2008 Sixth IEEE International Conference on Software Engineering and Formal Methods

Nullness Analysis in Boolean Form

Fausto Spoto, Università di Verona, Italy
fausto.spoto@univr.it

Abstract

Attempts to dereference `null` *result in an exception or a segmentation fault. Hence it is important to know those program points where this might occur and prove the others (or the entire program)* safe. *Nullness analysis of computer programs checks or infers non-*`null` *annotations for variables and object fields. Most nullness analyses currently use run-time checks or are incorrect or only verify manual annotations. We use here abstract interpretation to build and prove correct a static nullness analysis for Java bytecode which* infers *non-*`null` *annotations. It is based on Boolean formulas, implemented with binary decision diagrams. Our experiments show it faster and more precise than the correct nullness analysis by Hubert, Jensen and Pichardie. We deal with static fields and exceptions, which is not the case of most other analyses. We claim that the result is theoretically clean and the implementation strong and scalable.*

1 Introduction

Object-oriented languages such as Java and C# allow uninitialised object fields and let one store a `null` value into the fields. *Dereferences i.e.,* field accesses and method calls, work on a *receiver* value and are *safe* when the latter is never `null`. Othewise, an exception or a segmentation fault occurs. It is important to prove the absence of this error before the program is run or spot suspect program points. Even though dereferences are safe, languages such as Java do check the nullness of the receiver at run-time: removing useless checks improves the efficiency of the program and simplifies its control-flow graph by cutting spurious exceptional paths, which improves efficiency and precision of subsequent static analyses.

Most techniques proving dereferences safe use preliminary non-`null` annotations about fields and method arguments [8]. An extension [11] of class verification [10] propagates them intra-procedurally and exploits guards for better precision; it misses a global view of the code but fits inside the class verifier. Other analyses [2, 7] are more global but based on incorrect/incomplete tools like ESC/Java [5]. Also type systems can check non-`null` annotations [6].

The correct nullness analysis in [3] does not approximate the fields. By expanding its abstract domain, the constraint-based analysis in [9] *infers* non-`null` annotations for the fields. It is mechanically proved correct and more precise than [6] for typable programs. An implementation exists for the Java bytecode. This significant result shows that global nullness analysis can *infer* non-`null` annotations and lead to a reliable implementation. Nevertheless, the analysis is not context-sensitive and better precision can be achieved (Section 2). Moreover, the analysis looks too complex to us: a variable can have several approximations (Raw, $Raw(X)$, $MayBeNull$, $NotNull$, ...) and seven distinct abstract domains are used.

In this paper we make the following contributions:

- We define and prove correct a static nullness analysis to infer non-`null` annotations for Java bytecode; it is *natural i.e.,* a variable is only approximated as `null` or non-`null`; only one abstract domain of Boolean formulas is used, efficiently implemented with binary-decision diagrams [1];
- We identify non-`null` fields with an iterated *oracle* version of our analysis;
- We describe its implementation and show it more precise and faster than that in [9].

We have chosen the Java bytecode since we want to

978-0-7695-3437-4/08 $25.00 © 2008 IEEE
DOI 10.1109/SEFM.2008.8

check code downloaded from the net into client computers or phones. However, our analysis applies to all languages compiled into Java bytecode, so we often make examples easier by writing them in Java.

Section 2 shows an example where our analysis is more precise than others; Section 3 defines the concrete denotational semantics of Java bytecode that Section 4 abstracts into a nullness analysis; Section 5 describes the oracle approach for the fields; Section 6 shows some experiments; Section 7 concludes.

2 An Example of Nullness Analysis

Consider the Java program in Figure 1, devised to test the ability of a nullness analysis. Ours proves that fields `f` and `g` are non-`null` *i.e.,* they never hold `null` when they are read, and that a `java.lang.NullPointerException` might only be thrown at the statement `p.f=new Object()` in the second constructor. This result is optimal, since `n4` might actually hold `null` when `main()` calls it. *All* accesses to `g` inside `helper()` and `foo()` are marked as *safe*. Other analyses, such as [9] and [6], do not prove `f` nor `g` non-`null` nor the accesses inside `helper()` safe.

Our analysis initially assumes `f` and `g` optimistically non-`null` and then looks for a counterexample. Method `helper()` writes `g` and is called by both constructors. The first passes `this`, always non-`null`; the second `p`, non-`null` since otherwise the previous statement `p.f=new Object()` throws an exception and stops the execution. Hence no counterexample is found to the non-nullness of `g`. Both constructors write `f`. The second writes a non-`null` value (`this` or `new Object()`); the first requires to prove that its parameter `f` is always non-`null`. This is true for the call creating `n1`, since a `new Object()` is passed as `f`; the call creating `n3` passes `foo(n1)` which is non-`null` since `foo()` returns `n1.g`, assumed non-`null`, or `n1`, non-`null`; the call creating `n4` passes `n1.f`, assumed non-`null`; the call creating `n6` passes `n4`, non-`null` or otherwise the previous call to the second constructor throws an exception. Thus no counterexample is found to the non-nullness of `f`.

In this example, the following points are important:

1. the analyser must conclude that, after `p.f`, variable `p` is non-`null` or an exception is thrown;

```
public class Test {
 private Object f; private Test g;
 public Test(Object f) { // 1
   this.f = f; helper(this); }
 public Test(Test p) { // 2
  this.f = this; p.f = new Object(); helper(p); }
 private void helper(Test g) {
  this.g = g; try {
   if (this.g.g == this) this.g = this.g.g;
  } catch (NullPointerException e) {} }
 private static Object foo(Test p) {
  if (p != null) return p.g; else return p; }
 public static void main(String[] args) {
  Test n1 = new Test(new Object()); // 1
  Object o2 = foo(null);
  Test n3 = new Test(foo(n1)); // 1
  Test n4 = null;
  if (args.length > 0) n4 = new Test(n1.f); // 1
  // n4 might be null here
  Test n5 = new Test(n4); // 2
  Test n6 = new Test((Object)n4); // 1
 }
}
```

Figure 1. A program to analyse. We specify the constructor called by every `new Test`.

2. the analyser must conclude that if the last statement of `main()` is reached then the previous has thrown no exception *and hence* `n4` is non-`null`;

3. the analyser must not be fooled by the call `foo(null)`, which returns `null`, and conclude that also the subsequent call `foo(n1)` might return `null`: it must be *context-sensitive*.

We think these points outside the ability of current analyses. Ours, instead, fulfills them and proves both `f` and `g` non-`null`. Our experiments (Section 6) confirm that it is actually more precise than [9] and hence [6].

3 Denotational Semantics of Java Bytecode

We describe here the denotational semantics for Java bytecode proved in [12] equivalent to an operational one, but we also consider exceptions here. We assume a program P given as a collection of graphs of *basic blocks* of code, one for each method. Figure 2 shows it for the method `helper()` in Figure 1. Bytecodes which might throw exceptions are linked to a handler starting with a `catch`, possibly followed by bytecodes selecting the right kind of exception. In Figure 2, the topmost `putfield` has a default handler throwing back any exception to the

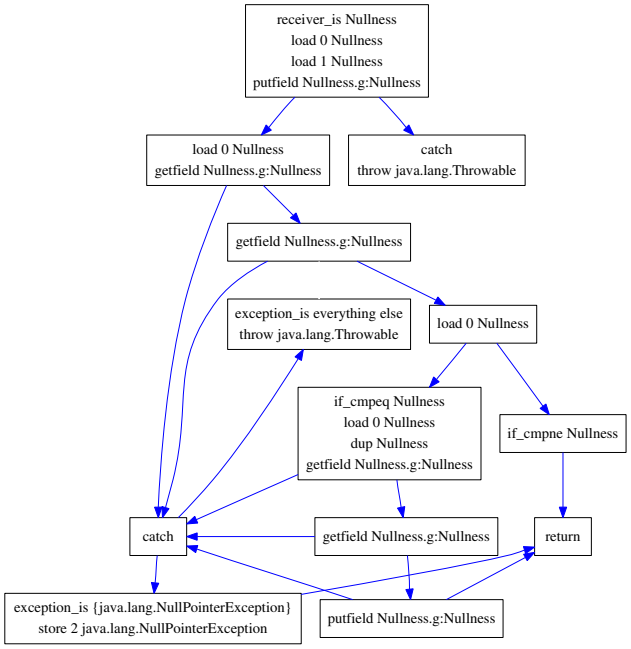

Figure 2. The blocks of `helper()` (Figure 1).

caller; the others have a handler for `java.lang.NullPointerException` and throw back the exception to the caller otherwise (`everything else` stands for the set of all other exceptions).

For simplicity, we assume that the only primitive type is `int` and the only reference types are the *classes*; we only allow *instance* fields and methods. Our implementation deals with full sequential code.

Definition 1 (Classes)**.** *The set of* classes $\mathbb{K}$ *is partially ordered w.r.t. the* subclass relation $\leq$. *A* type *is an element of* $\mathbb{T} = \mathbb{K} \cup \{\texttt{int}\}$. *A class* $\kappa \in \mathbb{K}$ *has* instance fields $\kappa.f : t$ *(field* f *of type* $t \in \mathbb{T}$ *defined in* κ*) and* instance methods $\kappa.m(t_1, \ldots, t_n) : t$ *(method* m *with arguments of type* $t_1, \ldots, t_n \in \mathbb{T}$*, returning a value of type* $t \in \mathbb{T} \cup \{\texttt{void}\}$*, defined in* κ*). We consider constructors as methods returning* `void`. □

Definition 2 (State)**.** *A* value *is an element of* $\mathbb{Z} \cup \mathbb{L} \cup \{\texttt{null}\}$*, where* $\mathbb{L}$ *is a set of* locations. *A* state *is a triple* $\langle l \parallel s \parallel \mu \rangle$ *where* l *is an array of values (the* local variables*),* s *a stack of values (the* operand stack*), which grows leftwards, and* μ *a* memory *which binds locations to* objects. *The empty stack is written* ε. *An object* o *belongs to class* $o.\kappa \in \mathbb{K}$ *(is an* instance *of* $o.\kappa$*) and maps identifiers (the fields* f *of* $o.\kappa$ *and of its superclasses) into values* $o.f$. *The set of states is* Ξ. *We write* $\Xi_{i,j}$ *when we want to fix the number* i *of local variables and* j *of stack elements. A value* v has type t *in a state* $\langle l \parallel s \parallel \mu \rangle$ *if* $v \in \mathbb{Z}$ *and* $t = \texttt{int}$*, or* $v = \texttt{null}$ *and* $t \in \mathbb{K}$*, or* $v \in \mathbb{L}$*,* $t \in \mathbb{K}$ *and* $\mu(v).\kappa \leq t$. □

Example 1. *State* $\sigma = \langle [\ell, \ell'] \parallel \ell'' :: \ell'' :: \ell' \parallel \mu \rangle \in \Xi_{2,3}$*, with* μ *mapping locations* ℓ, ℓ', ℓ'' *to some objects.* □

The Java Virtual Machine (JVM) allows exceptions. Hence we distinguish *normal* states $\sigma \in \Xi$, arising during the normal execution of a piece of code, from *exceptional* states $\underline{\sigma} \in \underline{\Xi}$, arising *just after* a bytecode that throws an exception and having only one stack element, the location of the thrown exception object, also in the presence of nested exception handlers [10].

Definition 3 (JVM State)**.** *The set of* JVM states *(from now just* states*) with* i *local variables and* j *stack elements is* $\Sigma_{i,j} = \Xi_{i,j} \cup \underline{\Xi}_{i,1}$. □

The semantics of a bytecode `ins` is a *denotation i.e.,* a map from an *initial* to a *final* state.

Definition 4 (Denotation)**.** *A* denotation *is a partial map from an* input *or* initial *state to an* output *or* final *state; the set of denotations is* Δ*; we also define* $\Delta_{i_1,j_1 \to i_2,j_2} = \Sigma_{i_1,j_1} \to \Sigma_{i_2,j_2}$ *to fix the number of local variables and stack elements. The* sequential composition *of* $\delta_1, \delta_2 \in \Delta$ *is* $\delta_1; \delta_2 = \lambda\sigma.\delta_2(\delta_1(\sigma))$*, undefined when* $\delta_1(\sigma)$ *or* $\delta_2(\delta_1(\sigma))$ *is undefined.* □

In $\delta_1; \delta_2$, the idea is that δ_1 describes the behaviour of an instruction ins_1, δ_2 that of an instruction ins_2 and $\delta_1; \delta_2$ that of the execution of ins_1 and then ins_2.

At each program point, the number i of local variables and j of stack elements and their types are statically known [10]. Hence, in the following we silently assume that the semantics of the bytecodes is undefined for input states of wrong sizes or types.

Basic instructions. Bytecode `const` v pushes $v \in \mathbb{Z} \cup \{\texttt{null}\}$ on the stack: $const\ v = \lambda\langle l \parallel s \parallel \mu \rangle.\langle l \parallel v :: s \parallel \mu \rangle$ (s might be ε). The λ-notation defines a partial map, undefined on exceptional states since $\langle l \parallel s \parallel \mu \rangle$ is not underlined. That is, $const\ v$ is executed when the JVM is in a normal state. This holds for *all* bytecodes but `catch`, that starts the exceptional handlers from an exceptional state. Bytecode `dup` t duplicates the top of

the stack, of type t: $dup\ t = \lambda\langle l \parallel top :: s \parallel \mu\rangle.\langle l \parallel top :: top :: s \parallel \mu\rangle$. Bytecode `load` k t pushes on the stack the value of local variable number k, which must exist and have type t: $load\ k\ t = \lambda\langle l \parallel s \parallel \mu\rangle.\langle l \parallel l[k] :: s \parallel \mu\rangle$. Conversely, bytecode `store` k t pops the top of the stack of type t and writes it in local variable k: $store\ k\ t = \lambda\langle l \parallel top :: s \parallel \mu\rangle.\langle l[k := top] \parallel s \parallel \mu\rangle$. If l has less than $k+1$ variables, the resulting set of local variables gets expanded. The semantics of a conditional bytecode is undefined when its condition is false. For instance, `ifne` t checks if the top of the stack, of type t, is not 0 when $t =$ `int` or is not `null` otherwise. Its semantics $ifne\ t$ is

$$\lambda\langle l \parallel top :: s \parallel \mu\rangle.\begin{cases}\langle l \parallel s \parallel \mu\rangle & \text{if } top \neq 0,\ top \neq \texttt{null},\\ undefined & \text{otherwise.}\end{cases}$$

Memory-manipulating instructions. Some bytecodes deal with objects in memory: `new` κ pushes on the stack a reference to a new object n of class κ, with reference fields set to `null`. Its semantics $new\ \kappa$ is

$$\lambda\langle l \parallel s \parallel \mu\rangle.\begin{cases}\langle l \parallel \ell :: s \parallel \mu[\ell := n]\rangle & \text{if there is memory}\\ \underline{\langle l \parallel \ell \parallel \mu[\ell := oome]\rangle} & \text{otherwise}\end{cases}$$

with $\ell \in \mathbb{L}$ fresh and $oome$ new instance of `java.lang.OutOfMemoryError`. This is the first bytecode that throws an exception. Bytecode `getfield` $\kappa.f : t$ reads the field $\kappa.f : t$ of the object pointed by the top rec (the *receiver*) of the stack, of type κ. Its semantics $getfield\ \kappa.f : t$ is

$$\lambda\langle l \parallel rec :: s \parallel \mu\rangle.\begin{cases}\langle l \parallel \mu(rec).f :: s \parallel \mu\rangle & \text{if } rec \neq \texttt{null},\\ \underline{\langle l \parallel \ell \parallel \mu[\ell \mapsto npe]\rangle} & \text{otherwise}\end{cases}$$

with $\ell \in \mathbb{L}$ fresh and npe new instance of `java.lang.NullPointerException`. This is the first example of a bytecode that might throw an exception while dereferencing a location (rec). Another is `putfield` $\kappa.f : t$ that moves the top of type t of the stack in the field $\kappa.f : t$ of the object pointed by a value rec of type κ below top. Its semantics $putfield\ \kappa.f : t$ is (ℓ and npe are as before)

$$\lambda\langle l \parallel top :: rec :: s \parallel \mu\rangle.\begin{cases}\langle l \parallel s \parallel \mu[\mu(rec).f := top]\rangle & \text{if } rec \neq \texttt{null},\\ \underline{\langle l \parallel \ell \parallel \mu[\ell := npe]\rangle} & \text{otherwise.}\end{cases}$$

Exception handling instructions. Bytecode `throw` κ throws the object of type $\kappa \leq$ `java.lang.Throwable` pointed by the top of the stack. Its semantics $throw\ \kappa$ is (ℓ and npe are as before)

$$\lambda\langle l \parallel top :: s \parallel \mu\rangle.\begin{cases}\underline{\langle l \parallel top \parallel \mu\rangle} & \text{if } top \neq \texttt{null},\\ \underline{\langle l \parallel \ell \parallel \mu[\ell \mapsto npe]\rangle} & \text{if } top = \texttt{null}.\end{cases}$$

Bytecode `catch` starts an exception handler from an exceptional state: it transforms it into a normal one, used by the implementation of the handler: $catch = \lambda\underline{\langle l \parallel top \parallel \mu\rangle}.\langle l \parallel top \parallel \mu\rangle$ where $top \in \mathbb{L}$ has type `java.lang.Throwable`. The handler is selected on the basis of the run-time class of the exception by a bytecode `exception_is` K that filters the states whose stack top points to an instance of a class in $K \subseteq \mathbb{K}$. Its semantics $exception\_is\ K$ is

$$\lambda\langle l \parallel top \parallel \mu\rangle.\begin{cases}\langle l \parallel top \parallel \mu\rangle & \text{if } top \in \mathbb{L},\ \mu(top).\kappa \in K,\\ undefined & \text{otherwise.}\end{cases}$$

Method call and return instructions. The code of a method $M = \kappa.m(t_1, \ldots, t_n) : t$ starts with a bytecode `receiver_is` K asserting that the run-time class of the receiver (local variable 0) is in a set K statically computed from the look-up rules of the language. Its semantics $receiver\_is\ K$ is

$$\lambda\langle l \parallel \varepsilon \parallel \mu\rangle.\begin{cases}\langle l \parallel \varepsilon \parallel \mu\rangle & \text{if } l[0] \in \mathbb{L},\ \mu(l[0]).\kappa \in K,\\ undefined & \text{otherwise,}\end{cases}$$

with ℓ and npe as before. At the beginning of M the stack is ε and local variables hold exactly its $n+1$ actual arguments (including `this`). At its end, a `return` t bytecode leaves on the stack the return value of type t only, or a `return` bytecode just returns, if $t =$ `void`: $return\ t = \lambda\langle l \parallel top :: s \parallel \mu\rangle.\langle l \parallel top \parallel \mu\rangle$ and $return = \lambda\langle l \parallel s \parallel \mu\rangle.\langle l \parallel \varepsilon \parallel \mu\rangle$. Overall, the semantics of the code of M is hence a denotation δ from a state $\langle [v_0, \ldots, v_n] \parallel \varepsilon \parallel \mu\rangle$ to a state $\sigma = \langle l' \parallel top \parallel \mu'\rangle$, with $top = \varepsilon$ when $t =$ `void`, if M returns normally, or to a state $\sigma = \underline{\langle l' \parallel top \parallel \mu'\rangle}$, with top pointing to an exception e if M throws e. From the point of view of the caller of M, its i local variables l are not affected by the call and the actual arguments $v_0, \ldots, v_n$ are popped from its stack, of height $j = b + n + 1$, and replaced with top (if any). We model this through $extend_M^{i,j} \in \Delta_{n+1,0\to i',r} \to \Delta_{i,j\to i,b+r}$, with $r = 0$ if $t =$ `void` and $r = 1$ otherwise, defined as

$$\lambda\langle l \parallel v_n :: \cdots :: v_0 :: s \parallel \mu\rangle.\begin{cases}\underline{\langle l \parallel \ell \parallel \mu[\ell := npe]\rangle} & \text{if } v_0 = \texttt{null}\\ \underline{\langle l \parallel top :: s \parallel \mu'\rangle} & \text{if } v_0 \in \mathbb{L},\ \sigma \in \Xi,\\ \underline{\langle l \parallel top \parallel \mu'\rangle} & \text{if } v_0 \in \mathbb{L},\ \sigma \in \underline{\Xi},\end{cases}$$

with ℓ and npe as before. This is the third place where a dereference might throw an exception.

The Denotational Semantics. A semantics ι of P is an *interpretation* that specifies the behaviour of each *block* b in P by providing a set $\iota(b)$ of denotations.

They represent possible executions starting at b and continuing with b's successor blocks until a block with no successor is reached. *Sets* are typical of a *collecting* semantics [4], able to model *properties* of denotations. The operators $extend$ and ; over denotations are consequently extended to sets of denotations.

Definition 5 (Interpretation)**.** *An* interpretation *is a map from P's blocks into $\wp(\Delta)$. The set of interpretations $\mathbb{I}$ is ordered by pointwise set-inclusion.* □

Given $\iota \in \mathbb{I}$, we define the set $[\![b]\!]^\iota \subseteq \Delta$ of all the executions, induced by ι, that start at b and continue with b's successors until a block with no successors is reached: we compose sequentially the denotations of the instructions inside b and those of the successor blocks $b_1, \ldots, b_n$, as given by ι. For `call`s, we $extend$ the denotations of the first block of the called method(s), as given by ι.

Definition 6 (Denotations of Instructions and Blocks)**.** *Let $\iota \in \mathbb{I}$. The* denotations in ι of an instruction *are $[\![\texttt{ins}]\!]^\iota = \{ins\}$ if* `ins` *is not a* `call`*, and $[\![\texttt{call}\ M_1, \ldots, M_q]\!]^\iota = \cup_{1 \le s \le q} extend^{i,j}_{M_s}(\iota(b_{M_s}))$ otherwise, where $\{M_1, \ldots, M_q\}$ is a superset of the methods that might be called (computed by some class analysis), b_{M_s} the block where M_s starts, i the number of local variables and j the height of the stack where the* `call` *occurs. Function $[\![\_]\!]^\iota$ is extended to blocks:*

$$\left[\!\!\left[\begin{array}{|c|}\hline \texttt{ins}_1 \\ \cdots \\ \texttt{ins}_n \\ \hline \end{array} \rightrightarrows \begin{array}{c} b_1 \\ \cdots \\ b_m \end{array} \right]\!\!\right]^\iota = [\![\texttt{ins}_1]\!]^\iota ; \cdots ; [\![\texttt{ins}_n]\!]^\iota \underbrace{; (\iota(b_1) \cup \cdots \cup \iota(b_m))}_{Cont}$$

where $Cont$ is missing when $m = 0$. □

Note that Definition 6 uses an operator $\cup$ over $\wp(\Delta)$.

Loops and recursion make the blocks of P interdependent and hence a denotational semantics is built with a fixpoint computation: one improves the *empty* interpretation ι_0, such that $\iota_0(b) = \emptyset$ for all blocks b of P, into $\iota_1 = T_P(\iota_0)$ and iterates the application of T_P until a fixpoint (for better efficiency, our implementation performs local, smaller fixpoints over the strongly-connected components of blocks).

Definition 7 (Denotational Semantics)**.** *We define $T_P : \mathbb{I} \to \mathbb{I}$ as $T_P(\iota)(b) = [\![b]\!]^\iota$ for every $\iota \in \mathbb{I}$ and block b of P. Its least fixpoint exists and can be computed with a (possibly infinite) iterative application of T_P from ι_0 [12]. It is the* denotational semantics *of P.* □

Safe abstractions of T_P (such that in Section 4) reach the abstract fixpoint in a finite number of iterations.

This semantics gives only an input/output description of a piece of code (hence of P). A preliminary *magic-sets transformation* [12] of P recovers information at selected internal program points. It adds new blocks whose denotation gives information at those points. For nullness analysis, we select those just before a dereferencing bytecode *i.e.,* a `getfield`, `putfield` or `call` (for full Java bytecode, also `arraylength`, `throw`, `arrayload` and `arraystore` [10]), so we can check if they are safe.

4 Nullness Analysis

We define here an abstract interpretation [4] of the semantics of Section 3. The latter works over $\wp(\Delta)$; it is built from basic sets, one for each bytecode, with three operators ;, $\cup$ and $extend$. Hence we define correct abstractions of those sets and operators.

Our abstract domain is a *natural* choice for nullness analysis since it expresses logical relations between nullness of variables (*i.e.,* local variables and stack elements) in the input or output state of denotations. We first define a function that extracts the variables holding `null` in a state. We use identifiers l_k for the kth local variable, s_k for the kth stack element (s_0 is the base of the stack) and e to mean that the state is in $\underline{\Xi}$.

Definition 8 (Nullness Extractor)**.** *Let $\sigma \in \Sigma_{i,j}$. We define the* nullness extractor

$$nullness(\sigma) = \begin{cases} \left\{ l_k \;\middle|\; \begin{array}{l} l[k] = \texttt{null} \\ 0 \le k < i \end{array} \right\} \cup \left\{ s_k \;\middle|\; \begin{array}{l} v_k = \texttt{null} \\ 0 \le k < j \end{array} \right\} \\ \textit{if } \sigma = \langle l \parallel v_{j-1} :: \cdots :: v_0 \parallel \mu \rangle \\ \{ l_k \mid l[k] = \texttt{null},\ 0 \le k < i \} \cup \{e\} \\ \textit{if } \sigma = \underline{\langle l \parallel v_0 \parallel \mu \rangle}. \end{cases}$$

□

We remind that the stack of the exceptional states contains one element only, a location (hence non-`null`).

Example 2. *Let $\sigma \in \Xi_{2,3}$ from Example 1. Since $\ell, \ell', \ell'' \in \mathbb{L}$, $nullness(\sigma) = \{l_0, l_1, s_0, s_1, s_2\}$.* □

We put $\check{}$ over the variables holding `null` in the input of a denotation and $\hat{}$ over those holding `null` in its output. If S is a set of identifiers, then $\check{S} = \{\check{v} \mid v \in S\}$ and $\hat{S} = \{\hat{v} \mid v \in S\}$.

Definition 9 ($\mathbb{NULL}$ Abstract Domain)**.** *Let* $i_1, j_1, i_2, j_2 \in \mathbb{N}$. *The abstract domain* $\mathbb{NULL}_{i_1,j_1 \to i_2,j_2}$ *is the set of Boolean formulas over* $\{\check{e}, \hat{e}\} \cup \{\check{l}_k \mid 0 \leq k < i_1\} \cup \{\check{s}_k \mid 0 \leq k < j_1\} \cup \{\hat{l}_k \mid 0 \leq k < i_2\} \cup \{\hat{s}_k \mid 0 \leq k < j_2\}$ *(modulo logical equivalence).* □

Example 3. *We have* $\phi = (\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\check{s}_0 \leftrightarrow \hat{s}_0) \wedge (\check{s}_1 \leftrightarrow \hat{s}_1) \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{s}_2 \leftrightarrow \hat{l}_0) \in \mathbb{NULL}_{2,3 \to 2,2}$. □

A formula $\phi \in \mathbb{NULL}$ abstracts those denotations that behave, *w.r.t.* nullness, in a way *compatible* with ϕ.

Definition 10 (Concretisation Map)**.** *We define* $\gamma : \mathbb{NULL}_{i_1,j_1 \to i_2,j_2} \to \wp(\Delta_{i_1,j_1 \to i_2,j_2})$ *as* $\gamma(\phi) = \{\delta \in \Delta_{i_1,j_1 \to i_2,j_2} \mid$ *for all* $\sigma \in \Sigma_{i_1,j_1}$ *s.t.* $\delta(\sigma)$ *is defined,* $\check{nullness}(\sigma) \cup \hat{nullness}(\delta(\sigma)) \models \phi\}$. □

Proposition 1. $\mathbb{NULL}_{i_1,j_1 \to i_2,j_2}$ *is an abstract interpretation of* $\wp(\Delta_{i_1,j_1 \to i_2,j_2})$ *with concretisation* γ. □

Example 4. *Consider the denotation* $store\ 0$ `java.lang.Object` *(Section 3) and* ϕ *from Example 3. Then* $(store\ 0$ `java.lang.Object`$) \in \gamma(\phi)$ *since that bytecode does not modify local variable* 0 *(*$\check{l}_0 \leftrightarrow \hat{l}_0$*) nor the base of the stack (*$\check{s}_0 \leftrightarrow \hat{s}_0$*) nor the element above (*$\check{s}_1 \leftrightarrow \hat{s}_1$*); it is only defined on normal states (*$\neg\check{e}$*) and yields a normal state (*$\neg\hat{e}$*); the output local variable* 0 *is an alias of the top of the input stack (*$\check{s}_2 \leftrightarrow \hat{l}_0$*).* □

Figure 3 defines correct abstractions for the bytecodes in Section 3. For a simple notation, a formula U (for *unchanged*) expresses that the input local variables L and the input stack elements S of a bytecode, which are also in the output and hold the same value as in the input, keep their nullness. For S, this is only checked when no exception is thrown, since otherwise the only output stack element is non-`null`.

Definition 11. *Let sets* S *(of stack elements) and* L *(of local variables) be the input variables that after all executions of a given bytecode in a given program point (only after the normal ones for* S*) survive with unchanged value. Then* $U = \wedge_{v \in L}(\check{v} \leftrightarrow \hat{v}) \wedge (\neg\hat{e} \to \wedge_{v \in S}(\check{v} \leftrightarrow \hat{v})) \wedge (\hat{e} \to \neg\hat{s}_0)$. □

Example 5. *Bytecode* `store 0 java.lang.Object`*, in a program point with* 2 *local variables and* 3 *stack elements, lets only* l_1 *and* s_0*,* s_1 *survive and keep their value. Here,* $U = (\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\neg\hat{e} \to ((\check{s}_0 \leftrightarrow \hat{s}_0) \wedge (\check{s}_1 \leftrightarrow \hat{s}_1))) \wedge (\hat{e} \to \neg\hat{s}_0)$. □

$$
\begin{aligned}
(const\ v)^{\mathbb{NULL}} &= \begin{cases} U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge \hat{s}_j & \text{if } v = \texttt{null} \\ U \wedge \neg\check{e} \wedge \neg\hat{e} & \text{if } v \neq \texttt{null} \end{cases} \\
(load\ k\ t)^{\mathbb{NULL}} &= U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{l}_k \leftrightarrow \hat{s}_j) \\
(store\ k\ t)^{\mathbb{NULL}} &= U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{l}_k) \\
(ifne\ t)^{\mathbb{NULL}} &= U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge \neg\check{s}_{j-1} \\
(new\ \kappa)^{\mathbb{NULL}} &= U \wedge \neg\check{e} \wedge (\neg\hat{e} \to \neg\hat{s}_j) \\
(getfield\ \kappa.f:t)^{\mathbb{NULL}} &= U \wedge \neg\check{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{e}) \\
(putfield\ \kappa.f:t)^{\mathbb{NULL}} &= U \wedge \neg\check{e} \wedge (\check{s}_{j-2} \leftrightarrow \hat{e}) \\
(throw\ \kappa)^{\mathbb{NULL}} &= U \wedge \neg\check{e} \wedge \hat{e} \\
(catch)^{\mathbb{NULL}} &= U \wedge \check{e} \wedge \neg\hat{e} \\
(exception\_is\ K)^{\mathbb{NULL}} &= U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge \neg\check{s}_0 \\
(receiver\_is\ K)^{\mathbb{NULL}} &= U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge \neg\check{l}_0 \\
(return\ t)^{\mathbb{NULL}} &= U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{s}_0) \\
(return)^{\mathbb{NULL}} &= U \wedge \neg\check{e} \wedge \neg\hat{e}.
\end{aligned}
$$

Figure 3. Bytecode abstraction for nullness, in a program point with j stack elements.

For simplicity, we do not distinguish variables of primitive and reference type. For instance, for `store 0 java.lang.Object` in Example 4, the sub-formula $\check{l}_1 \leftrightarrow \hat{l}_1$ of ϕ is useless if local 1 has primitive type, for which nullness is meaningless. For efficiency, our implementation removes useless sub-formulas, without affecting the precision of the analysis.

Let us comment on Figure 3. Bytecodes are run only if the preceding one does not throw any exception ($\neg\check{e}$) but `catch` requires an exception to be thrown ($\check{e}$). They state if they never throw any exception ($\neg\hat{e}$) or always do it ($\hat{e}$), like `throw`; bytecode `new` leaves this undefined since $\mathbb{NULL}$ knows nothing about the amount of available memory; the dereferencing bytecodes `getfield` and `putfield` throw an exception if and only if their receiver is `null` (namely, for `getfield`, we state $\check{s}_{j-1} \leftrightarrow \hat{e}$); for full Java bytecode we use $\to$ instead of $\leftrightarrow$ here, since an exception might also be thrown for other reasons. Bytecode `const null` states that it pushes `null` on the stack ($\hat{s}_j$). Bytecode `load` k t copies the nullness of input local variable k into that of the top of the output stack ($\check{l}_k \leftrightarrow \hat{s}_j$); `store` k t does the opposite. Bytecode `ifne` wants a non-`null` top of the input stack ($\check{s}_{j-1}$) or otherwise it is undefined. Bytecode `new` states that if it

throws no exception then the top of the output stack is non-null ($\neg\hat{s}_j$) since it is a reference to a new object. Bytecode getfield says nothing about the nullness of the field (Section 5 improves on this). Bytecode exception_is (respectively, receiver_is) requires the only stack element (respectively, local variable 0) non-null ($\check{s}_0$, respectively $\check{l}_0$) or otherwise it is undefined. Bytecode return t states that the top of the input stack is null if and only if the only output stack element is null ($\check{s}_{j-1} \leftrightarrow \hat{s}_0$).

Example 6. *Consider* new java.lang.Object*, run in a program point with $i = 2$ local variables and $j = 2$ stack elements. We have $U = (\check{l}_0 \leftrightarrow \hat{l}_0) \wedge (\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\neg\hat{e} \rightarrow ((\check{s}_0 \leftrightarrow \hat{s}_0) \wedge (\check{s}_1 \leftrightarrow \hat{s}_1)))\wedge(\hat{e} \rightarrow \neg\hat{s}_0)$. From Figure 3 its approximation is $\phi_1 = U \wedge \neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{s}_2) = (\check{l}_0 \leftrightarrow \hat{l}_0) \wedge (\check{l}_1 \leftrightarrow \hat{l}_1) \wedge \neg\check{e} \wedge (\neg\hat{e} \rightarrow ((\check{s}_0 \leftrightarrow \hat{s}_0) \wedge (\check{s}_1 \leftrightarrow \hat{s}_1) \wedge \neg\hat{s}_2)) \wedge (\hat{e} \rightarrow \neg\hat{s}_0)$ i.e., the bytecode is only run from a normal state ($\neg\check{e}$), local variables 0 and 1 are unchanged ($(\check{l}_0 \leftrightarrow \hat{l}_0) \wedge (\check{l}_1 \leftrightarrow \hat{l}_1)$) and if no exception is thrown ($\neg\hat{e}$) then no stack element is changed ($(\check{s}_0 \leftrightarrow \hat{s}_0) \wedge (\check{s}_1 \leftrightarrow \hat{s}_1)$) and the new top of the stack is non-null ($\neg\hat{s}_2$). Otherwise, the stack contains only a non-null exception ($\neg\hat{s}_0$).* □

Example 7. *Consider* store 0 java.lang.Object*, run in a program point with $i = 2$ local variables and $j = 3$ stack elements. Example 5 gives U. From Figure 3 its approximation is $\phi_2 = U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{s}_2 \leftrightarrow \hat{l}_0) = (\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\check{s}_0 \leftrightarrow \hat{s}_0) \wedge (\check{s}_1 \leftrightarrow \hat{s}_1) \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{s}_2 \leftrightarrow \hat{l}_0)$ i.e., ϕ from Example 3. Example 4 showed that (store 0* java.lang.Object*) $\in \gamma(\phi)$.* □

The result of Example 7 is not a coincidence.

Proposition 2. *The approximations of Figure 3 are correct w.r.t. the denotations of Section 3 i.e., for all bytecode* ins *we have $ins \in \gamma(ins^{\mathbb{NULL}})$.* □

Denotations are composed by ; and their abstractions by $;^{\mathbb{NULL}}$. The definition of $\phi_1;^{\mathbb{NULL}} \phi_2$ matches the output variables of ϕ_1 with the corresponding input variables of ϕ_2. To avoid name clashes, they are first renamed apart and then projected away.

Definition 12. *Let $\phi_1, \phi_2 \in \mathbb{NULL}$. Their* sequential composition *is $\exists_{\overline{V}}(\phi_1[\overline{V}/\hat{V}] \wedge \phi_2[\overline{V}/\check{V}])$, where $\overline{V}$ are fresh overlined variables.* □

Example 8. *Consider ϕ_1 from Example 6 and ϕ_2 from Example 7. Then $\phi_1;^{\mathbb{NULL}} \phi_2 = \exists_{\{\overline{e},\overline{l}_0,\overline{l}_1,\overline{s}_0,\overline{s}_1,\overline{s}_2\}}(\check{l}_0 \leftrightarrow \overline{l}_0) \wedge (\check{l}_1 \leftrightarrow \overline{l}_1) \wedge \neg\check{e} \wedge (\neg\overline{e} \rightarrow ((\check{s}_0 \leftrightarrow \overline{s}_0) \wedge (\check{s}_1 \leftrightarrow \overline{s}_1) \wedge \neg\overline{s}_2)) \wedge (\overline{l}_1 \leftrightarrow \hat{l}_1) \wedge (\overline{s}_0 \leftrightarrow \hat{s}_0) \wedge (\overline{s}_1 \leftrightarrow \hat{s}_1) \wedge \neg\overline{e} \wedge \neg\hat{e} \wedge (\overline{s}_2 \leftrightarrow \hat{l}_0) = (\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\check{s}_0 \leftrightarrow \hat{s}_0) \wedge (\check{s}_1 \leftrightarrow \hat{s}_1) \wedge \neg\check{e} \wedge \neg\hat{e} \wedge \neg\hat{l}_0$. That is, the sequential execution of* new java.lang.Object *and* store 0 java.lang.Object *keeps the nullness of local variable 1 and of the two stack elements; it is run in a normal state; at its end there is no exception and local variable 0 is non-null (it holds a new object).* □

The second semantical operator is $extend$. Let ϕ approximate the nullness behaviour of method $M = \kappa.m(t_1, \ldots, t_n) : t$; ϕ's variables are among $\check{l}_0, \ldots, \check{l}_n$ (the arguments including this), $\hat{s}_0$ (if M does not return void), $\hat{l}_0, \hat{l}_1 \ldots$ (the final values of M's local variables), $\check{e}$ and $\hat{e}$. Let method C call M. The final values of M's local variables are irrelevant to C and we remove them by computing $\exists_{\{\hat{l}_0,\hat{l}_1\ldots\}}\phi$; C holds the arguments in the $n+1$ topmost elements of its stack, of height $b+n+1$ (b is the number of non-argument stack elements of C); then we rename $\check{l}_0$ into $\check{s}_b$, $\check{l}_1$ into $\check{s}_{b+1}$ and so on; similarly, we rename $\hat{s}_0$ (if any) into $\hat{s}_b$. Finally, we state that $\check{s}_b$ is non-null or an exception is thrown and that the local variables of C and its b lowest stack elements keep their nullness (U).

Definition 13. *Let $i, j \in \mathbb{N}$ and $M = \kappa.m(t_1, \ldots, t_n) : t$ with $j = b + n + 1$ and $b \geq 0$. Define $(extend_M^{i,j})^{\mathbb{NULL}} : \mathbb{NULL}_{n+1,0\rightarrow i',r} \rightarrow \mathbb{NULL}_{i,j\rightarrow i,b+r}$ with $r = 0$ if $t =$* void *and $r = 1$ otherwise, as $(extend_M^{i,j})^{\mathbb{NULL}}(\phi) = U \wedge \neg\check{e} \wedge (\check{s}_b \rightarrow \hat{e}) \wedge (\neg\check{s}_b \rightarrow ((\exists_{\{\hat{l}_0,\hat{l}_1\ldots\}}\phi)[\check{s}_{i+b}/\check{l}_i \mid 0 \leq i \leq n][\hat{s}_b/\hat{s}_0]))$.* □

Example 9. *The body of the constructor $M =$* java.lang.Object.⟨init⟩() : void *of* java.lang.Object *is* receiver_is A; return*, where A is the set of all classes. From Figure 3, its approximation is $\phi = \neg\check{l}_0 \wedge \neg\hat{l}_0 \wedge \neg\check{e} \wedge \neg\hat{e}$. Let us call M in a program point with 2 local variables and 3 stack elements. We have $n = 0$ and $b = 2$. The approximation of the call is $(extend_M^{2,3})^{\mathbb{NULL}}(\phi) = U \wedge \neg\check{e} \wedge (\check{s}_2 \rightarrow \hat{e}) \wedge (\neg\check{s}_2 \rightarrow \exists_{\{\hat{l}_0\}}\phi[\check{s}_2/\check{l}_0]) = U \wedge \neg\check{e} \wedge (\check{s}_2 \rightarrow \hat{e}) \wedge (\neg\check{s}_2 \rightarrow (\neg\check{s}_2 \wedge \neg\check{e} \wedge \neg\hat{e})) = U \wedge \neg\check{e} \wedge (\check{s}_2 \leftrightarrow \hat{e}) = (\check{l}_0 \leftrightarrow \hat{l}_0) \wedge (\check{l}_1 \leftrightarrow \hat{l}_1) \wedge \neg\check{e} \wedge (\neg\hat{e} \rightarrow ((\check{s}_0 \leftrightarrow \hat{s}_0) \wedge (\check{s}_1 \leftrightarrow \hat{s}_1) \wedge \neg\check{s}_2)) \wedge (\hat{e} \rightarrow (\neg\hat{s}_0 \wedge \check{s}_2))$. It entails that, if the call does not throw any exception, then the top of the stack of the caller was non-null ($\neg\check{s}_2$).* □

The third semantical operator is $\cup$ of two sets of denotations. Its approximation is $\cup^{\mathbb{NULL}} = \vee$.

Proposition 3. *The operators* $;^{\mathbb{NULL}}$*,* $extend^{\mathbb{NULL}}$ *and* $\cup^{\mathbb{NULL}}$ *are correct.* □

The number of Boolean formulas over a given set of variables is finite (modulo equivalence) and the abstract fixpoint is reached in a finite number of iterations. For each `getfield`, `putfield` and `call` bytecode op in P that dereferences v_{op}, the magic-sets transformation gives us a formula ψ_{op} which holds just before op. If ψ_{op} entails $\neg\hat{v}_{op}$ then op is safe.

5 Oracle Semantics for Fields

The analysis in Section 4 never assumes that fields hold a non-`null` value (see `getfield` in Figure 3). This hypothesis is conservative but too strong. We show here how we determine non-`null` fields.

A *candidate* field is initialised before being read.

Definition 14 (Candidate Field)**.** *A field* $\kappa.f:t$ *is* candidate *if* $t \in \mathbb{K}$ *and for every execution path* x *in every constructor of* κ*, if* x *ends with* `return` *then there is a* `putfield` $\kappa.f:t$ *in* x *over the created object and if* x *contains a* `getfield` $\kappa.f:t$ *then it also contains a previous* `putfield` $\kappa.f:t$ *over the created object.* □

Fields `f` and `g` in Figure 1 are candidate, but only `k` in

```
public class C {
 private Object h, k;
 public C() {
  Object t = this.h; this.h = this; this.k = null;
 }
}
```

Definition 14 does not consider paths ending with `throw` since if the construction of an object o ends in an exception then o cannot be used [10].

We use a preliminary definite aliasing analysis to check if a `putfield` works on the created object, by checking if the receiver is an alias of local 0; we consider no field as candidate when a constructor contains a (legal but unusual) `store 0` t. After the alias analysis, being candidate is just a syntactical property, that we check through a graph algorithm taking into account helper functions for better precision (as `helper()` in Figure 1). However, being candidate does not guarantee that the field never contains `null`.

Definition 15 (Non-`null` Field)**.** *A candidate field* $\kappa.f:t$ *is* non-`null` *if* P *never writes* `null` *in it.* □

Then when P reads a non-`null` field, it does not find `null`. Being non-`null` is a semantical property, since we need to know what flows inside the field. Let us hence define an *oracle*, telling us if a field is non-`null`.

Definition 16 (Oracle)**.** *An* oracle *is a set of candidate fields. The set of oracles is* $\mathbb{O}$*. An oracle* $O \in \mathbb{O}$ *is* correct *if every* $\kappa.f:t \in O$ *is non-*`null`*.* □

For instance, $\{$`f`,`g`$\}$, $\{$`f`$\}$ and $\emptyset$ and correct oracles for Figure 1. We redefine the approximation of `getfield` so that, given $O \in \mathbb{O}$, the fields in O are assumed non-`null`. Namely, $(getfield\ \kappa.f:t)^{\mathbb{NULL}}_{O}$ is

$$\begin{cases} U \wedge \neg\check{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{e}) \wedge (\neg\hat{e} \rightarrow \neg\hat{s}_{j-1}) & \text{if } \kappa.f:t \in O \\ U \wedge \neg\check{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{e}) & \text{if } \kappa.f:t \notin O \end{cases}$$

i.e., if $\kappa.f:t \in O$, if no exception is thrown then the top of the output stack is non-`null` ($\neg\hat{e} \rightarrow \neg\hat{s}_{j-1}$). This analysis, parameterised *w.r.t.* O, is correct if O is correct, but may be incorrect otherwise; moreover, the larger a correct O, the more precise is the analysis.

Proposition 4. *If* $O \in \mathbb{O}$ *is correct, then the nullness analysis parameterised w.r.t.* O *is correct.* □

The problem is to find a correct $O \in \mathbb{O}$. The obvious choice $O = \emptyset$ is correct but leads us to the analysis in Section 4. The following result will help us.

Proposition 5. *Define* $F_P : \mathbb{O} \to \mathbb{O}$ *as*

$$F_P(O) = \left\{ \kappa.f:t \in O \;\middle|\; \begin{array}{l} \textit{our nullness analysis} \\ \textit{parameterised w.r.t. } O \textit{ proves} \\ \textit{that all } \texttt{putfield}\ \kappa.f:t \textit{ in } P \\ \textit{write a non-}\texttt{null}\textit{ value} \end{array} \right\}.$$

If O *is a fixpoint of* F_P *then* O *is correct.* □

By computing $F_P(O)$, one applies our nullness analysis parameterised *w.r.t.* O and checks in which fields of O the program write non-`null` values only. By definition, $F_P(O) \subseteq O$. Take O_0 equal to the set of all candidate fields and compute $O_1 = F_P(O_0)$. If $O_1 = O_0$ then O_0 is correct (Proposition 5); otherwise $O_0 \supset O_1$ and compute $O_2 = F_P(O_1)$; again, if $O_2 = O_1$ then O_1 is correct; otherwise $O_1 \supset O_2$ and compute $O_3 = F_P(O_2)$ and so on. Since the number of candidate fields of P is finite, the decreasing chain $O_0 \supset O_1 \supset O_2 \supset O_3 \supset \ldots$ must be finite

and converge to a correct oracle. In words, one starts with the optimistic hypothesis that all candidate fields are non-null and iteratively remove those that have no proof of being non-null. When no more fields are removed, one gets a correct oracle (a set of non-null fields) and the last iteration is correct (Proposition 4). For instance, take $O_0 = \{\texttt{f}, \texttt{g}\}$ in Figure 1; we have $F_P(O_0) = O_0$ so O_0 is correct. For the class C above, take $O_0 = \{\texttt{k}\}$ and $F_P(O_0) = \emptyset$ is correct. An upper bound to the number of needed iterations is the possibly large number of candidate fields of P. In practice, no more than four iterations are used even for the largest programs of Section 6. Moreover, the first iteration might be expensive but caching makes the subsequent iterations quicker than the first one.

Static fields are accomodated in our framework. A candidate static field is defined as in Definition 14 by using putstatic and getstatic instead of putfield and getfield and considering the only class constructor $\langle$clinit$\rangle$. Aliasing is not used for putstatic since there is no receiver object.

6 Experiments

We have implemented our analysis in Java inside our JULIA analyser, coding formulas as binary decision diagrams [1] with the BUDDY library. Figure 4 shows that it scales well on an AMD Opteron processor 280 at 2.4Ghz with 4 gigabytes of RAM and Sun jdk 1.5. We included library methods inside java.lang.* and java.util.* classes and approximated the others with a *worst-case assumption i.e.,* by assuming them to return a possibly null value.

Figure 5 compares our analysis with the implementation NIT of [9] (we thank Laurent Hubert for his help with NIT). Times are global (preprocessing plus analysis). Our analysis, coded in Java, is faster than the natively compiled OCaml of NIT. The latter did not analyse Kitten nor Julia for some error in the application extraction. *W.r.t.* precision, we observe that NIT is more precise than the analysis in [9]. Hence the results for NIT are upper bounds to the actual precision of [9]. Only instance fields are approximated by NIT; hence we have disabled the analysis of static fields from JULIA by assuming that they always hold a possibly null value. Our notion of non-null field (Definition 15) is stronger than that in [9], where a non-null field can be read before being initialised. Hence there are more non-null fields in the sense of [9] than in ours. Nevertheless, our analysis always finds more non-null fields (in our sense) than NIT (in the sense of [9]). The only exception is JavaCup, but the five non-null fields which are not found by JULIA are read before being initialised, so they cannot be non-null for our definition. A more fair comparison is the amount of getfields, putfields and instance calls proved safe, for which JULIA is more precise (Figure 5). By dealing with static fields also, precision increases further, as the last column of Figure 5 shows.

7 Conclusion

Our analysis is clean, fast, scalable and more precise than others but it can be improved. Namely, it cannot prove the call in if (this.f != null) this.f.foo() safe when f might hold null. When f is not initialised in a constructor but is made non-null by a method, say, set(), *always* called before reading f, then it fails to prove f non-null: it misses information about the calling order of methods. It also conservatively assumes that the elements of an array are potentially null since it does not know which portions of the array hold non-null values.

References

[1] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[2] M. Cielecki, J. Fulara, K. Jakubczyk, and L. Jancewicz. Propagation of JML non-null Annotations in Java Programs. In R. Gitzel, M. Aleksy, and M. Schader, editors, *Proc. of the 4th Int. Symposium on Principles and Practice of Programming in Java (PPPJ'06)*, pages 135–140, Mannheim, Germany, August-September 2006. ACM.

[3] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In *Proc. of the 2nd Int. Symposium on Programming*, pages 106–130, Paris, France, April 1976. Dunod.

[4] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.

program	methods	prepr.	analysis	derefs	get's	put's	calls	fields
JLex	447	4.90	2.79	69.54%	72.01%	71.76%	52.31%	48.07%
CaffeineMark	636	5.04	2.59	100.00%	100.00%	100.00%	100.00%	25.00%
JavaCup	934	7.80	10.08	86.39%	47.78%	96.04%	93.92%	55.55%
JavaCC	1550	10.37	33.66	84.54%	89.87%	96.16%	76.76%	57.91%
Kitten	2134	10.62	20.43	83.91%	62.70%	97.93%	85.56%	41.28%
Jess	3316	19.60	38.07	88.07%	97.51%	99.58%	79.59%	45.38%
Julia	4039	36.65	63.94	85.88%	81.80%	98.91%	82.07%	53.64%

Figure 4. Number of methods (including `java.lang.*` and `java.util.*` classes), time in seconds for preprocessing (application extraction, aliasing analysis), time in seconds for the nullness analysis, amount of dereferences proved safe, global and restricted to `getfield`s, `putfield`s and `call`s, and of reference fields proved non-`null`. Dereferences and fields in libraries are not counted.

	Our analysis					The analysis in [9]					All fields
program	time	fs	get's	put's	calls	time	fs	get's	put's	calls	calls
JLex	15.03	46	72.01%	71.76%	49.90%	42.76	43	68.67%	61.75%	36.57%	52.77%
CaffeineMark	14.41	1	100.00%	100.00%	61.94%	44.86	1	98.64%	100.00%	54.60%	100.00%
JavaCup	28.56	26	48.18%	96.04%	87.16%	44.21	31	47.41%	96.04%	76.38%	94.27%
JavaCC	55.01	47	89.87%	96.16%	73.26%	56.19	47	86.21%	94.17%	65.74%	76.76%
Kitten	48.36	52	62.64%	97.93%	82.28%	-	-	-	-	-	85.56%
Jess	288.15	107	97.53%	99.60%	76.52%	325.85	96	93.85%	97.49%	66.63%	78.40%
Julia	104.15	166	81.78%	98.91%	77.03%	-	-	-	-	-	81.75%

Figure 5. Time in seconds, number *fs* of instance reference fields proved non-`null` and of `getfield`s, `putfield`s and `call`s proved safe. The last column reports the number of instance `call`s proved safe by our analysis with all fields. Libraries are included but their dereferences are not counted.

[5] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, COMPAQ Systems Research Center, 1998.

[6] M. Fähndrich and K. R. M. Leino. Declaring and Checking non-null Types in an Object-Oriented Language. In R. Crocker and G. L. J. Steel, editors, *Proc. of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, pages 302–312, Anaheim, CA, USA, October 2003. ACM.

[7] C. Flanagan and K. R. M. Leino. Houdini, an Annotation Assistant for ESC/Java. In J. N. Oliveira and P. Zave, editors, *Proc. of the 2001 Int. Symposium of Formal Methods Europe (FME'01)*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517, Berlin, Germany, March 2001. Springer.

[8] D. Hovemeyer and W. Pugh. Finding More null Pointer Bugs, but not Too Many. In M. Das and D. Grossman, editors, *Proc. of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'07)*, pages 9–14, San Diego, CA, USA, June 2007. ACM.

[9] L. Hubert, T. Jensen, and D. Pichardie. Semantic Foundations and Inference of non-null Annotations. In G. Barthe and F. S. de Boer, editors, *Proc. of the 10th Int. Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*, volume 5051 of *Lecture Notes in Computer Science*, pages 132–149. Springer, 2008.

[10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[11] C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov. Java Bytecode Verification for `@NonNull` Types. In L. Hendren, editor, *Proc. of the 17th Int. Conference on Compiler Construction (CC'2008)*, volume 4959 of *Lecture Notes in Computer Science*, pages 229–244, Budapest, Hungary, March-April 2008. Springer.

[12] É. Payet and F. Spoto. Magic-Sets Transformation for the Analysis of Java Bytecode. In H. R. Nielson and G. Filé, editors, *Proc. of the 14th Int. Static Analysis Symposium (SAS'07)*, volume 4634 of *Lecture Notes in Computer Science*, pages 452–467, Kongens Lyngby, Denmark, August 2007. Springer.