Field-Sensitive Unreachability and Non-Cyclicity Analysis

Enrico Scapin and Fausto Spoto

Department of Computer Science, University of Verona, Italy
scapin@uni-trier.de, fausto.spoto@univr.it

Abstract

Field-sensitive static analyses of object-oriented code use approximations of the computational states where fields are taken into account, for better precision. This article presents a novel and sound *definite* analysis of Java bytecode that states two strictly related properties: field-sensitive unreachability between program variables and field-sensitive non-cyclicity of program variables. The latter exploits the former for better precision. We build a *data-flow* analysis based on *constraint graphs*, whose nodes are program points and whose arcs propagate information according to the semantics of each bytecode instruction. We follow *abstract interpretation* both to approximate the concrete semantics and to prove our results formally correct. Our analysis has been designed with the goal of improving client analyses such as *termination analysis*, asserting the non-cyclicity of variables *w.r.t.* specific fields.

Keywords: Static Analysis, Data-Flow Analysis, Constraint-Based Analysis, Field-Sensitive Analysis, Abstract Interpretation.

1 Introduction

Static analysis builds compile-time approximations of the set of values, states or behaviours arising dynamically, at run-time *i.e.*, during the execution of a computer program. This is important to improve the quality of software by detecting illegal operations, such as divisions by zero or dereferences of `null`, erroneous executions, such as infinite loops, or security flaws, such as unwanted disclosure of information. In order to make static analysis computable, we follow *abstract interpretation* [1] here, a framework that lets one define a static analysis from the formal specification of the property of interest and of the semantics of the language.

In modern object-oriented languages such as Java, a typical problem related to the verification of real, large software programs is how the dynamic allocation of objects shapes the heap: namely, objects can be instantiated on demand and reference other objects through *fields*, that can be updated at run-time. There are several articles in literature describing *memory*-related properties and providing *pointer analyses* that statically determine approximations of the possible run-time values of a pointer [3]. *Shape analysis* [10] builds the possible shapes that data structures might assume at run-time; *aliasing analysis* [6] determines which variables point to the same location; *sharing analysis* [13] infers which variables are bound to overlapping data structures; *reachability analysis* [7] looks for paths between locations and *non-cyclicity analysis* [9] spots variables bound to non-cyclical data. In this context, we present here a *definite* data-flow analysis for field-sensitive unreachability and non-cyclicity. Namely, we build an *under-approximation* of the program fields that are not used in any path between two variables or in a cycle bound to a variable, respectively. Under-approximations in the context of abstract interpretation have been studied in [12] through predicate transformers where the abstract transition function is a sound postcondition transformer of the state-transition function. A field-sensitive pointer analysis has been developed in [8], with a constraint-based approach as ours but not for object-oriented languages with dynamic memory allocation; instead, C and fields of structures are considered. Furthermore they extended a set constraint language and an inference system to model each field as a separate variable. Here instead, unreachability and non-cyclicity specify which fields cannot be used to establish the property. The work most related to ours is [2], that introduces an acyclicity analysis as the reduced product of abstract domains for reachability and cyclicity, over a semantics similar to ours. They highlight that cyclicity supports reachability *i.e.*, one can exploit unreachability information to improve non-cyclicity analysis. The main difference with our work is that we compute the fields not involved in reachability or cyclicity, getting higher precision. Furthermore, we have provided formal correctness proofs for the propagation rules of each bytecode instruction and method call, including its side-effects (see [11]).

Our analysis is designed with the goal of improving client analyses of the Julia analyzer for Java and Android bytecode (www.juliasoft.com). Namely, its *termination checker* finds method calls that might diverge at run-time, through the *path-length* property [15] *i.e.*, an estimation of the maximal length of a path of pointers rooted at each given program variable. For the Java instruction x = x.next, Julia estimates the path-length of x; in the original definition, it is decreasing only if it is possible to assert the non-cyclicity of x. With the analysis of this article, we can now assert it more precisely, by considering the accessed field: the path-length decreases if next belongs to the set of non-cyclical fields F_{x} for variable x.

2 Operational Semantics

We present here a formal operational semantics of Java bytecode, inspired by the standard informal semantics [4]. It has been first introduced in [14] and more widely explained in [11]. Java bytecode are the instructions executed by the Java Virtual Machine (JVM). Our formalization is at bytecode level for reasons already highlighted in [5]: it is more faithful, as it analyzes code that is actually executed; it enables the analysis of programs whose source code is not available; it lacks complexities such as inner classes and name resolution; the analyzer can be applied to all the many programming languages that compile to the JVM.

```
class Element{
 private Object value;
 private Element prec, next;

 public Element(Object value){
  this.value=value;
 }
 public Element(Object value,
                 Element prec){
  this.value=value;
  this.prec=prec;
  prec.next=this;
 }
```

Fig. 1: Our example

For simplicity, we assume that the only primitive type is int and reference types are *classes* with *instance fields and methods* only. Julia handles other Java types, fields and methods. We analyze bytecode preprocessed into a Control Flow Graph (CFG), a directed graph of *basic blocks*, with no jumps inside the blocks. Fig. 2 shows it for the second constructor from Fig. 1. Exception handlers start at a catch. A conditional, virtual method call, or selection of an exception handler is a block with many subsequent blocks, starting with *filtering* bytecodes such as excp_is K for exception handlers.

Definition 2.1 [Classes, Type environment, State] The set of *classes* $\mathbb{K}$ of a program is partially ordered *w.r.t.* the *subclass relation* $\leq$. A *type* is an element of $\mathbb{T} = \{\mathsf{int}\} \cup \mathbb{K}$, ordered by the extension of $\leq$ with $\mathsf{int} \leq \mathsf{int}$. A class $\kappa \in \mathbb{K}$ has *fields* $\kappa.f : \mathsf{t}$ *i.e.*, field f of type $\mathsf{t} \in \mathbb{T}$ defined in κ. By letting $\mathbb{F}(\kappa) = \{\kappa'.f : \mathsf{t}' \mid \kappa \leq \kappa'\}$ be the fields defined in κ or in any of its superclasses, we define the set of all fields $\mathcal{F} = \bigcup_{\kappa \in \mathbb{K}} \mathbb{F}(\kappa)$. A class κ has *methods* $\kappa.m(\mathbf{t}) : \mathsf{t}$ (method m, defined in κ, with arguments of type $\mathbf{t}$, returning a value of type $\mathsf{t} \in \mathbb{T} \cup \{\mathsf{void}\}$).

V is the set of *variables*, divided in $L = \{l_0, \ldots, l_m\}$ (*local* variables) and $S = \{s_0, \ldots, s_n\}$ (*stack* variables). A *type environment* is a function $\tau : V \to \mathbb{T}$, whose *domain* is written $\mathsf{dom}(\tau)$. The set of all type environments is $\mathcal{T}$. A *value* is an element of $\mathbb{Z} \cup \mathbb{L} \cup \{\mathtt{null}\}$, where $\mathbb{L}$ is an infinite set of *memory locations*. A *state* over $\tau \in \mathcal{T}$ is a pair $\langle\langle l \,\|\, s\rangle, \mu\rangle$: l is an array of values for the local variables in $\mathsf{dom}(\tau)$; s is a stack of values for the stack variables in $\mathsf{dom}(\tau)$, that grows leftwards; μ is a *memory* that binds locations to *objects*. We often use another representation: $\langle\rho, \mu\rangle$, where an *environment* ρ maps each $l_k \in L$ to its value $l[k]$ and each $s_k \in S$ to its value $s[k]$. An object o has class $o.\kappa$ and an internal environment $o.\phi$ that maps every field $\kappa'.f : \mathsf{t}'$ into its value $(o.\phi)(\kappa'.f : \mathsf{t}')$. The set of states is Ξ. We write Ξ_τ when we want to fix the type environment τ.

The semantics of an instruction ins is a partial map *ins* : $\Sigma_\tau \to \Sigma_{\tau'}$ from *initial* to *final* states. Number and type of local variables and stack elements at its start are specified by τ. The denotational semantics of each bytecode instruction and the transition rules of our Java bytecode small-step operational semantics are given and explained in [11].

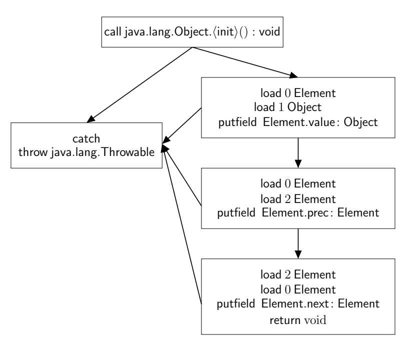

Fig. 2: A CFG

3 Field-Sensitive Properties

We formalize here field-sensitive *unreachability* and *non-cyclicity* [11].

Definition 3.1 [Locations reachable from a variable [7]] Let $\tau \in \mathcal{T}$. The set of *locations reachable from* $a \in \mathsf{dom}(\tau)$ *in a state* $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ is $\mathsf{L}_\sigma(a) = \bigcup_{i \geq 0} \mathsf{L}^i_\sigma(a)$, where $\mathsf{L}^i_\sigma(a)$ are the locations reachable from a in at most i steps: $\mathsf{L}^i_\sigma(a) = \{\rho(a)\} \cap \mathbb{L}$ if $i = 0$, and $\mathsf{L}^i_\sigma(a) = \mathsf{L}^{i-1}_\sigma(a) \cup \bigcup_{\ell \in \mathsf{L}^{i-1}_\sigma(a)} (\mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L})$ if $i > 0$.

Definition 3.2 [Path between Variables] Let $\tau \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$, $a, b \in \mathsf{dom}(\tau)$ and $\rho(a), \rho(b) \in \mathsf{dom}(\mu) \subseteq \mathbb{L}$. We define a path $\mathcal{P}$ from a to b in σ as an n-tuple $\langle \kappa_1.f_1 : \mathsf{t}_1, \ldots, \kappa_n.f_n : \mathsf{t}_n \rangle \subseteq \mathcal{F}$ such that $\exists \ell^1, \ldots, \ell^{n+1} \in \mathsf{dom}(\mu).\ \ell^1 = \rho(a), \ell^{n+1} = \rho(b) \wedge \forall i = 1, \ldots, n.\ (\mu(\ell^i).\phi)(\kappa_i.f_i : \mathsf{t}_i) = \ell^{i+1}$. We denote it by $a \rightsquigarrow^{\mathcal{P}}_\sigma b$.

Hence, a path from a to b is a tuple of fields starting at location $\rho(a)$ and reaching location $\rho(b)$ by following the fields in the tuple.

Definition 3.3 [Field-Sensitive Unreachability among Variables] Let $\tau \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$, $F \subseteq \mathcal{F}$ and $a, b \in \mathsf{dom}(\tau)$. If, for each path from a to b the fields in F are not in the path, *i.e.*, if $\forall \mathcal{P} \subseteq \mathcal{F} \left(a \rightsquigarrow^{\mathcal{P}}_\sigma b \Longrightarrow \mathcal{P} \cap F = \emptyset \right)$, then we write $a \not\rightsquigarrow^F_\sigma b$.

Definition 3.4 [Field-Sensitive Non-Cyclicity] Let $\tau \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$, $F \subseteq \mathcal{F}$ and $a \in \mathsf{dom}(\tau)$. If, for each cycle reachable from a, the fields in F are not in the cycle, *i.e.*, if $\forall \ell \in \mathsf{L}_\sigma(a), \forall \mathcal{P} \subseteq \mathcal{F} \left(\ell \rightsquigarrow^{\mathcal{P}}_\mu \ell \Rightarrow \mathcal{P} \cap F = \emptyset \right)$, then we write $a \rightsquigarrow^{\not\circlearrowleft F}_\sigma$.

4 Constraint-based Fields-Sensitive Analysis

We define here an abstract interpretation of the concrete semantics introduced in Section 2 *w.r.t.* the two properties introduced in Section 3. This will be an actual static analysis algorithm for interprocedural, whole-program analysis.

Definition 4.1 [Concrete and Abstract Domain] Given a type environment $\tau \in \mathcal{T}$, we define the *concrete lattice* over τ as $\mathsf{C}_\tau = \langle \wp(\Sigma_\tau), \subseteq \rangle$ and the *abstract lattice* over τ as $\mathsf{A}_\tau = \langle \mathsf{UR}_\tau \cup \mathsf{NC}_\tau, \supseteq \rangle$ *i.e.*, the union of two sets: $\mathsf{UR}_\tau = \wp(\mathsf{dom}(\tau) \times \mathsf{dom}(\tau) \times \wp(\mathcal{F}))$ and $\mathsf{NC}_\tau = \wp(\mathsf{dom}(\tau) \times \wp(\mathcal{F}))$. The former is the powerset of the product between the set of ordered pairs of variables $v, w \in \mathsf{dom}(\tau)$ and the powerset of the program fields $F \subseteq \mathcal{F}$. Its elements are written as $v \not\rightsquigarrow^F w$. The latter is the powerset of the product between the set of variables $v \in \mathsf{dom}(\tau)$ and the powerset of the program fields $F \subseteq \mathcal{F}$. Its elements are written as $v \rightsquigarrow^{\not\circlearrowleft F}$.

An abstract element $I \in \mathsf{A}_\tau$ represents those concrete states in Σ_τ whose unreachability and non-cyclicity information is *under-approximated* by the tokens in I. Thus, we induce a *definite* unreachability and non-cyclicity analysis *w.r.t.* an *under-approximation* of the set of program fields.

Definition 4.2 [Concretization map] Let $\tau \in \mathcal{T}$ and $I \in \mathsf{A}_\tau$. We define the *concretization map* $\gamma_\tau : \mathsf{A}_\tau \to \mathsf{C}_\tau$ in such a way that $\gamma_\tau(I)$ is

$$\left\{\sigma \in \Sigma_\tau \,\middle|\, \left(\forall a \not\rightsquigarrow^F b \in I, \exists F' \subseteq \mathcal{F}. a \not\rightsquigarrow_\sigma^{F'} b \wedge F \subseteq F'\right) \wedge \left(\forall c \rightsquigarrow^{\not\circlearrowleft F} \in I, \exists F' \subseteq \mathcal{F}. c \rightsquigarrow_\sigma^{\not\circlearrowleft F'} \wedge F \subseteq F'\right)\right\}$$

This map is co-additive in [11] and hence A_τ and C_τ are an abstract and concrete domain and γ_τ is the concretization map of a *Galois connection* [1] between them.

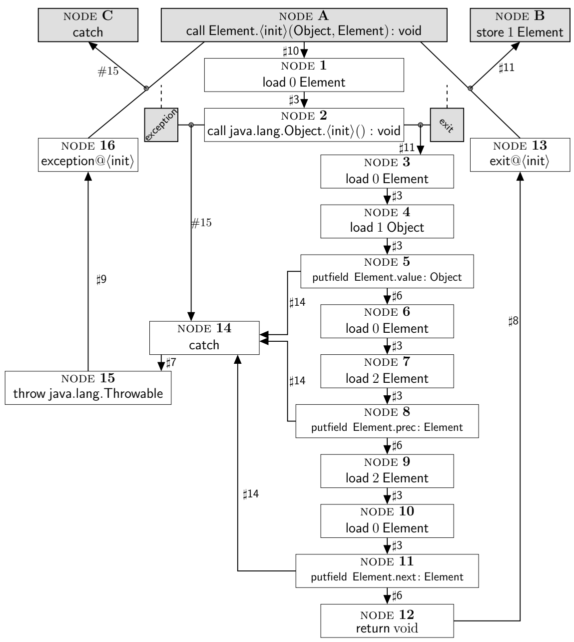

Fig. 3: The ACG of the CFG in Fig. 2

Our analysis is *constraint-based*, in the sense that it builds an Abstract Constraint Graph (ACG, see Figure 3) from the program under analysis.

Definition 4.3 [ACG] Let P be the program under analysis (*i.e.*, a control flow graph of basic blocks for each method or constructor). The *Abstract Constraint Graph* (ACG) of P is a directed graph $\langle V, E\rangle$ (nodes, arcs) where: *i)* V contains a node $\boxed{\mathsf{ins}}$, for every bytecode instruction ins of P; *ii)* V contains nodes $\boxed{\mathsf{exit}@m}$ and $\boxed{\mathsf{exception}@m}$ for each method or constructor m in P, and these nodes correspond to the normal and exceptional end of m; *iii)* E contains directed arcs from a source to a sink, reflecting, in abstract terms, the effects of the concrete semantics over the unreachability and non-cyclicity information; *iv)* for every arc in E, there is a *propagation rule* $\Pi^{\#i}$, *i.e.*, a function over A_τ, from the information at its source(s) to the information at its sink. Its exact definition depends on ins, since each bytecode instruction has different effects on unreachability and non-cyclicity. The arcs in E are built from P as follows. We assume that τ and τ' are the static type information at and immediately after the execution of a bytecode ins, respectively. Furthermore, we assume that τ contains j stack elements and i local variables. We define the propagation rules as a tuple of functions $\{\Pi^{\#1}, \ldots, \Pi^{\#n}\}$, where, if i identifies the Java bytecode instructions that we consider, then $\Pi^{\#i}$ is the propagation rule for that instruction. [11] discusses different types of arcs depending on the bytecode instructions that they link and defines each propagation rule: sequential arcs, final arcs, parameter passing arcs, exceptional arcs, side-effects arcs and return value arcs.

Each propagation rule Π is proved formally correct in [11] by using the standard technique of abstract interpretation: namely, by letting *ins* be a bytecode instruction, Π a propagation rule and $I \in \mathsf{A}_\tau$, we have proved the soundness of Π by showing that $ins(\gamma_\tau(I)) \subseteq \gamma_\tau(\Pi(I))$ *i.e.*, that for each $x \not\rightsquigarrow^F y \in \Pi(I)$ we have a state σ' such that $x \not\rightsquigarrow^{F'}_{\sigma'} y \wedge F \subseteq F'$ and for each $z \rightsquigarrow \not\circlearrowleft^F \in \Pi(I)$ we have another state σ' such that $z \rightsquigarrow_{\sigma'} \not\circlearrowleft^{F'} y \wedge F \subseteq F'$.

$$
\begin{aligned}
\lambda I.&\left\{ a \not\rightsquigarrow^F b \in I \,\middle|\, \begin{array}{l} a, b \notin \{s_{j-1}, s_{j-2}\} \wedge \\ (\langle a, s_{j-2}\rangle \notin \mathcal{MR}_\tau \vee \langle s_{j-1}, b\rangle \notin \mathcal{MR}_\tau) \end{array} \right\} \\
\cup &\left\{ a \not\rightsquigarrow^{F'} b \,\middle|\, \begin{array}{l} 1.\ a, b \notin \{s_{j-1}, s_{j-2}\} \wedge \langle a, s_{j-2}\rangle, \langle s_{j-1}, b\rangle \in \mathcal{MR}_\tau \\ 2.\ a \not\rightsquigarrow^{F_{ab}} b,\ a \not\rightsquigarrow^{F_{a2}} s_{j-2},\ s_{j-1} \not\rightsquigarrow^{F_{1b}} b \in I \\ 3.\ F' = \{F_{ab} \cap F_{a2} \cap F_{1b}\} \setminus \{\kappa.f : \mathsf{t}\} \end{array} \right\} \\
\cup &\left\{ c \rightsquigarrow \not\circlearrowleft^F \in I \,\middle|\, \begin{array}{l} c \notin \{s_{j-1}, s_{j-2}\} \wedge \\ \langle c, s_{j-1}\rangle, \langle c, s_{j-2}\rangle \notin \mathcal{MR}_\tau \end{array} \right\} \\
\cup &\left\{ c \rightsquigarrow \not\circlearrowleft^{F'} \,\middle|\, \begin{array}{l} 1.\ c \notin \{s_{j-1}, s_{j-2}\} \wedge \langle s_{j-1}, s_{j-2}\rangle \notin \mathcal{MR}_\tau \\ 2.\ \langle c, s_{j-2}\rangle \in \mathcal{MR}_\tau \vee \langle c, s_{j-1}\rangle \in \mathcal{MR}_\tau \\ 3.\ c \rightsquigarrow \not\circlearrowleft^{F_c},\ s_{j-1} \rightsquigarrow \not\circlearrowleft^{F_{j1}} \in I \wedge F' = F_c \cap F_{j1} \end{array} \right\} \\
\cup &\left\{ c \rightsquigarrow \not\circlearrowleft^{F'} \,\middle|\, \begin{array}{l} 1.\ c \notin \{s_{j-1}, s_{j-2}\} \wedge \langle s_{j-1}, s_{j-2}\rangle \in \mathcal{MR}_\tau \\ 2.\ \langle c, s_{j-1}\rangle \in \mathcal{MR}_\tau \vee \langle c, s_{j-2}\rangle \in \mathcal{MR}_\tau \\ 3.\ s_{j-1} \not\rightsquigarrow^{F_{12}} s_{j-2},\ c \rightsquigarrow \not\circlearrowleft^{F_c},\ s_{j-1} \rightsquigarrow \not\circlearrowleft^{F_{j1}} \in I \\ 4.\ F' = \{F_c \cap F_{j1} \cap F_{12}\} \setminus \{\kappa.f : \mathsf{t}\} \end{array} \right\}
\end{aligned}
$$

Fig. 4: The prop. rule for putfield

We report only the rule related to the putfield (Figure 4) since its execution changes dynamically the paths between locations and is hence significant for our heap-related properties. We note that we have heavily used the result of a preliminary *reachability* static analysis among pairs of variables to improve the precision: the set $\mathcal{MR}_\tau$ contains all pairs of variables such that the former might reach the latter at the putfield. The rule, starting from a node corresponding to putfield, states, in the first and third sets, that unreachability and non-cyclicity information of the pairs of variables that do not reach or are not reached by the topmost two values of the stack remains unchanged. Instead, if a pair of variables $\langle a, b\rangle$ is such that the former may reach s_{j-2} and the latter may be reached from s_{j-1}, then the putfield might create new paths between the two locations bound to these two variables. Hence the set of fields that define the unreachability information between a and b must be consistent both with the unreachability information of $a \not\rightsquigarrow^{F_{a2}} s_{j-2}$ and $s_{j-1} \not\rightsquigarrow^{F_{1b}} b$. Furthermore, since putfield links s_{j-2} with s_{j-1} through $\kappa.f : \mathsf{t}$, that field must be deleted from any set of unreachability information of pairs of variables such as $\langle a, b\rangle$, as shown in the second set. For non-cyclicity, we distinguish if s_{j-1} may reach s_{j-2} or not *i.e.*, if the putfield will create or not a new cycle in memory between the locations. If s_{j-1} may not reach s_{j-2}, then no new cycle is created. However, since s_{j-2} gets linked to s_{j-1}, all variables that may reach s_{j-2} will be able to reach s_{j-1} and hence their non-cyclicity information F_c must take into account also the non-cyclicity information of s_{j-1}, as shown in the fourth set. On the other hand, if s_{j-1} may reach s_{j-2}, then a new cycle could be created. Thus we delete the fields that might belong to that cycle: this information is provided by unreachability since we know which fields are not in a path from s_{j-1} to s_{j-2} *i.e.*, F_{12} such that $s_{j-1} \not\rightsquigarrow^{F_{12}} s_{j-2} \in I$. The resulting set is then: $F' = \{F_c \cap F_{j1} \cap F_{12}\} \setminus \{\kappa.f : \mathsf{t}\}$. Hence, unreachability information between two variables is crucial in order to correctly assert the non-cyclicity property.

Example 4.4 Consider nodes 11, 12 from Figure 3, and suppose that the unreachability and non-cyclicity information at node 11 is

$$I_{11} = \{l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, s_1\}, l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1, l_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} \{l_2, s_0\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\},$$
$$s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2, s_0, s_1\}, s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, s_1\}, s_1 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1, s_1 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} \{l_2, s_0\}, \{l_0, l_1, l_2, s_0, s_1\} \rightsquigarrow^{\not\circlearrowleft \mathcal{F}}\}$$

where $l_0 = s_1 =$ this, $l_1 =$ `value`, $l_2 = s_0 =$ `prec` and $\mathcal{F} = \{\mathsf{El}.value, \mathsf{El}.prec, \mathsf{El}.next\}$: El is an abbreviation for Element. In order to build I_{12}, we apply the propagation rule for putfield, since s_0 is linked to s_1 by the field $\mathsf{El}.next$. That application leads to the set:

$$I_{12} = \{l_0 \not\rightsquigarrow^{\mathcal{F}} l_0, l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec, \mathsf{El}.next\}} l_1, l_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.next\}} l_2, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\},$$
$$l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_1, l_2\}, l_2 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.prec\}} l_0, l_1 \rightsquigarrow^{\not\circlearrowleft \mathcal{F}}, \{l_0, l_2\} \rightsquigarrow^{\not\circlearrowleft \{\mathsf{El}.value\}}\}$$

We note that field $\mathsf{El}.next$ is no more inside the set of fields associated to the pair $\langle l_2, l_0\rangle$. This is because $\langle l_2, s_0\rangle, \langle s_1, l_0\rangle \in \mathcal{MR}_{\tau_{11}}$ and hence their set of fields changes according to the second set of Figure 4: $F_{l_2,l_0} = \{F_{l_2,l_0} \cap F_{l_2,s_0} \cap F_{s_1,l_0}\} \setminus \{\mathsf{El}.next\} = \{\mathsf{El}.value, \mathsf{El}.prec\}$. Moreover, also the non-cyclicity tokens for l_0 and l_2 changed, because a new cycle might be formed by executing this putfield. Indeed, since $\langle l_2, s_0\rangle, \langle l_0, s_1\rangle, \langle s_1, s_0\rangle \in \mathcal{MR}_{\tau_{11}}$, the rule modifies the non-cyclicity information of l_2 and l_0 and, hence, both the operation and the result are the same. We show it only for l_2, the latter being the same: $F_{l_2} = \{F_{l_2} \cap F_{l_0} \cap F_{s_1,s_0}\} \setminus \{\mathsf{El}.next\} = \{\mathsf{El}.value\}$. We note that, as shown in Figure 1, during the execution of the second constructor a new cycle between the two location bound to this and prec is actually built since this is linked to prec through field $\mathsf{El}.prec$, while prec is linked to this through field $\mathsf{El}.next$. Hence fields $\mathsf{El}.prec$ and $\mathsf{El}.next$ set up a path $\mathcal{P} = \{\mathsf{El}.prec, \mathsf{El}.next\}$ such that $\rho(\mathsf{this}) \rightsquigarrow^{\mathcal{P}} \rho(\mathsf{this})$ and, respectively, $\rho(\mathsf{prec}) \rightsquigarrow^{\mathcal{P}} \rho(\mathsf{prec})$. According to Definition 3.4, they must not be in $F_{\mathsf{this}} = F_{l_0}$ nor in $F_{\mathsf{prec}} = F_{l_2}$.

Definition 4.5 [Field-Sensitive Unreachability and Non-Cyclicity Analysis] A solution of an ACG is an assignment of an element $J_n \in \mathsf{A}_\tau$ to each node n of the ACG, where τ is the type environment associated to n, such that the propagation rules Π of the arcs is satisfied *i.e.*, for every arc from $n_1, \ldots, n_k$ to n' the condition $\Pi(J_{n_1}, \ldots, J_{n_k}) \supseteq J_{n'}$ holds. The unreachability and non-cyclicity analysis of the program is the *maximal solution i.e.*, the maximal *fixpoint*, of its ACG *w.r.t.* $\supseteq$.

In [11] we have also proved the soundness of the whole analysis *w.r.t.* the small-step operational semantics for Java bytecode previously explained: given an execution leading to a basic block $\boxed{ins}$ in a state σ and letting I_{ins} be the approximation of our properties at the correspondent node, we have proved that $\sigma \in \gamma(I_{\mathsf{ins}})$.

5 Conclusion

We have introduced and formalized a provably sound constraint-based field-sensitive unreachability and non-cyclicity analysis for Java bytecode. The work more similar similar to ours is the static analysis introduced in [2]. The main differences are that we provide specific information about the set of fields that cannot be used for reachability or cyclicity and that we deal directly with low-level Java bytecode, whereas they use a high-level language (pros and cons are explained in [5]). A conclusion of our investigation is that, in order to achieve a precise analysis for low-level code, we need finer domains and pre-processed information deriving from other static analyses. In this analysis we do not see any advantage with the use of a high-level representation instead of bytecode, since the main problem are field updates and the propagation of side effects, that are equally difficult to analyze, for both low and high level languages. Our domain is a refinement of reachability and non-cyclicity as introduced in [7] and [9], respectively; on the other hand, we exploit pre-processed information for possible sharing, possible reachability ($\mathcal{MR}_\tau$) and definite aliasing analyses, for better precision.

References

[1] Cousot, P. and Cousot, R., *Abstract interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '77).

[2] *Genaim, S. and Zanardini D.*, *Reachability-based Acyclicity Analysis by Abstract Interpretation*, In: The Computing Research Repository (CoRR'12).

[3] Hind, M., *Pointer Analysis: Haven't We Solved This Problem Yet?*, In: Proceedings of the 1st Program Analysis for Software Tools and Engineering (PASTE'01).

[4] Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification*, Java series. Addison- Wesley, 1999.

[5] Logozzo, F. and Fähndrich, M., *On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis*, In: Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction (CC'08/ETAPS'08).

[6] Nikolić, Đ. and Spoto, F., *Definite Expression Aliasing Analysis for Java Bytecode*, In: Proceedings of the 9th International Colloquium on Theoretical Aspects of Computing (ICTAC'12).

[7] Nikolić, Đ. and Spoto, F., *Reachability Analysis of Program Variables*, In: Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR'12).

[8] Pearce, David J. and Kelly, Paul H. J. and Hankin, Chris, *Efficient Field-Sensitive Pointer Analysis for C*, In: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, (PASTE '04).

[9] Rossignoli, S. and Spoto, F., *Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions*, In: Proceedings of the 7th international conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06).

[10] Sagiv, M. and Reps, T. and Wilhelm, R., *Parametric Shape Analysis via 3-Valued Logic*, In: ACM Trans. Program. Lang. Syst., 2002.

[11] Scapin, E. and Spoto, F., *Field-Sensitive Unreachability and Non-Cyclicity Analysis*, http://www.infsec.uni-trier.de/data/scapin/ScapinUnreachNonCyclicAnalysis.pdf

[12] Schmidt, D. A., *Underapproximating Predicate Transformers*, In: Proceedings of the 13th international conference on Static Analysis (SAS'06).

[13] Secci, S. and Spoto F., *Pair-Sharing Analysis of Object-Oriented Programs*, In: Proceedings of the 12th SAS, London, UK (2005).

[14] Spoto, F. and Ernst, M. D., *Inference of Field Initialization*, In: Proceedings of the 33rd ICSE. ACM, Waikiki, Honolulu, USA (2011).

[15] Spoto, F. and Mesnard, F. and Payet, É., *A Termination Analyzer for Java Bytecode Based on Path-Length*, In: ACM Trans. on Programming Languages and Systems (2010).