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 \( x = x . \text{next} \), 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 \( \text{next} \) 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 al-
ready 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.

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 \( T = [\text{int}] \cup \mathbb{K} \), ordered by the extension of \( \leq \) with \( \text{int} \leq \text{int} \). A class \( \kappa \in \mathbb{K} \) has fields \( \kappa.f : t \): i.e., field \( f \) of type \( t \in T \) defined in \( \kappa \). By letting \( \mathcal{F}(\kappa) = \{ \kappa'.f : t' \mid \kappa \leq \kappa' \} \) be the fields defined in \( \kappa \) or in any of its superclasses, we define the set of all fields \( \mathcal{F} = \bigcup_{\kappa \in \mathbb{K}} \mathcal{F}(\kappa) \). A class \( \kappa \) has methods \( \kappa.m(t) : t \) (method \( m \), defined in \( \kappa \), with arguments of type \( t \), returning a value of type \( t \in T \cup \{\text{void}\} \)).

\( V \) 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 \( \tau : V \rightarrow T \), whose domain is written \( \text{dom}(\tau) \). The set of all type environments is \( \mathcal{T} \). A value is an element of \( \mathbb{Z} \cup \mathbb{L} \cup \{\text{null}\} \), where \( \mathbb{L} \) is an infinite set of memory locations. A state over \( \tau \in \mathcal{T} \) is a pair \((\iota || s), \mu)\): \( l \) is an array of values for the local variables in \( \text{dom}(\tau) \); \( s \) is a stack of values for the stack variables in \( \text{dom}(\tau) \), that grows leftwards; \( \mu \) is a memory that binds locations to objects. We often use another representation: \((\rho, \mu)\), where an environment \( \rho \) 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 \( \kappa'.f : t' \) into its value \((o.\phi)(\kappa'.f : t')\). The set of states is \( \Xi \). We write \( \Xi_{\tau} \), when we want to fix the type environment \( \tau \).

The semantics of an instruction \( ins \) is a partial map \( ins : \Sigma_{\tau} \rightarrow \Sigma_{\nu} \) from initial to final states. Number and type of local variables and stack elements at its start are specified by \( \tau \). 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. 1:** Our example

```java
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. 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 $a \in \text{dom}(\tau)$ in a state $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ is $L_\tau(a) = \bigcup_{i \geq 0} L^i_\tau(a)$, where $L^i_\tau(a)$ are the locations reachable from $a$ in at most $i$ steps: $L^i_\tau(a) = \{\rho(a)\} \cap \mathbb{L}$ if $i = 0$, and $L^i_\tau(a) = L^{i-1}_\tau(a) \cup \bigcup_{\ell \in L^{i-1}_\tau(a)} (\text{rng}(\mu(\ell), \phi) \cap \mathbb{L})$ if $i > 0$.

**Definition 3.2** [Path between Variables] Let $\tau \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$, $a, b \in \text{dom}(\tau)$ and $\rho(a), \rho(b) \in \text{dom}(\mu) \subseteq \mathbb{L}$. We define a path $\mathcal{P}$ from $a$ to $b$ in $\sigma$ as an $n$-tuple $\langle \kappa_1, f_1 : t_1, \ldots, \kappa_n, f_n : t_n \rangle \subseteq F$ such that $\exists \ell^1, \ldots, \ell^{n+1} \in \text{dom}(\mu)$. $\ell^1 = \rho(a)$, $\ell^{n+1} = \rho(b)$ and $\forall i = 1, \ldots, n$. $(\mu(\ell^i), \phi)(\kappa_i, f_i : t_i) = \ell^{i+1}$. We denote it by $a \sim_{\sigma}^F b$.

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 $\tau \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$, $F \subseteq F$ and $a, b \in \text{dom}(\tau)$. If, for each path from $a$ to $b$ the fields in $F$ are not in the path, i.e., if $\forall \mathcal{P} \subseteq \mathcal{F} (a \sim_{\sigma}^F b \implies \mathcal{P} \cap F = \emptyset)$, then we write $a \not\sim_{\sigma}^F b$.

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

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 $\tau$ as $C_\tau = \langle \wp(\Sigma_\tau), \subseteq \rangle$ and the abstract lattice over $\tau$ as $A_\tau = \langle \mathcal{U}_r \cup \mathcal{N}_r, \supseteq \rangle$ i.e., the union of two sets: $\mathcal{U}_r = \wp(\text{dom}(\tau) \times \text{dom}(\tau) \times \wp(\mathcal{F}))$ and $\mathcal{N}_r = \wp(\text{dom}(\tau) \times \wp(\mathcal{F}))$. The former is the powerset of the product between the set of ordered pairs of variables $v, w \in \text{dom}(\tau)$ and the powerset of the program fields $F \subseteq \mathcal{F}$. Its elements are written as $v \not\sim^F \mathcal{W} w$. The latter is the powerset of the product between the set of variables $v \in \text{dom}(\tau)$ and the powerset of the program fields $F \subseteq \mathcal{F}$. Its elements are written as $v \not\sim^{\mathcal{F}} F$.

An abstract element $I \in A_\tau$ represents those concrete states in $\Sigma_\tau$ 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 $\tau \in T$ and $I \in A_\tau$. We define the concretization map $\gamma_\tau : A_\tau \rightarrow C_\tau$ in such a way that $\gamma_\tau(I)$ is

$$\{ \sigma \in \Sigma \left| \left( \forall a \mathrel{\#} b \in I, \exists F' \subseteq F, a \mathrel{\#} b \land F \subseteq F' \right) \land \left( \forall c \mathrel{\#} d \in I, \exists F' \subseteq F, c \mathrel{\#} d \land F \subseteq F' \right) \right. \}$$

This map is co-additive in [11] and hence $A_\tau$ and $C_\tau$ are an abstract and concrete domain and $\gamma_\tau$ is the concretization map of a Galois connection [1] between them.

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 $\text{ins}_m$ for every bytecode instruction $\text{ins}$ of $P$; ii) $V$ contains nodes $\text{exit}_m$ and $\text{exception}_m$ 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 $A_\tau$, from the information at its source(s) to the information at its sink. Its exact definition depends on $\text{ins}_m$, 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 $\tau$ and $\tau'$ are the static type information at and immediately after the execution of a bytecode $\text{ins}_m$, respectively. Furthermore, we assume that $\tau$ contains $j$ stack elements and $i$ local variables. We define the propagation rules as a tuple of functions $\langle \Pi^1, \ldots, \Pi^m \rangle$, 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 $\Pi$ is proved formally correct in [11] by using the standard technique of abstract interpretation: namely, by letting $\text{ins}_m$ be a bytecode instruction, $\Pi$ a propagation rule and $I \in A_\tau$, we have proved the
soundness of \( \Pi \) by showing that \( \text{ins}(\gamma_T(I)) \subseteq \gamma_T(\Pi(I)) \) i.e., that for each \( x \not \rightarrow F y \in \Pi(I) \) we have a state \( \sigma' \) such that \( x \not \rightarrow_{\sigma'} F y \wedge F \subseteq F' \) and for each \( z \not \rightarrow_{\sigma'} F \in \Pi(I) \) we have another state \( \sigma' \) such that \( z \not \rightarrow_{\sigma'} F \).

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 \( \mathcal{MR}_r \) 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 \( a \not \rightarrow F_{a_2} s_{j-2} \) and \( s_{j-1} \not \rightarrow F_{b_1} b \). Furthermore, since putfield links \( s_{j-2} \) with \( s_{j-1} \) through \( \kappa.f : t \), the 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 \( s_{j-1} \not \rightarrow F_{12} s_{j-2} \in I \). The resulting set is then: \( F' = [F_c \cap F_{j1} \cap F_{12}] \setminus \{\kappa.f : t\} \). 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

\[ I_{11} = \{ l_0 \phi_{\mathcal{A}}^\sigma(l_0, s_1), l_0 \phi_{\mathcal{A}}^\rho(l_0, \text{value}, \text{El}_n) \{ l_0, s_0 \} \}, l_1 \phi_{\mathcal{A}}^\rho(l_0, l_1, l_2, s_0, s_1), l_2 \phi_{\mathcal{A}}^\rho(l_0, l_1, l_2, s_0, s_1), s_0 \phi_{\mathcal{A}}^\rho(l_0, l_1, l_2, s_0, s_1), s_1 \phi_{\mathcal{A}}^\rho(l_0, \text{prec}_n, \text{El}_n) \{ l_0, s_0 \}, \{ l_0, l_1, l_2, s_0, s_1 \} \} \]

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

\[ I_{12} = \{ l_0 \phi_{\mathcal{A}}^\rho(l_0, \text{El}_n) \{ l_0, s_0 \}, l_0 \phi_{\mathcal{A}}^\rho(l_0, \text{El}_n) \{ l_0, l_1, l_2 \}, l_1 \phi_{\mathcal{A}}^\rho(l_0, l_1, l_2), l_2 \phi_{\mathcal{A}}^\rho(l_0, l_1, l_2) \}
\]

We note that field \( \text{El}_n \) is no more inside the set of fields associated to the pair \( \langle l_2, l_0 \rangle \). This is because \( \langle l_2, s_0 \rangle, \langle s_1, l_0 \rangle \in \mathcal{MR}_{I_{11}} \) and hence their set of fields changes according to the second set of Figure 4: \( F_{l_2, l_0} = \{ F_{l_2, l_0} \cap F_{l_1, s_0} \} \setminus \{ \text{El}_n \} \) = \{ \text{El}_n, \text{El}_\text{prec} \}. Moreover, also the non-cyclicity tokens for \( l_0 \) and \( l_2 \) changed, because a new cycle might be formed by executing this \( \text{putfield} \). Indeed, since \( \langle l_2, s_0 \rangle, \langle l_0, s_1 \rangle, \langle s_1, s_0 \rangle \in \mathcal{MR}_{I_{11}} \), 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: \( F_{l_2} = \{ F_{l_2} \cap F_{l_0} \cap F_{s_1, s_0} \} \setminus \{ \text{El}_n \} \) = \{ \text{El}_\text{prec} \}. 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 \( \text{prec} \) is actually built since this is linked to \( \text{prec} \) through field \( \text{El}_n \), while \( \text{prec} \) is linked to this through field \( \text{El}_\text{next} \). Hence fields \( \text{El}_\text{prec} \) and \( \text{El}_\text{next} \) set up a path \( P = \{ \text{El}_\text{prec}, \text{El}_\text{next} \} \) such that \( \rho(\text{this}) \leadsto_P \rho(\text{this}) \) and, respectively, \( \rho(\text{prec}) \leadsto_P \rho(\text{prec}) \). According to Definition 3.4, they must not be in \( F_{\text{this}} = F_{l_0} \) nor in \( F_{\text{prec}} = F_{l_0} \).

Definition 4.5 [Field-Sensitive Unreachability and Non-Cyclicity Analysis] A solution of an ACG is an assignment of an element \( J_n \in \mathcal{A}_\tau \) to each node \( n \) of the ACG, where \( \tau \) is the type environment associated to \( n \), such that the propagation rules \( \Pi \) 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 \( \text{ins} \) in a state \( \sigma \) and letting \( I_{\text{ins}} \) be the approximation of our properties at the correspondent node, we have proved that \( \sigma \in \gamma(I_{\text{ins}}) \).

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 differ-
ences 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 ($MR_r$) and definite aliasing analyses, for better precision.

References


