Reachability Analysis of Program Variables

Đurica Nikolić[1,2] and Fausto Spoto[1]

[1] Dipartimento di Informatica, University of Verona
[2] Microsoft Research - University of Trento Centre for Computational and Systems Biology
{durica.nikolic,fausto.spoto}@univr.it

Abstract. A variable v reaches a variable w if there is a path from the memory location bound to v to the one bound to w. This information is important for improving the precision of other static analyses, such as side-effects, field initialization, cyclicity and path-length, as well as of more complex analyses built upon them, such as nullness and termination. We present a provably correct constraint-based reachability analysis for Java bytecode. Our constraint is a graph whose nodes are program points and whose arcs propagate reachability information according to the semantics of bytecodes. The analysis has been implemented in the Julia static analyzer. Experiments that we performed on non-trivial Java and Android programs show a gain in precision due to a reachability information, whose presence also reduces the cost of nullness and termination analyses.

1 Introduction

Static analysis of computer programs allows us to statically gather information about their run-time behavior, making it possible to prove that these programs do not perform illegal operations (such as division by zero or dereference of `null`), do not give rise to erroneous executions (such as infinite loops) or do not divulge information (such as security authorizations or GPS position) in an incorrect way.

Dynamic allocation of objects is heavily used in real life programs. These objects are instantiated on demand, their number is not statically known and they can reference other objects (through *fields*). Such references can be updated at run-time. In this paper we present, formalize and implement a provably correct abstraction of the run-time, dynamically allocated memory, that we call *reachability*. We say that a variable v *reaches* a variable w if w holds an object reachable from v, by following (different objects') fields from the object held in the location bound to v. For instance, after an assignment `v.next.next = w`, we can state that v reaches w. Reachability is distinct from *sharing* i.e., being able to reach a shared object. For instance, after the statement `v.next = w.next`, we can state that v and w share. If v reaches w then v and w share, but the converse might not hold. Hence reachability is more *precise*, i.e., it induces a finer, more concrete abstraction of the computational states than sharing analysis. Our analysis is constraint-based: constraints are built from the syntax of the program and their solution is a correct approximation of reachability. A companion paper [14] includes full definitions and proofs.

B. Gramlich, D. Miller, and U. Sattler (Eds.): IJCAR 2012, LNAI 7364, pp. 423–438, 2012.

Reachability has been applied to several static analyses:

Side-Effects Analysis: Side-effects analysis tracks (among other things) which parameters `p` of a method might be affected by its execution in the sense that the method might update a field of an object reachable from `p`. Namely, if the method performs an assignment `a.f=b`, this affects `p` only if `p` reaches `a`. If we used sharing rather than reachability information, that would lead to a loss of precision, since it might be the case that `p` and `a` share but the assignment modifies an object unreachable from `p`.

Field Initialization Analysis: It is often the case that a field is initialized by all of the constructors of its defining class *before being read* by these constructors. Spotting this frequent situation is important for many analyses, including nullness [15,22]. Hence, we want to know whether a field read operation `a=expression.f` inside a constructor can actually read field `f` of the `this` object, being initialized by the constructor. This happens only if `this` reaches `expression`. Again, sharing would be less precise here.

Cyclicity Analysis: An assignment `a.f=b` might make `a` *cyclical* (i.e., point to a cyclical data structure), but only if `b` reaches `a`. Originally, this analysis was built upon sharing information [16], but analysis of reachable variables helps here.

Path-Length Analysis: Path-length is a data structure measure used in termination analysis [23]. It is the maximum number of pointer dereferences that can be followed from a program variable. An assignment `a.f=b` can only modify the path-length of the program variables that share with `a`, according to the original definition of path-length [23]. Reachability analysis improves this approximation, since the path-length of a program variable `v` is actually modified only if `v` *reaches* `a`.

These analyses, among others, are implemented in our Julia tool (`http://www.juliasoft.com`). They are building blocks of larger *tools*, such as a nullness and a termination checker. The former spots where a program might throw a null-pointer exception at run-time; the latter if method calls might diverge. A tool performs its supporting analyses (the *building blocks*) in distinct threads, parallel on multi-core hardware.

Our experiments show that reachability improves side-effects, field initialization and nullness analysis of non-trivial Java and Android programs. However there is no improvement for cyclicity, path-length and termination analysis of the same programs, but only of sample programs from the international termination competition. That is because termination often depends on loops over integer counters rather than on recursion over data structures, as is the case in those samples (probably unusual and artificial). An unexpected effect of reachability is, however, an increase in the speed of both tools.

Reachability analysis belongs to the group of *pointer analyses*, that support other static analyses. Plenty of papers consider them: [9] surveys more than 75 papers. Different properties of pointers give rise to different kinds of pointer analyses: *alias*, *sharing*, *points-to* and *shape* analyses. Possible (definitive) alias analysis discovers the pairs of variables that might (must) point to the same memory location. If two variables are alias, they are also reachable from each other, but the opposite might not hold. Sharing analysis [21] determines whether two variables might ever reach the same object at run-time. Reachability entails sharing, but the opposite, in general, does not hold. Points-to analysis [20,10,11,17,8] computes the objects that a pointer variable might refer to at run-time. Usually, points-to analysis performs a conservative approximation of the heap, which is then used to compute points-to information for the whole program. In [20], points-to graphs are precise approximations of the run-time heap memory and can be used to over-approximate the reachability information. Points-to information is much more concrete than our reachability information. Shape analysis determines heap *shape invariants* [18,19,3,7]. These analyses are quite concrete and capture aliasing and points-to information, as well as other properties such as cyclicity or acyclicity. These are often encoded as first-order logic formulae and theorem provers are used to determine their validity. Reachability can, of course, be abstracted from these very precise approximations of the memory, but we wanted here an analysis that uses the most abstract (i.e., the simplest) domain able to express reachability between variables.

There is also another notion of reachability [13], slightly different from ours. The *reachability predicate* determines whether a memory location reaches another one, usually along *one* particular *field* of *one* particular *data structure*, while our definition of reachable locations deals with *arbitrary fields* of *arbitrary data structures*. That predicate is used in [6,1,4] for abstraction of programs, as one particular case of predicate abstraction [2].

2 Operational Semantics

We present here a formal operational semantics of Java bytecode, inspired by the standard informal semantics [12]. The same semantics is used in [22], while [23] uses its denotational form. Java bytecode is the form of instructions executed by the Java Virtual Machine (JVM). Our formalization is at bytecode level for several reasons: there is a small number of bytecode instructions, compared to varieties of source statements; bytecode lacks complexities such as inner classes; our implementation of reachability analysis is at bytecode level, bringing formalism, implementation and proofs closer.

For simplicity, we assume that the only primitive type is int and that reference types are *classes* containing *instance fields* and *instant methods* only. Our implementation handles all Java types and bytecodes, as well as classes with static fields and methods. We analyze bytecode preprocessed into a control flow graph, i.e., a directed graph of *basic blocks*, with no jumps inside the blocks. $\boxed{\begin{array}{c}\mathsf{ins}\\ \mathit{rest}\end{array}}\begin{array}{l}\rightarrow b_1\\ \cdots\\ \rightarrow b_m\end{array}$ denotes a block of code starting at instruction ins, possibly followed by more bytecodes *rest* and linked to m subsequent blocks $b_1, \ldots, b_m$. Exception handlers start with a catch. A conditional, virtual method call, or selection of an exception handler becomes a block with many subsequent blocks, starting with a *filtering* bytecode such as exception_is K for exception handlers.

Example 1. Fig. 2 shows the basic blocks of the constructor in Fig. 1. There is a branch at the call to the constructor of `java.lang.Object`, that might throw an exception (like every call). If this happens, the exception is first caught and then re-thrown to the caller of the constructor. Otherwise, the execution continues with 2 blocks storing the formal parameters (locals 1 and 2) into the fields of `this` (local 0) and then returns. □

Bytecodes operate on *variables*, which encompass both stack elements and local variables. A standard algorithm [12] infers their static types.

```
public class ListStudent {
  public Student head;
  public ListStudent tail;

  public ListStudent(Student head,
                ListStudent tail) {
    this.head = head;
    this.tail = tail;
  }
}
```

Fig. 1. Our running example

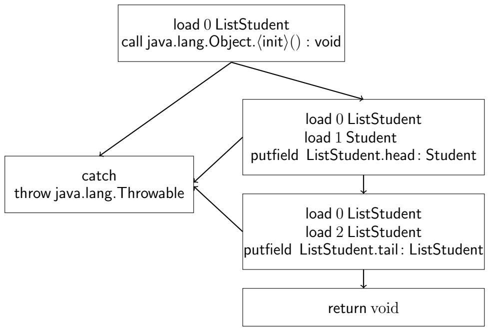

Fig. 2. Representation of the constructor from Fig. 1

Definition 1 (Classes). *The set of* classes $\mathbb{K}$ *of a program is partially ordered w.r.t. the* subclass relation $\leq$*:* $\mathsf{t} \leq \mathsf{t}'$ *if* t *(respectively* t'*) is a subclass (respectively superclass) of* t' *(respectively* t*). Every class has at most one* direct superclass *and an arbitrary number of direct subclasses. A* type *is an element of* $\mathbb{T} = \{\mathsf{int}\} \cup \mathbb{K}$*, ordered by the extension of* $\leq$ *with* $\mathsf{int} \leq \mathsf{int}$*. A class* $\kappa \in \mathbb{K}$ *has* fields $\kappa.f : \mathsf{t}$ *(field* f *of type* $\mathsf{t} \in \mathbb{T}$ *defined in* κ*), where* κ *and* t *are often omitted. We let* $\mathbb{F}(\kappa) = \{\kappa'.f : \mathsf{t}' \mid \kappa \leq \kappa'\}$ *be the fields defined in* κ *or in any of its superclasses. 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* $\mathsf{t} \in \mathbb{T} \cup \{\mathsf{void}\}$*), where* κ*,* $\vec{\mathsf{t}}$*, and* t *are often omitted. Constructors are methods named* init *that return* void*.*

Definition 2 (Type environment). *Let* V *be the set of* variables *from* $L = \{l_0, \ldots, l_m\}$ *(*local *variables) and* $S = \{s_0, \ldots, s_n\}$ *(*stack *variables). A* type environment *is a function* $\tau : V \to \mathbb{T}$*. Its* domain *is written as* $\mathsf{dom}(\tau)$*. The set of all type environments is* $\mathcal{T}$*.*

Definition 3 (State). *A* value *is an element of* $\mathbb{Z} \cup \mathbb{L} \cup \{\mathtt{null}\}$*, where* $\mathbb{L}$ *is an infinite set of* memory locations*. A* state *over* $\tau \in \mathcal{T}$ *is a pair* $\langle\langle l \parallel s\rangle, \mu\rangle$ *where* 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* $\mathsf{dom}(\tau)$*, which grows leftwards, and* μ *is a* memory*, or* heap*, that binds locations to* objects*. The empty stack is denoted by* ε*. We often use another representation for a state:* $\langle\rho, \mu\rangle$*, 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* $o.\kappa$ *(is an* instance *of* $o.\kappa$*) and has an internal environment* $o.\phi$ *that maps every field* $\kappa'.f : \mathsf{t}' \in \mathbb{F}(o.\kappa)$ *into its value* $(o.\phi)(\kappa'.f : \mathsf{t}')$*. A value* v has type t *in* $\langle\rho, \mu\rangle$ *if:* $v \in \mathbb{Z}$ *and* $\mathsf{t} = \mathsf{int}$*, or* $v = \mathtt{null}$ *and* $t \in \mathbb{K}$*, or* $v \in \mathbb{L}$*,* $t \in \mathbb{K}$ *and* $\mu(v).\kappa \leq \mathsf{t}$*. In a state* $\langle\rho, \mu\rangle$ *over* τ*, we require that* $\rho(v)$ *has type* $\tau(v)$ *for any* $v \in \mathsf{dom}(\tau)$ *and* $(o.\phi)(\kappa'.f : \mathsf{t}')$ *has type* t' *for every* $o \in \mathsf{rng}(\mu)$ *(range* μ*) and every* $\kappa'.f : \mathsf{t}' \in \mathbb{F}(o.\kappa)$*. The set of states is* Ξ*. We write* Ξ_τ *when we want to fix the type environment* τ*.*

Example 2. Let $\tau = [l_1 \mapsto \mathsf{ListStudent}; l_2 \mapsto \mathsf{int}; l_3 \mapsto \mathsf{Student}; l_4 \mapsto \mathsf{ListStudent}] \in \mathcal{T}$ and consider the state $\sigma = \langle\rho, \mu\rangle \in \Sigma_\tau$ shown in Fig. 3. The environment ρ maps variables l_1, l_2, l_3 and l_4 to values ℓ_2, 2, ℓ_3 and ℓ_4, respectively; the memory μ maps locations ℓ_2 and ℓ_4 to objects o_2 and o_4 of class ListStudent and location ℓ_3 to object o_3 of class Student. Objects are shown as boxes with a class tag and an internal environment mapping fields to values. For instance, fields head and tail of o_4 contain ℓ_3 and ℓ_2, respectively. □

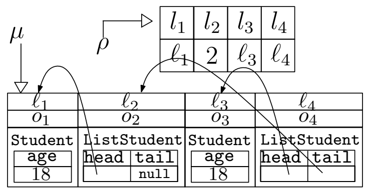

Fig. 3. A JVM state $\sigma = \langle \rho, \mu \rangle$

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 throwing 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 $\underline{\langle\langle l \parallel s\rangle, \mu\rangle}$.

The semantics of an instruction ins is a partial map $ins : \Sigma_\tau \to \Sigma_{\tau'}$ from *initial* to *final* states. The number and type of local variables and stack elements at its start are specified by τ. The formal semantics is given in [14]. We discuss it informally below.

Basic Instructions. $\mathsf{const}\ v$ pushes $v \in \mathbb{Z}$ on the top of the stack. Like any other bytecode except catch, it is defined only when the JVM is in a normal state. The latter starts the exceptional handlers from an exceptional state and is, therefore, undefined on a normal state. $\mathsf{dup\,t}$ duplicates the top of the stack, of type t. $\mathsf{load}\,k\,\mathsf{t}$ pushes on the stack the value of local variable number k, l_k, which must exist and have type t. Conversely, $\mathsf{store}\ k\ \mathsf{t}$ 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 our formalization, conditional bytecodes are used in complementary pairs (such as $\mathsf{ifne\ t}$ and $\mathsf{ifeq\ t}$), at a conditional branch. For instance, $\mathsf{ifeq\ t}$ checks whether the top of the stack, of type t, is 0 when $\mathsf{t} = \mathsf{int}$ or `null` when $\mathsf{t} \in \mathbb{K}$. Otherwise, its semantics is undefined.

Object-Manipulating Instructions. These bytecode instructions create or access objects in memory. $\mathsf{new}\ \kappa$ 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 [12]. $\mathsf{getfield}\ \kappa.f\!:\!\mathsf{t}$ reads the field $\kappa.f : \mathsf{t}$ of a receiver object r popped from the stack, of type κ. $\mathsf{putfield}\ \kappa.f\!:\!\mathsf{t}$ writes the top of the stack, of type t, inside field $\kappa.f : \mathsf{t}$ of the object pointed to by the underlying value r, of type κ.

Exception-Handling Instructions. $\mathsf{throw}\ \kappa$ throws the top of the stack, of type $\kappa \le \mathsf{Throwable}$. 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, $\mathsf{exception\_is}\,K$ selects an appropriate handler depending on the run-time class of the exception.

Method Call and Return. We use an activation stack of states. Methods can be redefined in object-oriented code, so a call instruction has the form $\mathsf{call}\ m_1 \ldots m_k$, enumerating an over-approximation of the set of possible run-time targets [14].

3 Reachability

In this section we formalize our notion of *reachability* between two program variables.

Definition 4 (Locations reachable from a variable). *Let* $\tau \in \mathcal{T}$. *The set of* locations reachable from a variable $a \in \mathsf{dom}(\tau)$ in a state $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ *is* $\mathsf{L}_\sigma(a) = \bigcup_{i \ge 0} \mathsf{L}^i_\sigma(a)$,

$\mathsf{L}^0_\sigma(l_1) = \{\ell_2\}$
$\mathsf{L}^1_\sigma(l_1) = \mathsf{L}_\sigma(l_1) = \{\ell_1, \ell_2\}$
$\mathsf{L}^0_\sigma(l_2) = \mathsf{L}_\sigma(l_2) = \varnothing$
$\mathsf{L}^0_\sigma(l_3) = \mathsf{L}_\sigma(l_3) = \{\ell_3\}$
$\mathsf{L}^0_\sigma(l_4) = \{\ell_4\}$
$\mathsf{L}^1_\sigma(l_4) = \{\ell_2, \ell_3, \ell_4\}$
$\mathsf{L}^2_\sigma(l_4) = \mathsf{L}_\sigma(l_4) = \{\ell_1, \ell_2, \ell_3, \ell_4\}$

$\mathsf{T}^0(\mathsf{Object})$	$= \mathsf{T}(\mathsf{Object})$
	$= \{\mathsf{Object}, \mathsf{Student}, \mathsf{ListStudent}\}$
$\mathsf{T}^0(\mathsf{Student})$	$= \{\mathsf{Object}, \mathsf{Student}\}$
$\mathsf{T}^1(\mathsf{Student})$	$= \mathsf{T}(\mathsf{Student})$
	$= \{\mathsf{int}, \mathsf{Object}, \mathsf{Student}\}$
$\mathsf{T}^0(\mathsf{ListStudent})$	$= \{\mathsf{ListStudent}, \mathsf{Object}\}$
$\mathsf{T}^1(\mathsf{ListStudent})$	$= \{\mathsf{ListStudent}, \mathsf{Object}, \mathsf{Student}\}$
$\mathsf{T}^2(\mathsf{ListStudent})$	$= \mathsf{T}(\mathsf{ListStudent})$
	$= \{\mathsf{int}, \mathsf{ListStudent}, \mathsf{Object}, \mathsf{Student}\}$

Fig. 4. Example of computation of reachable locations and types

where $\mathsf{L}^i_\sigma(a)$ *are the locations reachable from* a *in at most* i *steps:* $\mathsf{L}^i_\sigma(a) = \{\rho(a)\} \cap \mathbb{L}$ *if* $i = 0$*, and* $\mathsf{L}^i_\sigma(a) = \mathsf{L}^{i-1}_\sigma(a) \cup \bigcup_{\ell \in \mathsf{L}^{i-1}_\sigma(a)} (\mathsf{rng}(\mu(\ell).\phi) \cap \mathbb{L})$ *if* $i > 0$*.*

Definition 5 (Reachability between variables). *Let* $\tau \in \mathcal{T}$*,* $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ *and variables* $a, b \in \mathsf{dom}(\tau)$*. We say that* b *is* reachable *from* a *in* σ *or, equivalently, that* a *reaches* b *in* σ*, denoted as* $a \rightsquigarrow^\sigma b$*, iff* $\rho(b) \in \mathsf{L}_\sigma(a)$*.*

We also introduce a notion of static reachability between types.

Definition 6 (Reachability between types). *Let* $\mathsf{t} \in \mathbb{T}$*. The set of types* compatible *with* t *is* $\mathsf{compatible}(\mathsf{t}) = \{\mathsf{t}' \mid \mathsf{t} \le \mathsf{t}' \text{ or } \mathsf{t}' \le \mathsf{t}\}$*. The* set of types reachable from t *is* $\mathsf{T}(\mathsf{t}) = \bigcup_{i \ge 0} \mathsf{T}^i(\mathsf{t})$*, where* $\mathsf{T}^i(\mathsf{t})$ *are the types reachable from* t *in at most* i *steps:* $\mathsf{T}^i(\mathsf{t}) = \mathsf{compatible}(\mathsf{t})$ *if* $i = 0$*, and* $\mathsf{T}^i(\mathsf{t}) = \mathsf{T}^{i-1}(\mathsf{t}) \cup \bigcup_{\kappa \in \mathsf{T}^{i-1}(\mathsf{t}) \cap \mathbb{K},\ \kappa'.f:\mathsf{t}' \in \mathbb{F}(\kappa)} \mathsf{compatible}(\mathsf{t}')$ *if* $i > 0$*. We say that* $\mathsf{t}' \in \mathbb{T}$ *is reachable from* t *if* $\mathsf{t}' \in \mathsf{T}(\mathsf{t})$*, and we denote it as* $\mathsf{t} \rightsquigarrow \mathsf{t}'$*.*

Example 3. Consider $\sigma \in \Sigma_\tau$ from Ex. 2. On the left of Fig. 4 we give, for each $l_i \in \mathsf{dom}(\tau)$ and $j \ge 0$, the set of reachable locations from l_i in σ in at most j steps until the fixpoint is reached. Hence, $l_1 \rightsquigarrow^\sigma l_1$, $l_1 \rightsquigarrow^\sigma l_2$, $l_3 \rightsquigarrow^\sigma l_3$, $l_4 \rightsquigarrow^\sigma l_1$, $l_4 \rightsquigarrow^\sigma l_2$, $l_4 \rightsquigarrow^\sigma l_3$, $l_4 \rightsquigarrow^\sigma l_4$. Assume that class Student contains only one field, of type int. ListStudent and Student are subclasses of Object. Fig. 4 reports on the right the types reachable from these three classes: $\mathsf{ListStudent} \rightsquigarrow \mathsf{Student}$, $\mathsf{Object} \rightsquigarrow \mathsf{Student}$, $\mathsf{Student} \rightsquigarrow \mathsf{Object}$, $\mathsf{Object} \rightsquigarrow \mathsf{Student}$, etc. □

Reachability between types can be used to conservatively approximate possible pairs of variables that might reach each other.

Lemma 1. *Let* $\tau \in \mathcal{T}$*,* $\sigma \in \Sigma_\tau$ *and* $a, b \in \mathsf{dom}(\tau)$*. If* $a \rightsquigarrow^\sigma b$*, then* $\tau(a) \rightsquigarrow \tau(b)$*.*

Example 4. Since $l_4 \rightsquigarrow^\sigma l_3$ (Ex. 3), by Lemma 1, also $\tau(l_4) \rightsquigarrow \tau(l_3)$ holds. In fact, Ex. 3 shows that $\tau(l_4) = \mathsf{ListStudent} \rightsquigarrow \mathsf{Student} = \tau(l_3)$. □

4 Reachability Analysis

We define here an abstract interpretation of the concrete semantics of Section 2 w.r.t. the property of reachability between variables (Definition 5). This will be an actual algorithm for interprocedural, whole-program reachability analysis. We follow here the abstract interpretation approach [5], that allows us to define a static analysis from the formal specifications of the property of interest and the semantics of the language.

The concrete semantics works over concrete states (Definition 3), that our abstract interpretation abstracts into ordered pairs of variables.

Definition 7 (Concrete and Abstract Domain). *Given a type environment* $\tau \in \mathcal{T}$*, we define the* concrete domain *over* τ *as* $\mathsf{C}_\tau = \langle \wp(\Sigma_\tau), \subseteq \rangle$ *and the* abstract domain *over* τ *as the powerset of the set of ordered pairs of variables* $\mathsf{A}_\tau = \langle \wp(\mathsf{dom}(\tau) \times \mathsf{dom}(\tau)), \subseteq \rangle$*. For every* $v, w \in \mathsf{dom}(\tau)$*, we write* $v \rightsquigarrow w$ *to denote the ordered pair* $\langle v, w \rangle$*.*

An abstract element $R \in \mathsf{A}_\tau$ represents those concrete states whose reachability information is over-approximated by the pairs of variables in R (*possible* reachability).

Definition 8 (Concretization map). *For every* $\tau \in \mathcal{T}$*, we define the* concretization map $\gamma_\tau : \mathsf{A}_\tau \to \mathsf{C}_\tau$ *as* $\gamma_\tau = \lambda R.\{\sigma \in \Sigma_\tau \mid \forall a, b \in \mathsf{dom}(\tau). a \rightsquigarrow^\sigma b \Rightarrow a \rightsquigarrow b \in R\}$*.*

Both C_τ and A_τ are complete lattices. Moreover, we proved γ_τ co-additive, and therefore it is the concretization map of a Galois connection [5] and A_τ is actually an abstract domain, in the sense of abstract interpretation.

Our analysis is constraint-based: we build an *abstract constraint graph* from the source code of a Java bytecode program. There is a node for each bytecode b in the program, containing an element of A_τ, where τ is the static type information at the beginning of b. An arc linking the nodes corresponding to two bytecodes b_1 and b_2 propagates the reachability information from b_1 to b_2. Here, the exact meaning of *propagates* depends on b_1, since each bytecode has different effects on reachability.

Definition 9 (ACG). *Let* P *be the program under analysis (i.e., a control flow graph of basic blocks for each method or constructor). The* abstract constraint graph *(*ACG*) of* P *is a directed graph* $\langle V, E \rangle$ *(nodes, arcs) where:*

- V *contains a node* $\boxed{\mathsf{ins}}$*, for every bytecode instruction* ins *of* P*;*
- 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*;*
- E *contains directed (multi-)arcs with one or two sources and always one sink;*
- *for every arc in* E*, there is a* propagation rule*, i.e., a function over* A*, from the reachability information at its source(s) to the reachability information at its sink.*

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. Moreover, we assume that* τ *contains* j *stack elements and* i *local variables. In the following we discuss different types of arcs.*

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* #1-#7 *in Fig. 5.*

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-#10 *in Fig. 5.*

#1	dup t	$\lambda R.R \cup R[s_{j-1} \mapsto s_j] \cup \{s_{j-1} \rightsquigarrow s_j, s_j \rightsquigarrow s_{j-1} \mid s_{j-1} \rightsquigarrow s_{j-1} \in R\}$
#2	new κ	$\lambda R.R \cup \{s_j \rightsquigarrow s_j\}$
#3	load k t	$\lambda R.R \cup R[l_k \mapsto s_j] \cup \{l_k \rightsquigarrow s_j, s_j \rightsquigarrow l_k \mid l_k \rightsquigarrow l_k \in R\}$
#4	store k t	$\lambda R.\{(a \rightsquigarrow b)[s_{j-1} \mapsto l_k] \mid a \rightsquigarrow b \in R \wedge a, b \neq l_k\}$
#5	getfield f: t	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \neq s_{j-1}\} \cup \{s_{j-1} \rightsquigarrow b \in R \mid \mathsf{t} \rightsquigarrow \tau(b)\} \cup$ $\{a \rightsquigarrow s_{j-1} \mid a \in \mathsf{dom}(\tau) \wedge \tau(a) \rightsquigarrow \mathsf{t} \wedge [a$ and s_{j-1} might share at getfield f: t$]\}$
#6	putfield f: t	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \notin \{s_{j-1}, s_{j-2}\}\} \cup$ $\{a \rightsquigarrow b \mid a, b \notin \{s_{j-1}, s_{j-2}\} \wedge a \rightsquigarrow s_{j-2} \in R \wedge s_{j-1} \rightsquigarrow b \in R\}$
#7	const v, catch, ifne t, ifeq t	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \in \mathsf{dom}(\tau')\}$
#8	return void	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \notin \{s_0, \ldots, s_{j-1}\}\}$
#9	return t	$\lambda R.\{(a \rightsquigarrow b)[s_{j-1} \mapsto s_0] \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\}$
#10	throw κ	$\lambda R.\{(a \rightsquigarrow b)[s_{j-1} \mapsto s_0] \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\} \cup \{s_0 \rightsquigarrow s_0\}$
#11	throw κ	$\lambda R.\{(a \rightsquigarrow b)[s_{j-1} \mapsto s_0] \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-2}\}\} \cup \{s_0 \rightsquigarrow s_0\}$
#12	call $m_1 \ldots m_k$	$\lambda R.\{a \rightsquigarrow b \in R \mid a, b \notin \{s_0, \ldots, s_{j-1}\}\} \cup \{s_0 \rightsquigarrow s_0\}$ $\cup\{a \rightsquigarrow s_0 \mid a \in \{l_0, \ldots, l_{i-1}\} \wedge \tau(a) \rightsquigarrow \mathtt{Throwable}\}$ $\cup\{s_0 \rightsquigarrow a \mid a \in \{l_0, \ldots, l_{i-1}\} \wedge \mathtt{Throwable} \rightsquigarrow \tau(a)\}$
#13	new κ, getfield f: t, putfield f: t	$\lambda R.\{a \rightsquigarrow b \mid a \rightsquigarrow b \in R \wedge a, b \notin \{s_0, \ldots, s_{j-1}\}\} \cup \{s_0 \rightsquigarrow s_0\}$
#14	call $m_1 \ldots m_k$	$\lambda R.\left\{(a \rightsquigarrow b)\left[\begin{smallmatrix} s_{j-\pi} \mapsto l_0 \\ \cdots \\ s_{j-1} \mapsto l_{\pi-1} \end{smallmatrix}\right] \middle\| a \rightsquigarrow b \in R \wedge a, b \in \{s_{j-\pi}, \ldots, s_{j-1}\}\right\}$

Fig. 5. Propagation rules of simple arcs

Exceptional Arcs. *For each* ins *throwing an exception, immediately followed by a* catch*, a arc is built from* $\boxed{\text{ins}}$ *to* $\boxed{\text{catch}}$*, with one of the propagation rules* #11 – #13 *in Fig. 5.*

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* #14 *in Fig. 5, for each* $1 \leq w \leq k$*.*

Return Value Arcs. *For each* $\mathsf{ins}_c = \mathsf{call}\ m_1 \ldots m_k$ *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* $\boxed{\mathsf{ins}_c}$ *and* $\boxed{\mathsf{exit}@m_w}$ *(2 sources, in that order) to* $\boxed{\mathsf{ins}'}$ *with the propagation rule* #15 *defined in Fig. 6, for each* $1 \leq w \leq k$*.*

Side-Effects Arcs. *For each* $\mathsf{ins}_c = \mathsf{call}\ m_1 \ldots m_k$ *to a method with* π *parameters (including* this*) and each subsequent bytecode* ins′ *, we build a multi-arc from* $\boxed{\mathsf{ins}_c}$ *and* $\boxed{\mathsf{exit}@m_w}$ *(2 sources, in that order) to* $\boxed{\mathsf{ins}'}$*, where* ins′ *is not a* catch*, or from* $\boxed{\mathsf{ins}_c}$ *and* $\boxed{\mathsf{exception}@m_w}$ *(2 sources, in that order) to* $\boxed{\text{catch}}$*, for each* $1 \leq w \leq k$*. The propagation rule* #16 *is given in Fig. 6, where* $\max = j - \pi$ *if* ins′ *is not a* catch *and* $\max = 0$ *otherwise.*

The **sequential arcs** link an instruction to its immediate successors. For instance, the arc #1, starting from a node corresponding to a dup t, states that the reachability approximation at that node can be found at its successor's node as well ($\lambda R.R$). On the other hand, since s_j, the new topmost stack element (new top), is an alias of s_{j-1}, the former topmost stack element (old top), it is clear that every variable reaching s_{j-1} (or, respectively, that is reachable from s_{j-1}) also reaches s_j (respectively, is reachable from s_j): $\lambda R.R \cup R[s_{j-1} \mapsto s_j]$. For the same reason, we must assume that, if s_{j-1} reaches itself (i.e., if the old top was not `null`) then, immediately after the dup t, s_j might reach

#15	$\lambda R_1.\lambda R_2.\{s_{j-\pi} \leadsto s_{j-\pi} \mid s_0 \leadsto s_0 \in R_2\}$ $\cup \left\{ a \leadsto s_{j-\pi} \;\middle\vert\; \begin{array}{l} 1.\ a \in \mathsf{dom}(\tau') \setminus \{s_{j-\pi}\} \wedge \\ 2.\ \tau'(a) \leadsto \mathsf{t} \wedge \\ 3.\ \exists j-\pi \le p < j \text{ s.t. } a \text{ might share with } s_p \text{ at } \mathsf{call}\ m_1 \ldots m_k \wedge \\ 4.\ \text{if } a \text{ is definitely alias of } s_p \text{ at } \mathsf{call}\ m_1 \ldots m_k \text{ and no } \mathsf{store}\ l_{p-j+\pi} \\ \quad \text{occurs in } m_w \text{, then } l_{p-j+\pi} \leadsto s_0 \in R_2 \end{array} \right\}$ $\cup \left\{ s_{j-\pi} \leadsto b \;\middle\vert\; \begin{array}{l} 1.\ b \in \mathsf{dom}(\tau') \setminus \{s_{j-\pi}\} \wedge \\ 2.\ \mathsf{t} \leadsto \tau'(b) \wedge \\ 3.\ \exists j-\pi \le p < j \text{ s.t. } s_p \leadsto b \in R_1 \wedge \\ 4.\ \text{if } b \text{ is definitely alias of } s_p \text{ at } \mathsf{call}\ m_1 \ldots m_k \text{ and no } \mathsf{store}\ l_{p-j+\pi} \\ \quad \text{occurs in } m_w \text{, then } s_0 \leadsto l_{p-j+\pi} \in R_2 \end{array} \right\}$
#16	$\lambda R_1.\lambda R_2.$ $\left\{ a \leadsto b \;\middle\vert\; \begin{array}{l} [a \leadsto b \in R_1 \wedge a, b \in \{l_0, \ldots, l_{i-1}, s_0, \ldots, s_{\max-1}\}] \vee \\ \left[\begin{array}{l} 1.\ a, b \in \{l_0, \ldots, l_{i-1}, s_0, \ldots, s_{\max-1}\} \wedge \\ 2.\ \tau'(a) \leadsto \tau'(b) \wedge \\ 3.\ \exists j-\pi \le p_a < j \text{ s.t. } a \text{ might share with } s_{p_a} \text{ at } \mathsf{call}\ m_1 \ldots m_k \wedge \\ 4.\ \exists j-\pi \le p_b < j \text{ s.t. } p_b \leadsto b \in R_1 \wedge \\ 5.\ \text{if } \exists j-\pi \le q_a < j \text{ s.t. } a \text{ is definitely alias of } s_{q_a} \text{ at } \mathsf{call}\ m_1 \ldots m_k \text{ and} \\ \quad \text{if } \exists j-\pi \le q_b < j \text{ s.t. } b \text{ is definitely alias of } s_{q_b} \text{ at } \mathsf{call}\ m_1 \ldots m_k \text{ and} \\ \quad \text{no } \mathsf{store}\ l_{q_a-j+\pi} \text{ nor } \mathsf{store}\ l_{q_b-j+\pi} \text{ occurs in } m_i \text{, then } l_{q_a-j+\pi} \leadsto l_{q_b-j+\pi} \in R_2 \end{array} \right] \end{array} \right\}$

Fig. 6. Propagation rules of mulit-arcs

s_{j-1} and vice versa, which leads to rule #1. Rule #5 is more interesting: getfield f : t replaces the old top of the stack, s_{j-1}, with the value of its field f. Hence all reachability pairs that do not consider s_{j-1} are still valid after the execution of the getfield f : t: $\lambda R.\{a \leadsto b \in R \mid a, b \neq s_{j-1}\}$. But we have to consider which variable b might be reached from the field ($s_{j-1} \leadsto b$) and which variable a might reach the field ($a \leadsto s_{j-1}$). For b, we observe that if the field reaches b, then also its containing object (i.e., the old top of the stack) had to reach b before the getfield f: t (i.e., $s_{j-1} \leadsto b \in R$); for better precision we consider only those pairs of variables that satisfy type reachability requirement, i.e., $\mathsf{t} \leadsto \tau(b)$. For a, we rely on a pessimistic (but conservative) assumption: every variable a might reach the field after the getfield$f$: t, as long as the field has a reference type such that $\tau(a) \leadsto \mathsf{t}$ and as long as a shares with the top of the stack before the instruction. Rule #6 states that a reachability pair at a putfield f: t instruction remains valid just after that instruction, provided that it did not deal with the topmost two values of the stack s_{j-1} and s_{j-2}, that disappear. Moreover, since this instruction writes s_{j-1} in a field of s_{j-2}, it might introduce reachability from a to b, when a reaches the receiver s_{j-2} and the value s_{j-1} reaches b before the putfield f: t.

The **final arcs** feed nodes $\boxed{\mathsf{exit}@m}$ and $\boxed{\mathsf{exception}@m}$ for each method or constructor m. The former contains all states at the end of a normal execution of m; the latter contains those at the end of an exceptional execution of m. Hence $\boxed{\mathsf{exit}@m}$ is the sink of an arc from every return t in m. The propagation rule states that the stack is emptied at the end of execution of m (#8) or only one element survives, the return value (#9). Similarly, $\boxed{\mathsf{exception}@m}$ is the sink node of every throwκ instruction that has no exception handler in m (i.e., it has no successors in m). Rule #10 states that all stack elements, but the topmost one s_{j-1}, disappear. The latter is renamed into the exception object s_0, and is always non-`null` (thus, $s_0 \leadsto s_0$). We observe that only a throw κ is 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 two putfield in Fig. 2).

The **exceptional arcs** link every instruction that might throw an exception to the catch at the beginning of their exception handler(s). Rules #10 and #11 are identical, but the latter is applied when throw κ has a successor. Rule #12 states a pessimistic assumption about the exceptional states after a method call: the reachability pairs before the call can survive as long as they do not deal with stack elements. The thrown object s_0 is non-`null` (thus, $s_0 \rightsquigarrow s_0$) and conservatively assumed to reach and be reached from every local variable a, as long as the static types allow it.

The **parameter passing arcs** connect each method call to the beginning of a method m_w that it might call. Rule #14 renames the actual parameters of m_w, i.e., $s_{j-\pi}, \ldots, s_{j-1}$, into its formal parameters, i.e., $l_0, \ldots, l_{\pi-1}$.

There exists a **return value multi-arc** for each target m_w of a call. Rule #15 considers R_1 and R_2, approximations at the node corresponding to the call and at node $\boxed{\mathsf{exit}@m_w}$. It builds the reachability pairs related to the returned value $s_{j-\pi}$, in the caller. Namely, $s_{j-\pi}$ reaches itself if the return value in the callee (held in the only stack element s_0 at its end) reaches itself. Moreover, a variable a of the caller might reach that returned value ($a \rightsquigarrow s_{j-1}$) if it exists after the call and it is not $s_{j-\pi}$ itself (condition 1); if the static types allow it (condition 2); if a shares with at least one actual parameter s_p (condition 3); moreover, if a is a definite alias of the actual parameter s_p whose corresponding formal parameter $l_{p-j+\pi}$ is never re-assigned inside the callee m_w, then it must also be the case that $l_{p-j+\pi}$ reaches the returned value s_0 (condition 4). Variables b that might be reachable from the returned value $s_{j-\pi}$ are determined in a symmetrical way. It is worth noting that the result of the call can reach a variable b only if b is reachable from at least one actual parameter s_p of the call at call-time ($s_p \rightsquigarrow b \in R_1$).

The **side-effects multi-arcs** enrich the reachability information already known at call-time with some additional pairs of variables whose presence is due to the side-effects of the call. Rule #16 adds a new pair $a \rightsquigarrow b$ if it satisfies the following conditions: a and b must exist after the call and must not be the returned value nor the exception thrown by m_w (condition 1); the static types of a and b must allow their reachability (condition 2); moreover, a must share with at least one actual parameter of the call and b must be reachable from at least one actual parameter of the call (conditions 3 and 4, respectively); finally, if a and b are definite aliases of two actual parameters q_a and q_b of the call whose corresponding formal parameters $l_{q_a-j+\pi}$ and $l_{q_b-j+\pi}$ are not re-assigned inside m_w, then $l_{q_a-j+\pi}$ must reach $l_{q_b-j+\pi}$ at the end of m_w (condition 5).

Propagation rules #15 and #16 use possible sharing and definite aliasing between program variables. If these data are missing, one can always assume the worst, least precise hypothesis. In our experiments (Section 5) reachability analysis is performed inside the nullness and termination tools of Julia, that already perform definite aliasing and possible sharing analyses, so they have no additional cost. The precision of the analysis would benefit from a possible inlining of frequently used methods, so that their calling contexts are not merged into one. However, this is not implemented in Julia.

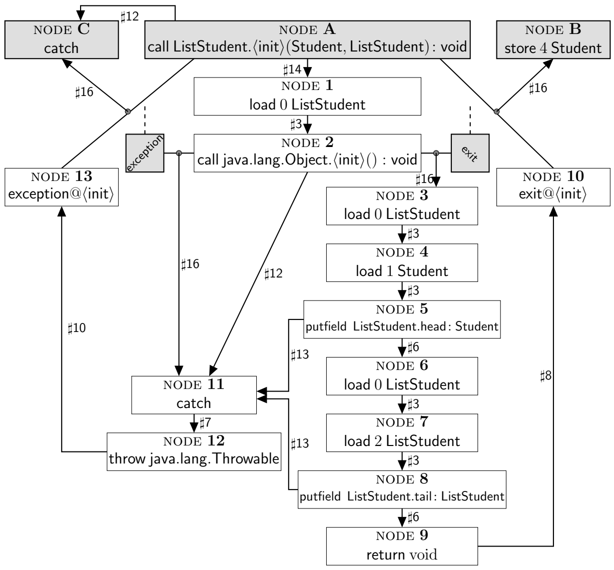

Fig. 7. The ACG for the constructor in Fig. 2

An ACG is *solved* by finding a reachability approximation at each node, consistent with the propagation rules of the arcs. Since these propagation rules are monotonic, a minimal solution exists and can be computed through a fixpoint calculation. This solution is the *reachability analysis* of the program, and has been proven sound [14].

Theorem 1 (Soundness). *Let* ins *and* $\sigma \in \Sigma_\tau$ *be a bytecode instruction and a state reached by an execution of the* main *method of a program, and let* $R_{\mathsf{ins}} \in \mathsf{A}_\tau$ *be the reachability approximation computed by our analysis at* $\boxed{\mathsf{ins}}$. *Then,* $\sigma \in \gamma_\tau(R_{\mathsf{ins}})$.

Example 5. Fig. 7 shows the ACG built for the constructor in Fig. 2. It also shows, in grey, three nodes of a caller of this constructor (nodes A, B and C) and two nodes of the callee of call `java.lang.Object.⟨init⟩() : void`, to exemplify the arcs related to method call and return. Arcs are decorated with the number of their associated propagation rule. Note that the graph for the whole program includes other nodes and arcs. Suppose that at node A, which invokes the constructor, there are four stack elements and four local variables and that we know, from previous static analyses, that a correct possible sharing information is $\mathsf{share}_A = \{\langle s_0, s_1\rangle, \langle l_3, s_2\rangle, \langle l_1, s_3\rangle\}$ (only these pairs of variables might share), while a correct definite aliasing information is $\mathsf{alias}_A = \{\langle s_0, s_1\rangle, \langle l_3, s_2\rangle\}$ (those pairs of variables must be alias, but there might be others). Moreover, suppose that this call occurs in a context with reachability information $S_A = \{l_1 \rightsquigarrow l_1, l_3 \rightsquigarrow l_3, l_1 \rightsquigarrow s_3, l_3 \rightsquigarrow s_2, s_2 \rightsquigarrow l_3, s_0 \rightsquigarrow s_0, s_0 \rightsquigarrow s_1, s_1 \rightsquigarrow s_0, s_1 \rightsquigarrow s_1, s_2 \rightsquigarrow s_2, s_3 \rightsquigarrow s_3\}$. The constructor stores the locations held in its parameters s_2 and s_3 into the fields `head` and `tail` of the newly created object, whose location is, in turn, held in s_0 and s_1. Moreover, s_2 and l_3 are definite aliases at node A, hence we expect that, after any non-exceptional execution of the call (node B), l_3 is reachable from s_0. Node A is linked to node 1 through an arc with propagation rule #14, whose application on S_A gives an approximation of the reachability information at node 1, $S_1 = \{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}$. Similarly, we determine the approximations of the reachability information of the other nodes. For instance, $S_2 = \{l_0 \rightsquigarrow l_0, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2, l_0 \rightsquigarrow s_0, s_0 \rightsquigarrow l_0, s_0 \rightsquigarrow s_0\}$, $S_3 = S_1$, etc. In particular, $S_{10} = \{l_0 \rightsquigarrow l_0, l_0 \rightsquigarrow l_1, l_0 \rightsquigarrow l_2, l_1 \rightsquigarrow l_1, l_2 \rightsquigarrow l_2\}$ and there is a side-effect arc from nodes A and 10 to node B, whose propagation rule #16 applied to S_A and S_{10} gives $S_B = \{l_1 \rightsquigarrow l_1, l_1 \rightsquigarrow s_0, l_1 \rightsquigarrow l_3, l_3 \rightsquigarrow l_3, s_0 \rightsquigarrow l_3, s_0 \rightsquigarrow s_0\}$. As expected, $s_0 \rightsquigarrow l_3 \in S_B$. □

5 Experiments

We have implemented our reachability analysis inside the Julia analyzer for Java and Android (`http://www.juliasoft.com`). Our first aim was to evaluate the cost of the reachability analysis itself and verify whether it actually improves the precision of side-effects, field initialization and cyclicity, as hinted in Section 1. The second aim was to verify if the extra reachability information improves the precision of the nullness and termination checking tools available in Julia, that use side-effects, field initialization, cyclicity and path-length as (some of their) supporting analyses. We do not have any measure of precision for path-length analysis, so we do not evaluate its improvements directly but only as a component of the termination checking tool. To reach these goals, we have analyzed some Java and Android programs, with reachability analysis turned off and then on. Most of these samples are Android applications: Mileage, OpenSudoku, Solitaire and TiltMazes[1]; ChimeTimer, Dazzle, OnWatch and Tricorder[2]; TxWthr[3]. There are also some Java programs: JFlex is a lexical analyzers generator[4]; Plume is a library by Michael D. Ernst[5]; Nti is a non-termination analyzer by Étienne Payet[6]; Lisimplex is a numerical simplex implementation by Ricardo Gobbo[7]. The others are sample programs taken from the Android 3.1 distribution by Google.

Fig. 8 reports time and precision of reachability analysis on a Linux quad-core Intel Xeon machine running at 2.66GHz, with 8 gigabytes of RAM. Times are always below 41 seconds. Average precision is 45.07% which means that, given two variables v and w of reference type at a given program point, in more than half of the cases the analysis proves that v does not reach w. A smaller percentage, here, means better precision. Fig. 8 shows that reachability analysis improves the precision of the side-effects analysis and has positive effects on field initialization as well. Instead, cyclicity analysis seems unaffected. Sharing analysis is always used in these experiments, both when we use reachability information and when we do not compute it. Thus, this figure shows the importance of having also reachability information instead of just sharing information.

Fig. 9 presents our experiments with the nullness and termination tools of Julia and reports their runtime, including reachability analysis. In 8 cases over 24, the extra reachability information improves the precision of the nullness checking tool. But

[1] `http://f-droid.org/repository/browse/`
[2] `http://moonblink.googlecode.com/svn/trunk/`
[3] `http://typoweather.googlecode.com/svn/trunk/`
[4] `http://jflex.de`
[5] `http://code.google.com/p/plume-lib`
[6] `http://personnel.univ-reunion.fr/epayet/Research/NTI/NTI.html`
[7] `http://sourceforge.net/projects/lisimplex`

program	language	source lines	analyzed lines	reach. analysis		prec. of side-effects analysis		prec. of field initial. analysis		prec. of cyclicity analysis	
				time	prec	without reach.	with reach.	without reach.	with reach.	without reach.	with reach.
BluetoothChat	Android	616	84415	21.26	56.01%	645.99	540.23	2185	2325	12.85%	12.85%
ChimeTimer	Android	1090	89565	23.39	47.04%	730.68	618.08	2348	2486	13.54%	13.54%
Dazzle	Android	1791	77828	24.23	46.99%	309.89	225.96	2417	2447	22.19%	22.19%
GestureBuilder	Android	502	84346	23.90	64.11%	667.70	557.52	2162	2282	16.57%	16.57%
Home	Android	870	87413	18.54	55.78%	693.80	584.01	2274	2415	10.89%	10.89%
HoneycombGallery	Android	948	71558	16.71	23.84%	333.25	242.32	2131	2175	33.33%	33.33%
JFlex	Java	7681	40779	7.19	39.59%	357.59	243.89	1092	1146	33.67%	33.67%
JetBoy	Android	839	65174	16.37	64.54%	281.48	198.71	2173	2202	11.79%	11.79%
Lisimplex	Java	768	49303	16.26	47.98%	637.69	347.96	1356	1433	14.13%	14.13%
LunarLander	Android	538	57675	14.92	66.40%	270.87	191.07	1880	1911	18.11%	18.11%
Mileage	Android	5877	104009	32.12	43.73%	959.30	804.98	2636	2794	25.45%	25.45%
NotePad	Android	705	73742	17.96	36.59%	293.57	218.17	2108	2139	37.50%	37.50%
Nti	Java	2372	13486	2.44	47.90%	24.11	13.51	465	467	32.59%	32.59%
OnWatch	Android	6295	112423	29.59	41.00%	1299.51	796.89	3232	3399	32.59%	32.59%
OpenSudoku	Android	5877	90810	40.68	44.81%	440.36	344.92	2622	2660	22.57%	22.57%
Plume	Java	8586	43637	17.75	24.17%	186.31	126.71	1316	1335	57.11%	57.11%
Real3D	Android	1228	74350	17.81	43.55%	497.94	400.73	2093	2189	36.43%	36.43%
SampleSyncAdapter	Android	978	65971	18.48	34.59%	328.80	235.68	2111	2142	42.77%	42.77%
SoftKeyboard	Android	703	58088	10.96	51.90%	174.01	116.96	2112	2131	11.21%	11.21%
Solitaire	Android	3905	62065	18.67	32.23%	243.19	166.57	1957	1982	50.06%	50.06%
TicTacToe	Android	607	59160	13.40	58.56%	228.27	154.35	1919	1943	20.73%	20.73%
TiltMazes	Android	1853	89653	21.14	15.66%	650.45	562.57	2313	2454	71.57%	71.57%
Tricorder	Android	5317	98389	26.69	46.39%	783.59	663.23	2806	2942	33.91%	33.91%
TxWthr	Android	2024	74537	16.97	48.33%	309.24	229.79	2220	2258	15.39%	15.39%
average precision					**45.07%**	**472.81**	**361.86 (-23.47%)**	**2080.33**	**2152.37 (+3.46%)**	**26.84%**	**26.84% (+0.00%)**

Fig. 8. Cost and precision of reachability analysis, and its effects on the precision of side-effects, field initialization and cyclicity analyses. *Source lines* counts non-comment non-blank lines of codes. *Analyzed lines* includes the portion of `java.*`, `javax.*` and `android.*` libraries analyzed with each program and is a more faithful measure of the analyzed codebase. Times are in seconds. For reachability analysis, precision is the ratio of pairs of variables $\langle v, w\rangle$ s.t. the analysis concludes that v might reach w, over the total number of pairs of variables of reference type: the lower the ratio, the higher the precision (the reatio never reaches 0% in practice, since real-life programs contain reachability). For side-effects analysis, precision is the average number of fields modified or read by a method or constructor: the lower the numbers, the better the precision. For field initialization analysis, precision is the number of fields of reference type proven to be always initialized before being read, in all constructors of their defining class: the higher the numbers, the better the precision. For cyclicity analysis, precision is the average number of variables of reference type proven to hold a non-cyclical data structure; the higher the numbers, the better the precision

program	null. without reach.			null. with reach.			term. without reach.			term. with reach.		
	time	ws	prec	time	ws	prec	time	ws	prec	time	ws	prec
BluetoothChat	368.43	**22*****	93.65%	301.31	**19*****	94.23%	158.96	2	33.33%	141.78	2	33.33%
ChimeTimer	343.01	4	98.36%	360.28	4	98.36%	178.87	1	83.33%	183.81	1	83.33%
Dazzle	223.16	26	97.99%	220.78	26	97.99%	120.34	0	100.00%	126.07	0	100.00%
GestureBuilder	261.25	16	92.37%	288.51	16	92.37%	153.33	0	100.00%	151.83	0	100.00%
Home	314.66	27	94.27%	312.55	27	94.27%	166.98	8	38.46%	163.39	8	38.46%
HoneycombGallery	177.32	12	97.79%	179.90	12	97.79%	105.96	0	100.00%	101.47	0	100.00%
JFlex	87.06	71	97.03%	86.10	71	97.03%	300.84	66	53.52%	321.03	66	53.52%
JetBoy	138.99	20**	97.42%	140.64	20**	97.42%	85.91	3	57.14%	85.38	3	57.14%
Lisimplex	251.09	20**	96.94%	202.76	20**	96.94%	160.07	9	70.97%	153.36	9	70.97%
LunarLander	118.75	4	99.30%	121.25	4	99.30%	72.49	3*	0.00%	68.41	3*	0.00%
Mileage	503.90	**102**	97.40%	501.02	**95**	97.67%	387.68	12	69.23%	381.99	12	69.23%
NotePad	194.52	**18**	96.50%	199.19	**17**	96.50%	103.64	0	100.00%	101.49	0	100.00%
Nti	14.06	12	98.93%	16.15	12	98.93%	43.70	70	36.94%	43.53	70	36.94%
OnWatch	898.36	**74**	97.91%	518.55	**65**	98.18%	385.00	6	86.96%	371.32	6	86.96%
OpenSudoku	284.30	124*	95.93%	286.72	124*	95.93%	458.01	6	90.32%	467.34	6	90.32%
Plume	106.67	**59**	98.82%	116.75	**58**	98.83%	208.81	86	60.00%	187.92	86	60.00%
Real3D	203.62	19*	98.14%	195.76	19*	98.14%	116.42	2	60.00%	112.22	2	60.00%
SampleSyncAdapter	156.31	3	99.51%	152.45	3	99.51%	91.90	2	60.00%	89.61	2	60.00%
SoftKeyboard	104.21	**14**	95.78%	103.83	**13**	95.94%	70.45	0	100.00%	67.96	0	100.00%
Solitaire	153.51	63	92.59%	147.54	63	92.59%	207.09	11	86.08%	203.92	11	86.08%
TicTacToe	115.38	0	100.00%	118.27	0	100.00%	79.69	1	85.71%	78.02	1	85.71%
TiltMazes	281.43	**18**	98.20%	276.54	**14**	98.83%	188.56	1	88.89%	174.63	1	88.89%
Tricorder	415.17	**54**	98.29%	407.51	**52**	98.41%	252.25	12	80.33%	257.36	12	80.33%
TxWthr	200.16	48	97.85%	191.88	48	97.85%	109.76	6	70.00%	105.08	6	70.00%
sum of the times	**5915.32**			**5456.24 (-7.77%)**			**4206.71**			**4138.92 (-1.62%)**		
sum of the warnings	**830**			**802 (-3.38%)**			**307**			**307 (+0.00%)**		

Fig. 9. Our experiments with the nullness and termination tools of Julia. Times are in seconds. For nullness analysis, *ws* counts the warnings issued by Julia (possible dereference of null, possibly passing null to a library method) and *prec* reports its precision, as the ratio of the dereferences proved safe over their total number (100% is the maximal precision). For termination analysis, *ws* counts the warnings issued by Julia (constructors or methods possibly diverging) and *prec* reports its precision, as the ratio of the constructors or methods proved to terminate over the total number of constructors or methods containing loops or recursive (100% is the maximal precision). Asterisks stand for actual bugs in the programs. Boldface highlights the cases where reachability improves the precision of the tools

this never happens for termination, consistently with the fact that cyclicity is not improved (Fig. 8). This is because the methods of the programs that we have analyzed terminate since they perform loops over numerical counters or iterators. There is no complex case of recursion over data structures dynamically allocated in memory (lists or trees) where cyclicity would help. To investigate further the case of termination analysis, we have applied Julia to the set of (very tiny) programs used in the international termination competition that is performed every year. Those programs, although small and often unrealistic, are nevertheless interesting since the proof of their termination often requires non-trivial arguments, also related to objects dynamically allocated in memory. Over a total of 164 test programs, the reachability information allows Julia to prove the termination of six more tests: LinkedList, List, ListDuplicate, PartitionList, Test5 and Test6, by supporting a more precise cyclicity and path-length analysis.

For both nullness and termination checking, the presence of reachability analysis actually reduces the total runtime of the tools. This is because reachability helps subsequent analyses, in particular side-effects analysis, and prevents them from generating too much spurious information. For instance, side-effects analysis computes much smaller sets of affected fields per method (Fig. 8, compare the 7th and the 8th columns).

6 Conclusion

We have introduced, formalized and implemented a provably sound (see [14] for proofs) constraint-based reachability analysis for Java bytecode. Its implementation inside the Julia static analyzer is able to scale to programs containing 100k lines of code. Our experiments show that the reachability analysis improves the precision and efficiency of the side-effects, field initialization and nullness analyses, already performed by Julia.

Our constraint-based approach has been used to develop aliasing and sharing analyses of our tool (never published and with completely different propagation rules). We plan to use it in the future to formalize and prove correct other static analyses as well.

References

1. Balaban, I., Pnueli, A., Zuck, L.D.: Shape Analysis by Predicate Abstraction. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 164–180. Springer, Heidelberg (2005)
2. Ball, T., Millstein, T., Rajamani, S.K.: Polymorphic Predicate Abstraction. ACM Trans. on Programming Languages and Systems 27, 314–343 (2005)
3. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional Shape Analysis by Means of Bi-Abduction. In: Proc. of the 36th POPL, pp. 289–300. ACM, New York (2009)
4. Chatterjee, S., Lahiri, S., Qadeer, S., Rakamaric, Z.: A Low-Level Memory Model and an Accompanying Reachability Predicate. STTT 11(2), 105–116 (2009)
5. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proceedings of the 4th POPL, pp. 238–252. ACM (1977)
6. Dams, D.R., Namjoshi, K.S.: Shape Analysis through Predicate Abstraction and Model Checking. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 310–323. Springer, Heidelberg (2002)

7. Distefano, D., O'Hearn, P.W., Yang, H.: A Local Shape Analysis Based on Separation Logic. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
8. Hardekopf, B.C.: Pointer Analysis: Building a Foundation for Effective Program Analysis. Ph.D. thesis, University of Texas at Austin, Austin, TX, USA (2009)
9. Hind, M.: Pointer Analysis: Haven't We Solved This Problem Yet? In: Proceedings of PASTE 2001, pp. 54–61. ACM, New York (2001)
10. Lhoták, O.: Program Analysis Using Binary Decision Diagrams. Ph.D. thesis, McGill University (2006)
11. Lhoták, O., Chung, K.C.A.: Points-to Analysis with Efficient Strong Updates. In: Proceedings of the 38th POPL, pp. 3–16. ACM (2011)
12. Lindholm, T., Yellin, F.: The Java^{TM} Virtual Machine Specification, 2nd edn. Addison-Wesley (1999)
13. Nelson, G.: Verifying Reachability Invariants of Linked Structures. In: Proc. of the 10th POPL, pp. 38–47 (1983)
14. Nikolić, D., Spoto, F.: Reachability Analysis of Program Variables, `http://profs.sci.univr.it/~nikolic/download/IJCAR2012/IJCAR2012Ext.pdf`
15. Papi, M.M., Ali, M., Correa, T.L., Perkins, J.H., Ernst, M.D.: Practical Pluggable Types for Java. In: Proceedings of the ISSTA 2008, pp. 201–212. ACM, Seattle (2008)
16. Rossignoli, S., Spoto, F.: Detecting Non-cyclicity by Abstract Compilation into Boolean Functions. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 95–110. Springer, Heidelberg (2005)
17. Rountev, A., Milanova, A., Ryder, B.G.: Points-to Analysis for Java Using Annotated Constraints. In: Proceedings of the 16th OOPSLA, pp. 43–55. ACM (2001)
18. Sagiv, M., Reps, T., Wilhelm, R.: Solving Shape-Analysis Problems in Languages with Destructive Updating. ACM Trans. on Programming Languages and Systems 20, 1–50 (1998)
19. Sagiv, M., Reps, T., Wilhelm, R.: Parametric Shape Analysis via 3-Valued Logic. ACM Trans. Program. Lang. Syst. 24, 217–298 (2002)
20. Salcianu, A.D.: Pointer Analysis for Java Programs: Novel Techniques and Applications. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (2006)
21. Secci, S., Spoto, F.: Pair-Sharing Analysis of Object-Oriented Programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 320–335. Springer, Heidelberg (2005)
22. Spoto, F., Ernst, M.D.: Inference of Field Initialization. In: Proceedings of the 33rd ICSE, pp. 231–240. ACM, Waikiki (2011)
23. Spoto, F., Mesnard, F., Payet, E.: A Termination Analyzer for Java Bytecode Based on Path-Length. ACM Trans. on Programming Languages and Systems 32(3), 1–70 (2010)