}z\rangle, \langle x{\cdot}y\rangle, \langle x{\cdot}z\rangle, \langle y{\cdot}z\rangle, y{\rightsquigarrow}x, y{\rightsquigarrow}z, z{\rightsquigarrow}x, x{\rightsquigarrow}z\}.\end{aligned}$$

The reachability statement $x{\rightsquigarrow}z$ 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 $f^\#_1 = \mathcal{E}^\tau_\zeta[\![v.f]\!]$ for field access, which is optimal with respect to $f_1 = E^\iota_\tau[\![v.f]\!]$, and $f^\#_2 = \mathcal{C}^\tau_\zeta[\![v.f{:=}\rho]\!]$ for field updates, which is optimal with respect to $f_2 = C^\iota_\tau[\![v.f{:=}\rho]\!]$. The use of ρ 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 $f^\#_2$ will be applied to the abstract state I'_0 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_1 the corresponding output; therefore, J' 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 γ and α are the ones which are induced by the reduced product $\mathcal{I}^\tau_{rc} \sqcap \mathcal{I}^\tau_s$ in the standard way. This means that optimality is

[4] The notation $\langle\_{\bullet}\_\rangle$ and $\langle\_{\cdot}\_\rangle$ is used in the beginning of Section 5.

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

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 cyclicity, and that it will be clear from the context how to refer to each part.

By soundness, the non-strict inequalities $\mathcal{E}^{\tau}_{\zeta}[\![v.f]\!](J) \supseteq \alpha(E^{\iota}_{\tau}[\![v.f]\!](\gamma(J)))$ and $\mathcal{C}^{\tau}_{\zeta}[\![v.f{:=}\rho]\!](J) \supseteq \alpha(C^{\iota}_{\tau}[\![v.f{:=}\rho]\!](\gamma(J)))$ already hold, where set inclusion is the partial order on $\mathcal{I}^{\tau}_{rc} \sqcap \mathcal{I}^{\tau}_{s}$. 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 J, there is a concrete state $\sigma \in \gamma(J)$ such that $\sigma_1 = C^{\iota}_{\tau}[\![v.f{:=}\rho]\!](\sigma)$ (the case of $E^{\iota}_{\tau}[\![v.f]\!](\sigma)$ is similar) is a concrete state whose abstraction $\alpha(\{\sigma_1\})$ contains st. In other words, σ_1 is a state where the may-information represented by st is *actually* happening (for example, if st is some $v{\rightsquigarrow}w$, then there must *actually* be a path in the heap from v to w in σ_1), so that the abstraction of σ_1 will forcefully contain such a statement. In the proof, this idea of "a statement st actually happening in a state σ" will be phrased as "σ *justifies st*".

- Case $f^{\#}_1$: the output abstract state J_1 is basically the union of four sets: (a) J; (b) $J[v/\rho]$; (c) $\{w{\rightsquigarrow}\rho \mid \langle w{\bullet}v\rangle{\in}J_s\}$; and (d) $\{\rho{\rightsquigarrow}\rho \mid \circlearrowleft^{v} \in J\}$. 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 σ such that the corresponding output state $\sigma_1 = f_1(\sigma)$ justifies st.
  (a) Clearly, every statement st which was already in J, and is therefore maintained in J_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 σ justifying st, and the corresponding output σ_1 still justifies such statement.
  (b) In this case, relevant statements in J 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 $J[v/\rho]$ (where v is replaced by ρ) are justified.
  $v{\rightsquigarrow}w$: In this case, there certainly exists σ in the concretization of J such that v actually reaches w in at least two steps, and the first step goes through f; then, the location pointed to by the expression actually reaches w (in fact, it is on the path from v to w), so that σ_1 justifies the statement $\rho{\rightsquigarrow}w$ contained in $J[v/\rho]$, corresponding to $v{\rightsquigarrow}w$;
  $w{\rightsquigarrow}v$: This case is easy since there exists σ such that w actually reaches v, and it is straightforward to see that ρ will be actually reached by w in σ_1 (transitivity of reachability at the concrete level), thus justifying the corresponding statement $w{\rightsquigarrow}\rho$ in $J[v/\rho]$;
  $v{\rightsquigarrow}v$: This case is also easy because there certainly exists σ such that v is cyclic, and the first step of the cycle when starting from v goes through f; 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 $\rho{\rightsquigarrow}\rho$ in $J[v/\rho]$;
  $\circlearrowleft^{v}$: This case is similar to the previous one.
  (c) In this case, every $w{\rightsquigarrow}\rho$ 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 ρ 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 σ where v is cyclic (but not necessarily reaching itself), and the location pointed to by $v.f$ reaches itself.
- Case $f^{\#}_2$: 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 := new C();
2   x.f := x;
3   x.f := null;
```

shows the lack of optimality under the condition that f is the only field of C. In fact, let the abstract value J before line 3 be $\{\mathtt{x}{\rightsquigarrow}\mathtt{x}, \circlearrowleft^{\mathtt{x}}\}$ as it would be obtained by the analysis, so that $\gamma(J)$ contains all the states where x is cyclic and reaches itself. However, the abstract semantics without the optimization would generate the same abstract value $\{\mathtt{x}{\rightsquigarrow}\mathtt{x}, \circlearrowleft^{\mathtt{x}}\}$ 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 statements 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 $J_r = \{w_1{\rightsquigarrow}w_2 \mid ((\langle w_1{\cdot}v\rangle{\in}J'_s) \vee (w_1{\rightsquigarrow}v \in J')) \wedge ((\langle \rho{\cdot}w_2\rangle{\in}J'_s) \vee (\rho{\rightsquigarrow}w_2 \in J'))\}$; and (b) the one coming from $J_c = \{\circlearrowleft^{w} \mid ((\rho{\rightsquigarrow}v \in J') \vee (\langle \rho{\cdot}v\rangle{\in}J'_s) \vee (\circlearrowleft^{\rho} \in J')) \wedge ((\langle w{\cdot}v\rangle{\in}J'_s) \vee (w{\rightsquigarrow}v \in J'))\}$.
  (a) In order to justify a statement $w_1{\rightsquigarrow}w_2$, it is enough to take a concrete state $\sigma \in \gamma_s(J)$ (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 σ_1, so that the statement is justified.
  (b) A statement $\circlearrowleft^{w}$ can be easily justified by taking σ 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 ρ 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 $v_{\kappa.f}$ 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

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

hold, where $\subseteq$ is the ordering on $\mathcal{I}_\flat^\tau$.

Part (a). We show that $v{\rightsquigarrow}w \in I_r \Leftrightarrow v{\rightsquigarrow}w \in \gamma_r^\tau(\alpha_r^\tau(I_\flat))$. ($\Rightarrow$) assume $v{\rightsquigarrow}w \in I_r$; then, according to the definition of $\mathcal{I}_r^\tau$ and class reachability, there must be a concrete state $\sigma \in \Sigma_\tau$ in which v reaches w, since, otherwise, the statement $v{\rightsquigarrow}w$ cannot be part of the domain $\mathcal{I}_r^\tau$. We construct a state σ' from σ by setting all reference variables but v and w to **null**. By the definition of γ_r^τ, this specific σ' must be in $\gamma_r^\tau(I_r)$. This, according to the definition of α_r^τ, implies that $v{\rightsquigarrow}w \in \alpha_r^\tau(\gamma_r^\tau(I_r))$. ($\Leftarrow$) assume $v{\rightsquigarrow}w \in \alpha_r^\tau(\gamma_r^\tau(I_r))$. According to the definition of α_r^τ, this means that there exists at least one $\sigma \in \gamma_r^\tau(I_r)$ in which v reaches w, and, according to the definition of γ_r^τ, this can only happen if $v{\rightsquigarrow}w \in I_r$.

Part (b). We show that $\sigma \in I_\flat \Rightarrow \sigma \in \gamma_r^\tau(\alpha_r^\tau(I_\flat))$. Let $\sigma \in I_\flat$, and let I_r be the set of all reachability relations in σ, i.e., v reaches w in σ iff $v{\rightsquigarrow}w \in I_r$. Clearly, $I_r \subseteq \alpha_r^\tau(I_\flat)$. Then, according to the definition of γ_r^τ, σ must be in $\gamma_r^\tau(\alpha_r^\tau(I_\flat))$ since it satisfies $\forall v, w \in \tau.\ v$ reaches w in $\sigma \Rightarrow v{\rightsquigarrow}w \in \alpha_r^\tau(I_\flat)$. □

A.2. Proof of Lemma 4.6

Very similar to the proof of Lemma 4.3. □

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

A.3. *Proof of Lemma 4.7*

(⇒). We show that:

$$\underbrace{\gamma_{rc}^{\tau}(\langle I_r^1, I_c^1\rangle)=\gamma_{rc}^{\tau}(\langle I_r^2, I_c^2\rangle)}_{F} \quad\Rightarrow\quad \underbrace{I_c^1=I_c^2}_{G}\wedge\underbrace{(I_r^1\setminus\{v\rightsquigarrow v \mid \circlearrowright^{v}\notin I_c^1\})=(I_r^2\setminus\{v\rightsquigarrow v \mid \circlearrowright^{v}\notin I_c^2\})}_{H}.$$

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

1. proving that $I_c^1 \neq I_c^2$ implies $\gamma(\langle I_r^1, I_c^1\rangle) \neq \gamma(\langle I_r^2, I_c^2\rangle)$; and
2. proving that $(I_r^1 \setminus \{v\rightsquigarrow v|\circlearrowright^{v}\notin I_c^1\}) \neq (I_r^2 \setminus \{v\rightsquigarrow v|\circlearrowright^{v}\notin I_c^2\})$ implies $\gamma(\langle I_r^1, I_c^1\rangle) \neq \gamma(\langle I_r^2, I_c^2\rangle)$.

The proof goes as follows.

1. Suppose $I_c^1 \neq I_c^2$, and let $X_1 = \{v \mid \circlearrowright^{v} \in I_c^1 \setminus I_c^2\}$, and $X_2 = \{v \mid \circlearrowright^{v} \in I_c^2 \setminus I_c^1\}$. Note that, by hypothesis, at least one of X_1 and X_2 must be non-empty. For $i \in \{1, 2\}$, let σ_i be a state where
   (a) Every $v \in X_i$ is cyclic, but does not reach itself, and no other variable is cyclic; and
   (b) No variables reach any variables, i.e., $\alpha_r^{\tau}(\{\sigma_i\}) = \emptyset$. 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 σ_1 and σ_2 both belong to $\gamma_r(I_r^1) \cap \gamma_r(I_r^2)$, since they do not include any reachability statement; therefore, if $X_1 \neq \emptyset$, then σ_1 belongs to $\gamma(\langle I_r^1, I_c^1\rangle)$, but not to $\gamma(\langle I_r^2, I_c^2\rangle)$, since $\langle I_r^2, I_c^2\rangle$ does not allow the cyclicity on variables from X_1. Dually, if $X_2 \neq \emptyset$, then σ_2 belongs to $\gamma(\langle I_r^2, I_c^2\rangle)$ but not to $\gamma(\langle I_r^1, I_c^1\rangle)$.
2. Suppose $R_1 = I_r^1 \setminus \{v\rightsquigarrow v \mid \circlearrowright^{v}\notin I_c^1\}$ is different from $R_2 = I_r^2 \setminus \{v\rightsquigarrow v \mid \circlearrowright^{v}\notin I_c^2\}$, and let $S_1 = R_1 \setminus R_2$ and $S_2 = R_2 \setminus R_1$. Note that at least one between S_1 and S_2 is non-empty. If S_1 is not empty, then let $p \in S_1$ be one of the statements which in R_1 but not in R_2. A state σ_1 can be chosen such that
   (a) If $p = v\rightsquigarrow v$, then v is the only cyclic variable in σ_1 (note that the cyclicity of v must be allowed by I_c^1 since, otherwise, p would not be included in R_1 and thus not in S_1 too); and
   (b) If $p = v\rightsquigarrow w$, with $v \neq w$; then, v must reach w in σ_1, and no other variable reaches any other variable. Also, no variables can be cyclic.

   Clearly, in both cases above such state belongs to $\gamma(\langle I_r^1, I_c^1\rangle)$, but it cannot be in $\gamma(\langle I_r^2, I_c^2\rangle)$ because: in (a), either $v\rightsquigarrow v \notin I_r^2$ (so that $\sigma_1 \notin \gamma_r^{\tau}(I_r^2)$), or $\circlearrowright^{v} \notin I_c^2$ (so that thus $\sigma_1 \notin \gamma_c^{\tau}(I_c^2)$); and, in (b), $v\rightsquigarrow w \notin I_r^2$, so that $\sigma_1 \notin \gamma_r^{\tau}(I_r^2)$. Dually, if S_2 is empty, then S_1 cannot be empty, and, with a similar reasoning, a state σ_2 can be found which belongs to $\gamma(\langle I_r^2, I_c^2\rangle)$, but not to $\gamma(\langle I_r^1, I_c^1\rangle)$.

(⇐). We prove that:

$$I_c^1=I_c^2\wedge(I_r^1\setminus\{v\rightsquigarrow v|\circlearrowright^{v}\notin I_c^1\})=(I_r^2\setminus\{v\rightsquigarrow v|\circlearrowright^{v}\notin I_c^2\}) \Rightarrow \gamma_{rc}^{\tau}(\langle I_r^1, I_c^1\rangle)=\gamma_{rc}^{\tau}(\langle I_r^2, I_c^2\rangle).$$

It follows easily from observing that, under the hypothesis of the above implication, the only difference between $\langle I_r^1, I_c^1\rangle$ and $\langle I_r^2, I_c^2\rangle$ is that $\langle I_r^1, I_c^1\rangle$ may contain some statements $v\rightsquigarrow v$ for variables v such that $\circlearrowright^{v} \notin I_c^1$, and $\langle I_r^2, I_c^2\rangle$ may contain some (different) statements $v\rightsquigarrow v$ for variables v such that or $\circlearrowright^{v} \notin I_c^2$. However, adding such statements to both abstract values does not change the set of concrete states they represent, since the possibility that v reaches itself in any concrete state is contradicted by the lack of the $\circlearrowright^{v}$ statement. In other words, there is no concrete state which belongs either to $\gamma_{rc}^{\tau}(\langle I_r^1, I_c^1\rangle)$ or $\gamma_{rc}^{\tau}(\langle I_r^2, I_c^2\rangle)$, but not to $\gamma_{rc}^{\tau}(\langle I_r^1\setminus\{v\rightsquigarrow v|\circlearrowright^{v}\notin I_c^1\},\ I_c^1\rangle)$ and $\gamma_{rc}^{\tau}(\langle I_r^2\setminus\{v\rightsquigarrow v|\circlearrowright^{v}\notin I_c^2\},\ I_c^2\rangle)$ (which are equal by the hypothesis.) □

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 ι and a corresponding abstract one ζ which correctly approximates ι 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 ι and ζ).

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

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

Note that, if σ^* is obtained after evaluating an expression, then $\rho \in \mathrm{dom}(\sigma^*)$, while, if it is obtained after executing a command, then $\mathrm{dom}(\sigma^*) = \mathrm{dom}(\sigma)$.

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

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

Denotations (1_e), (2_e), *and* (3_e). Suppose $\sigma^* \notin \gamma_{rc}^{\tau\cup\{\rho\}}(I^*)$. Then, according to the definition of $\gamma_{rc}^{\tau\cup\{\rho\}}$, it must be the case that (i) w_1 reaches w_2 in σ^* but $w_1{\rightsquigarrow}w_2 \notin I^*$; or (ii) w is cyclic in σ^* but $\circlearrowleft^w \notin I^*$. This contradicts the soundness hypothesis $\sigma \in \gamma_{rc}^{\tau}(I)$, since $I^* = I$ and σ and σ^* have the same reachability and cyclicity information.[5]

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

- If $w_1 \neq \rho \wedge w_2 \neq \rho$, then $\hat{\sigma}(w_2) = \hat{\sigma}^*(w_2) \in R(\hat{\sigma}^*(w_1), \check{\sigma}^*) = R(\hat{\sigma}(w_1), \check{\sigma})$, i.e., w_1 reaches w_2 in σ. By the soundness hypothesis $\sigma \in \gamma_{rc}^{\tau}(I)$ we have $w_1{\rightsquigarrow}w_2 \in I \subseteq I^*$, which contradicts $w_1{\rightsquigarrow}w_2 \notin I^*$.
- If $w_1 = \rho \wedge w_2 \neq \rho$, then $\hat{\sigma}(w_2) = \hat{\sigma}^*(w_2) \in R(\hat{\sigma}^*(\rho), \check{\sigma}^*) = R(\hat{\sigma}(v), \check{\sigma})$, i.e., v reaches w_2 in σ. By the soundness hypothesis $\sigma \in \gamma_{rc}^{\tau}(I)$, we have $v{\rightsquigarrow}w_2 \in I$ and thus $\rho{\rightsquigarrow}w_2 \in I[v/\rho] \subseteq I^*$, which contradicts $\rho{\rightsquigarrow}w_2 \notin I^*$.
- If $w_1 \neq \rho \wedge w_2 = \rho$, then $\hat{\sigma}(v) = \hat{\sigma}^*(\rho) \in R(\hat{\sigma}^*(w_1), \check{\sigma}^*) = R(\hat{\sigma}(w_1), \check{\sigma})$, i.e., w_1 reaches v in σ. By the soundness hypothesis $\sigma \in \gamma_{rc}^{\tau}(I)$, we have $w_1{\rightsquigarrow}v \in I$ and thus $w_1{\rightsquigarrow}\rho \in I[v/\rho] \subseteq I^*$, which contradicts $w_1{\rightsquigarrow}\rho \notin I^*$.
- If $w_1 = \rho \wedge w_2 = \rho$, then $\hat{\sigma}(v) = \hat{\sigma}^*(\rho) \in R(\hat{\sigma}^*(\rho), \check{\sigma}^*) = R(\hat{\sigma}(v), \check{\sigma})$, i.e., v reaches v in σ. By the soundness hypothesis $\sigma \in \gamma_{rc}^{\tau}(I)$, we have $v{\rightsquigarrow}v \in I$ and thus $\rho{\rightsquigarrow}\rho \in I[v/\rho] \subseteq I^*$, which contradicts $\rho{\rightsquigarrow}\rho \notin I^*$.

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

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

- If $w_1 \neq \rho \wedge w_2 \neq \rho$, then $\hat{\sigma}(w_2) = \hat{\sigma}^*(w_2) \in R(\hat{\sigma}^*(w_1), \check{\sigma}^*) = R(\hat{\sigma}(w_1), \check{\sigma})$, i.e., w_1 reaches w_2 in σ. By the soundness hypothesis $\sigma \in \gamma_{rc}^{\tau}(I)$, we have $w_1{\rightsquigarrow}w_2 \in I \subseteq I^*$, which contradicts $w_1{\rightsquigarrow}w_2 \notin I^*$.
- If $w_1 = \rho \wedge w_2 \neq \rho$, then $\hat{\sigma}(w_2) = \hat{\sigma}^*(w_2) \in R(\hat{\sigma}^*(\rho), \check{\sigma}^*) \subseteq R(\hat{\sigma}(v), \check{\sigma})$, i.e., v reaches w_2 in σ. By the soundness hypothesis $\sigma \in \gamma_{rc}^{\tau}(I)$, we have $v{\rightsquigarrow}w_2 \in I$ and thus $\rho{\rightsquigarrow}w_2 \in I[v/\rho] \subseteq I^*$, which contradicts $\rho{\rightsquigarrow}w_2 \notin I^*$.
- If $w_1 \neq \rho \wedge w_2 = \rho$, then $\hat{\sigma}^*(\rho) \in R(\hat{\sigma}(w_1), \check{\sigma}^*) = R(\hat{\sigma}(w_1), \check{\sigma})$, we also have $\hat{\sigma}^*(\rho) \in R(\hat{\sigma}(v), \check{\sigma}))$ (since $\rho = v.f$), i.e., w_1 shares with v in σ. Thus, $w_1{\rightsquigarrow}\rho \in \{w{\rightsquigarrow}\rho \mid \langle w \bullet v\rangle \in I_s\} \subseteq I^*$, which contradicts $w_1{\rightsquigarrow}\rho \notin I^*$.
- If $w_1 = \rho \wedge w_2 = \rho$, then $\hat{\sigma}^*(\rho) \in R(\hat{\sigma}^*(\rho), \check{\sigma}))$, which means that v is cyclic in σ, and by the soundness hypothesis we have $\circlearrowleft^v \in I$, and thus $\rho{\rightsquigarrow}\rho \in \{\rho{\rightsquigarrow}\rho | \circlearrowleft^v \in I\} \subseteq I^*$, which contradicts $w_1{\rightsquigarrow}\rho \notin I^*$.

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

Denotation (6_e). The proof for this case is by structural induction on expressions, where the base-case include the non-compound expressions of cases (1_e)–(5_e) and (7_e), for which we have seen already (case (7_e) is done below) that the abstract denotations correctly approximate the concrete ones. Let $I_1 = \mathcal{E}_{\zeta}^{\tau}[\![exp_1]\!](I)$ and $\sigma_1 = E_{\tau}^{\iota}[\![exp_1]\!](\sigma)$. By the (structural) induction hypothesis, we have $\sigma_1 \in \gamma_{rc}^{\tau\cup\{\rho\}}(I_1)$. Moreover, since the state $\langle\hat{\sigma}, \check{\sigma}_1\rangle$ is basically obtained by removing ρ from σ_1, we also have $\langle\hat{\sigma}, \check{\sigma}_1\rangle \in \gamma_{rc}^{\tau}(\exists\rho.I_1)$. Now, let $I_2 = \mathcal{E}_{\zeta}^{\tau}[\![exp_2]\!](\exists\rho.I_1)$, and $\sigma_2 = E_{\tau}^{\iota}[\![exp_2]\!](\langle\hat{\sigma}, \check{\sigma}_1\rangle)$; then, by the (structural) induction hypothesis, we have $\sigma_2 \in \gamma_{rc}^{\tau\cup\{\rho\}}(I_2)$. Since σ^* is obtained from σ_2 by setting ρ to a number (i.e., there is no reachability or cyclicity relations in σ^* that involve ρ), and since $I^* = \exists\rho.I_2$, we can conclude that $\sigma^* \in \gamma_{rc}^{\tau\cup\{\rho\}}(I^*)$.

Denotation (7_e). Calling a method m consists of an abstract execution of its body on the actual parameters, followed by the propagation of the effects of m to the calling context (i.e., the input abstract state I). First, note that, in the abstract semantics, reachability and cyclicity statements are only removed when a variable is assigned. Due to the use of shallow variables for the parameters, statements about the formal parameters of m are never removed during an abstract execution of its body. Therefore, if, during the execution of m, the variable v reaches w, then, at the end of the method, v will be said to possibly reach w, even if this reachability is destroyed at 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 cyclicity is rather easy. In addition to keeping all cyclicity which is in I, a safe approximation is taken, which states that, if an argument v might become cyclic during the execution of m, then anything that shares with it before the execution might also become cyclic. This is accounted for in the definition of I_4, and is clearly safe. In fact, variables of the calling method which are not arguments of the call, and do not share with any argument v_i, cannot be affected by the execution of m.

The treatment of reachability is more complicated: in addition to I and I_m (which is introduced by the method for $\bar{v}$), it is necessary to take into account the effect of the method call on variables which are not arguments. This is done in the definition of I_1, I_2, and I_3, which model the effects of m on variables which share with its actual arguments. Consider two

[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.

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

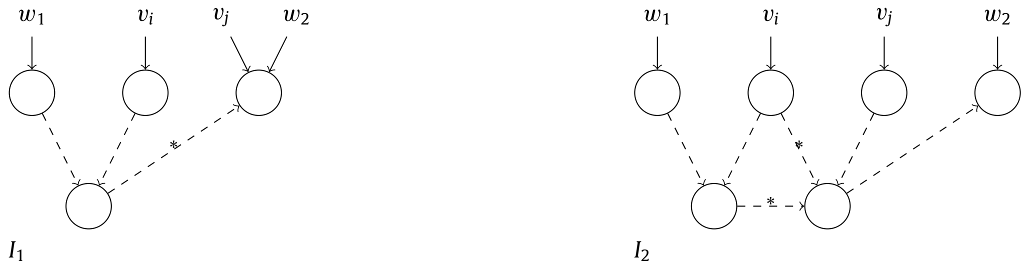


Fig. A.7. Scenarios where a path from w_1 to w_2 can be created inside m. Dashed arrows represent reachability: they connect a variable to a reachable location (represented as a circle). Solid arrows connect a variable u to the location $\hat{\sigma}(u)$ directly bound to it. Arrows labeled with * are paths which are created inside m (strictly speaking, they could also exist before the call), while the others existed before the method call. In both cases, it can be seen that a reachability path from w_1 to w_2 is created, which contains a sub-path created inside m by modifying its arguments.

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 $\dot{v}_i \in sh'$), and that v_i and v_j do not point to disjoint regions of the heap (i.e., either v_i reaches v_j, or they simply share). As a result, if these conditions are met, then the statement $w_1 \rightsquigarrow w_2$ is added. It can be seen that this accounts for all cases where some change in the arguments of m affects the reachability between non-argument variables.

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

Denotation (1_c). Suppose $\sigma^* \notin \gamma_{rc}^{\tau\cup\{\rho\}}(I^*)$. Then, according to the definition of $\gamma_{rc}^{\tau\cup\{\rho\}}$, it must be the case that (i) w_1 reaches w_2 in σ^* but $w_1 \rightsquigarrow w_2 \notin I^*$; or (ii) w is cyclic in σ^* but $\circlearrowleft^w \notin I^*$. Suppose we are in case (i), and let $\sigma_e = E^\iota_\tau[\![exp]\!](\sigma)$ and $I_1 = \mathcal{E}^\tau_\zeta[\![exp]\!](I)$.

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

Case (ii) can be done with similar reasoning.

Denotation (2_c). This case is trivial when f has type **int**, since only side effects during the evaluation of *exp* have to be taken into account. If f has reference type, then this command is equivalent to first evaluating *exp*, and then executing $v.f := \rho$. Let $\sigma' = E^\iota_\tau[\![exp]\!](\sigma)$, and $\ell_e = \hat{\sigma}'(\rho)$. If v and ℓ_e are considered, then there are two main cases (Fig. A.8): (a) $\hat{\sigma}(v) = \ell_e$; or (b) $\hat{\sigma}(v) \neq \ell_e$.

(a) In this case, a cycle on v is created, whose length is 1. If another variable u (possibly, v itself) sharing with v in σ^* is considered, then there are several possible scenarios in the heap, and soundness has to be proven for each of them.
  - u aliases with v or reaches v (cases u_1 and u_2 in the left-hand side of Fig. A.8). In this case, u reaches v via f, and this is taken into account in the definition of I_r, where u plays the role of w_1, and v also plays the role of w_2. The result is that I_r includes $u \rightsquigarrow v$, as expected. The semantics correctly adds $v \rightsquigarrow v$ as well (in fact, v can play the role of both w_1 and w_2). As for cyclicity, the definition of I_c guarantees that $\circlearrowleft^v$ and $\circlearrowleft^u$ will belong to I^*.
  - v reaches u (case u_3 in the same figure). In this case, $v \rightsquigarrow u \in I^*$ since, in the definition of I_r, u plays the role of w_2 (note that v and ρ alias). v will also be considered as cyclic by the definition of I_c;
  - v and u both reach a common location ℓ (case u_4). If none of the previous cases happens, then v and u do not reach each other, so that I^* does not need to contain reachability statements between them. In general, only v will be considered as cyclic in this case (in the same way as the previous cases).

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

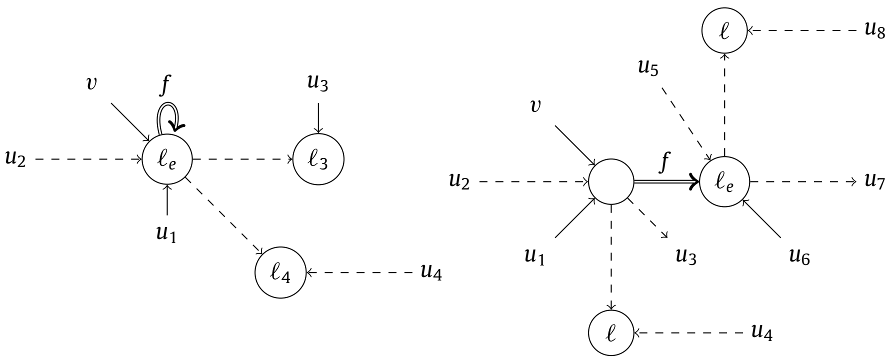


Fig. A.8. The possible scenarios for case (2_c): (a) ℓ_e and $\hat{\sigma}(v)$ coincide (left); and (b) they do not coincide. Variables u_i represent the possible relations between the variable u used in the proof and the data structure modified by the field update. Double solid arrows stand for field dereferencing, and are labeled with the name of the field. For the other kinds of arrows, see Fig. A.7.

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

- u reaches v or aliases with it (cases u_1 and u_2 in the right-hand side of Fig. A.8). In this case, it was also reaching v (or aliasing with it) in σ', so that (in the case of reachability) $u \rightsquigarrow v \in I'$, which implies $u \rightsquigarrow v \in I''$, as soundness requires. As for cyclicity, in case (i), the cyclicity of u is detected because it reaches v.
- Cases u_3, u_4, and u_5. These cases are easy, because nothing changes with respect to the reachability between u and v, and all the statements were already contained in I.
- u points to ℓ_e or is reached by it (cases u_6 and u_7). In this case, u plays the role of w_2 in the definition of I_r, and is correctly considered to be reached by v. As for cyclicity, u will only become cyclic in case (i) if it points to ℓ_e, or belongs to the cyclic path. In both cases, the semantics accounts for it since u would reach v, thus being considered as cyclic (definition of I_c).
- w and ℓ_e reach some common location ℓ (case u_8). Also easy since nothing changes with respect to the reachability between u and v.

Note that, due to the discussion in Section 5.2.3, the single-field optimization introduced by *condRemove* is not problematic for soundness, since the removal of statements is only applied if the required conditions about v and f are guaranteed to hold. In any case, the conservative choice of taking $condRemove(I'_0, v, f)$ to be I'_0 itself is also sound.

Denotation (3_c). This case is quite straightforward, given the inductive hypothesis on com_1 and com_2, and the assumption that *exp* has no side effects and returns an **int**. Suppose $\sigma^* = C^{\iota}_{\tau}[\![com_i]\!](\sigma)$ for $i \in \{1, 2\}$, then, by the induction hypothesis, $\sigma^* \in \gamma^{\tau}_{rc}(\mathcal{C}^{\tau}_{\zeta}[\![com_i]\!](I)) \subseteq \gamma^{\tau}_{rc}(\mathcal{C}^{\tau}_{\zeta}[\![com_i]\!](I)) \cup \gamma^{\tau}_{rc}(\mathcal{C}^{\tau}_{\zeta}[\![com_2]\!](I)) = I^*$.

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

Having proven that all abstract denotations are sound with respect to the concrete denotational semantics, together with Definition 5.7 and the definition of a denotational semantics, proves the theorem. □

References

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, D. Zanardini, Termination analysis of Java bytecode, in: G. Barthe, F.S. de Boer (Eds.), Proceedings of International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS, in: Lecture Notes in Computer Science, vol. 5051, 2008, pp. 2–18.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, COSTA: design and implementation of a cost and termination analyzer for java bytecode, in: Proceedings of International Symposium on Formal Methods for Components and Objects, FMCO, Revised Lectures, in: Lecture Notes in Computer Science, vol. 5382, 2008, pp. 113–132.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost analysis of object-oriented bytecode programs, Theoretical Computer Science 413 (1) (2012) 142–159. (Special Issue on Quantitative Aspects of Programming Languages).

[4] I. Balaban, A. Pnueli, L.D. Zuck, Shape analysis by predicate abstraction, in: Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI, January 2005, pp. 164–180.

[5] I. Balaban, A. Pnueli, L.D. Zuck, Shape analysis of single-parent heaps, in: Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI, January 2007, pp. 91–105.

[6] S. Bardin, A. Finkel, D. Nowak, Toward symbolic verification of programs handling pointers, in: Proceedings of the 3rd International Workshop on Automated Verification of Infinite-State Systems, AVIS'04, April 2004.

[7] J. Berdine, B. Cook, D. Distefano, P. O'Hearn, Automatic termination proofs for programs with shape-shifting heaps, in: Proceedings of International Conference on Computer Aided Verification, CAV, in: Lecture Notes in Computer Science, vol. 4144, 2006, pp. 386–400.

Please cite this article in press as: S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by Abstract Interpretation, Theoretical Computer Science (2013), doi:10.1016/j.tcs.2012.12.018

[8] A. Bossi, M. Gabbrielli, G. Levi, M. Martelli, The *s*-semantics approach: theory and applications, Journal of Logic Programming 19&20 (1994) 149–197.

[9] J. Brotherston, R. Bornat, C. Calcagno, Cyclic proofs of program termination in separation logic, in: Proceedings of ACM Symposium on Principles of Programming Languages, POPL, January 2008, pp. 101–112.

[10] B. Cook, A. Podelski, A. Rybalchenko, Termination proofs for systems code, in: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, June 2006, pp. 415–426.

[11] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of ACM Symposium on Principles of Programming Languages, POPL, January 1977, pp. 238–252.

[12] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Proceedings of ACM Symposium on Principles of Programming Languages, POPL, January 1979, pp. 269–282.

[13] S.K. Debray, N.W. Lin, Cost analysis of logic programs, ACM Transactions on Programming Languages and Systems 15 (5) (1993) 826–875.

[14] S. Genaim, F. Spoto, Constancy analysis, in: Proceedings of Workshop on Formal Techniques for Java-Like Programs, FTfJP, July 2008.

[15] S. Genaim, D. Zanardini, The acyclicity inference of COSTA, in: Proceedings of International Workshop on Termination, WST, July 2010.

[16] S. Genaim, D. Zanardini, Automatic inference of acyclicity, Technical report, 2010.

[17] R. Ghiya, L.J. Hendren, Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c, in: Proceedings of ACM Symposium on Principles of Programming Languages, POPL, January 1996, pp. 1–15.

[18] R. Giacobazzi, E. Quintarelli, Incompleteness, counterexamples and refinements in abstract model-checking, in: Proceedings of the International Static Analysis Symposium, SAS, in: Lecture Notes in Computer Science, vol. 2126, 2001, pp. 356–373.

[19] A. Gotsman, J. Berdine, B. Cook, Interprocedural shape analysis with separated heap abstractions, in: Proceedings of the International Static Analysis Symposium, SAS, in: Lecture Notes in Computer Science, vol. 4134, 2006, pp. 240–260.

[20] S. Graf, H. Saïdi, Construction of abstract state graphs with PVS, in: Proceedings of the International Conference on Computer Aided Verification, CAV, in: Lecture Notes in Computer Science, vol. 1254, 1997, pp. 72–83.

[21] M. Hind, Pointer analysis: haven't we solved this problem yet? in: Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE, June 2001, pp. 54–61.

[22] R. Jones, R. Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons, Inc., New York, NY, USA, 1996.

[23] M. Müller-Olm, D.A. Schmidt, B. Steffen, Model-checking: a tutorial introduction, in: Proceedings of International Static Analysis Symposium, SAS, in: Lecture Notes in Computer Science, vol. 1694, 1999, pp. 330–354.

[24] J.C. Reynolds, Separation logic: a logic for shared mutable data structures, in: Proceedings of the IEEE Symposium on Logic in Computer Science, LICS, July 2002, pp. 55–74.

[25] N. Rinetzky, J. Bauer, T.W. Reps, S. Sagiv, R. Wilhelm, A semantics for procedure local heaps and its abstractions, in: Proceedings of ACM Symposium on Principles of Programming Languages, POPL, January 2005, pp. 296–309.

[26] S. Rossignoli, F. Spoto, Detecting non-cyclicity by abstract compilation into Boolean functions, in: International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'06, in: Lecture Notes in Computer Science, vol. 3855, 2006, pp. 95–110.

[27] S. Sagiv, T.W. Reps, R. Wilhelm, Parametric shape analysis via 3-valued logic, ACM Transactions on Programming Languages and Systems 24 (3) (2002) 217–298.

[28] S. Secci, F. Spoto, Pair-sharing analysis of object-oriented programs, in: Proceedings of International Static Analysis Symposium, SAS, in: Lecture Notes in Computer Science, vol. 3672, 2005, pp. 320–335.

[29] F. Spoto, F. Mesnard, É Payet, A termination analyser for java bytecode based on path-length, Transactions on Programming Languages and Systems 32 (3) (2010) 8:1–8:70.

[30] B. Wegbreit, Mechanical program analysis, Communications of the ACM 18 (9) (1975) 528–539.

[31] R. Wilhelm, S. Sagiv, T.W. Reps, Shape analysis, in: Proceedings of the International Conference on Compiler Construction, CC, in: Lecture Notes in Computer Science, vol. 1781, 2000, pp. 1–17.