Science of Computer Programming 95 (2014) 359–375

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Field-sensitive unreachability and non-cyclicity analysis

Enrico Scapin [a,*], Fausto Spoto [b]

[a] Department of Computer Science, University of Trier, Germany
[b] Department of Computer Science, University of Verona, Italy

HIGHLIGHTS

- We model a novel and sound data-flow analysis for Java bytecode.
- Our interprocedural definite analysis approximates two related heap properties.
- Abstract Interpretation is used to soundly approximate the program semantics.
- An abstract constraint graph of the program is built to compute the solution.
- The final aim is to improve other analyses by also considering program fields.

ARTICLE INFO

Article history:
Received 17 January 2014
Accepted 11 March 2014
Available online 8 April 2014

Keywords:
Data-flow analysis
Interprocedural static analysis
Constraint-based analysis
Field-sensitive analysis
Abstract interpretation

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 approximates 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 with respect to specific fields.

1. Introduction

Static analysis builds compile-time approximations of the set of values, states or behaviors 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 approximated but sound static analyses from the formal specification of the properties 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 can 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

* Corresponding author.
E-mail addresses: scapin@uni-trier.de (E. Scapin), fausto.spoto@univr.it (F. Spoto).

http://dx.doi.org/10.1016/j.scico.2014.03.012
0167-6423/

values of a pointer [3]. *Shape analysis* [11] 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* [14] infers which variables are bound to overlapping data structures; *reachability analysis* [7] looks for paths between locations and *non-cyclicity analysis* [10] 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 never used in the paths between two variables or in a cycle bound to a variable, respectively. Under-approximations in the context of abstract interpretation have been studied in [13] 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 [9], 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 [12]).

Our analysis is designed with the goal of improving client analyses of the Julia analyzer for Java and Android bytecode (`http://www.juliasoft.com`). Namely, its *termination checker* finds method calls that might diverge at run-time, through the *path-length* property [16] *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. Overview of the analyses

We provide here a high-level description of the analyses that we are going to define in the next sections.

Our definite unreachability and non-cyclicity analyses are built over the assumption that the program has been already processed into a graph of basic blocks. There is a subgraph for every method or constructor and those subgraphs are linked at method calls, where each call is bound to an over-approximation of the runtime targets of the call. Bytecodes are assumed to be typed. While Java bytecodes are often untyped, they are guaranteed to be typable by the traditional type inference Kindall algorithm [4]. The construction of the graph and the type inference is already performed in fully implemented tools, such as the Julia analyzer.

Once this preprocessing has been performed, actual static analyses can be performed. We assume that three preliminary static analyses are already available before we run our unreachability and non-cyclicity analyses. They are a definite aliasing analysis between program variables and possible sharing and reachability analyses between program variables. Note that these preliminary analyses do not use field names and are consequently much simpler to define and implement that the new analyses described in this article. They are actually all already available and highly optimized in the Julia analyzer. They are used for these reasons:

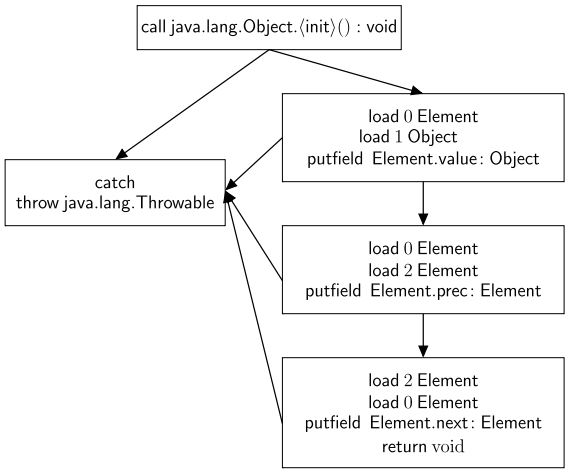
- definite aliasing analysis is used to determine variables of a caller method that are definite alias of parameters passed to a callee. If the parameter is not reassigned inside the callee, then its value at the end of the callee stands for the variable of the caller as well and can be used to reconstruct the side-effects of the callee on that variable;
- possible sharing analysis is used at method call, again, since the locations reachable from a variable of the caller that does not share with any parameter of the callee are unaffected by the call itself. This improves the approximation of the side-effects of the call;
- possible reachability analysis is used to clean-up our new unreachability analysis. If a variable does not possibly reach another, then the latter is unreachable from the former, for any set of fields that might be used to state unreachability. This removes spurious pairs of unreachable variables from the approximation and can be seen as the basis over which our new unreachability analysis builds, by providing more fine-grained information that considers the fields used for reachability as well.

These supporting analyses and our new analyses are plugged inside the same framework of analysis. Namely, the graph of basic blocks is translated into a graph where nodes stand for bytecodes and arcs propagate abstract information among nodes, in a monotonic way. Abstract information is propagated until fixpoint, by using any fixpoint strategy. The Julia analyzer includes a fixpoint strategy that uses a workset of arcs still to propagate. Arcs are sequentially picked up from the workset and propagated; other arcs are added to the workset when the approximation of the heads changes. Token of abstract information are kept inside bitsets, for compact representation and efficient propagation. This propagation is extremely efficient for relatively simple analyses such as definite aliasing, possible sharing and reachability. Instead, it might be expensive for complex analyses as those described in this article, that are still to be implemented. This complexity can

```
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;
 }
```

(a)



(b)

Fig. 1. The Java code of our examples (a) and the CFG of its second constructor (b).

be tamed by fixing a worst-case approximation at the beginning of each method or constructor, so that the analyses fall back to being intraprocedural. It is also possible, and much better for precision, to apply a worst-case assumption when and where the analyses require too much time to converge to the fixpoint. This situation should be compared to that of shape analysis, that might be more precise than our analyses but requires the specification of the shapes of interest for each program under analysis and has a very high computational cost. This is why a shape analysis for general purpose Java programs is not yet available.

Since our new analyses are examples of definite analyses, approximation means in this context that the analyses feature false negatives. That is, if they state the unreachability between two variables then those variables are actually unreachable, but there might be unreachable variables that are not proved to be unreachable by our analyses.

3. 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 [15] and more widely explained in [12]. 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.

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. 1b shows it for the second constructor from Fig. 1a. 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 1 *(Classes, type environment, state).* The set of *classes* $\mathbb{K}$ of a program is partially ordered *w.r.t.* the *subclass relation* $\leqslant$. A *type* is an element of $\mathbb{T} = \{int\} \cup \mathbb{K}$, ordered by the extension of $\leqslant$ with $int \leqslant int$. 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 $\mathbb{F}(\kappa) = \{\kappa'.f : \mathsf{t}' \mid \kappa \leqslant \kappa'\}$ 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* $\kappa.m(\vec{\mathsf{t}}) : \mathsf{t}$ (method m, defined in κ, with arguments of type $\vec{\mathsf{t}}$, 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 as $\mathsf{dom}(\tau)$. The set of all type environments is $\mathcal{T}$.

A *value* $\mathbb{V}$ 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 $\langle\langle l \parallel s\rangle, \mu\rangle$: 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.

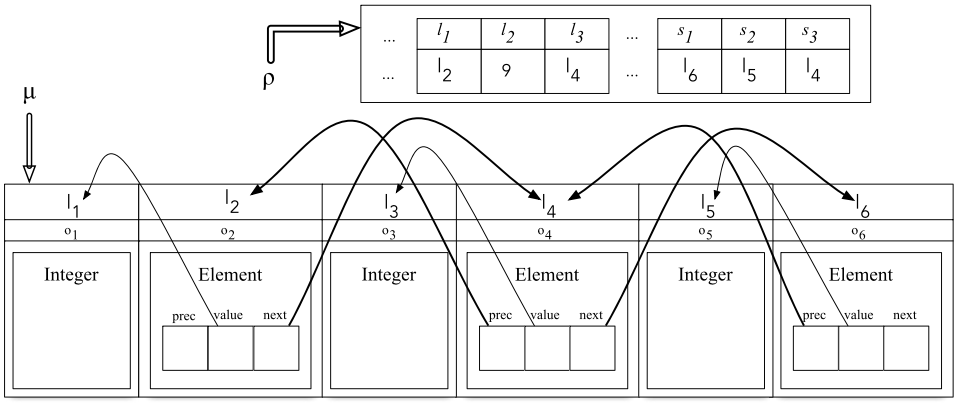


Fig. 2. A JVM state $\sigma = \langle \rho, \mu \rangle$.

$$
\begin{array}{rl}
\mathsf{const}\ z = & \lambda\langle\langle l \parallel s\rangle, \mu\rangle \,.\, \langle\langle l \parallel z :: s\rangle, \mu\rangle \\
\mathsf{dup\ t} = & \lambda\langle\langle l \parallel v :: s\rangle, \mu\rangle \,.\, \langle\langle l \parallel v :: v :: s\rangle, \mu\rangle \\
\mathsf{load}\ k\ \mathsf{t} = & \lambda\langle\langle l \parallel s\rangle, \mu\rangle \,.\, \langle\langle l \parallel l[k] :: s\rangle, \mu\rangle \\
\mathsf{store}\ k\ \mathsf{t} = & \lambda\langle\langle l \parallel v :: s\rangle, \mu\rangle \,.\, \langle\langle l[k \mapsto v] \parallel s\rangle, \mu\rangle \\
\mathsf{ifne\ t} = & \lambda\langle\langle l \parallel v :: s\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle l \parallel s\rangle, \mu\rangle & \text{if } v \in \{0, \mathtt{null}\} \\ \mathit{undefined} & \text{otherwise} \end{cases} \\
\mathsf{ifeq\ t} = & \lambda\langle\langle l \parallel v :: s\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle l \parallel s\rangle, \mu\rangle & \text{if } v \notin \{0, \mathtt{null}\} \\ \mathit{undefined} & \text{otherwise} \end{cases} \\
\mathsf{new}\ \kappa = & \lambda\langle\langle l \parallel s\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle l \parallel \ell :: s\rangle, \mu[\ell \mapsto o]\rangle & \text{if enough memory} \\ \underline{\langle\langle l \parallel \ell\rangle, \mu[\ell \mapsto \mathit{oome}]\rangle} & \text{otherwise} \end{cases} \\
\mathsf{getfield}\ \kappa.f{:}\mathsf{t} = & \lambda\langle\langle l \parallel \ell :: s\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle l \parallel (\mu(\ell).\phi)(f) :: s\rangle, \mu\rangle & \text{if } \ell \neq \mathtt{null} \\ \underline{\langle\langle l \parallel \ell\rangle, \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{otherwise} \end{cases} \\
\mathsf{putfield}\ \kappa.f{:}\mathsf{t} = & \lambda\langle\langle l \parallel v :: \ell :: s\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle l \parallel s\rangle, \mu[(\mu(\ell).\phi)(f) \mapsto v]\rangle & \text{if } \ell \neq \mathtt{null} \\ \underline{\langle\langle l \parallel \ell\rangle, \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{otherwise} \end{cases} \\
\mathsf{throw}\ \kappa = & \lambda\langle\langle l \parallel v :: s\rangle, \mu\rangle \,.\, \begin{cases} \underline{\langle\langle l \parallel \ell\rangle, \mu\rangle} & \text{if } \ell \neq \mathtt{null} \\ \underline{\langle\langle l \parallel \ell\rangle, \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{otherwise} \end{cases} \\
\mathsf{catch} = & \underline{\lambda\langle\langle l \parallel v\rangle, \mu\rangle} \,.\, \langle\langle l \parallel v\rangle, \mu\rangle \\
\mathsf{excp\_is}\ K = & \lambda\langle\langle l \parallel v\rangle, \mu\rangle \,.\, \begin{cases} \langle\langle l \parallel v\rangle, \mu\rangle & \text{if } v \in \mathbb{L} \text{ and } \mu(v).\kappa \in \mathbb{K} \\ \mathit{undefined} & \text{otherwise} \end{cases} \\
\mathsf{return\ void} = & \lambda\langle\langle l \parallel s\rangle, \mu\rangle \,.\, \langle\langle l \parallel \epsilon\rangle, \mu\rangle \\
\mathsf{return\ t} = & \lambda\langle\langle l \parallel v :: s\rangle, \mu\rangle \,.\, \langle\langle l \parallel v\rangle, \mu\rangle, \qquad \text{where } v \neq \mathsf{void}
\end{array}
$$

Fig. 3. Bytecode semantics. Each instruction is modelled as a function mapping a pre-state to a post-state. $\mathbb{V} \in \mathbb{Z} \cup \mathbb{L} \cup \{\mathtt{null}\}$ is a value; $\mathsf{int} \in \mathbb{Z}$ is an integer constant and $\ell \in \mathbb{L} \cup \{\mathtt{null}\}$ is a location. For exceptions, *oome* is a new instance of `OutOfMemoryException`, while *npe* a new instance of `NullPointerException`.

Example 1. Let $\tau = [l_1 \mapsto \mathsf{Element}; l_2 \mapsto \mathsf{int}; l_3 \mapsto \mathsf{Element}; s_1 \mapsto \mathsf{Element}; s_2 \mapsto \mathsf{Object}; s_3 \mapsto \mathsf{Element}]$ be a type environment and consider Fig. 2 representing a state $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$. Environment ρ maps variables l_1, l_2, l_3 and s_1, s_2, s_3 to values $\ell_2, 9, \ell_4$ and ℓ_6, ℓ_5, ℓ_4, respectively. Memory μ maps locations ℓ_1, ℓ_3 and ℓ_5 to objects o_1, o_3 and o_5 of class Integer; it also maps locations ℓ_2, ℓ_4 and ℓ_6 to objects o_2, o_4 and o_6 of class Element. Objects are represented as boxes with a class tag and a local environment mapping fields to integers, locations or `null`. For instance, fields value and next of objects o_2 contain locations ℓ_1 and ℓ_4, respectively.

We assume that states are well-typed *i.e.*, variables hold values consistent with their static types. Since the JVM supports exceptions, we distinguish between *normal* states Ξ and *exceptional* states $\underline{\Xi}$, which arise *immediately after* bytecode instructions that throw an exception and have a stack of height 1 containing a location bound to the thrown exception. When we denote a state by σ, we do not specify if it is normal or exceptional. If we want to stress that, we write $\langle\langle l \parallel s\rangle, \mu\rangle$ or $\underline{\langle\langle l \parallel s\rangle, \mu\rangle}$, respectively. The semantics of an instruction ins is a partial map $\mathit{ins} : \Sigma_\tau \to \Sigma_{\tau'}$ from *initial* to *final* states. Number and type of local variables and stack elements at its start are specified by τ.

We now explain the denotational semantics of each bytecode instruction, as it is reported in Fig. 3.

Basic instructions. const int pushes int $\in \mathbb{Z}$ on the top of the stack. Like any other bytecode except catch, it is defined only when the JVM is in a normal state. Only catch starts the exceptional handlers from an exceptional state and is, therefore, undefined on a normal state. dup t duplicates the top of the stack, of type t. load k t pushes on the stack the value of local variable number k, l_k, which must exist and have type t. Conversely, store k t pops the top of the stack of type t and writes it in local variable l_k; it might potentially enlarge the set of local variables, in the sense that it can provide a value for a local variable that was still uninitialized before that program point and consequently not usable, according to the static constraints of the JVM. In our formalization, conditional bytecodes are used in complementary pairs (such as ifne t and ifeq t), at a conditional branch. For instance, ifeq t checks whether the top of the stack, of type t, is 0 when t $= int$ or `null` when t $\in \mathbb{K}$. Otherwise, its semantics is undefined.

Object-manipulating instructions. These bytecode instructions create or access objects in memory. new κ pushes on the stack a reference to a new object o of class κ, whose fields are initialized to a default value: `null` for reference fields, and 0 for integer fields. getfield $\kappa.f$: t reads the field $\kappa.f$: t of an object bound to a receiver location ℓ popped from the stack, of type κ. putfield $\kappa.f$: t writes the top of the stack, of type t, inside field $\kappa.f$: t of the object pointed by the underlying location ℓ, of type κ.

Exception-handling instructions. Bytecode throw κ throws the top of the stack, of type $\kappa \leqslant$ Throwable. Bytecode catch starts an exception handler: it takes an exceptional state and transforms it into a normal state at the beginning of the handler. After catch, excp_is K selects an appropriate handler depending on the run-time class of the exception.

Method call and return. When a caller transfers control to a callee $\kappa.m(\vec{\mathrm{t}})$: t, the JVM runs an operation *makescope* $\kappa.m(\vec{\mathrm{t}})$: t that copies the topmost stack elements, holding the actual arguments of the call, to local variables that correspond to the formal parameters of the callee, and then clears the stack. We only consider instance methods, where this is a special argument held in the local variable l_0 of the callee. The *makescope* $\kappa.m(\vec{\mathrm{t}})$: t function is formally defined as follows: let $\kappa.m(\vec{\mathrm{t}})$: t be a method and π the number of stack elements holding its actual parameters, including the implicit parameter this; (*makescope* $\kappa.m(\vec{\mathrm{t}})$: t): $\Sigma \to \Sigma$ is defined as $\lambda\langle\langle l \parallel v_{\pi-1} :: \cdots :: v_1 :: \mathsf{rec} :: \mathrm{s}\rangle, \mu\rangle.\langle\langle[\mathsf{rec}, v_1, \ldots, v_{\pi-1}] \parallel \epsilon\rangle, \mu\rangle$, provided that rec $\neq$ `null` and the look-up of $m(\vec{\mathrm{t}})$: t from the class $\mu(\mathsf{rec}).\kappa$ leads to $\kappa.m(\vec{\mathrm{t}})$: t; it is undefined otherwise.

We define the operational semantics of our language. It uses a stack of activation records to model method and constructor calls.

Definition 2 *(Configuration).* A *configuration* is a pair $\langle b \parallel \sigma\rangle$ of a block b and a state σ representing the fact that the JVM is about to execute b in state σ. An *activation stack* is a stack $c_1 :: c_2 :: \cdots :: c_n$ of configurations, where c_1 is the *active* configuration.

The *operational semantics* of a Java bytecode program is a relation between activation stacks. It models the transformation of the activation stack induced by the execution of each single bytecode.

Definition 3 *(Operational semantics).* The small step operational semantics of a Java Bytecode program P is a relation $a' \Rightarrow_P a''$ (P is usually omitted) providing the immediate successor activation stack a'' of an activation stack a'. It is defined by the rules in Table 1.

Rule **(1)** runs the first instruction ins of a block, different from call, by using its semantics *ins*.

Rule **(2)** calls a method on a non-`null` receiver: the call instructions are decorated with an over-approximation of the set of their possible run-time target methods. This approximation can be computed by class analysis [8]. The dynamic semantics of call looks up for the exact target implementation $\kappa_i.m(\vec{\mathrm{t}})$: t that is executed, by using the look-up rules of the language, builds its initial state σ' by using *makescope*, and creates a new current configuration containing the first block of the target implementation and σ'. It pops the actual arguments from the previous configuration and the call from the instructions to be executed at return time. A method call may lead to many implementations, depending on the run-time class of the receiver. Since in Java bytecode the look-up rule of methods is deterministic, only one thread of execution actually continues here. If, instead, a call occurs on a `null` receiver, no actual call happens and Rule **(3)** creates a new stack containing only a reference to a `NullPointerException`.

After the execution of the method, if the callee ends in a normal state, control returns to the caller by Rule **(4)**: it rehabilitates the caller configuration but keeps the memory at the end of the execution of the callee and pushes the return value on the stack of the caller. If, instead, the callee ends in an exceptional state, Rule **(5)** propagates the exception back to the caller.

Rule **(6)** applies when all instructions inside a block have been executed; it runs one of its immediate successors, if any. In our formalization this rule is always deterministic: if a block has two or more immediate successors then they start with mutually exclusive conditional instructions and only one thread of control is actually followed.

Table 1
The transition rules of our semantics.

$$\frac{\text{ins is not a call, } ins(\sigma) \text{ is defined}}{\left\langle \begin{array}{l} ins \\ rest \end{array} \right| \begin{array}{l} \to \\ \to \end{array} \begin{array}{c} b_1 \\ \cdots \\ b_m \end{array} \| \sigma \rangle :: a \Rightarrow \left\langle \begin{array}{l} rest \end{array} \right| \begin{array}{l} \to \\ \to \end{array} \begin{array}{c} b_1 \\ \cdots \\ b_m \end{array} \| ins(\sigma) \rangle :: a} \quad (1)$$

$$\frac{\begin{array}{c}\pi \text{ is the number of parameters of the target method, including this} \\ \sigma = \langle\langle \ell \parallel v_{\pi-1} :: \ldots :: v_1 \ldots \mathsf{rec} \ldots s\rangle, \mu\rangle, \mathsf{rec} \neq \mathtt{null} \\ 1 \leq i \leq n \text{ is such that } \sigma' = (\mathsf{makescope}\ \kappa_i.m)(\sigma) \text{ is defined} \\ f = first(\kappa_i.m) \text{ is the block where the implementation starts}\end{array}}{\left\langle \begin{array}{l} \mathsf{call}\ \kappa_1.m \ldots \kappa_n.m \\ rest \end{array} \right| \begin{array}{l} \to \\ \to \end{array} \begin{array}{c} b_1 \\ \cdots \\ b_m \end{array} \| \sigma \rangle :: a \Rightarrow \langle f \parallel \sigma' \rangle :: \left\langle \begin{array}{l} rest \end{array} \right| \begin{array}{l} \to \\ \to \end{array} \begin{array}{c} b_1 \\ \cdots \\ b_m \end{array} \| \langle\langle l \parallel s\rangle, \mu\rangle\rangle :: a} \quad (2)$$

$$\frac{\begin{array}{c}\pi \text{ is the number of parameters of the target method, including this} \\ \sigma = \langle\langle \ell \parallel v_{\pi-1} :: \ldots :: v_1 \ldots \mathsf{rec} \ldots s\rangle, \mu\rangle \\ \ell \in \mathbb{L} \text{ is fresh and } npe \text{ is a new instance of } \mathtt{NullPointerException}\end{array}}{\left\langle \begin{array}{l} \mathsf{call}\ \kappa_1.m \ldots \kappa_n.m \\ rest \end{array} \right| \begin{array}{l} \to \\ \to \end{array} \begin{array}{c} b_1 \\ \cdots \\ b_m \end{array} \| \sigma \rangle :: a \Rightarrow \left\langle \begin{array}{l} rest \end{array} \right| \begin{array}{l} \to \\ \to \end{array} \begin{array}{c} b_1 \\ \cdots \\ b_m \end{array} \| \langle\langle l \parallel \ell\rangle, \mu[\ell \mapsto npe]\rangle\rangle :: a} \quad (3)$$

$$\frac{}{\left\langle \begin{array}{l} \\ \end{array} \right| \| \langle\langle l \parallel top\rangle, \mu\rangle\rangle :: \langle b \parallel \langle\langle l' \parallel s'\rangle, \mu'\rangle\rangle :: a \Rightarrow \langle b \parallel \langle\langle l' \parallel top :: s'\rangle, \mu\rangle\rangle :: a} \quad (4)$$

$$\frac{}{\left\langle \begin{array}{l} \\ \end{array} \right| \| \underline{\langle\langle l \parallel e\rangle, \mu\rangle\rangle} :: \langle b \parallel \langle\langle l' \parallel s'\rangle, \mu'\rangle\rangle :: a \Rightarrow \langle b \parallel \underline{\langle\langle l' \parallel e\rangle, \mu\rangle\rangle} :: a} \quad (5)$$

$$\frac{1 \leq i \leq m}{\left\langle \begin{array}{l} \\ \end{array} \right| \begin{array}{l} \to \\ \to \end{array} \begin{array}{c} b_1 \\ \cdots \\ b_m \end{array} \| \sigma \rangle :: a \Rightarrow \langle b_i \parallel \sigma \rangle :: a} \quad (6)$$

4. Field-sensitive properties

In this section, we formalize the two properties that we want to state for each program point: field-sensitive *unreachability* and field-sensitive *non-cyclicity* of program variables. To that purpose, we first introduce two preliminary definitions.

Definition 4 *(Locations reachable from a variable).* (See [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 $\mathsf{L}_\sigma(a) = \bigcup_{i \geqslant 0} \mathsf{L}^i_\sigma(a)$, where $\mathsf{L}^i_\sigma(a)$ are the locations reachable from a in at most i steps:

$$\mathsf{L}^i_\sigma(a) = \begin{cases} \{\rho(a)\} \cap \mathbb{L} & \text{if } i = 0 \\ \mathsf{L}^{i-1}_\sigma(a) \cup \bigcup_{\ell \in \mathsf{L}^{i-1}_\sigma(a)} \mathsf{range}(\mu(\ell).\phi) \cap \mathbb{L} & \text{if } i > 0 \end{cases}$$

where $\mathsf{range}(\mu(\ell).\phi) = \{\mathbb{V} \in \mathbb{Z} \cup \mathbb{L} \cup \{\mathtt{null}\} \mid \exists\, \kappa.f : \mathsf{t} \text{ such that } (\mu(\ell).\phi)(\kappa.f : \mathsf{t}) = \mathbb{V}\}$ *i.e.*, the set of values bound to the fields of the object $\mu(\ell)$.

Intuitively, the locations reachable from a variable are computed by first collecting all the locations held in the fields of the object bound to the variable, then by considering the contents of the fields of the objects held at these locations and so on until reaching a fixpoint.

Example 2. Consider Fig. 2 representing the state $\sigma = \langle \rho, \mu \rangle$ discussed in Example 2. Then, for instance, the locations reachable from l_1 are: $\mathsf{L}^0_\sigma(l_1) = \{\ell_2\}$, $\mathsf{L}^1_\sigma(l_1) = \{\ell_2, \ell_1, \ell_4\}$, $\mathsf{L}^2_\sigma(l_1) = \{\ell_2, \ell_1, \ell_4, \ell_3, \ell_6\}$, $\mathsf{L}^3_\sigma(l_1) = \mathsf{L}^4_\sigma(l_1) = \{\ell_2, \ell_1, \ell_4, \ell_3, \ell_6 \ell_5\}$. Hence, since we reached the fixpoint, we can assert that $\mathsf{L}_\sigma(l_1) = \{\ell_2, \ell_1, \ell_4, \ell_3, \ell_6, \ell_5\}$.

Definition 5 *(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

$$\exists \ell^1, \ldots, \ell^{n+1} \in \mathsf{dom}(\mu).\ \ell^1 = \rho(a), \quad \ell^{n+1} = \rho(b) \wedge \forall i = 1, \ldots, n.\ \left(\mu\left(\ell^i\right).\phi\right)(\kappa_i.f_i : \mathsf{t}_i) = \ell^{i+1}$$

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 $\rho(b)$ simply by following the fields in the tuple.

Example 3. Consider Fig. 2 representing the state $\sigma = \langle \rho, \mu \rangle$ discussed in Example 2. Then, for instance the path from l_1 to s_2, $l_1 \not\rightsquigarrow_\sigma^{\mathcal{P}_1} s_2$, is $\mathcal{P}_1 = \langle \mathsf{El}.next, \mathsf{El}.next, \mathsf{El}.value \rangle$: El is an abbreviation for Element. On the other hand, the path from s_1 to l_1, $s_1 \not\rightsquigarrow_\sigma^{\mathcal{P}_2} l_1$, is $\mathcal{P}_2 = \langle \mathsf{El}.prec, \mathsf{El}.prec \rangle$.

We can now introduce the two properties that we want to approximate at each point of the program under analysis.

Definition 6 *(Field-sensitive unreachability between variables).* Let $\tau \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$, $F \subseteq \mathcal{F}$ and $a, b \in \mathsf{dom}(\tau)$. We say that "for each path from a to b the fields in F are not in the path" if

$$\forall \mathcal{P} \subseteq \mathcal{F} (a \rightsquigarrow_\sigma^{\mathcal{P}} b \Longrightarrow \mathcal{P} \cap F = \emptyset)$$

We denote it as $a \not\rightsquigarrow_\sigma^F b$.

In other words, the fields in F are all program fields that do not belong to any path from the location bound to a to the location bound to b. We note that, when $a \not\rightsquigarrow_\sigma^{\mathcal{F}} b$, then either there are no paths from a to b *i.e.*, the antecedent is false, or $\rho(a) = \rho(b)$.

Example 4. Consider Fig. 2 representing the state $\sigma = \langle \rho, \mu \rangle$ discussed in Example 2 and let $\mathcal{F} = \langle \mathsf{El}.next, \mathsf{El}.prec, \mathsf{El}.value \rangle$. Then, for instance, the unreachability between variables l_1 and s_2 is stated as $l_1 \not\rightsquigarrow_\sigma^{F_1} s_2$ where $F_1 = \{\mathsf{El}.prec\}$, since the path between them is $\mathcal{P}_1 = \langle \mathsf{El}.next, \mathsf{El}.next, \mathsf{El}.value \rangle$ (Example 3) and therefore the only field which is not in $\mathcal{P}_1$ is $\{\mathsf{El}.prec\}$. On the other hand, the unreachability between variables s_1 and l_1 is stated as $s_1 \not\rightsquigarrow_\sigma^{F_2} l_1$ where $F_2 = \{\mathsf{El}.next, \mathsf{El}.value\}$, since the path between them is $\mathcal{P}_2 = \langle \mathsf{El}.prec, \mathsf{El}.prec \rangle$ (Example 3) and therefore the fields that are not in $\mathcal{P}_2$ are $\{\mathsf{El}.next, \mathsf{El}.value\}$.

Definition 7 *(Field-sensitive non-cyclicity).* Let $\tau \in \mathcal{T}$, $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$, $F \subseteq \mathcal{F}$ and $a \in \mathsf{dom}(\tau)$. We say that "for each cycle reachable from a, the fields in F are not in the cycle" if

$$\forall \ell \in \mathsf{L}_\sigma(a), \forall \mathcal{P} \subseteq \mathcal{F} (\ell \rightsquigarrow_\mu^{\mathcal{P}} \ell \Rightarrow \mathcal{P} \cap F = \emptyset)$$

We denote it as $a \rightsquigarrow_\sigma^{\not\circlearrowleft F}$.

It is worth noting that the expression $\ell \rightsquigarrow_\mu^{\mathcal{P}} \ell$ denotes a cyclical-path and hence the cyclicity of ℓ itself. That is, there is a path with at least a field of the object stored at ℓ through which we can reach the location itself. Therefore the fields in F are all program fields that do not belong to any cycle reachable from the location bound to a. If from that location we cannot reach any cycle, the we have $a \rightsquigarrow^{\not\circlearrowleft \mathcal{F}}$.

Example 5. Consider Fig. 2 representing the state $\sigma = \langle \rho, \mu \rangle$ discussed in Example 2 and let $\mathcal{F} = \langle \mathsf{El}.next, \mathsf{El}.prec, \mathsf{El}.value \rangle$. Then, for instance the non-cyclicity of variable l_1 is stated as $l_1 \rightsquigarrow_\sigma^{\not\circlearrowleft F_1}$, where $F_1 = \{\mathsf{El}.value\}$: from the location bound to l_1 *i.e.*, ℓ_2, we can reach the location itself by two paths, $\mathcal{P}_1 = \langle \mathsf{El}.next, \mathsf{El}.prec \rangle$ and $\mathcal{P}_2 = \langle \mathsf{El}.next, \mathsf{El}.next, \mathsf{El}.prec, \mathsf{El}.prec \rangle$, but in both cases the field $\mathsf{El}.value$ is not part of the path. On the other hand, the non-cyclicity of variable s_2 is stated as $s_2 \rightsquigarrow_\sigma^{\not\circlearrowleft F_2}$, where $F_2 = \langle \mathsf{El}.prec, \mathsf{El}.next, \mathsf{El}.value \rangle$: there are no paths starting from the location bound to s_2 *i.e.*, ℓ_5, and therefore it cannot reach any cycle.

It is worth noting that, when we assert $v \not\rightsquigarrow_\sigma^F w$ or $v \rightsquigarrow_\sigma^{\not\circlearrowleft F}$ for some $v, w \in \mathsf{dom}(\tau)$, these properties hold also for every $F' \subseteq F$. This is important in our approximation.

5. Constraint-based fields-sensitive analysis

We define here an abstract interpretation of the concrete semantics introduced in Section 3 *w.r.t.* the two properties introduced in Section 4. This will be an actual static analysis algorithm for interprocedural, whole-program analysis. While the concrete semantics works over concrete states, the abstract interpretation abstracts it into ordered pairs of variables for the unreachability property and into variables for the non-cyclicity one.

The fact that we define a whole-program analysis entails that the computational cost of our analysis might explode for large programs. A solution, in that case, would be to assume that the approximation at the beginning of each method or constructor is empty, which corresponds to a worst-case assumption at the calling context and to an intra-procedural only analysis. On the other hand, experiments with our Julia analyzer (for other, equally expensive static analyses) have shown that it is better to aim at a precise, interprocedural static analysis and let the analyzer downgrade that precision only when

a timeout occurs. For instance, a worst-case (empty) approximation can be used, in our case, when the cost of the analysis of a piece of code takes too long to be computed.

Definition 8 *(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 $v \rightsquigarrow^{\not\emptyset F}$.

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 9 *(Concretization map).* Let $\tau \in \mathcal{T}$ and $I \in \mathsf{A}_\tau$. We define the *concretization map* $\gamma_\tau : \mathsf{A}_\tau \to \mathsf{C}_\tau$ as

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

This map is co-additive, as shown in [12], 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.

Before introducing the field-sensitive analysis, we note that we use the result of other static analyses in order to achieve better precision of the assertions that we are able to state. The analyses that we exploit are: *Possible Reachability*, an over–approximation of the concrete reachability information [7], *Possible Sharing*, an over-approximation of the concrete sharing information [14], and *Definite Aliasing*, an under-approximation of the concrete aliasing information [6], all between pairs of variables and/or fields of the program. In particular, for each program point, we assume that three sets are available: $\mathcal{MR}_\tau$ containing all pairs of variables/fields such that the former might reach the latter, $\mathcal{MS}_\tau$ containing all pairs of variables/fields such that they might be bound to an overlapped data structure and $\mathcal{DA}_\tau$ containing all pairs of variables/fields definitely pointing to the same location. Furthermore, we assume that these analyses are processed asynchronously *i.e.*, when we build our analysis we already have the related approximated information at each program point.

Let now introduce our analysis. It is *constraint-based*, in the sense that it builds an Abstract Constraint Graph (ACG, see Fig. 4) from the program under analysis, by creating a node of the graph for each bytecode instruction *ins* of the program. This node contains an element from A_τ, where τ is the static type information at the beginning of *ins*. Arcs of this graph propagate abstract domain elements, reflecting, in abstract terms, the effects of the concrete semantics over the reachability information. In other words, an arc from the node for bytecode instruction ins_1 to the node for bytecode instruction ins_2 propagates the information at ins_1 into that at ins_2. The exact meaning of *propagates* depends here on ins_1, since each bytecode instruction has different effects on our unreachability and non-cyclicity properties. Ours is a *forward* analysis, since abstract information is propagated from the beginning of each instruction to its end.

Definition 10 *(Abstract Constraint Graph).* 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* Π *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. Each arc is associated with its propagation rule Π. We now discuss different types of arcs depending on the bytecode instructions that they link and define their propagation rules.

Sequential arcs. If ins is a bytecode in P, distinct from call, immediately followed by a bytecode ins′, distinct from catch, then a simple arc is built from $\boxed{\mathsf{ins}}$ to $\boxed{\mathsf{ins}'}$, with one of the propagation rules between #1 and #7.

Final arcs. For each return t and throw κ occurring in a method or in a constructor m of P, there are simple arcs from $\boxed{\mathsf{return\ t}}$ to $\boxed{\mathsf{exit@}m}$ and from $\boxed{\mathsf{throw}\ \kappa}$ to $\boxed{\mathsf{exception@}m}$, respectively, with one of the propagation rules #8 or #9.

Parameter passing arcs. For each $\mathsf{ins}_c = \mathsf{call}\ m_1 \ldots m_k$ to a method with π parameters (including this), we build a simple arc from $\boxed{\mathsf{ins}_c}$ to the node corresponding to the first bytecode of m_w with the propagation rule #10, for each $1 \leqslant w \leqslant k$.

Side-effects arc. For each $\mathsf{ins}_c = \mathsf{call}\ m_1 \ldots m_k$ to a method with π parameters (including this) returning no value (void) and each subsequent bytecode ins′, we build a multi-arc from nodes $\mathsf{C} = \boxed{\mathsf{call}\ m_1 \ldots m_k}$ and $\mathsf{E} = \boxed{\mathsf{exit@}m_w}$ (2 sources,

in that order) to $\mathsf{Q} = \boxed{\mathsf{ins}'}$, where ins′ is not a catch for each $1 \leqslant w \leqslant k$. The propagation rule is #13, where $\max = j - \pi$. Furthermore, we denote the static type information at C and Q as τ and τ', respectively.

Return value arc. For each $\mathsf{ins}_c = \mathsf{call}\ m_1 \dots m_k$ to a method with π parameters (including this) returning a value of type $\mathsf{t} \in \mathbb{K}$ and each subsequent bytecode ins′ distinct from catch, we build a multi-arc from $\mathsf{C} = \boxed{\mathsf{call}\ m_1 \dots m_k}$ and $\mathsf{E} = \boxed{\mathsf{exit@}m_w}$ (2 sources, in that order) to $\mathsf{Q} = \boxed{\mathsf{ins}'}$ with propagation rule #14, for each $1 \leqslant w \leqslant k$. Furthermore, we denote the static type information at C and Q as τ and τ', respectively.

Exceptional arcs. For each ins different from call $m_1 \dots m_k$ that might throw an exception, immediately followed by a catch, an arc is built from $\boxed{\mathsf{ins}}$ to $\boxed{\mathsf{catch}}$, with propagation rules #11 or #12. For each $\mathsf{ins} = \mathsf{call}\ m_1 \dots m_k$, method with π parameters (including this), immediately followed by a catch, we build a multi-arc from nodes $\mathsf{C} = \boxed{\mathsf{call}\ m_1 \dots m_k}$ and $\mathsf{E} = \boxed{\mathsf{exception@}m_w}$ (2 sources, in that order) to $\mathsf{Q} = \boxed{\mathsf{catch}}$, for each $1 \leqslant w \leqslant k$, with propagation rule #15. Furthermore, we denote the static type information at C and Q as τ and τ', respectively.

Π	Instruction	Propagation Rule
#1	dup t	$\lambda I.I \cup I[s_{j-1} \mapsto s_j] \cup \left\{ s_{j-1} \not\rightsquigarrow^F s_j, s_j \not\rightsquigarrow^F s_{j-1} \,\middle|\, s_{j-1} \not\rightsquigarrow^F s_{j-1} \in I \right\}$
#2	new κ, const i	$\lambda I.I \cup \left\{ s_j \not\rightsquigarrow^{\mathcal{F}} s_j,\ s_j \rightsquigarrow \not\Box^{\mathcal{F}} \right\} \cup \left\{ a \not\rightsquigarrow^{\mathcal{F}} s_j, s_j \not\rightsquigarrow^{\mathcal{F}} a \,|\, a \in dom(\tau) \right\}$
#3	load k t	$\lambda I.I \cup I[l_k \mapsto s_j] \cup \left\{ l_k \not\rightsquigarrow^F s_j, s_j \not\rightsquigarrow^F l_k \,\middle|\, l_k \not\rightsquigarrow^F l_k \in I \right\}$
#4	store k t	$\lambda I. \left\{ \left(a \not\rightsquigarrow^F b \right) \left[s_{j-1} \mapsto l_k \right] \,\middle|\, a \not\rightsquigarrow^F b \in I \wedge a, b \neq l_k \right\}$ $\cup \left\{ \left(c \rightsquigarrow \not\Box^F \right) \left[s_{j-1} \mapsto l_k \right] \,\middle|\, c \rightsquigarrow \not\Box^F \in I \wedge a \neq l_k \right\}$
#5	catch, excp_is K ifne t, ifeq t	$\lambda I. \left\{ a \not\rightsquigarrow^{F_{ab}} b,\ c \rightsquigarrow \not\Box^{F_c} \in I \,\middle|\, a, b, c \in \mathsf{dom}(\tau') \right\}$

#6 getfield $\kappa.f$:t

$$
\begin{aligned}
&\lambda I. \left\{ a \not\rightsquigarrow^{F_{ab}} b,\ c \rightsquigarrow \not\Box^{F_c} \in I \,\middle|\, a, b, c \neq s_{j-1} \right\} \\
&\cup \left\{ a \not\rightsquigarrow^{\mathcal{F}} s_{j-1} \,\middle|\, a \in \mathsf{dom}(\tau) \setminus \{s_{j-1}\} \wedge \langle a, s_{j-1}.(\kappa.f : \mathsf{t}) \rangle \notin \mathcal{MR}_\tau \right\} \\
&\cup \left\{ a \not\rightsquigarrow^F s_{j-1} \,\middle|\, \begin{array}{l} a \neq s_{j-1} \wedge \langle a, s_{j-1}.(\kappa.f : \mathsf{t}) \rangle \in \mathcal{MR}_\tau \wedge \\ a \not\rightsquigarrow^F b \in I \wedge \left(b, s_{j-1}.(\kappa.f : \mathsf{t}) \right) \in \mathcal{DA}_\tau \end{array} \right\} \\
&\cup \left\{ s_{j-1} \not\rightsquigarrow^{\mathcal{F}} b \,\middle|\, b \in \mathsf{dom}(\tau) \setminus \{s_{j-1}\} \wedge \langle s_{j-1}.(\kappa.f : \mathsf{t}), b \rangle \notin \mathcal{MR}_\tau \right\} \\
&\cup \left\{ s_{j-1} \not\rightsquigarrow^{F'} b \,\middle|\, \begin{array}{l} b \neq s_{j-1} \wedge \langle s_{j-1}.(\kappa.f : \mathsf{t}), b \rangle \in \mathcal{MR}_\tau \wedge \\ s_{j-1} \not\rightsquigarrow^F b \in I \wedge F' = F \cup \{\kappa_k.f_k : \mathsf{t}_k \in \mathcal{F} \,|\, \mathsf{t}_k \notin \mathsf{T(t)}\} \end{array} \right\} \\
&\cup \left\{ s_{j-1} \not\rightsquigarrow^F s_{j-1} \,\middle|\, a \not\rightsquigarrow^F a \in I \wedge \left(a, s_{j-1}.(\kappa.f : \mathsf{t}) \right) \in \mathcal{DA}_\tau \right\} \\
&\cup \left\{ s_{j-1} \rightsquigarrow \not\Box^{F'} \,\middle|\, \begin{array}{l} s_{j-1} \rightsquigarrow \not\Box^F \in I \wedge \\ F' = F \cup \{\kappa_k.f_k : \mathsf{t}_k \in \mathcal{F} \,|\, \mathsf{t}_k \notin \mathsf{T(t)}\} \end{array} \right\}
\end{aligned}
$$

#7 putfield $\kappa.f$:t

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

Definition 10 specifies how the ACG is built from the program under analysis, for each bytecode instruction $\boxed{\mathsf{ins}}$ in the graph, *decorated* by an element of our abstract domain defined in 8 and representing an under-approximation of our two properties that we want to state at that point. These rules state how this approximation is propagated along the arcs of the ACG; before starting to fully explain them, we introduce the concept of Normalized Propagation Rules that help us to formalize how to compute the abstract set I for every node whose in-degree is greater than 1.

Definition 11 *(Normalized propagation rules).* Let Π be a propagation rule, $\tau \in \mathcal{T}$, $I \in \mathsf{A}_\tau$. We define the *Abstract Function* Φ as the composition of two functions:

$$\Phi = \mathsf{norm} \circ \Pi$$

where the norm operator is defined as:

$$\lambda I.I \cup \left\{ a \not\rightsquigarrow^{F'} b \,\middle|\, a \not\rightsquigarrow^F b \in I \wedge F' \subseteq F \right\} \cup \left\{ c \rightsquigarrow \not\Box^{F'} \,\middle|\, c \rightsquigarrow \not\Box^F \in I \wedge F' \subseteq F \right\}$$

In other words, given an element $I \in \mathsf{A}_\tau$, for each unreachability and non-cyclicity token *w.r.t.* a set F, the norm operator adds the same relations with every $F' \subseteq F$. In this way, to build the abstract set of a node with in-degree greater than 1 we simply use the join operator $\cap$. This is because, since the norm operator takes every token depending on a set of fields F and adds the same relations for every $F' \subseteq F$, we can simply use the join operator to estimate the least upper bound of these sets.

Π	Instruction	Propagation Rule
#8	return void	$\lambda I.\left\{a \not\rightsquigarrow^{F_{ab}} b,\ c \rightsquigarrow \not\emptyset^{F_c} \in I \,\middle|\, a, b, c \notin \{s_0, \ldots, s_{j-1}\}\right\}$
#9	return t with t ≠ void throw κ	$\lambda I.\left\{\left(a \not\rightsquigarrow^{F} b\right)\left[s_{j-1} \mapsto s_0\right] \,\middle|\, a \not\rightsquigarrow^{F} b \in I \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\right\}$ $\cup\left\{\left(c \rightsquigarrow \not\emptyset^{F}\right)\left[s_{j-1} \mapsto s_0\right] \,\middle|\, c \rightsquigarrow \not\emptyset^{F} \in I \wedge c \notin \{s_0, \ldots, s_{j-2}\}\right\}$
#10	call $m_1 \ldots m_k$	$\lambda I.\left\{\left(a \not\rightsquigarrow^{F} b\right)\left[\begin{smallmatrix} s_{j-\pi} \mapsto l_0 \\ \ldots \\ s_{j-1} \mapsto l_{\pi-1} \end{smallmatrix}\right] \,\middle|\, a \not\rightsquigarrow^{F} b \in I \wedge a, b \in \{s_{j-\pi}, \ldots, s_{j-1}\}\right\}$ $\cup\left\{\left(c \rightsquigarrow \not\emptyset^{F}\right)\left[\begin{smallmatrix} s_{j-\pi} \mapsto l_0 \\ \ldots \\ s_{j-1} \mapsto l_{\pi-1} \end{smallmatrix}\right] \,\middle|\, c \rightsquigarrow \not\emptyset^{F} \in I \wedge c \in \{s_{j-\pi}, \ldots, s_{j-1}\}\right\}$
#11	throw κ	$\lambda I.\left\{\left(a \not\rightsquigarrow^{F} b\right)\left[s_{j-1} \mapsto s_0\right] \,\middle|\, a \not\rightsquigarrow^{F} b \in I \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\right\}$ $\cup\left\{\left(c \rightsquigarrow \not\emptyset^{F}\right)\left[s_{j-1} \mapsto s_0\right] \,\middle|\, c \rightsquigarrow \not\emptyset^{F} \in I \wedge c \notin \{s_0, \ldots, s_{j-2}\}\right\}$
#12	new κ getfield $\kappa.f$:t putfield $\kappa.f$:t	$\lambda I.\left\{a \not\rightsquigarrow^{F_{ab}} b,\ c \rightsquigarrow \not\emptyset^{F_c} \in I \,\middle|\, a, b, c \notin \{s_0, \ldots, s_{j-1}\}\right\}$ $\cup\left\{a \not\rightsquigarrow^{\mathcal{F}} s_0,\ s_0 \not\rightsquigarrow^{\mathcal{F}} a \,|\, a \in L\right\} \cup \left\{s_0 \not\rightsquigarrow^{\mathcal{F}} s_0,\ s_0 \rightsquigarrow \not\emptyset^{\mathcal{F}}\right\}$

#13 call $m_1 \ldots m_k$ (side effects)

$$\lambda I_1.\lambda I_2.\lambda max.\left\{a \not\rightsquigarrow^{\mathcal{F}} b \,\middle|\, a, b \in \mathsf{dom}(\tau') \wedge \langle a, b\rangle \notin \mathcal{MR}_{\tau'}\right\}$$
$$\cup\left\{\begin{array}{l|l} a \not\rightsquigarrow^{F_1} b, & a, b \in L \cup \{s_0, \ldots, s_{max-1}\} \wedge \\ a \rightsquigarrow \not\emptyset^{F_2} \in I_1 & \forall j - \pi \leq p < j, (a, s_p) \notin \mathcal{MS}_\tau \end{array}\right\}$$
$$\cup\left\{a \not\rightsquigarrow^{F} b \,\middle|\, \begin{array}{l} 1.\ a, b \in L \cup \{s_0, \ldots, s_{max-1}\} \wedge \langle a, b\rangle \in \mathcal{MR}_{\tau'} \\ 2.\ \exists j - \pi \leq p_a, p_b < j \,|\, (a, s_{p_a}), (b, s_{p_b}) \in \mathcal{DA}_\tau \wedge \\ \quad \left[\text{no } \boxed{\text{store } l_{\mathsf{p_a}-j+\pi}} \text{ nor } \boxed{\text{store } l_{\mathsf{p_b}-j+\pi}} \text{ occurs in } m_w\right] \\ 3.\ l_{p_a-j+\pi} \not\rightsquigarrow^{F} l_{p_b-j+\pi} \in I_2 \end{array}\right\}$$
$$\cup\left\{c \rightsquigarrow \not\emptyset^{F} \,\middle|\, \begin{array}{l} 1.\ c \in L \cup \{s_0, \ldots, s_{max-1}\} \\ 2.\ \exists j - \pi \leq p < j \,|\, (c, s_p) \in \mathcal{DA}_\tau \wedge \\ \quad \left[\text{no } \boxed{\text{store } l_{\mathsf{p}-j+\pi}} \text{ occurs in } m_w\right] \\ 3.\ l_{p-j+\pi} \rightsquigarrow \not\emptyset^{F} \in I_2 \end{array}\right\}$$

#14 call $m_1 \ldots m_k$ (return value)

$$\lambda I_1.\lambda I_2.\left\{s_{j-\pi} \not\rightsquigarrow^{F_1} s_{j-\pi},\ s_{j-\pi} \rightsquigarrow \not\emptyset^{F_2} \,\middle|\, s_0 \not\rightsquigarrow^{F_1} s_0,\ s_0 \rightsquigarrow \not\emptyset^{F_2} \in I_2\right\}$$
$$\cup\left\{a \not\rightsquigarrow^{\mathcal{F}} s_{j-\pi} \,\middle|\, a \in \mathsf{dom}(\tau') \setminus \{s_{j-\pi}\} \wedge \langle a, s_{j-\pi}\rangle \notin \mathcal{MR}_{\tau'}\right\}$$
$$\cup\left\{a \not\rightsquigarrow^{F} s_{j-\pi} \,\middle|\, \begin{array}{l} 1.\ a \in \mathsf{dom}(\tau') \setminus \{s_{j-\pi}\} \wedge \langle a, s_{j-\pi}\rangle \in \mathcal{MR}_{\tau'} \\ 2.\ \exists j - \pi \leq p < j \,|\, (s_p, a) \in \mathcal{DA}_\tau \wedge \\ \quad \left[\text{no } \boxed{\text{store } l_{\mathsf{p}-j+\pi}} \text{ occurs in } m_w\right] \\ 3.\ l_{p-j+\pi} \not\rightsquigarrow^{F} s_0 \in I_2 \end{array}\right\}$$
$$\cup\left\{s_{j-\pi} \not\rightsquigarrow^{\mathcal{F}} b \,\middle|\, b \in \mathsf{dom}(\tau') \setminus \{s_{j-\pi}\} \wedge \langle s_{j-\pi}, b\rangle \notin \mathcal{MR}_{\tau'}\right\}$$
$$\cup\left\{s_{j-\pi} \not\rightsquigarrow^{F} b \,\middle|\, \begin{array}{l} 1.\ b \in \mathsf{dom}(\tau') \setminus \{s_{j-\pi}\} \wedge \langle s_{j-\pi}, b\rangle \in \mathcal{MR}_{\tau'} \\ 2.\ \exists j - \pi \leq p < j \,|\, (s_p, b) \in \mathcal{DA}_\tau \wedge \\ \quad \left[\text{no } \boxed{\text{store } l_{\mathsf{p}-j+\pi}} \text{ occurs in } m_w\right] \\ 3.\ s_0 \not\rightsquigarrow^{F} l_{p-j+\pi} \in I_2 \end{array}\right\}$$
$$\cup\ \Pi^{\#13}(I_1, I_2, j - \pi)$$

#15 call $m_1 \ldots m_k$ (exceptional)

$$\lambda I_1.\lambda I_2.\Pi^{\#13}(I_1, I_2, 0) \cup \left\{s_0 \not\rightsquigarrow^{\mathcal{F}} s_0\right\}$$
$$\cup\left\{a \not\rightsquigarrow^{\mathcal{F}} s_0 \,\middle|\, a \in \mathsf{dom}(\tau') \setminus \{s_0\} \wedge \langle a, s_0\rangle \notin \mathcal{MR}_{\tau'}\right\}$$
$$\cup\left\{s_0 \not\rightsquigarrow^{\mathcal{F}} b \,\middle|\, b \in \mathsf{dom}(\tau') \setminus \{s_0\} \wedge \langle s_0, b\rangle \notin \mathcal{MR}_{\tau'}\right\}$$
$$\cup\left\{s_0 \rightsquigarrow \not\emptyset^{\mathcal{F}} \,\middle|\, \forall a \in \mathsf{dom}(\tau') \setminus \{s_0\}, \langle s_0, a\rangle \notin \mathcal{MR}_{\tau'}\right\}$$

Example 6. Let I_X the abstract set of the node X with in-degree greater than 1 ($k \geqslant 1$); let $I_1, \ldots, I_k$ be the abstract sets of its predecessors $Y_1, \ldots, Y_k$ and $\Pi_1, \ldots, \Pi_k$ the propagation rules associated to the arcs linking $Y_1, \ldots, Y_k$, respectively, to node X. Then we can simply calculate I_X as the intersection of the normalized propagation rules *i.e.*, $I_X = \Phi_1(I_1) \cap \cdots \cap \Phi_k(I_k)$. Example 11 shows how to handle nodes with in-degree greater than 1.

In the following lemma we are going to show that the application of the concretization map γ both on the propagation rules Π and on the normalized counterpart Φ leads to the same result. In other words the set of states remains unchanged.

Lemma 1. *Let $\tau \in \mathcal{T}$, $I \subseteq \mathsf{A}_\tau$, Π be a propagation rule and Φ the corresponding normalized rule. Then $\gamma_\tau(\Pi(I)) = \gamma_\tau(\Phi(I))$.*

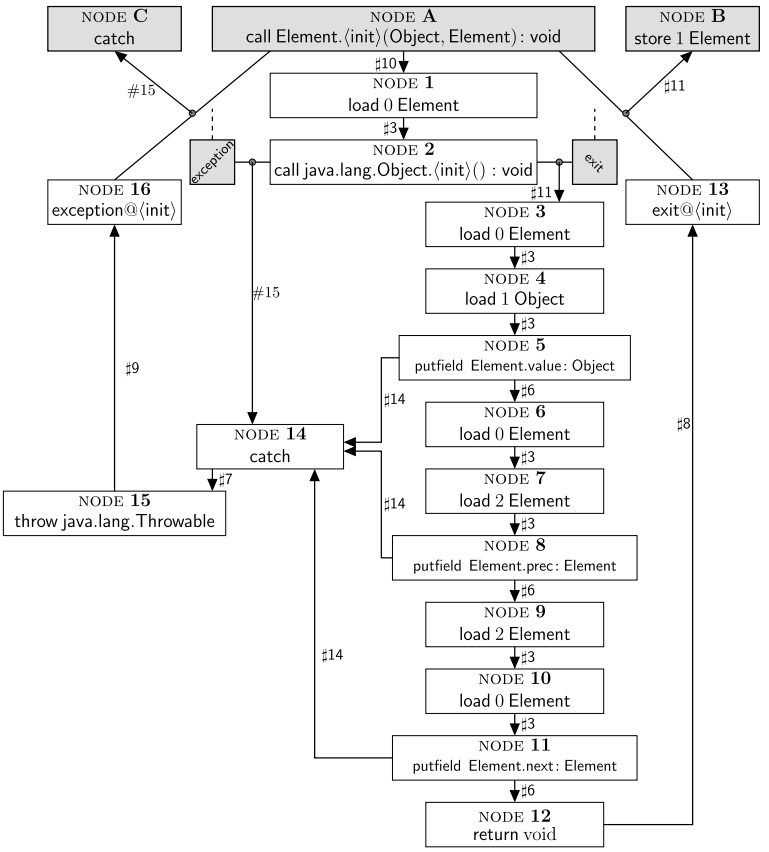


Fig. 4. The ACG of the CFG in Fig. 1b.

Proof. $\subseteq$) By Definition 11 we have: $\Pi(I) \subseteq \mathsf{norm}(\Pi(I)) = \Phi(I)$. Then, since γ_τ is part of a Galois connection, it preserves the order *i.e.*, $\gamma_\tau(\Pi(I)) \subseteq \gamma_\tau(\Phi(I))$.

$\supseteq$) We are going to prove it for the unreachability tokens; for the non-cyclical ones we can follow the same reasoning. Let $\sigma \in \gamma_\tau(\Phi(I))$. Then, by Definition 9, we have that there exists $F' \subseteq \mathcal{F}$ such that $a \not\rightsquigarrow_\sigma^{F'} b$ and $F \subseteq F'$ such that $a \not\rightsquigarrow^F b \in \Phi(I)$. By Definition of Normalized Propagation Rules (Definition 11), it also exists a maximal set $\overline{F}$ such that $F \subseteq \overline{F}$ and $a \not\rightsquigarrow^{\overline{F}} b \in \Phi(I)$. Since every set of fields related to an abstract token in $\Phi(I)$ must not be larger than the concrete counterpart (Definition 9), we also have $\overline{F} \subseteq F'$ and, therefore, $\sigma \in \gamma_\tau(a \not\rightsquigarrow^{\overline{F}} b)$. Since $a \not\rightsquigarrow^{\overline{F}} b$ is also inside $\Pi(I)$, we can conclude $\sigma \in \gamma_\tau(\Pi(I))$. $\square$

We are now able to define how to build the solution of our ACG with respect to the unreachability and non-cyclicity properties in order to obtain the most precise abstract set of tokens related to our properties for every node of it.

Definition 12 *(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 normalized propagation rule Φ of the arcs is satisfied *i.e.*, for every arc from nodes $n_1, \ldots, n_k$ to n' the condition $\Phi(J_{n_1}, \ldots, J_{n_k}) \supseteq J_{n'}$ 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.* the partial order $\supseteq$.

A maximal solution has to exist since all propagation rules are monotonic *w.r.t.* set inclusion $\supseteq$: it can hence be computed by starting from the complete approximation for every node *i.e.*, $a \not\rightsquigarrow^{\mathcal{F}} b$ and $a \rightsquigarrow^{\emptyset_{\mathcal{F}}} b$ for every $a, b \in \mathsf{dom}(\tau)$ of that node, and by propagating this approximation along the arcs, until the greatest fixpoint is reached.

Let now explain the propagation rules that we introduced. In order to provide some examples, we also use the Abstract Constraint Graph in Fig. 4. We note that some nodes are in gray: three nodes (A, B, C) of a caller of this constructor and two nodes of the callee of call `java.lang.Object.⟨init⟩ : void`, which invokes the superclass ⟨*init*⟩ method, in this case `Object`. Each arc is decorated with the number of its associated normalized propagation rule. Finally, we note that the graph for the whole program includes other nodes and arcs; in Fig. 4 is shown only the relevant subgraph of its second

constructor. In [12], we present the whole execution of our analysis under this ACG, while here we provide just the relevant examples to better explain the propagation rules.

Sequential arcs link an instruction to its immediate successor. Arc **#1**, starting from a node corresponding to a dup t and going to the node corresponding to its immediate successor in the Control Flow Graph, states that the information of the node remains unchanged at its successor's node as well: $\lambda I.I$.

Furthermore, since the new top of the stack, s_j, is an alias of the old one, s_{j-1}, its unreachability and non-cyclicity information is the same as s_j: $I[s_{j-1} \mapsto s_j]$. Finally, we have to add the new unreachability information between these two variables: since they are alias, we add two new unreachability relations between s_{j-1} and s_j *w.r.t.* the same set of fields of the unreachability relations between s_{j-1} and itself.

Arc **#2** starts from a node corresponding either to new κ or to const z. In both cases, a new *fresh* variable is pushed on top of the stack. It means that this new variable, s_j, is only reachable from itself and hence, in addition to the information provided by the previous node ($\lambda I.I$), we have to add new unreachability relations *w.r.t.* all program fields between s_j and all the other variables in dom(τ): $\{a \not\rightsquigarrow^{\mathcal{F}} s_j, s_j \not\rightsquigarrow^{\mathcal{F}} a | a \in dom(\tau)\}$.

Furthermore, we add the non-cyclicity information of the new top of the stack, $s_j \rightsquigarrow^{\emptyset\mathcal{F}}$, and also the unreachability information between this new variable and itself, $s_j \not\rightsquigarrow^{\mathcal{F}} s_j$; in both cases the set of fields is $\mathcal{F}$ because s_j is fresh and, therefore, no path can exist starting or ending at the location bound to it.

Arc **#3**, starting from a node corresponding to load k t, has been built through the same reasoning of the propagation rule #1 with l_k instead of s_{j-1}.

Arc **#4**, starting from a node corresponding to store k t, states that the information that does not relate to l_k and s_{j-1} (the old top of the stack) remains unchanged. Instead, the information on these two variables is replaced with information on l_k, having the same unreachability and non-cyclicity properties of the old top of the stack: $(a \not\rightsquigarrow^F b)[s_{j-1} \mapsto l_k]$ and $(c \rightsquigarrow^{\emptyset F})[s_{j-1} \mapsto l_k]$.

Arc **#5**, starting from a node corresponding to catch, excp_is K, ifne t or ifeq t, states that these instructions do not change the unreachability and non-cyclicity information. That is because the state after their execution is the same as before.

Arc **#6**, starting from a node corresponding to getfield $\kappa.f$: t, is more complicated. First of all we exploit two other static analyses: possible reachability and definite aliasing. Since getfield $\kappa.f$: t replaces the old top of the stack, s_{j-1}, with the value of its field $\kappa.f$: t, all reachability properties that do not consider s_{j-1} are still valid after its execution, as expressed in the first set of the corresponding rule. The reachability information between this new variable and itself is inferred by checking whether it is aliased to another variable: $(a, s_{j-1}.(\kappa.f : \mathsf{t})) \in \mathcal{DA}_\tau$. In that case, as shown in its sixth set, the unreachability information of s_{j-1} is the same as that of the variable it is alias of. Additionally, for all variables in dom(τ) $\setminus s_{j-1}$ that may not reach this field or that may not be reached by this field, we can add the unreachability information with the new variables *w.r.t.* all program fields. Instead, the variables that may reach the field of s_{j-1} are treated in the second set: if we found a relation $a \not\rightsquigarrow^F b \in I$ where b is alias to the field of s_{j-1} *i.e.*, $(b, s_{j-1}.(\kappa.f : \mathsf{t})) \in \mathcal{DA}_\tau$, we obtain *safety* information about the new top of the stack and then we can use it to produce the new token $a \not\rightsquigarrow^F s_{j-1}$. Regarding the variables that may be reached by this field (fifth set) we maintain all the unreachability information regarding s_{j-1} and we also add all the other fields whose type cannot be reached by the type of our field $\kappa.f$: t: $F' = F \cup \{\kappa_k.f_k : \mathsf{t}_k \in \mathcal{F} | \mathsf{t}_k \notin \mathsf{T(t)}\}$. That is because there could be other paths in the heap that involve both the old top of the stack and b without having $\kappa.f$: t inside them. Therefore, since F does not contain the fields of these paths, in order to be more precise, we can add them in F', the set stating the unreachability information between the new top of the stack and b. Finally, the last set, with the non-cyclicity information of the new top of the stack, is built by following the same reasoning just explained for the fifth set.

Arc **#7**, 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 remain 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 $\kappa.f$: 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 7. Consider nodes 11, 12 from Fig. 4, and suppose that the unreachability and non-cyclicity information at node 11 is

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

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

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

We note that field 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 rule #7: $F_{l_2,l_0} = \{F_{l_2,l_0} \cap F_{l_2,s_0} \cap F_{s_1,l_0}\} \setminus \{\mathsf{El}.next\} = \{\mathsf{El}.value, \mathsf{El}.prec\}$. 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 Fig. 1a, during the execution of the second constructor a new cycle between the two locations bound to this and prec is actually built since this is linked to prec through field El.*prec*, while prec is linked to this through field El.*next*. Hence fields El.*prec* and 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 7, they must not be in $F_{\mathsf{this}} = F_{l_0}$ nor in $F_{\mathsf{prec}} = F_{l_2}$.

Final arcs feed nodes exit@m or exception@m for each method or constructor of m. The former (respectively the latter) contains the information present in all states at a non-exceptional (respectively exceptional) end of m. Hence, exit@m is the sink of the arcs starting from the return t bytecodes inside m. The propagation rule states that either the stack is emptied at the end of execution of m *i.e.*, when t = void (rule **#8** is applied) or only one element survives *i.e.*, the returned value (rule **#9**).

Similarly, exception@m is the sink of a bytecode instruction throw κ with no exception handler in m (*i.e.*, not followed by a catch inside m). The rule is the same as return t with t ≠ void (rule #9) since only a stack element, the topmost s_{j-1}, survives and it is renamed into the exception object s_0. We observe that only instructions throw κ are allowed to throw an exception to the caller since, in our representation of the code as basic blocks, all other instructions that might throw an exception are always linked to an exception handler, possibly minimal (as the three putfield $\kappa.f$: t in Fig. 4).

Example 8. Consider nodes 12 and 13 in Fig. 4, and suppose that the unreachability and non-cyclicity approximation at node 12 is

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

Nodes 12 and 13 are linked by a final arc with propagation rule #8. By Definition 10, I_{13} contains all the tokens of I_{12} containing no stack variable, and since $j_{12} = 0$ we conclude that $I_{13} = I_{12}$.

Parameter passing arcs link every node corresponding to a method call to the node corresponding to the first bytecode instruction of method(s) m_w that might be called here. Propagation rule **#10** simply states that the actual parameters of m_w, held in the stack variables $s_{j-\pi}, \ldots, s_{j-1}$, are renamed into its formal parameters *i.e.*, the local variables $l_0, \ldots, l_{\pi-1}$. No other variables exist at the beginning of m_w.

Example 9. Consider, for instance, nodes A and 1 in Fig. 4. We assume that the unreachability and non-cyclicity information at node A is:

$$I_A = \{l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\}, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\},$$
$$s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\}, s_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\}, s_3 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\}\}$$
$$\cup \{\{l_0, l_1, s_0, s_1, s_2, s_3\} \rightsquigarrow^{\not\emptyset\mathcal{F}}\}$$

Nodes A and 1 are linked by a parameter passing arc with propagation rule #10. We have $j_A = 4$ and $\pi = 3$ (stack element s_1, s_2 and s_3 hold the actual parameters of a call to this constructor). Hence, by Definition 10, it must be:

$$\left\{(a \not\rightsquigarrow^{F} b)\begin{bmatrix} s_1 \mapsto l_0 \\ s_2 \mapsto l_1 \\ s_3 \mapsto l_2 \end{bmatrix} \;\middle|\; a \not\rightsquigarrow^{F} b \in I_A \wedge a, b \in \{s_1, s_2, s_3\}\right\} \cup \left\{(c \rightsquigarrow^{\not\emptyset F})\begin{bmatrix} s_1 \mapsto l_0 \\ s_2 \mapsto l_1 \\ s_3 \mapsto l_2 \end{bmatrix} \;\middle|\; c \rightsquigarrow^{\not\emptyset F} \in I_A \wedge c \in \{s_1, s_2, s_3\}\right\} \supseteq I_1$$

In particular, we obtain:

$$I_1 = \left\{ l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\}, l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\}, \{l_0, l_1, l_2\} \rightsquigarrow^{\not\emptyset\mathcal{F}} \right\}$$

Side-effect multi-arcs link the nodes $\boxed{\text{call } m_1 \dots m_k}$ and $\boxed{\text{exit@}m_w}$ to the next instruction after the non-exceptional ending of method m_w only if it does not return any value (void). Furthermore, since the propagation rule **#13** of this arc determines unreachability and non-cyclicity information of the caller's variables after the execution of method m_w, it is also used as part of the propagation rules #14 and #15. For this reason, we add the (meta-)parameter *max* as input: $max = j - \pi$ when I_2 is the information set of a node $\boxed{\text{exit@}m_w}$, while $max = 0$ when I_2 is the information set of a node $\boxed{\text{exception@}m_w}$. Finally, before explaining the rule, we remind that, when we use τ' as subscript of another static analysis result (e.g. $\mathcal{MR}_{\tau'}$), we mean that it is the set after the abstract execution of that node and we can exploit this *forward information* because we have assumed that these analyses are processed asynchronously.

Rule #13 states that between all the variables pairs $\langle a, b\rangle$ such that a cannot reach b after the execution of method m_w the unreachability information is $a \not\rightsquigarrow^{\mathcal{F}} b$.

On the other hand, all the unreachability tokens before the method execution are still valid (first set), provided that the first variable a cannot *share* with any actual parameters at that program point: $\forall j - \pi \leqslant p < j, (a, s_p) \notin \mathcal{MS}_\tau$. In this way, inside m_w we are sure that no paths through locations which share with a are modified and hence its unreachability information remains unchanged. The same reasoning about non-cyclicity tokens leads to the second set of the rule.

Regarding the tokens coming from I_2 we exploit them to add new unreachability and non-cyclicity information that are available at the end of method m_w. We add new unreachability tokens $a \not\rightsquigarrow^{F} b$ if the following conditions are satisfied:

1. a may reach b after the execution of the method: $\langle a, b\rangle \in \mathcal{MR}_{\tau'}$;
2. a and b must be alias of at least a couple of actual parameters s_{p_a}, s_{p_b} respectively: $(a, s_{p_a}), (b, s_{p_b}) \in \mathcal{DA}_\tau$;
3. the formal parameters $l_{p_a - j + \pi}$ and $l_{p_b - j + \pi}$, corresponding to these two actual parameters, are not reassigned inside m_w through a store $l_{p_a - j + \pi}$ or store $l_{p_b - j + \pi}$.

If these conditions are satisfied the unreachability information between $l_{p_a - j + \pi}$ and $l_{p_b - j + \pi}$ in I_2 (*i.e.*, at the end of the callee method m_w) corresponds to the unreachability information of a and b in the caller: $I_2 \ni l_{p_a - j + \pi} \not\rightsquigarrow^{F} l_{p_b - j + \pi} = a \not\rightsquigarrow^{F} b$.

For the non-cyclicity tokens in I_2, we add the new information in a way similar to that used for unreachability, but considering only one variable in both the caller and callee methods.

Example 10. Consider, for instance, nodes A and 13 in Fig. 4. The approximation information of these nodes, as already presented above, is:

$$\begin{aligned} I_A = \big\{ & l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_2, s_3\}, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\}, \\ & s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\}, s_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\}, s_3 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0, s_1, s_3, s_3\} \big\} \\ & \cup \big\{ \{l_0, l_1, s_0, s_1, s_2, s_3\} \rightsquigarrow^{\not\emptyset\mathcal{F}} \big\} \end{aligned}$$

and

$$\begin{aligned} I_{13} = \big\{ & l_0 \not\rightsquigarrow^{\mathcal{F}} l_0, l_0 \not\rightsquigarrow^{\{\text{El}.prec, \text{El}.next\}} l_1, l_0 \not\rightsquigarrow^{\{\text{El}.value, \text{El}.next\}} l_2, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, l_2\}, \\ & l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_1, l_2\}, l_2 \not\rightsquigarrow^{\{\text{El}.value, \text{El}.prec\}} l_0, l_1 \rightsquigarrow^{\not\emptyset\mathcal{F}}, \{l_0, l_2\} \rightsquigarrow^{\not\emptyset\{\text{El}.value\}} \big\}. \end{aligned}$$

Consider the side-effect multi-arc linking nodes A and 13 with node B; we illustrate the application of rule #13 on the presence of the sharing, reachability and aliasing information $\mathcal{MS}_{\tau_S}, \mathcal{MR}_{\tau_A}, \mathcal{MR}_{\tau_B}, \mathcal{DA}_{\tau_A}$. We remember that there are $\pi = 3$ parameters: the implicit this and two parameters of type Object and Element. Since the return type of the constructor is void we obtain $i_B = 3$ while $j_B = 1$, and for each variable $v \in \text{dom}(\tau_B) = \{l_0, l_1, l_2, s_0\}$, $\tau_B(v) = \tau_A(v)$. By the application of the propagation rule #13 we obtain:

$$\begin{aligned} I_B = \big\{ & l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1, s_0\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0, l_1\}, l_1 \not\rightsquigarrow^{\{\text{El}.value, \text{El}.prec\}} s_0, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0, s_0\}, \\ & s_0 \not\rightsquigarrow^{\{\text{El}.value, \text{El}.next\}} l_1, l_0 \rightsquigarrow^{\not\emptyset\mathcal{F}}, \{l_1, s_0\} \rightsquigarrow^{\not\emptyset\{\text{El}.value\}} \big\} \end{aligned}$$

First of all we have $max = j_A - \pi = 4 - 3 = 1$ *i.e.*, regarding the stack elements, we have only to take into account the information about s_0. For the tokens that remain unchanged after the execution of the method, we have:

$$\left\{ \begin{array}{l} l_1 \not\rightsquigarrow^{\mathcal{F}} l_0, s_0 \not\rightsquigarrow^{\mathcal{F}} l_0, \\ l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_1, s_0\} \end{array} \right\} \subseteq \left\{ a \not\rightsquigarrow^{\mathcal{F}} b \;\middle|\; a, b \in \text{dom}(\tau') \wedge \langle a, b\rangle \notin \mathcal{MR}_{\tau_B} \right\}$$

$$\text{with } \mathcal{MR}_{\tau_B} = \left\{ \langle s_0, l_1\rangle, \langle l_1, s_0\rangle, \langle l_0, l_0\rangle, \langle l_1, l_1\rangle, \langle s_0, s_0\rangle \right\}$$

$$\left\{l_0 \not\rightsquigarrow^{\mathcal{F}} l_0, l_0 \rightsquigarrow^{\not\emptyset\mathcal{F}}\right\} \subseteq \left\{ \begin{array}{l} a \not\rightsquigarrow^{F_1} b, \\ a \rightsquigarrow^{\not\emptyset F_2} \in I_A \end{array} \;\middle|\; \begin{array}{l} a, b \in \{l_0, l_1, s_0\} \wedge \\ \forall 1 \leqslant p < 4, (a, s_p) \notin \mathcal{MS}_{\tau_A} \end{array} \right\}$$

$$\text{with } \mathcal{MS}_{\tau_A} = \left\{(s_0, s_1), (l_1, s_3), (l_0, l_0), (l_1, l_1), (s_0, s_0), (s_1, s_1), (s_2, s_2), (s_3, s_3)\right\}$$

For the unreachability and non-cyclicity tokens that derive from the information available at the end of method m_w, we have:

$$\left\{ \begin{array}{l} l_1 \not\rightsquigarrow^{\mathcal{F}} l_1, s_0 \not\rightsquigarrow^{\mathcal{F}} s_0, \\ l_1 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.prec\}} s_0, \\ s_0 \not\rightsquigarrow^{\{\mathsf{El}.value, \mathsf{El}.prec\}} l_1 \end{array} \right\} \subseteq \left\{ a \not\rightsquigarrow^{F} b \;\middle|\; \begin{array}{l} 1.\ a, b \in \{l_0, l_1, s_0\} \wedge \langle a, b\rangle \in \mathcal{MR}_{\tau_B} \\ 2.\ \exists 1 \leqslant p_a, p_b < 4 | (a, s_{p_a}), (b, s_{p_b}) \in \mathcal{DA}_{\tau_A} \wedge \\ \quad [\text{ no } \boxed{\mathsf{store}\, l_{p_a - j + \pi}} \text{ nor } \boxed{\mathsf{store}\, l_{p_b - j + \pi}} \text{ occurs in } m_w] \\ 3.\ l_{p_a - j + \pi} \not\rightsquigarrow^{F} l_{p_b - j + \pi} \in I_{13} \end{array} \right\}$$

$$\text{with } \mathcal{DA}_\tau A = \left\{(s_0, s_1), (l_1, s_3), (l_0, l_0), (l_1, l_1), (s_0, s_0), (s_1, s_1), (s_2, s_2), (s_3, s_3)\right\}$$

and

$$\left\{\{l_1, s_0\} \rightsquigarrow^{\not\emptyset\{\mathsf{El}.value\}}\right\} \subseteq \left\{ c \rightsquigarrow^{\not\emptyset F} \;\middle|\; \begin{array}{l} 1.\ c \in \{l_0, l_1, s_0\} \\ 2.\ \exists 1 \leqslant p < 4 | (c, s_p) \in \mathcal{DA}_{\tau_A} \wedge \\ \quad [\text{ no } \boxed{\mathsf{store}\, l_{p - j + \pi}} \text{ occurs in } m_w] \\ 3.\ l_{p - j + \pi} \rightsquigarrow^{\not\emptyset F} \in I_{13} \end{array} \right\}$$

Return value multi-arcs link the nodes $\boxed{\mathsf{call}\ m_1 \ldots m_k}$ and $\boxed{\mathsf{exit@}m_w}$ to the next instruction after the non-exceptional ending of method m_w only if it returns a value. The rule #14 extends #13, since also in this case in the caller we have side-effects due to the method execution, but now we have also to handle the unreachability and non-cyclicity tokens related to the returned value $s_{j-\pi}$. The returned value in the callee corresponds to the stack variable s_0 and hence its information at node $\boxed{\mathsf{exit@}m_w}$.

First of all the non-cyclicity information of $s_{j-\pi}$ in the caller is the same as that of s_0 at the end of the callee. Furthermore the rule states that also the unreachability information between $s_{j-\pi}$ and itself is the same of that of s_0 and itself in the callee.

For the other variables after the call and different from the returned value *i.e.*, $a \in \mathsf{dom}(\tau') \setminus \{s_{j-\pi}\}$, if they do not reach $s_{j-\pi}$ after the call we can add the triples such that $a \not\rightsquigarrow^{\mathcal{F}} s_{j-\pi}$, as shown in the second set. If they are not reached by $s_{j-\pi}$ we can add the triples such that $s_{j-\pi} \not\rightsquigarrow^{\mathcal{F}} a$, as shown in the fourth set.

For the variables that may reach $s_{j-\pi}$ after the method execution *i.e.*, $\langle a, s_{j-\pi}\rangle \in \mathcal{MR}_{\tau'}$, we add unreachability tokens if the following conditions are satisfied:

1. a must be alias of at least an actual parameter s_{p_a}: $(a, s_{p_a}) \in \mathcal{DA}_\tau$;
2. the formal parameter corresponding to this actual parameter, $l_{p_a - j + \pi}$, is not reassigned inside m_w through a $\mathsf{store}\, l_{p_a - j + \pi}$.

If these conditions are satisfied the unreachability information between $l_{p_a - j + \pi}$ and s_0 in I_2 *i.e.*, at the end of the callee method m_w, corresponds to the unreachability information of a and $s_{j-\pi}$ in the caller: $I_2 \ni l_{p_a - j + \pi} \not\rightsquigarrow^{F} s_0 = a \not\rightsquigarrow^{F} s_{j-\pi}$.

For the variables that may be reached by $s_{j-\pi}$ after the method execution *i.e.*, $\langle s_{j-\pi}, b\rangle \in \mathcal{MR}_{\tau'}$, we add unreachability tokens if the following conditions are satisfied:

1. b must be alias of at least an actual parameter s_{p_b}: $(s_{p_b}, b) \in \mathcal{DA}_\tau$;
2. the formal parameter corresponding to this actual parameter, $l_{p_b - j + \pi}$, is not reassigned inside m_w through a $\mathsf{store}\, l_{p_b - j + \pi}$.

If these conditions are satisfied the unreachability information between s_0 and $l_{p_b - j + \pi}$ in I_2 *i.e.*, at the end of the callee method m_w, corresponds to the unreachability information of $s_{j-\pi}$ and b in the caller: $I_2 \ni s_0 \not\rightsquigarrow^{F} l_{p_b - j + \pi} = s_{j-\pi} \not\rightsquigarrow^{F} b$.

Exceptional arcs link every instruction that might throw an exception to the catch at the beginning of their exception handler(s).

Rule **#11** is identical to #9 since it deals with the same source node, $\boxed{\mathsf{throw}\ \kappa}$, but it is applied in the case of an exceptional execution.

Rule **#12** deals with all other bytecode instructions that might throw an exception ($\mathsf{new}\,\kappa$, $\mathsf{getfield}\,\kappa.f : \mathsf{t}$, $\mathsf{putfield}\,\kappa.f : \mathsf{t}$): it states that the stack disappears but the unreachability between local variables remains unaffected. Moreover a new stack element containing a new fresh variable s_0 of type $\kappa \leqslant \texttt{Throwable}$ is created. It means that this new variable, s_0, is only reachable from itself and hence, in addition to the information provided by the previous node ($\lambda I.I$), we have to add new unreachability information *w.r.t.* all program fields between s_0 and all the other variables in L: $\{a \not\rightsquigarrow^{\mathcal{F}} s_0, s_0 \not\rightsquigarrow^{\mathcal{F}} a | a \in L\}$. Finally, we add the non-cyclicity information of the new top of the stack, $s_0 \rightsquigarrow^{\not\emptyset\mathcal{F}}$, and also the unreachability information

between this new variable and itself, $s_0 \not\rightsquigarrow^{\mathcal{F}} s_0$; in both cases the set of fields is $\mathcal{F}$ because s_0 is fresh and, therefore, it is isolated from the rest of the heap.

Rule **#15** states a pessimistic assumption about the exceptional states after a method call. The peculiarity of this rule is the handling of unreachability and non-cyclicity tokens about the new variable s_0 of type $\kappa \leqslant \texttt{Throwable}$: we deal with s_0 as in rule #12 with the difference that we check if the other variables may not reach nor be reached from s_0. That is because this variable, bound to an exception object, may reach or be reached by another exception.

Example 11. Consider nodes 2, 5, 8, 11 and 14 in Fig. 4. We assume that the approximation information of the first three nodes are:

$$I_2 = \{l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0\}, l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0\}, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0\}, \{l_0,l_1,l_2,s_0\} \rightsquigarrow^{\emptyset_{\mathcal{F}}}\}$$

$$I_5 = \{l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0,s_1\}, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0,s_1\}, l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0,s_1\},$$
$$s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0,s_1\}, s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0,s_1\}, \{l_0,l_1,l_2,s_0,s_1\} \rightsquigarrow^{\emptyset_{\mathcal{F}}}\}$$

$$I_8 = \{l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_2,s_0,s_1\}, l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec,\mathsf{El}.next\}} l_1, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0,s_1\}, l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0,s_1\},$$
$$s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_2,s_0\}, s_0 \not\rightsquigarrow^{\{\mathsf{El}.prec,\mathsf{El}.next\}} l_1, s_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0,s_1\}, \{l_0,l_1,l_2,s_0,s_1\} \rightsquigarrow^{\emptyset_{\mathcal{F}}}\}$$

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

These three nodes are all linked to node 14 through an exceptional arc with the same propagation rule #12. Thus the node 14 has in-degree greater than 1 and hence to establish the set I_{14} we have to compute the least upper bound of the sets obtained by applying the normalized propagation rules of these arcs *i.e.*,

$$I_{14} \subseteq \Phi^{\#12}(I_2) \cap \Phi^{\#12}(I_5) \cap \Phi^{\#12}(I_8) \cap \Phi^{\#12}(I_{11})$$

The resulting set (not normalized) is:

$$I_{14} = \{l_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0,s_0\}, l_0 \not\rightsquigarrow^{\{\mathsf{El}.prec,\mathsf{El}.next\}} l_1, l_0 \not\rightsquigarrow^{\{\mathsf{El}.value,\mathsf{El}.next\}} l_2, l_1 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0,s_1\},$$
$$l_2 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0\}, s_0 \not\rightsquigarrow^{\mathcal{F}} \{l_0,l_1,l_2,s_0\}, \{l_0,l_1,l_2,s_0\} \rightsquigarrow^{\emptyset_{\mathcal{F}}}\}$$

We note that this set is exactly $I_{14} \subseteq \Pi^{\#12}(I_{11})$, since I_{11} is, among the sets of the three $\mathsf{putfield}\,\kappa.f : \mathsf{t}$ nodes, that with the least number of tokens concerning the local variables.

6. Soundness

Each propagation rule Φ is proved formally correct in [12] by using the standard technique of abstract interpretation: namely, by letting *ins* be a bytecode instruction, Φ a propagation rule and $I \in \mathsf{A}_\tau$, we have proved the soundness of Φ by showing that

$$ins\big(\gamma_\tau(I)\big) \subseteq \gamma_\tau\big(\Phi(I)\big)$$

i.e., that for each $x \not\rightsquigarrow^{F} y \in \Phi(I)$ we have a state σ' such that $x \not\rightsquigarrow^{F'}_{\sigma'} y \wedge F \subseteq F'$ and for each $z \rightsquigarrow^{\emptyset_F} \in \Phi(I)$ we have another state σ'' such that $z \rightsquigarrow^{\emptyset_{F'}}_{\sigma''} y \wedge F \subseteq F'$. In [12] we have also proved the soundness of the whole analysis with respect to the small-step operational semantics for Java bytecode previously explained.

Theorem 1 *(Soundness). Let* $\langle b_{first(\texttt{main})} \parallel \xi \rangle \Rightarrow^* \left\langle \begin{array}{|c|}\hline \text{ins} \\ \text{rest} \\ \hline \end{array} \begin{array}{l} \rightarrow b_1 \\ \cdots \\ \rightarrow b_m \end{array} \parallel \sigma \right\rangle :: a$ *be an execution of our operational semantics, from the first bytecode instruction of the method* `main` *i.e., first*(`main`) *and an initial state* ξ *containing no reachability between variables. Suppose that* $\sigma \in \Sigma_\tau$ *and let* $I_{\mathsf{ins}} \in A_\tau$ *be the approximation on the actual unreachability and non-cyclicity w.r.t. a set of fields at the ACG node corresponding to ins. Then,* $\sigma \in \gamma_\tau(I_{\mathsf{ins}})$.

7. 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 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 in 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 [10], respectively; on the other hand, we exploit pre-processed information of possible sharing ($\mathcal{MS}_\tau$), possible reachability ($\mathcal{MR}_\tau$) and definite aliasing ($\mathcal{DA}_\tau$) analyses, for better precision.

We still miss an implementation of the analysis, which is relatively complex because of the complexity of the abstract domain. However, it would support evidence of its usefulness and practicability on real examples of programs.

Acknowledgements

The first author was partially supported by the *Deutsche Forschungsgemeinschaft* (DFG) under Grant KU 1434/6-2 within the priority programme 1496 "Reliably Secure Software Systems – RS^3".

References

[1] P. Cousot, R. Cousot, 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] S. Genaim, D. Zanardini, Reachability-based acyclicity analysis by abstract interpretation, Theor. Comp. Sci. 474 (2013) 60–79.
[3] M. Hind, Pointer analysis: haven't we solved this problem yet?, in: Proceedings of the 1st Program Analysis for Software Tools and Engineering (PASTE'01).
[4] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Java Series, Addison-Wesley, 1999.
[5] F. Logozzo, M. Fähndrich, 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] Đ. Nikolić, F. Spoto, Definite expression aliasing analysis for Java bytecode, in: Proceedings of the 9th International Colloquium on Theoretical Aspects of Computing (ICTAC'12).
[7] Đ. Nikolić, F. Spoto, Reachability analysis of program variables, in: Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR'12).
[8] J. Palsberg, M.I. Schwartzbach, Object-oriented type inference, in: Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91).
[9] David J. Pearce, Paul H.J. Kelly, Chris Hankin, 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).
[10] S. Rossignoli, F. Spoto, 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).
[11] M. Sagiv, T. Reps, R. Wilhelm, Parametric shape analysis via 3-valued logic, ACM Trans. Program. Lang. Syst. 24 (3) (2002) 8:1–8:70.
[12] E. Scapin, Field-sensitive unreachability and non-cyclicity analysis, Master's thesis, University of Verona, Italy, 2012.
[13] D.A. Schmidt, Underapproximating predicate transformers, in: Proceedings of the 13th International Conference on Static Analysis (SAS'06).
[14] S. Secci, F. Spoto, Pair-sharing analysis of object-oriented programs, in: Proceedings of the 12th SAS, London, UK, 2005.
[15] F. Spoto, M.D. Ernst, Inference of field initialization, in: Proceedings of the 33rd ICSE, ACM, Waikiki, Honolulu, USA, 2011.
[16] F. Spoto, F. Mesnard, É. Payet, A termination analyzer for Java bytecode based on path-length, ACM Trans. Program. Lang. Syst. 32 (3) (2010) 60–79.