The Acyclicity Inference of COSTA

Samir Genaim
DSIC, Complutense University of Madrid (UCM), Spain
Damiano Zanardini
CLIP, DIA, Technical University of Madrid (UPM), Spain

1 Introduction

Programming languages with dynamic memory allocation, such as Java, allow creating and manipulating *cyclic* data structures. The presence of cyclic data structures in the program memory (the *heap*) is a challenging issue in the context of termination analysis [4, 5, 1, 14], resource usage analysis [15, 7, 2], garbage collection [11], etc. Consider the loop "**while** (x!=**null**) **do** x:=x.next;". If x points to an acyclic data structure before the loop, then the *depth* of the data structure to which x points strictly decreases after each iteration; therefore, the number of iterations is bounded by the initial depth of (the structure pointed to by) x.

Automatic inference of such information is typically done by (1) *abstracting* the loop to a numeric loop "$while(x) \leftarrow \{x{>}0, x{>}x'\}, while(x')$"; and (2) bounding the number of iterations of the numeric loop. The numeric loop means that, if the loop entry is reached with x pointing to a data structure with depth $x > 0$, 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 the condition $x > 0$, 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 meaningful for any structure, the latter holds only if x is acyclic. Therefore, acyclicity information is essential in order to apply such abstractions.

In mainstream programming languages with dynamic memory manipulation, data structures can only be modified by means of *field updates*. If, before x.f:=y, x and y are guaranteed to point to disjoint parts of the heap, then there is no possibility to create a cycle. On the other hand, if they are not disjoint, i.e., *share* a common part of the heap, then a cyclic structure might be created. This simple observation has been used in previous work [12] in order to declare x and y, among others, as cyclic whenever they were sharing before the update. Such approach is simple and efficient. However, there can be an important loss of precision in typical programming patterns. E.g., consider "y:=x.next.next;x.next:=y;" (which typically removes an element from a linked list), and let x be initially acyclic. After the first command, x and y clearly share, so that they should be finally declared as 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. There are four possible cases: (1) x and y alias; (2) x reaches y; (3) y reaches x; (4) they both reach a common location. The update x:=y.f might create a cycle only in cases (1) and (3).

This abstract summarizes an acyclicity analysis which is based on the above observation as described in [9]. The analysis has been first developed in [10]; more recent work [9] formalizes it in the theory of *abstract interpretation*, and reports on an implementation for *Java bytecode*. The analysis defines an *abstract domain* $\mathscr{I}^{\tau}_{rc}$ which captures the *reachability* information among program variables (i.e., whether there can be a path in the heap from the location ℓ_v bound to some variable v and the location ℓ_w bound to some w), and the *acyclicity* of data structures (i.e., whether there can be a cyclic path starting from the location bound to some variable). A provably sound *abstract semantics* $\mathscr{C}^{\tau}_{\zeta}[\![\_]\!](\_)$ of a simple object-oriented language is developed, that works on $\mathscr{I}^{\tau}_{rc}$, and can often guarantee the acyclicity of *Directed Acyclic Graphs* (DAGs), which most likely will be considered as cyclic if only sharing, not reachability, is taken into account. The semantics has been implemented in the COSTA [3] COSt and Termination Analyzer as a component whose result is an essential information for proving the termination or inferring the resource usage of programs.

2 The abstract domain

The analysis works on the *reduced product* of two abstract domains. The first domain captures *may-reachability*, while the second deals with the *may-be-cyclic* property of variables. Let μ be a heap, $\ell \in\in \mathrm{dom}(\mu)$ be a location, $\mathscr{L}$ be the set of valid locations, and $\mu(\ell).\mathsf{frm}$ be the set of locations corresponding to the fields of the object located at ℓ The set of *reachable locations* from $\ell \in \mathrm{dom}(\mu)$ is $R(\mu,\ell) = \cup \{R^i(\mu,\ell) \mid i \geq 0\}$, where $R^0(\mu,\ell) = \mathrm{rng}(\mu(\ell).\mathsf{frm}) \cap \mathscr{L}$ (i.e., the locations reachable by directly accessing the fields of ℓ), and $R^{i+1}(\mu,\ell) = \cup \{\mathrm{rng}(\mu(\ell').\mathsf{frm}) \cap \mathscr{L} \mid \ell' \in R^i(\mu,\ell)\}$ (the inductive case). The set of ε-reachable locations from $\ell \in \mathrm{dom}(\mu)$ is $R^\varepsilon(\mu,\ell) = R(\mu,\ell) \cup \{\ell\}$. Note that ε-reachable locations include the source location ℓ itself, while reachable locations do not (unless ℓ is reachable from itself through a cycle). The rest of this section is developed in the context of a type environment τ which specifies the type of variables at a given program point. The set Σ_τ represents the states which are compatible with τ; every state contains a frame ϕ (a function from variables to locations) and a heap μ.

Reachability. Given a state $\sigma = (\phi,\mu) \in \Sigma_\tau$ containing a heap μ, a reference variable v is said to *reach* w in σ if $\phi(w) \in R(\mu,\phi(v))$. This means that, starting from v and applying *at least one dereference operation*, it is possible to reach the object to which w points. Due to strong typing, τ puts some restrictions on reachability; i.e., it might not be possible to have a heap where a variable of type κ_1 reaches one of type κ_2. Following [13], a class $\kappa_2 \in \mathscr{K}$ is said to be *reachable* from $\kappa_1 \in \mathscr{K}$ if there exists $(\phi,\mu) \in \Sigma_\tau$, and two locations $\ell,\ell' \in \mathrm{dom}(\mu)$ s.t. (a) $\mu(\ell).\mathsf{tag} = \kappa_1$ (where $\mu(\ell).\mathsf{tag}$ is the *class tag* of the object located at $\mu(\ell)$); (b) $\mu(\ell').\mathsf{tag} = \kappa_2$; and (c) $\ell' \in R(\mu,\ell)$. The reachability abstract domain is the complete lattice $\mathscr{I}_r^\tau = \langle \wp(\mathscr{R}^\tau), \subseteq, \emptyset, \mathscr{R}^\tau, \cap, \cup \rangle$ where $\mathscr{R}^\tau = \{v \rightsquigarrow w \mid v,w \in \mathrm{dom}(\tau)$ the class $\tau(w)$ is reachable from the class $\tau(v)\}$. The abstraction and concretization functions α_r^τ and γ_r^τ are defined in the standard way. *May-reach* information is described by *abstract values* $I_r \in \wp(\mathscr{R}^\tau)$. 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 $\mathsf{x.f}$ point to the same location. The top element $\mathscr{R}^\tau$ is $\alpha_r^\tau(\Sigma_\tau)$, and represents all states which are compatible with τ. The bottom element $\emptyset$ models the set of all states where, for every two reference variables v and w (possibly the same variable), v does not reach w. Intuitively, reachability is a transitive property; i.e., if x reaches y and y reaches z, then x also reaches z. However, values in $\mathscr{I}_r^\tau$ are *not* closed by transitivity: e.g., it is possible to have $I_r = \{\mathsf{x} \rightsquigarrow \mathsf{y}, \mathsf{y} \rightsquigarrow \mathsf{z}\}$ which contains $\mathsf{x} \rightsquigarrow \mathsf{y}$ and $\mathsf{y} \rightsquigarrow \mathsf{z}$, but not $\mathsf{x} \rightsquigarrow \mathsf{z}$. Such abstract value is a reasonable one, and approximates, for example, the execution of "x=**new** C; y=**new** C; **if** (w>0) **then** $\mathsf{x.f}$=y; **else** $\mathsf{y.f}$=z;".

Cyclicity. Given a state $\sigma = (\phi,\mu) \in \Sigma_\tau$, a variable v is said to be *cyclic* in σ if there exists $\ell \in R^\varepsilon(\mu,\phi(v))$ such that $\ell \in R(\mu,\ell)$. In other words, v is cyclic if it reaches some memory location ℓ (which can possibly be $\phi(v)$ itself) through which a cyclic path goes. The notion of cyclic class is defined similarly to that of reachable classes [12]. The cyclicity domain is the dual of the non-cyclicity domain of [12]. The abstract domain for cyclicity is represented as the complete lattice $\mathscr{I}_c^\tau = \langle \wp(\mathscr{Y}^\tau), \subseteq, \emptyset, \mathscr{Y}^\tau, \cap, \cup \rangle$ where $\mathscr{Y}^\tau = \{\circlearrowleft^v \mid v \in \tau, \tau(v)$ is a cyclic class$\}$. *May-be-cyclic* information is described by *abstract values* $I_c \in \wp(\mathscr{Y}^\tau)$. E.g., $\{\circlearrowleft^{\mathsf{x}}\}$ represents states where no variable but x can be cyclic. The top element $\mathscr{Y}^\tau$ corresponds to Σ_τ; the bottom $\emptyset$ does not allow any variable to be cyclic.

The reduced product. As explained below, the abstract semantics uses reachability information in order to detect cycles, and cyclicity information in order to produce, in some cases, reachability information. Therefore, it makes sense to combine both kinds of information: in Abstract Interpretation, this amounts to computing the *reduced product* [6] of the corresponding abstract domains. In the present

context, the reduced product can be computed by *reducing* the Cartesian product $\mathscr{I}^{\tau}_{rc} = \mathscr{I}^{\tau}_{r} \times \mathscr{I}^{\tau}_{c}$. Elements of $\mathscr{I}^{\tau}_{rc}$ 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 of $\mathscr{I}^{\tau}_{c}$ and $\mathscr{I}^{\tau}_{r}$:

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

However, it can happen that two elements of $\mathscr{I}^{\tau}_{rc}$ are mapped to the same concrete element, which prevents having a Galois insertion between $\mathscr{I}^{\tau}_{rc}$ and the concrete domain $\wp(\Sigma_{\tau})$. Computing the reduced product deals exactly with this problem. In order to compute it, an equivalence relation $\equiv$ has to be defined, which satisfies $I^1_{rc} \equiv I^2_{rc}$ iff $\gamma^{\tau}_{rc}(I^1_{rc}) = \gamma^{\tau}_{rc}(I^2_{rc})$. Functions γ^{τ}_{rc} and α^{τ}_{rc} define a Galois insertion between $\mathscr{I}^{\tau}_{rc\equiv}$ and $\mathscr{I}^{\tau}_{\flat}$, where $\mathscr{I}^{\tau}_{rc\equiv}$ is $\mathscr{I}^{\tau}_{rc}$ equipped (reduced) with the equivalence relation. The equivalence relation can be based on the following observation: For every $I^1_r, I^2_r \in \mathscr{I}^{\tau}_{r}$ and $I^1_c, I^2_c \in \mathscr{I}^{\tau}_{c}$, 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 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 $I_{rc} \in \mathscr{I}^{\tau}_{rc}$ results in representing a larger set of states; and (b) adding a pair $v \rightsquigarrow v$ to $I_{rc} \in \mathscr{I}^{\tau}_{rc}$, when v cannot be cyclic, does not make it represent more concrete states, since the acyclicity of v excludes that it can reach itself. From now on, $\mathscr{I}^{\tau}_{rc}$ will be a shorthand for $\mathscr{I}^{\tau}_{rc\equiv}$, where $\equiv$ is left implicit.

Denotational semantics. Abstract denotations for expressions and commands are depicted in Fig. 1. *Possible sharing*, *possible aliasing* and *purity* analysis are used as pre-existent components, i.e., programs are assumed to have been analyzed w.r.t. these properties. Two reference variables v and w *share* in $(\phi/heap)$ iff $R^{\varepsilon}(\mu, \phi(v)) \cap R^{\varepsilon}(\mu, \phi(w)) \neq \emptyset$; also, they *alias* if they point to the same location, namely, if $\phi(v) = \phi(w) \in \mathrm{dom}(\mu)$. The i-th argument of a method m is said to be *pure* if m does not update the data structure to which the argument initially points. For *sharing* and *purity*, the analysis described in [8] (based on [13]) is applied: with it, (1) it is possible to know if v might share with w at any program point (denoted by the pair $\langle v \bullet w \rangle$); and (2) for each method m, a denotation SH_{m} is given: for a set of pairs sh which safely describes the sharing between actual arguments in the input state, $sh' = \mathsf{SH}_{\mathsf{m}}(I)$ is such that (i) if $\langle v \bullet w \rangle \in sh'$, then v and w might share during the execution of m; and (ii) $\dot{v}_i \in sh'$ means that the i-th argument might be non-pure. As for *aliasing*, it is assumed that, at each program point, the pair $\langle v \cdot w \rangle$ tells if v and w can alias. Importantly, any non-null reference variable shares and aliases with itself; also, both are symmetric relations (i.e., $\langle v \bullet w \rangle$ iff $\langle w \bullet v \rangle$, and $\langle v \cdot w \rangle$ iff $\langle w \cdot v \rangle$). An abstract element $\langle I_r, I_c \rangle \in \mathscr{I}^{\tau}_{rc}$ will be represented by the set $I = I_r \cup I_c$; therefore, $v \rightsquigarrow w \in I$ and $\circlearrowleft^v \in I$ are shorthands for, resp., $v \rightsquigarrow w \in I_r$ and $\circlearrowleft^v \in I_c$. The operation $\exists v.I$ (*projection*) removes any statement about v from I, while $I[v/w]$ (*renaming*) v to w in I. For the sake of simplicity, class-reachability and class-cyclicity are taken into account *implicitly*: a new statement $v \rightsquigarrow w$ (resp., $\circlearrowleft^v$) is not added to an abstract state if $v \rightsquigarrow w \notin \mathscr{R}^{\tau}$ (resp., $\circlearrowleft^v \notin \mathscr{Y}^{\tau}$). The abstract semantics has been proven to be sound; i.e., (1) whenever v reaches w in a concrete state σ at a given program point, the statement $v \rightsquigarrow w$ is included in the abstract description I of σ at the same program point; and (2) whenever v is cyclic in σ, $\circlearrowleft^v$ must be present in I. For a detailed explanation of the abstract semantics and the proof of its correctness, the reader can refer to [9].

3 The analysis by examples

This section explains the behavior of the abstract semantics on a couple of interesting examples. The semantics instruments the program code with abstract values I_n, where n is the line number. A statement $v \rightsquigarrow w \in I_n$ means that v could reach w at line n, while $\circlearrowleft^v$ means that v could be cyclic.

Consider the class OrderedList depicted in Fig. 2. It implements an *ordered linked list* where head points to the first element, and lastInserted points to the last element which has been inserted. The class

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

Figure 1: Abstract denotations for expressions and commands

Node (not shown) implements a linked list in the usual way. The method insert adds a new element to the ordered list: it takes an integer i, creates a new node n for i, looks for its position, adds it to the list, and makes lastInserted point to the new node. Suppose insert appears inside a loop (for example, when inserting into the ordered list elements which are stored in an array). The goal is to infer that a call "x.insert(i)" never makes x cyclic. This is important since, if x cannot be proven to be acyclic after insert, then it must be assumed to be cyclic from the second iteration of the loop on. This, in turn, prevents from proving the termination of the loop at lines 10–12, since it might traverse a cycle. The challenge in this example is to prove that the instructions at lines 16 and 17 do not make any data structure cyclic. This is not trivial since **this**, p, and n share between each other at line 15; depending on how they share, the corresponding data structures might become cyclic or remain acyclic. Consider line 17: 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 17, but n does not reach p, which, in turn, guarantees that no data structure ever becomes cyclic (as evident from the absence of any cyclicity statement $\circlearrowleft^v$). Note that reachability information is essential for proving acyclicity, since the mere information that p and n share, without knowing how they do, requires to consider then as possibly cyclic, as done in [12].

As another example, consider the method mirror in Fig. 2, and suppose class Tree implements a binary tree in the standard way, with fields left and right. The call mirror(t) exchanges the values of left and right of each node in t. An initial state $\emptyset$ is transformed by mirror as follows. 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 initially mirror ha a denotation

```
1  class OrderedList {
2   Node head, lastInserted;
3
4   void insert(int i) {
5     Node c,p,n;
6                          //I6=∅
7     n:=new Node;         //I7=∅
8     n.value:=i;          //I8=∅
9     c:=this.head;        //I9={this⇝c}
10    while (c!=null && c.value<i) do
11        p:=c;            //I11={this⇝c,this⇝p}
12        c:=c.next;       //I12={this⇝c,this⇝p,p⇝c}
13                         //I13={this⇝c,this⇝p,p⇝c}
14    n.next:=c;           //I14={this⇝c,this⇝p,p⇝c,n⇝c}
15    if (p=null) then
16      this.head:=n;      //I16=I15∪{this⇝n}
17    else p.next:=n;      //I17=I15∪{this⇝n,p⇝n}
18                                 //I18=I16∪I17=I17
19    this.lastInserted:=n;//I19=I18
20  }
21 }
```

```
1  void 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     mirror(l);
12     mirror(r);
13 }
1  Node connect() {
2    Node c=this;
3
4    while (c.next!=null) do
5       c:=c.next;
6    c.next:=this;
7    return c;
8  }
```

Figure 2: The running example and the analysis results (in comments).

$\emptyset$) 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.

Finally, consider the method connect, 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 2 does not add any statements, while line 5 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 6 adds $\{curr\rightsquigarrow this, curr\rightsquigarrow curr, this\rightsquigarrow this\}$, and $\{\circlearrowleft^{curr}, \circlearrowleft^{this}\}$. Finally, line 7 clones *curr* to *out*. In conclusion, the analysis correctly infers that l.connect() makes both l and the return value cyclic (i.e., statements $\circlearrowleft^{l}$ and $\circlearrowleft^{out}$ are produced).

The result of the abstract semantics for the above examples has been taken from the outcome of the implementation in the COSTA [3] COSt and Termination Analyzer on the Java bytecode version of these programs. The acyclicity analysis is a component of the system which is used in order to help to infer statically the termination of a program or information about its resource consumption. For example, suppose that, as pointed out above, insert is called inside a Java-style loop like

```
1     for (i=0; i<n; i++) { l.insert(a[i]); }
```

where the content of an array is copied to the ordered list. If l cannot be proven to be acyclic on exit from insert, then the list must be considered as possibly cyclic when entering the **for** loop the second time. This implies that the loop at line 10–12 in the code of insert could be non-terminating from the second iteration of the **for**. On the other hand, COSTA can prove that this code is terminating since the acyclicity analysis guarantees that the list is always acyclic, so that every iteration of the **for** loop terminates.

References

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *FMOODS*, LNCS 5051, pages 2–18, 2008.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, LNCS 4421, pages 157–172. Springer, 2007.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *FMCO'07*, number 5382 in LNCS, pages 113–133. Springer, 2008.

[4] J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proc. CAV*, 2006.

[5] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

[6] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *POPL'79*, pages 269–282. ACM, 1979.

[7] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *ACM TOPLAS*, 15(5):826–875, November 1993.

[8] S. Genaim and F. Spoto. Constancy analysis. In *10th Workshop on Formal Techniques for Java-like Programs*, July 2008.

[9] S. Genaim and D. Zanardini. The acyclicity inference of COSTA. In *11th International Workshop on Termination*, July 2010.

[10] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL*, pages 1–15, 1996.

[11] R. Jones and R. Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[12] S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *VMCAI*, LNCS 3855. Springer, 2006.

[13] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *SAS*, number 3672 in LNCS, pages 320–335, 2005.

[14] F. Spoto, F. Mesnard, and É. Payet. A Termination Analyser for Java Bytecode based on Path-Length. *ACM TOPLAS*, 32(3), 2010.

[15] B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9), 1975.