Softw Syst Model (2011) 10:219–252
DOI 10.1007/s10270-009-0132-5

SPECIAL SECTION PAPER

Precise null-pointer analysis

Fausto Spoto

Received: 12 February 2009 / Revised: 22 August 2009 / Accepted: 18 September 2009 / Published online: 2 October 2009

Abstract In Java, C or C++, attempts to dereference the `null` value result in an exception or a segmentation fault. Hence, it is important to identify those program points where this undesired behaviour might occur or prove the other program points (and possibly the entire program) *safe*. To that purpose, null-pointer analysis of computer programs checks or infers non-`null` annotations for variables and object fields. With few notable exceptions, null-pointer analyses currently use run-time checks or are incorrect or only verify manually provided annotations. In this paper, we use abstract interpretation to build and prove correct a first, flow and context-sensitive static null-pointer analysis for Java bytecode (and hence Java) which *infers* non-`null` annotations. It is based on Boolean formulas, implemented with binary decision diagrams. For better precision, it identifies instance or static fields that remain *always* non-`null` after being initialised. Our experiments show this analysis faster and more precise than the correct null-pointer analysis by Hubert, Jensen and Pichardie. Moreover, our analysis deals with exceptions, which is not the case of most others; its formulation is theoretically clean and its implementation strong and scalable. We subsequently improve that analysis by using local reasoning about fields that are not *always* non-`null`, but happen to hold a non-`null` value *when they are accessed*. This is a frequent situation, since programmers typically check a field for non-nullness before its access. We conclude with an example of use of our analyses to infer null-pointer annotations which are more precise than those that other inference tools can achieve.

Communicated by Dr. Antonio Cerone.

F. Spoto (✉)
Università di Verona, Verona, Italy
e-mail: fausto.spoto@univr.it

Keywords Null-pointer analysis · Java bytecode · Static analysis · Abstract interpretation · Automatic software verification

1 Introduction

Imperative programming languages such as C, C++ and Java let one store a `null` value into a variable or field. *Dereferences*, i.e. field accesses, method calls and synchronisations on a lock, work on a *receiver* value and are *safe* when the latter is never `null`. Otherwise, an exception or a segmentation fault occurs. It is important to prove the absence of this programming error before the program is run or spot suspect program points where it can be generated. Moreover, even though dereferences are safe, programming languages such as Java do check for the nullness of the receiver at run-time: hence, by removing useless checks, one improves the efficiency of the program. Furthermore, proving dereferences safe simplifies the control-flow graph of the program by cutting spurious exceptional paths, which in turn improves efficiency and precision of subsequent static analyses. Finally, null-pointer annotations about the source code of a program are increasingly used as an important part of software documentation. A preliminary static *null-pointer analysis*, which is executed before the program is run, is however complex for modern object-oriented languages such as Java and C#, which allow uninitialised object fields, holding `null` by default. Moreover, objects can be just partially initialised by the constructors and they can be passed, in a partially unitialised state, from the constructor to auxiliary methods (*helper functions*) or to other constructors.

Current main-stream programming languages do not allow the specification of null-pointer annotations in the code. Nevertheless, this situation might change in the future and programmers will be allowed to annotate the code with warnings about possible null-pointers or with explicit nullness types. The syntax of these annotations is being standardised for the Java 7 release [4]. Extensions exist, in particular that used by the *Checker Framework* [24], where type annotations can be used wherever a type is allowed, also in casts or parameters of generic types. In the case of null-pointer annotations, both the dual `@Nullable` and `@NonNull` annotations are allowed, although it has been argued that, by letting `@NonNull` be the default, the resulting annotations are statistically smaller [6].

Many static null-pointer analyses have been developed in the past. A survey can be found in [11]. We briefly present here the main results, in order to appreciate the difference with the present work. A first class of analyses are those that do not require any preliminary null-pointer annotation of the code but rather *infer* those annotations in an automatic way. This class includes the first null-pointer analysis, presented in [8], where Cousot and Cousot defined a simple abstract domain for nullness of program variables. As a consequence, it completely misses the ability to approximate the nullness of the object fields. Nevertheless, it has been formally proved correct by using abstract interpretation [9], which is a generic methodology for defining new static analyses and proving their correctness as well as optimality. This work can be seen as the starting point of other works based on abstract interpretation. Namely, by expanding this abstract domain, the constraint-based analysis in [19] infers non-`null` annotations for the fields. It builds constraints about the initialisation status of fields by following the structure of the code. Its correctness proof is based on abstract interpretation and has been mechanically verified. An implementation exists for the Java bytecode. This significant result shows that global nullness analysis can *infer* non-`null` annotations, work for modern programming languages and lead to a reliable implementation. However, the analysis is not context-sensitive and better precision can be achieved (we show some examples in Sect. 2). Moreover, the analysis looks too complex to us: a variable can have several approximations (Raw, $Raw(X)$, $MayBeNull$, $NotNull$, $\ldots$) and seven distinct abstract domains are used. Nevertheless, it remains the best analysis for evaluating new, more precise automatic tools for null-pointer inference, working without any preliminary annotation. This is why in Sect. 7 we have compared our results with those of this analysis and shown ours more precise. In this first class of analyses there are tools that infer null-pointer annotations by using type systems, aiming at finding a consistent type annotation for the program. An example is the null-pointer analysis inside the JUSTADD tool [10], based on a type system similar to that in [13]. We note that the analysis in [19] has been proved more precise than that type system for typable programs. Finally, the DAIKON [12] tool is able to infer *likely* annotations by running the program on a test suite and by analysing the resulting traces. Since there is no guarantee of a complete coverage of all possible execution traces, the inferred `@NonNull` annotations only hold *likely*, while the inferred `@Nullable` annotations are always correct.

The second class of analyses requires preliminary null-pointer annotations about fields and methods of the analysed code, which can both be manually provided or derived by one of the techniques in the first class. The analysis in [22] works by propagating such annotations inside a class file; namely, the class verification algorithm [21] has been extended to propagate annotations intra-procedurally, by exploiting the explicit tests in the guards for achieving higher precision. This technique misses a global view of the code but has the great advantage of fitting inside the class verifier. Other analyses [7,15] are more global but based on incorrect/incomplete tools such as ESC/Java [20]. Type systems can also be used to check non-`null` annotations [13,14]. The results, however, are less precise than those in [19]. Some techniques infer null-pointer annotations from the tests in the guards of the conditionals [17,18,22].

Our null-pointer analysis belongs to the first class. Namely, we use abstract interpretation to define a simple abstract domain expressing logical constraints among the nullness of program variables in Java bytecode. We have chosen the Java bytecode since we want to check code downloaded from the net into client computers or phones. We use Java examples, but the analysis works for every language compiled to Java bytecode, or implemented by hand. Those nullness constraints of our analysis take the form of Boolean formulas, which opens the way to a fast implementation based on binary decision diagrams [5]. Fields are considered as *non*-`null` whenever it can be proved that they are always initialised, by every constructor of the class they belong to, and always get assigned a definitely non-`null` value. This is achieved by using an iterated *oracle-based* static analysis which looks for counterexamples to the non-nullness of the fields. The analysis is flow- and context-sensitive, provably correct and fully implemented for Java bytecode. It is more precise than that in [19] for an acceptable extra cost. We further improve its precision with local non-nullness information gathered from the guards of the conditionals, as in [17,18,22].

In more details, in this paper we make the following contributions:

- We formalise the semantics of Java bytecode and of its exception handling mechanism, getting a concrete semantics that we later abstract into null-pointer analyses;
- We define and prove correct a first static null-pointer analysis to infer non-`null` annotations for Java bytecode; it is *natural*, i.e. a variable is only approximated as `null` or non-`null`; it uses only one abstract domain of Boolean

formulas, efficiently implemented with binary-decision diagrams [5];
- We identify non-`null` fields with an iterated *oracle* version of the analysis above;
- We couple our first analysis with another static analysis, modelling those fields that are not *always* non-`null`, but rather non-`null` *in the context where they are used*, because they are *protected* by a preliminary nullness check;
- We show experimentally that the implementation of both our analyses is more precise than that of the analysis in [19]. Moreover, the second analysis is more precise than the first.

An important aspect of our work is that the concrete semantics of Java bytecode has been carefully devised in order to allow its simple and modular abstraction into static analyses. Hence it should also be appreciated the resulting high quality of the proofs, which are mostly *automatic*, modular and easily verifiable.

Formalisations of the exception mechanism of Java exist already, but they do not seem to have been used to derive and prove correct static null-pointer analyses. This is somewhat surprising since the handling of the `null` pointers in a Java program is strictly connected to its exception mechanism. Boolean formulas have been used to express *groundness* relationships between variables [3]. Here we also model exceptions with Boolean formulas and use all formulas, not just the positive ones. The use of an *oracle* for iterated applications of an analysis is new and we believe that it applies to other cases of analysis as well. Namely, it could, more generally, be applied to all those analyses that need to approximate properties of the fields that are often invariant after an object is created.

This paper is organised as follows. Section 2 shows examples of null-pointer analysis where our two analyses are more precise than others; Sect. 3 defines the concrete denotational semantics of Java bytecode that Sect. 4 abstracts into a null-pointer analysis; Sect. 5 describes the oracle approach for the fields; Sect. 6 shows the improvement of the analysis of Sect. 4 by collecting information on locally non-`null` fields, hence getting our second, improved null-pointer analysis; Sect. 7 shows the high precision of our two analyses through practical experiments over large software and describes their application for the automatic annotation of Java programs with nullness information. Section 8 concludes.

A preliminary and partial version of this paper appeared in [27]. This version

- contains the algorithm for computing the candidate fields in Sect. 5;
- contains the second, more precise null-pointer analysis dealing with locally non-`null` fields, in Sect. 6;
- shows updated experiments, larger introduction and conclusions;
- gives a description of the generation of null-pointer annotations from the results of our analyses, in Sect. 7;
- contains extra examples and all the proofs.

In particular, note that all the material in Sects. 6 and 8, as well as most of Sect. 7, is completely new and never published before.

2 Some examples of null-pointer analysis

Consider the Java program in Fig. 1, devised to test the ability of a null-pointer analysis and test its flow and context-sensitivity. The analysis that we will describe in Sect. 4 (and hence also the more precise analysis of Sect. 6) proves that fields `f` and `g` are always non-`null`, i.e. they never hold `null` after being initialised. It also proves that a `java.lang.NullPointerException` might only be thrown at the statement `p.f=new Object()` in the second constructor. This is an optimal result, since `n4` might actually hold `null` when `main()` calls the second constructor. *All*

```
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 n2 = 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
  }
}
```

Fig. 1 A program to analyse. We specify the constructor called by every `new Test` accesses to `g` inside `helper()` and `foo()` are marked instead as *safe*. Other analyses, such as [13,19], do not prove `f` nor `g` non-`null` nor the accesses inside `helper()` safe.

Our analysis in Sect. 4 assumes, initially, `f` and `g` optimistically non-`null` and then looks for a counterexample. Let us describe how it reasons. Method `helper()` writes `g` and is called by both constructors. The first passes `this`, always non-`null`, to `helper()`; the second passes `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, we see that the following considerations are exploited during the analysis:

1. the analyser must conclude that, after evaluating `p.f`, variable `p` is non-`null` or an exception is thrown: it must be *flow-sensitive*;
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`: it must be, again, flow-sensitive;
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 that these points are outside the reach of current analyses, since they require flow and context-sensitivity, as well as non-trivial reasonings about the exception mechanism of Java. Our analysis, instead, fulfills them and proves both `f` and `g` non-`null`. Our experiments (Sect. 7) confirm that it is actually more precise than the analysis in [19] and hence also more precise that in [13].

Despite these positive results, the analysis of Sect. 4 leaves space for improvements. Consider for instance the program in Fig. 2, which implements a linked list and operations over it. The analysis from Sect. 4 issues three false alarms (spurious warnings that do not correspond to an actual error):

```
unsafe operations inside private List.append
 (List):List:
 * calling method List.append(List):List
```

```
public class List {
  private Object head;
  private List tail;

  public static void main(String[] args) {
    List l1 = new List(new Object(),
                  new List(new Object(),null));
    List l2 = new List(new Object(),
                  new List(new Object(),null));
    l1.alternate(l2);
    l1.append(l2);
    l1.iter();
    l1.reverse();
  }

  public List(Object head, List tail) {
    this.head = head;
    this.tail = tail;
  }

  private void iter() {
    if (tail != null) tail.iter();
  }

  private List append(List other) {
    if (tail == null) return new List(head,other);
    else return new List(head,tail.append(other));
  }

  private List reverse() {
    if (tail == null) return this;
    else return tail.reverse().append(new List(head,null));
  }

  private List alternate(List other) {
    if (other == null) return this;
    else return new List(head,other.alternate(tail));
  }
}
```

Fig. 2 A class implementing a list. Field `tail` is not always non-`null`

```
unsafe operations inside private
 List.reverse():List:
 * calling method List.reverse():List

unsafe operations inside private
 List.iter():void:
 * calling method List.iter():void
```

This is due to the lack of precision about the nullness of field `tail`: it is true that `tail` actually holds `null` inside some object of class `List` (for instance, the tails of `l1` and `l2`), but its uses are safe since they are protected by explicit nullness checks (as for instance `tail != null` in `iter()`). However, the analysis of Sect. 4 and all other null-pointer analyses, as far as we know, are not able to exploit that extra information. Since this programming pattern is very frequent in practice, it must be considered for a precise null-pointer analysis. This is what we do with the analysis of Sect. 6 which, correctly, issues no warning at all when applied to the program in Fig. 2. To the best of our knowledge, no other null-pointer analysis is able to reach such level of precision. Our experiments in Sect. 7 show that this improvement significantly increases the precision of the analysis of real, large software.

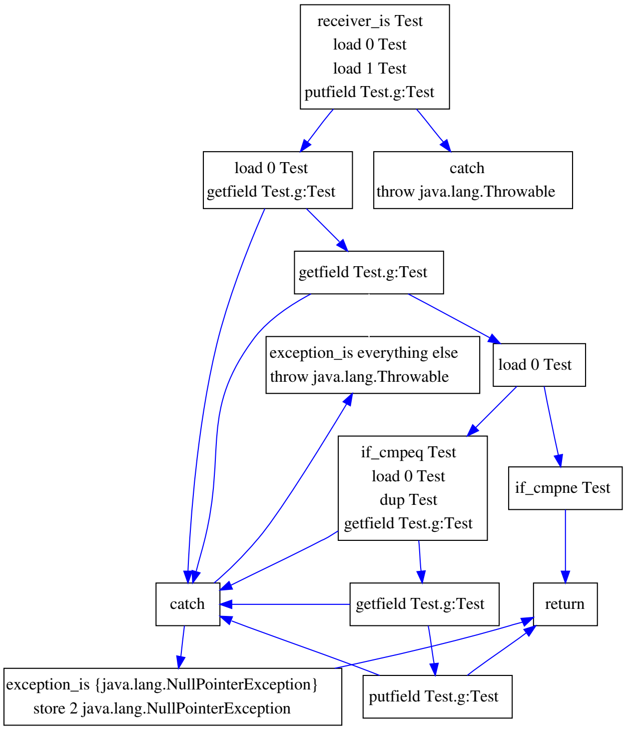

Fig. 3 The blocks of method `helper()` from Fig. 1

3 Denotational semantics of java bytecode

We describe here the denotational semantics for Java bytecode, that in [25] has been proved equivalent to an operational semantics. The only difference is that we also consider exceptions here, since they are an important ingredient of the semantics of the dereference of a value. We assume a program P given as a collection of graphs of *basic blocks* of code, one for each method. Figure 3 shows this graph for the method `helper()` in Fig. 1. A basic block does not contain jumping instructions, but for their last instruction, and is not the target of any jumping instruction, but for its first instruction [1]. In the class files, exception handlers are actually represented by suitable exception tables, that specify which kind of exception is caught by each exception handler inside a given portion of the code. That notation is compact but awkward for static analysis, since the control-flow of the program is not apparent. For this reason, we prefer to link bytecodes which might throw exceptions to an exception handler starting with a `catch`, possibly followed by bytecodes that select the right kind of exception by using appropriate `exception_is` bytecodes. Those bytecodes check the run-time class of the exception, which in Java bytecode is, by definition, held in the first and only stack element of an exceptional state. In Fig. 3, the topmost `putfield` has a default exception handler throwing back any exception to the caller; the others have a handler for `java.lang.NullPointerException` and throw back the other kinds of exceptions to the caller otherwise (the writing `everything else` used in the central block in Fig. 3 stands for the set of all other exceptions). Conditional bytecodes, such as `if_cmpeq` in Fig. 1, are compiled into two branches of computation, the first starting with the conditional bytecode itself and the other with its negation, that is `if_cmpne` in Fig. 1. In this way, we can interpret those conditional bytecodes as *filters*, that just pop their arguments from the stack if the condition that they embed is satisfied and block the computation otherwise. Formally, this will be translated in a semantics that makes them undefined when the condition is not satisfied, so that there is no next state. Our analyser constructs these graphs from Java bytecode `.class` files.

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. The extension to the full sequential Java bytecode is only a technical issue; consequently, our implementation and examples deal with the full language, so that for instance we allow Fig. 1 to contain some static methods.

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 $\mathbb{T} = \mathbb{K} \cup \{\texttt{int}\}$. The set of fields is $\mathbb{F}$ and the set of methods is $\mathbb{M}$. A class $\kappa \in \mathbb{K}$ has *instance fields* $\kappa.f : t \in \mathbb{F}$ (field f of type $t \in \mathbb{T}$ defined in κ) and *instance methods* $\kappa.m(t_1, \ldots, t_n) : t \in \mathbb{M}$ (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`.

A *state* provides *values* to program variables.

Definition 2 (State) A *value* is an element of $\mathbb{Z} \cup \mathbb{L} \cup \{\texttt{null}\}$, where $\mathbb{L}$ is a set of *memory locations*. A *state* is a triple $\langle l \,\|\, s \,\|\, \mu \rangle$ 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, i.e. 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 $\langle l \,\|\, s \,\|\, \mu \rangle$ if $v \in \mathbb{Z}$ and $t = \texttt{int}$, or if $v = \texttt{null}$ and $t \in \mathbb{K}$, or if $v \in \mathbb{L}$, $t \in \mathbb{K}$ and $\mu(v).\kappa \leq t$.

Example 1 State $\sigma = \langle [\texttt{null}, \ell] \,\|\, \ell'' :: \ell'' :: \ell' \,\|\, \mu \rangle \in \Xi_{2,3}$, 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. The latter contain one stack element only, which is the location of the thrown exception object. This is true also in the presence of nested exception handlers [21].

Definition 3 (JVM State) The set of *JVM states* (from now on 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 ins* : $\Sigma \to \Sigma$, i.e. a map from an *initial* to a *final* state. We require that the number of local variables can only increase from the initial to the final state. This corresponds to the fact that new local variables can be defined during the execution of a piece of code. Also the set of locations can only grow. This corresponds to the fact that during the execution of a piece of code new objects can be allocated in memory. If we had to consider the presence of a garbage collector, the latter constraint should be modified by requiring that denotations do not erase *reachable* locations. For simplicity, we do not consider a garbage collector here.

Definition 4 (Denotation) A *denotation* δ is a partial map from an *input* or *initial* state to an *output* or *final* state; we require that if $\delta(\langle l \parallel s \parallel \mu\rangle) = \langle l' \parallel s' \parallel \mu'\rangle$ (both states are possibly underlined) then l is not longer than l' and $dom(\mu) \subseteq dom(\mu')$. 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 in the states. The *sequential composition* of $\delta_1, \delta_2 \in \Delta$ is $\delta_1; \delta_2 = \lambda\sigma.\delta_2(\delta_1(\sigma))$, which is undefined when $\delta_1(\sigma)$ is undefined or when $\delta_2(\delta_1(\sigma))$ is undefined.

In the composition of denotations $\delta_1; \delta_2$, the idea is that δ_1 describes the behaviour of a piece of code c_1, while δ_2 describes the behaviour of a piece of code c_2; hence, denotation $\delta_1; \delta_2$ describes the behaviour of the sequential execution of c_1 followed by c_2.

We define now a denotation for each bytecode instruction in our language. This will be the *semantics* of the bytecode, since it specifies, in a denotational way, the behaviour of the bytecode when it is run from each given initial state. At a given program point, the number i of local variables and j of stack elements and their types are statically known [21]. Hence, in the following, we silently assume that the semantics of the bytecodes is undefined for input states of wrong sizes or types.

3.1 Basic instructions

Bytecode `const` v pushes $v \in \mathbb{Z} \cup \{\texttt{null}\}$ on the stack. Its semantics is the denotation

$$const\ v = \lambda\langle l \parallel s \parallel \mu\rangle.\langle l \parallel v :: s \parallel \mu\rangle$$

(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. Its semantics is

$$dup\ t = \lambda\langle l \parallel top :: s \parallel \mu\rangle.\langle l \parallel top :: top :: s \parallel \mu\rangle.$$

Bytecode `load` k t pushes on the stack the value of local variable number k, which must exist and have type t. Hence

$$load\ k\ t = \lambda\langle l \parallel s \parallel \mu\rangle.\langle l \parallel l[k] :: s \parallel \mu\rangle.$$

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 $t = \texttt{int}$ and is not `null` otherwise. Its semantics is

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

The bytecode `ifeq` t performs the opposite check:

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

3.2 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 is

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

with $\ell \in \mathbb{L}$ fresh and *oome* new instance of `java.lang.OutOfMemoryError`. This is the first bytecode that throws an exception. Note the use of an underlined output state to represent that situation. 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 is

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

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 example is `putfield` $\kappa.f{:}t$ that moves the *top* of type t of the stack inside the field $\kappa.f:t$

of the object pointed by a value *rec* of type κ below *top*. Its semantics is (ℓ and *npe* are as before)

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

3.3 Exception handling instructions

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

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

Bytecode `catch` starts an exception handler from an exceptional state: it transforms it into a normal state, subsequently used by the implementation of the handler:

$$catch = \lambda\underline{\langle l \| top \| \mu\rangle}.\langle l \| top \| \mu\rangle,$$

where $top \in \mathbb{L}$ has type `java.lang.Throwable`. After `catch`, a handler is selected on the basis of the run-time class of the exception object, by using 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 is

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

3.4 Method call and return instructions

In order to model the dynamic look-up of methods, the code of a method $M = \kappa.m(t_1, \ldots, t_n) : t$ starts with a `receiver_is` K bytecode 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 is

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

At the beginning of M the stack is ε and local variables hold exactly the $n + 1$ actual arguments of the call (including `this`). At the end of M, a `return` t bytecode leaves on the stack the return value of type t only, or a `return` bytecode just returns, if $t =$ `void`:

$$return\ t = \lambda\langle l \| top::s \| \mu\rangle.\langle l \| top \| \mu\rangle,$$
$$return = \lambda\langle l \| s \| \mu\rangle.\langle l \| \varepsilon \| \mu\rangle.$$

Overall, the semantics of the code of M is hence a denotation δ from a state $\langle[v_0, \ldots, v_n] \| \varepsilon \| \mu\rangle$ to a state $\sigma = \langle l' \| top \| \mu'\rangle$, with $top = \varepsilon$ when $t =$ `void`, if M returns normally, or to a state $\sigma = \underline{\langle l' \| top \| \mu'\rangle}$, 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 the operator $extend_M^{i,j} \in \Delta_{n+1,0\to i',r} \to \Delta_{i,j\to i,b+r}$, with $r = 0$ if $t =$ `void` and $r = 1$ otherwise, defined as

$$extend_M^{i,j}(\delta) = \lambda\langle l \| v_n::\cdots::v_0::s \| \mu\rangle \cdot \begin{cases} \underline{\langle l \| \ell \| \mu[\ell := npe]\rangle} & \text{if } v_0 = \texttt{null} \\ \langle l \| top::s \| \mu'\rangle & \text{if } v_0 \in \mathbb{L}, \sigma \in \Xi, \\ \underline{\langle l \| top \| \mu'\rangle} & \text{if } v_0 \in \mathbb{L}, \sigma \in \underline{\Xi}, \end{cases}$$

with ℓ and *npe* as before. The meaning of this definition is that a call to a method will throw a `NullPointerException` is the receiver v_0 is `null` (first case of the definition). In that case, local variables are not modified and the stack only contains a reference to the exception *npe*. Otherwise (second case of the definition), the method might return normally, without throwing any exception, and leave on the stack its return value *top* instead of the parameters $v_n :: \cdots :: v_0$ which get popped from the stack. The local variables of the caller are not modified. The called method is allowed to have side-effects and this is why the final memory is μ' instead of the original memory μ. Finally (third case), the method might throw an exception *top* during its execution. In this last case, the exception will be propagated back to the caller, which continues with an exceptional state having *top* as its only stack element. The local variables of the caller are not modified. Also in this case, we allow the method to produce side-effects and we hence use μ' instead of μ in the final state of *extend*. Note that *extend* is the third place where a dereference might throw an exception.

3.5 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. These denotations represent possible executions starting at b and continuing with b's successor blocks until a block with no successor is reached (hence ending with `return` or `throw`). *Sets* are typical of a *collecting* semantics [9], 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 $\iota \in \mathbb{I}$, providing some executions for each block of P, 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. To that purpose, we compose sequentially the denotations of the instructions inside b and then compose the result with 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

$$[\![\texttt{ins}]\!]^\iota = \{ins\} \quad \text{if } \texttt{ins} \textit{ is not a } \texttt{call}$$
$$\left[\!\left[\texttt{call } M_1, \ldots, M_q\right]\!\right]^\iota = \cup_{1 \le s \le q} extend^{i,j}_{M_s}(\iota(b_{M_s})) \quad \text{otherwise},$$

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 method M_s starts, i the number of local variables and j the height of the stack at the program point where the `call` occurs. Function $[\![\_]\!]^\iota$ is extended to blocks:

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

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

We note here that Definition 6 uses an operator $\cup$ over $\wp(\Delta)$ which, together with the already discussed operators ; and *extend*, forms the three operators that must be abstracted by each abstract interpretation of this concrete semantics, as those in Sects. 4 and 6.

The set $\{M_1, \ldots, M_q\}$ of methods that follow a `call` bytecode must include the set A of all methods that can actually be called at run-time at that program point. Since that information is in general non-computable, $\{M_1, \ldots, M_q\}$ is just an over-approximation of A. This is not a problem in our concrete semantics of Definition 6, since the code of each method is prefixed by a `receiver_is` bytecode that filters the right kind of receiver (Sect. 3.4). When we will define abstractions of this semantics, however, it will typically be the case that the abstraction does not provide any hint about the run-time class of the receiver (this is for instance the case for both the abstract semantics of Sects. 4 and 6). There, a finer over-approximation $\{M_1, \ldots, M_q\}$ of A also means a more precise and faster null-pointer analysis. In our implementation, we have used the very precise class analysis defined in [23] for computing a good, quite small over-approximation $\{M_1, \ldots, M_q\}$ of A.

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 $\iota_0 \in \mathbb{I}$, which is such that $\iota_0(b) = \emptyset$ for all blocks b of P, into $\iota_1 = T_P(\iota_0) \in \mathbb{I}$ and iterates the application of T_P until a fixpoint, i.e. the *denotational semantics* of P (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 [25]. It is the *denotational semantics* of P.

Definition 7 does not provide an effective way for computing the least fixpoint of T_P, since it might require an infinite number of iterations. But safe abstractions of T_P (such as those in Sects. 4 and 6) can be devised in such a way that they always reach the abstract fixpoint in a finite number of iterations.

4 Null-pointer analysis

We define here an abstract interpretation [9] of the concrete semantics of Sect. 3. The latter works over sets of denotations in $\wp(\Delta)$; 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 here.

The abstract domain of this section is a *natural* choice for null-pointer 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 an exceptional state in $\underline{\Xi}$.

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

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

We remind (Definition 2) that the stack of the exceptional states contains one element only, which is a location (and hence is non-`null`).

Example 2 Let $\sigma \in \Xi_{2,3}$ from Example 1. Since $\ell, \ell', \ell'' \in \mathbb{L}$, then we have $nullness(\sigma) = \{l_1\}$.

Denotations are maps from an input state to an output state. To distinguish the variables in those two states, we use Boolean formulas where we put ˇ over the variables holding `null` in the input of a denotation and ˆ over the variables holding `null` in the output of a denotation. If S is a set of identifiers, then we let $\check{S} = \{\check{v} \mid v \in S\}$ 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 *nullness abstract domain* $\mathbb{NULL}_{i_1,j_1\to i_2,j_2}$ is the set of Boolean formulas over $\{\check{e}, \hat{e}\} \cup \{\check{l}_k \mid 0 \le k < i_1\} \cup \{\check{s}_k \mid 0 \le k < j_1\} \cup \{\hat{l}_k \mid 0 \le k < i_2\} \cup \{\hat{s}_k \mid 0 \le k < j_2\}$ (modulo logical equivalence). It is a complete lattice whose greatest lower bound operator is $\wedge$.

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. the variables holding `null`, in a way *compatible* with ϕ.

Definition 10 (Concretisation Map) We define the *concretisation map*

$$\gamma : \mathbb{NULL}_{i_1,j_1\to i_2,j_2} \to \wp(\Delta_{i_1,j_1\to i_2,j_2})$$

as

$$\gamma(\phi) = \left\{ \delta \in \Delta_{i_1,j_1\to i_2,j_2} \,\middle|\, \begin{array}{l}\text{for all } \sigma \in \Sigma_{i_1,j_1} \text{ s.t.} \delta(\sigma) \text{ is defined} \\ \check{nullness}(\sigma) \cup \hat{nullness}(\delta(\sigma)) \models \phi\end{array} \right\}.$$

The following lemma shows a useful property of this concretisation map.

Lemma 1 *The map γ of Definition 10 is co-additive.*

Proof Let $i_1, i_1, j_2, j_2 \in \mathbb{N}, I \subseteq \mathbb{N}$ and $\{\phi_i\}_{i\in I} \subseteq \mathbb{NULL}_{i_1,j_1\to i_2,j_2}$. We prove that $\gamma(\wedge_{i\in I}\phi_i) = \cap_{i\in I}\gamma(\phi_i)$:

$$\begin{aligned}
&\gamma(\wedge_{i\in I}\phi_i) \\
&= \left\{ \delta \in \Delta_{i_1,j_1\to i_2,j_2} \,\middle|\, \begin{array}{l}\text{for all } \sigma \in \Sigma_{i_1,j_1} \text{ s.t. } \delta(\sigma) \text{is defined} \\ \check{nullness}(\sigma) \cup \hat{nullness}(\delta(\sigma)) \models \wedge_{i\in I}\phi_i\end{array} \right\} \\
&= \left\{ \delta \in \Delta_{i_1,j_1\to i_2,j_2} \,\middle|\, \begin{array}{l}\text{for all } \sigma \in \Sigma_{i_1,j_1} \text{ s.t. } \delta(\sigma) \text{ is defined} \\ \check{nullness}(\sigma) \cup \hat{nullness}(\delta(\sigma)) \models \phi_i \ \forall i \in I\end{array} \right\} \\
&= \bigcap_{i\in I} \left\{ \delta \in \Delta_{i_1,j_1\to i_2,j_2} \,\middle|\, \begin{array}{l}\text{for all } \sigma \in \Sigma_{i_1,j_1} \text{ s.t. } \delta(\sigma) \text{ is defined} \\ \check{nullness}(\sigma) \cup \hat{nullness}(\delta(\sigma)) \models \phi_i\end{array} \right\} \\
&= \cap_{i\in I}\gamma(\phi_i).
\end{aligned}$$

□

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 γ as concretisation map.*

Proof The domain $\mathbb{NULL}_{i_1,j_1\to i_2,j_2}$ is a complete lattice w.r.t. logical entailment with $\wedge$ as greatest lower bound operator. The domain $\wp(\Delta_{i_1,j_1\to i_2,j_2})$ is a complete lattice w.r.t. set inclusion with $\cap$ as greatest lower bound operator. The map γ is co-additive (Lemma 1). By a general result of abstract interpretation [9], we have the thesis. □

Example 4 Consider the denotation *store 0* `java.lang.Object` (Sect. 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 it ($\check{s}_1 \leftrightarrow \hat{s}_1$); it is only defined on normal states ($\neg\check{e}$) and always yields a normal state ($\neg\hat{e}$); the output local variable 0 is an alias of the top of the input stack ($\check{s}_2 \leftrightarrow \hat{l}_0$).

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

Fig. 4 Bytecode abstractions for nullness, in a program point with j stack elements

Figure 4 defines correct abstractions for the bytecodes in Sect. 3. To keep the notation simple, we use a formula U (for *unchanged*) which expresses the fact 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 stated only 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 we define $U = \wedge_{v\in L}(\check{v} \leftrightarrow \hat{v}) \wedge \big(\neg\hat{e} \to \wedge_{v\in S}(\check{v} \leftrightarrow \hat{v})\big) \wedge (\hat{e} \to \neg\hat{s}_0)$.

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. There, $U = (\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\neg\hat{e} \to ((\check{s}_0 \leftrightarrow \hat{s}_0) \wedge (\check{s}_1 \leftrightarrow \hat{s}_1))) \wedge (\hat{e} \to \neg\hat{s}_0)$.

For simplicity, in the construction of the formulas we do not distinguish between variables of primitive type and variables of reference type. For instance, for the bytecode `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 in Fig. 4. These formulas model the concrete behaviour of the bytecodes, as specified in Sect. 3 and as it is reflected on the nullness of the variables. First of all, bytecodes are run only if the preceding one does not throw any exception, so that we require that, at their beginning, the Boolean variable identifying exceptional states is false. Namely, we use the conjunct $\neg\check{e}$ in the formula abstracting all bytecodes but `catch`, which is the only one that requires an exception to be thrown just before it is run. For the latter, symmetrically, we use the conjunct $\check{e}$. The abstraction of the bytecodes states whether they never throw any exception (in which case the formula $\neg\hat{e}$ is used) or always do it ($\hat{e}$ is used), as it is the case of `throw`; bytecode `new` leaves this information undefined since the abstract domain $\mathbb{NULL}$ knows nothing about the amount of available memory, so that it is not possible to evince if an exception will be thrown by the bytecode; the dereferencing bytecodes `getfield` and `putfield` throw an exception if and only if their receiver is `null` at the beginning of the execution of the bytecode (namely, for `getfield`, we state $\check{s}_{j-1} \leftrightarrow \hat{e}$, where $\check{s}_{j-1}$ is the nullness of the topmost element of the input stack, i.e. of the receiver); for full Java bytecode we use $\rightarrow$ instead of $\leftrightarrow$ here, since an exception might also be thrown for other reasons than nullness. The abstraction of bytecode `const null` states that it pushes `null` on the stack and hence the formula $\hat{s}_j$ is used. Bytecode `load` k t copies the nullness of the input local variable k into that of the top of the output stack and hence its abstraction contains the formula $\check{l}_k \leftrightarrow \hat{s}_j$. The bytecode `store` k t does the opposite. Bytecode `ifne` wants a non-`null` top of the input stack or otherwise it is undefined (Sect. 3). Hence its abstraction contains the formula $\neg\check{s}_{j-1}$. Conversely, bytecode `ifeq` wants a `null` top of the stack or otherwise it is undefined, so that the formula $\check{s}_{j-1}$ is used. Bytecode `new` states that if it throws no exception then the top of the output stack is non-`null` (hence the formula $\neg\hat{s}_j$) since it is a reference to a new object. Bytecode `getfield` says nothing about the nullness of the field (Sect. 5 improves on this). Bytecode `exception_is` (respectively, `receiver_is`) requires the only stack element (respectively, local variable 0) to be non-`null` or otherwise it is undefined; hence it uses a formula $\check{s}_0$ (respectively, $\check{l}_0$). Bytecode `return` t states that the top of the input stack is `null` if and only if the only output stack element is `null`, as expressed by the formula $\check{s}_{j-1} \leftrightarrow \hat{s}_0$.

Example 6 Consider the bytecode `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 Fig. 4, it follows that the approximation of that bytecode 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 reference to an exception, hence non-`null` ($\neg\hat{s}_0$).

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

The result of Example 7 is not a coincidence. First of all, the formula U actually expresses the fact that the nullness of some variables does not change.

Lemma 2 *Let* `ins` *be a bytecode and let U be the formula constructed for* `ins` *according to Definition 11. Then U is correct w.r.t. ins, i.e. ins* $\in \gamma(U)$.

Proof Let S and L be as in Definition 11 and $\sigma \in \Sigma$ be such that $\sigma' = ins(\sigma)$ is defined.

Let $v \in L$. Since v survives to all executions of `ins`, it is a local variable of both σ and σ' where it has the same value. Hence, either $\{\check{v}, \hat{v}\} \subseteq \check{nullness}(\sigma) \cup \hat{nullness}(\sigma')$ or $\{\check{v}, \hat{v}\} \cap (\check{nullness}(\sigma) \cup \hat{nullness}(\sigma')) = \emptyset$. In both cases we conclude that $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \check{v} \leftrightarrow \hat{v}$, i.e. $ins \in \gamma(\check{v} \leftrightarrow \hat{v})$.

Let now $v \in S$. If $\sigma' \in \Xi$, since v survives to all normal executions of `ins`, it is a stack element of both σ and σ' where it has the same value. Hence $\hat{e} \notin \hat{nullness}(\sigma')$ and either $\{\check{v}, \hat{v}\} \subseteq \check{nullness}(\sigma) \cup \hat{nullness}(\sigma')$ or $\{\check{v}, \hat{v}\} \cap (\check{nullness}(\sigma) \cup \hat{nullness}(\sigma')) = \emptyset$, so that $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \neg\hat{e} \rightarrow (\check{v} \leftrightarrow \hat{v})$. If $\sigma' \in \underline{\Xi}$ we have $\hat{e} \in \hat{nullness}(\sigma')$ and also in this case $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \neg\hat{e} \rightarrow (\check{v} \leftrightarrow \hat{v})$. We conclude that $ins \in \gamma(\neg\hat{e} \rightarrow (\check{v} \leftrightarrow \hat{v}))$.

If $\sigma' \in \Xi$ then $\hat{e} \notin \hat{nullness}(\sigma')$ and $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \hat{e} \rightarrow \neg\hat{s}_0$. If $\sigma' \in \underline{\Xi}$ then σ' has a stack of one element only, which is a location (Definition 3). We conclude that $\hat{s}_0 \notin \hat{nullness}(\sigma')$ so that, also in this case, we have $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \hat{e} \rightarrow \neg\hat{s}_0$. We conclude that $ins \in \gamma(\hat{e} \rightarrow \neg\hat{s}_0)$.

The result follows by Lemma 1. □

By using Lemma 2, we can prove the correctness of the abstract bytecodes in Fig. 4.

Proposition 2 (Correctness of the Abstract Bytecodes) *The approximations in Fig. 4 are correct w.r.t. the denotations of Sect. 3, i.e. for all bytecode* `ins` *we have ins* $\in \gamma(ins^{\mathbb{NULL}})$.

Proof By Lemma 2 we know that $ins \in \gamma(U)$. Let σ be such that $\sigma' = ins(\sigma)$ is defined. If `ins` is not `catch` then

$\sigma \in \Xi$ (Sect. 3). Hence $\check{e} \notin \check{nullness}(\sigma)$ and $ins \in \gamma(\neg\check{e})$. If instead `ins` is `catch`, we must have $\sigma \in \underline{\Xi}$ and hence $\check{e} \in \check{nullness}(\sigma)$. Then $ins \in \gamma(\check{e})$. By Lemma 1, it remains to prove that $ins \in \gamma(\phi)$, where ϕ is the portion of the formulas in Fig. 4 that follows the $U \wedge \neg\check{e}$ prefix ($U \wedge \check{e}$ for `catch`).

`const` v

We have $\phi = \neg\hat{e} \wedge \hat{s}_j$ if $v = \texttt{null}$ and $\phi = \neg\hat{e}$ if $v \neq \texttt{null}$. We have $\sigma' \in \Xi$ so $\hat{e} \notin \hat{nullness}(\sigma')$. Moreover, the top s_j of the stack of σ' holds v. If $v = \texttt{null}$ then $\hat{s}_j \in \hat{nullness}(\sigma')$ while if $v \in \mathbb{Z}$ then $\hat{s}_j \notin \hat{nullness}(\sigma')$. We conclude that $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \phi$ and hence $const\ v \in \gamma(\phi)$.

`load` $k\ t$

We have $\phi = \neg\hat{e} \wedge (\check{l}_k \leftrightarrow \hat{s}_j)$. Since $\sigma' \in \Xi$ we have $\hat{e} \notin \hat{nullness}(\sigma')$. Moreover, the ith local variable of σ is a copy of the top of the stack of σ'. Hence they are both `null`, in which case $\{\check{l}_k, \hat{s}_j\} \subseteq \check{nullness}(\sigma) \cup \hat{nullness}(\sigma')$, or they are both non-`null`, in which case $\{\check{l}_k, \hat{s}_j\} \cap (\check{nullness}(\sigma) \cup \hat{nullness}(\sigma')) = \emptyset$. In both cases we have $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \phi$ and hence $load\ k\ t \in \gamma(\phi)$.

`store` $k\ t$

We have $\phi = \neg\hat{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{l}_k)$. Since $\sigma' \in \Xi$ we have $\hat{e} \notin \hat{nullness}(\sigma')$. Moreover, the top of the stack of σ is a copy of the kth local variable of σ'. Hence they are both `null`, in which case $\{\check{s}_{j-1}, \hat{l}_k\} \subseteq \check{nullness}(\sigma) \cup \hat{nullness}(\sigma')$, or they are both non-`null`, in which case $\{\check{s}_{j-1}, \hat{l}_k\} \cap (\check{nullness}(\sigma) \cup \hat{nullness}(\sigma')) = \emptyset$. In both cases we have $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \phi$ and hence $store\ k\ t \in \gamma(\phi)$.

`ifne` t

We have $\phi = \neg\hat{e} \wedge \neg\check{s}_{j-1}$. Since $\sigma' \in \Xi$ we have $\hat{e} \notin \hat{nullness}(\sigma')$. The top of the stack of σ is non-`null` since otherwise $(ifne\ t)(\sigma)$ would be undefined. Hence $\check{s}_{j-1} \notin \check{nullness}(\sigma)$. We conclude that $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \phi$ and hence $ifne\ t \in \gamma(\phi)$. The proof for `ifeq` is similar.

`new` κ

We have $\phi = \neg\hat{e} \rightarrow \neg\hat{s}_j$. If $\sigma' \in \underline{\Xi}$ we have $\hat{e} \in \hat{nullness}(\sigma')$ and hence $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \phi$. If $\sigma' \in \Xi$ then the top of the stack of σ' is a reference to a new object of class κ, hence non-`null`. Then $\hat{s}_j \notin \hat{nullness}(\sigma')$ and, also in this case, we have $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \phi$. In conclusion, $new\ \kappa \in \gamma(\phi)$.

`getfield` $\kappa.f : t$

We have $\phi = \check{s}_{j-1} \leftrightarrow \hat{e}$. If the top of the stack of σ is `null` then $\sigma' \in \underline{\Xi}$ and then $\{\check{s}_{j-1}, \hat{e}\} \subseteq \check{nullness}(\sigma) \cup \hat{nullness}(\sigma')$. Otherwise $\sigma' \in \Xi$ and hence $\{\check{s}_{j-1}, \hat{e}\} \cap (\check{nullness}(\sigma) \cup \hat{nullness}(\sigma')) = \emptyset$. In both cases $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \phi$. In conclusion, $getfield\ \kappa.f : t \in \gamma(\phi)$.

`putfield` $\kappa.f : t$

We have $\phi = \check{s}_{j-2} \leftrightarrow \hat{e}$. If the element under the top of the stack of σ is `null` then $\sigma' \in \underline{\Xi}$ and hence $\{\check{s}_{j-2}, \hat{e}\} \subseteq \check{nullness}(\sigma) \cup \hat{nullness}(\sigma')$. If instead the element under the top of the stack of σ is non-`null` then $\sigma' \in \Xi$ and hence $\{\check{s}_{j-2}, \hat{e}\} \cap (\check{nullness}(\sigma) \cup \hat{nullness}(\sigma')) = \emptyset$. In both cases we conclude that we have $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \phi$. In conclusion, $putfield\ \kappa.f : t \in \gamma(\phi)$.

`throw` κ

We have $\phi = \hat{e}, \sigma' \in \underline{\Xi}$ and hence $\hat{e} \in \hat{nullness}(\sigma')$. Then we have $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \phi$. In conclusion, $throw\ \kappa \in \gamma(\phi)$.

`catch`

We have $\phi = \neg\hat{e}, \sigma' \in \Xi$ and hence $\hat{e} \notin \hat{nullness}(\sigma')$. Then we have $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \phi$. In conclusion, $catch \in \gamma(\phi)$.

`exception_is` K

We have $\phi = \neg\hat{e} \wedge \neg\hat{s}_0, \sigma' \in \Xi$ and hence $\hat{e} \notin \hat{nullness}(\sigma')$. Moreover, the stack of σ contains only one element and it is non-`null`, since otherwise we would have that $(exception\_is\ K)(\sigma)$ is undefined. Then $\check{s}_0 \notin \hat{nullness}(\sigma)$ and we have $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \phi$. In conclusion, $exception\_is\ K \in \gamma(\phi)$. The proof for `receiver_is` K is similar.

`return` t

We have $\phi = \neg\hat{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{s}_0)$. We have $\sigma' \in \Xi$ and hence $\hat{e} \notin \hat{nullness}(\sigma')$. Moreover, the top of the stack of σ is a copy of the top of the stack of σ', which has height one. Hence either they are both `null`, in which case $\{\check{s}_{j-1}, \hat{s}_0\} \subseteq \check{nullness}(\sigma) \cup \hat{nullness}(\sigma')$, or they are both non-`null`, in which case $\{\check{s}_{j-1}, \hat{s}_0\} \cap (\check{nullness}(\sigma) \cup \hat{nullness}(\sigma')) = \emptyset$. In both cases $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \phi$. In conclusion, $return\ t \in \gamma(\phi)$. The proof for `return` is similar. □

Denotations are composed by ; and their abstractions by $;^{\mathbb{NULL}}$. In $\phi_1 ;^{\mathbb{NULL}} \phi_2$ we match 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 (Abstract Sequential Composition) Let $\phi_1, \phi_2 \in \mathbb{NULL}$. Their *sequential composition* is

$$\phi_1 ;^{\mathbb{NULL}} \phi_2 = \exists_{\overline{V}}(\phi_1[\overline{V}/\hat{V}] \wedge \phi_2[\overline{V}/\check{V}]),$$

where $\overline{V}$ are fresh overlined variables.

Example 8 Consider ϕ_1 from Example 6 and ϕ_2 from Example 7. Then we have $\phi_1;^{\mathbb{NULL}}\phi_2 = \exists_{\{\overline{e},\overline{l}_0,\overline{l}_1,\overline{s}_0,\overline{s}_1,\overline{s}_2\}}(\check{l}_0 \leftrightarrow \overline{l}_0) \wedge (\check{l}_1 \leftrightarrow \overline{l}_1) \wedge \neg\check{e} \wedge (\neg\overline{e} \rightarrow ((\check{s}_0 \leftrightarrow \overline{s}_0) \wedge (\check{s}_1 \leftrightarrow \overline{s}_1) \wedge \neg\overline{s}_2)) \wedge (\overline{l}_1 \leftrightarrow \hat{l}_1) \wedge (\overline{s}_0 \leftrightarrow \hat{s}_0) \wedge (\overline{s}_1 \leftrightarrow \hat{s}_1) \wedge \neg\overline{e} \wedge \neg\hat{e} \wedge (\overline{s}_2 \leftrightarrow \hat{l}_0) = (\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\check{s}_0 \leftrightarrow \hat{s}_0) \wedge (\check{s}_1 \leftrightarrow \hat{s}_1) \wedge \neg\check{e} \wedge \neg\hat{e} \wedge \neg\hat{l}_0$. 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 formula 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`), $\check{e}$, $\hat{e}$ and $\hat{l}_0, \hat{l}_1 \ldots$ (the final values of M's local variables). 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 it exists and only if the renaming is needed, that is, only when $t \neq$ `void` and $b > 0$) into $\hat{s}_b$, but this must be performed only when no exception is thrown by the callee. To that purpose, we first rename $\hat{s}_0$ into a temporary variable w and then state that when no exception is thrown then w entails $\hat{s}_b$. At the end, we remove w. 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 (Abstract *extend*) 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 $(extend_M^{i,j})^{\mathbb{NULL}} : \mathbb{NULL}_{n+1,0\to i',r} \to \mathbb{NULL}_{i,j\to i,b+r}$ with $r = 0$ if $t =$ `void` and $r = 1$ otherwise, as

$$(extend_M^{i,j})^{\mathbb{NULL}}(\phi) = U \wedge \neg\check{e} \wedge (\check{s}_b \rightarrow \hat{e}) \wedge \left(\neg\check{s}_b \rightarrow \exists_w \left((\exists_{\{\hat{l}_0,\hat{l}_1\ldots\}}\phi)[\check{s}_{i+b}/\check{l}_i \mid 0 \leq i \leq n] \underbrace{[w/\hat{s}_0] \wedge ((\neg\hat{e} \wedge w) \leftrightarrow \hat{s}_b)}_{\text{only when } t\neq\texttt{void} \text{ and } b>0}\right)\right).$$

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 Fig. 4, 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` ($\neg\check{s}_2$).

Example 10 Consider the method

```
public Object build() {
    return new Object();
}
```

Its denotation over $\mathbb{NULL}$ is the formula $\phi = (\check{l}_0 \leftrightarrow \hat{l}_0) \wedge \neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{s}_0)$, accordingly to the abstraction of `new` in Fig. 4. Assume to call that method in a context `this.f=this.build()` where `this` is the only local variable of the caller. At the calling point, the stack already contains the value of `this` twice: it is needed as receiver of the call but also as receiver of the subsequent `putfield` that writes the return value of the call into field `f`. Hence we have $i = 1, j = 2, n = 0, b = 1$ and $r = 1$. We have $\exists_{\hat{l}_0}\phi = \neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{s}_0)$. According to Definition 13, the denotation of this call `this.build()` is $(\check{l}_0 \leftrightarrow \hat{l}_0) \wedge (\check{s}_0 \leftrightarrow \hat{s}_0) \wedge \neg\check{e} \wedge (\check{s}_1 \rightarrow \hat{e}) \wedge (\neg\check{s}_1 \rightarrow \exists_w(\neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg w) \wedge ((\neg\hat{e} \wedge w) \leftrightarrow \hat{s}_1)))$ which is equal to $(\check{l}_0 \leftrightarrow \hat{l}_0) \wedge (\check{s}_0 \leftrightarrow \hat{s}_0) \wedge \neg\check{e} \wedge (\check{s}_1 \rightarrow \hat{e}) \wedge (\neg\check{s}_1 \rightarrow (\neg\check{e} \wedge \neg\hat{s}_1))$, that is $(\check{l}_0 \leftrightarrow \hat{l}_0) \wedge (\check{s}_0 \leftrightarrow \hat{s}_0) \wedge \neg\check{e} \wedge (\check{s}_1 \rightarrow \hat{e}) \wedge (\neg\check{s}_1 \rightarrow \neg\hat{s}_1)$. This result means that the nullness of local variable 0 of the caller, i.e. `this`, does not change, nor the nullness of the base s_0 of the stack, that holds the first copy of `this`; the call is executed if there is no exception before it ($\neg\check{e}$); if the receiver of the call holds `null` then an exception is thrown ($\check{s}_1 \rightarrow \hat{e}$); otherwise, the return value is non-`null` ($\neg\check{s}_1 \rightarrow \neg\hat{s}_1$).

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

Proposition 3 (Correctness of the Abstract Operators) *The operators* $;^{\mathbb{NULL}}$, $extend^{\mathbb{NULL}}$ *and* $\cup^{\mathbb{NULL}}$ *are correct.*

Proof Let $\phi_1, \phi_2 \in \mathbb{NULL}$, $d_1 \subseteq \gamma(\phi_1)$ and $d_2 \subseteq \gamma(\phi_2)$. We must prove that $d_1; d_2 \in \gamma(\phi_1;^{\mathbb{NULL}}\phi_2)$. Let $\delta_1 \in d_1$ and $\delta_2 \in d_2$. It is enough to prove that $\delta_1; \delta_2 \in \gamma(\phi_1;^{\mathbb{NULL}}\phi_2)$. Let hence σ be such that $(\delta_1; \delta_2)(\sigma)$ is defined, i.e. both $\sigma' = \delta_1(\sigma)$ and $\sigma'' = \delta_2(\sigma')$ are defined (Definition 4). From $\delta_1 \in \gamma(\phi_1)$ we conclude that $\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \phi_1$. From $\delta_2 \in \gamma(\phi_2)$ we conclude that $\check{nullness}(\sigma') \cup \hat{nullness}(\sigma'') \models \phi_2$. Hence

$$\check{nullness}(\sigma) \cup \{\overline{v} \mid \hat{v} \in \hat{nullness}(\sigma')\} \models \phi_1[\overline{V}/\hat{V}]$$

$$\{\overline{v} \mid \check{v} \in \check{nullness}(\sigma')\} \cup \hat{nullness}(\sigma'') \models \phi_2[\overline{V}/\check{V}]$$

so that $\check{nullness}(\sigma) \cup \{\overline{v} \mid v \in nullness(\sigma')\} \cup \hat{nullness}(\sigma'') \models \phi_1[\overline{V}/\hat{V}] \wedge \phi_2[\overline{V}/\check{V}]$. We conclude that

$$\check{nullness}(\sigma) \cup \hat{nullness}(\sigma'') \models \exists_{\overline{V}}(\phi_1[\overline{V}/\hat{V}] \wedge \phi_2[\overline{V}/\check{V}]) = \phi_1;^{\mathbb{NULL}}\phi_2.$$

Hence $\delta_1; \delta_2 \in \gamma(\phi_1;^{\mathbb{NULL}}\phi_2)$.

Let $\phi \in \mathbb{NULL}_{n+1,0\to i',r}$ as in Definition 13. Let $d \subseteq \gamma(\phi)$. We must prove that for all $i, j \in \mathbb{N}$ with $j = b+n+1$ and $b \geq 0$ we have $extend_M^{i,j}(d) \subseteq \gamma((extend_M^{i,j})^{\mathbb{NULL}}(\phi))$. Let hence $\delta \in d$. It is enough to prove that $extend_M^{i,j}(\delta) \in \gamma((extend_M^{i,j})^{\mathbb{NULL}}(\phi))$. We show the case when $b > 0$ and M does not return `void`. The other cases are similar. Let σ be such that $\sigma' = extend_M^{i,j}(\delta)(\sigma)$ is defined. By the definition of $extend_M^{i,j}$ (Sect. 3) we know that σ and σ' have the same set of local variables with unchanged values; moreover, when $\sigma' \in \Xi$ the b lowest stack elements are both in σ and σ' with unchanged value. From Lemma 2 we conclude that $extend_M^{i,j}(\delta) \in \gamma(U)$. Moreover, $extend_M^{i,j}(\delta)$ is defined only on normal states of the form $\sigma = \langle l \,\|\, v_n :: \cdots :: v_0 :: s \,\|\, \mu\rangle \in \Xi$ (see Sect. 3.4). Then $extend_M^{i,j}(\delta) \in \gamma(\neg\check{e})$. We also know that if $v_0 = \texttt{null}$ then $\sigma' \in \underline{\Xi}$ so that $extend_M^{i,j}(\delta) \in \gamma(\check{s}_b \to \hat{e})$. If instead $v_0 \in \mathbb{L}$, then:

- $\check{s}_b \notin \check{nullness}(\sigma)$,
- $\sigma' = \langle l \,\|\, top :: s \,\|\, \mu'\rangle$ if $\delta(\langle [v_0, \ldots, v_n] \,\|\, \varepsilon \,\|\, \mu\rangle) = \langle l' \,\|\, top \,\|\, \mu'\rangle$,
- $\sigma' = \underline{\langle l \,\|\, top \,\|\, \mu'\rangle}$ if $\delta(\langle [v_0, \ldots, v_n] \,\|\, \varepsilon \,\|\, \mu\rangle) = \underline{\langle l' \,\|\, top \,\|\, \mu'\rangle}$.

Let $\sigma_1 = \langle [v_0, \ldots, v_n] \,\|\, \varepsilon \,\|\, \mu\rangle$ and $\sigma_2 = \delta(\langle [v_0, \ldots, v_n] \,\|\, \varepsilon \,\|\, \mu\rangle)$. Then we have that

$$\check{nullness}(\sigma_1) \cup \hat{nullness}(\sigma_2) \models \phi.$$

Let $\sigma_2' = \langle l \,\|\, top \,\|\, \mu'\rangle$ if $\sigma_2 = \langle l' \,\|\, top \,\|\, \mu'\rangle$ and $\sigma_2' = \underline{\langle l \,\|\, top \,\|\, \mu'\rangle}$ if $\sigma_2 = \underline{\langle l' \,\|\, top \,\|\, \mu'\rangle}$. We have

$$\check{nullness}(\sigma_1) \cup \hat{nullness}(\sigma_2') \models \exists_{\{\hat{l}_0,\hat{l}_1,\ldots\}}\phi.$$

Let $\sigma_1' = \langle l \,\|\, v_n :: \cdots :: v_0 \,\|\, \mu\rangle$. We have

$$\check{nullness}(\sigma_1') \cup \hat{nullness}(\sigma_2') \models (\exists_{\{\hat{l}_0,\hat{l}_1,\ldots\}}\phi)[\check{s}_i/\check{l}_i \mid 0 \leq i \leq n]$$

and hence

$$\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \neg\check{s}_b \to (\exists_{\{\hat{l}_0,\hat{l}_1,\ldots\}}\phi)[\check{s}_{i+b}/\check{l}_i \mid 0 \leq i \leq n][\hat{s}_b/\hat{s}_0]$$

when $\sigma' \in \Xi$ and

$$\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \neg\check{s}_b \to (\exists_{\{\hat{l}_0,\hat{l}_1,\ldots\}}\phi)[\check{s}_{i+b}/\check{l}_i \mid 0 \leq i \leq n]$$

when $\sigma' \in \underline{\Xi}$. Since, it the latter case, we have $\hat{e} \in \hat{nullness}(\sigma')$, we conclude that in both cases we have

$$\check{nullness}(\sigma) \cup \hat{nullness}(\sigma') \models \neg\check{s}_b \to \exists_w((\exists_{\{\hat{l}_0,\hat{l}_1,\ldots\}}\phi)[\check{s}_{i+b}/\check{l}_i \mid 0 \leq i \leq n][w/\hat{s}_0] \wedge((\neg\hat{e} \wedge w) \leftrightarrow \hat{s}_b)).$$

So $extend_M^{i,j}(\delta) \in \gamma(\neg\check{s}_b \to \exists_w((\exists_{\{\hat{l}_0,\hat{l}_1,\ldots\}}\phi)[\check{s}_{i+b}/\check{l}_i \mid 0 \leq i \leq n][w/\hat{s}_0] \wedge((\neg\hat{e}\wedge w) \leftrightarrow \hat{s}_b)))$. By Lemma 1 we conclude that the denotation $extend_M^{i,j}(\delta)$ must belong to

$$\gamma\begin{pmatrix} U \wedge \neg\check{e} \wedge (\check{s}_b \to \hat{e}) \wedge \\ (\neg\check{s}_b \to \exists_w((\exists_{\{\hat{l}_0,\hat{l}_1,\ldots\}}\phi)[\check{s}_{i+b}/\check{l}_i \mid 0 \leq i \leq n][w/\hat{s}_0] \\ \wedge((\neg\hat{e} \wedge w) \leftrightarrow \hat{s}_b)) \end{pmatrix} = \gamma((extend_M^{i,j})^{\mathbb{NULL}}(\phi)).$$

Let $\phi_1, \phi_2 \in \mathbb{NULL}$, $d_1 \subseteq \gamma(\phi_1)$ and $d_2 \subseteq \gamma(\phi_2)$. We must prove that $d_1 \cup d_2 \subseteq \gamma(\phi_1 \cup^{\mathbb{NULL}} \phi_2) = \phi_1 \vee \phi_2$. Let hence $\delta \in d_1 \cup d_2$. It is enough to prove that $\delta \in \gamma(\phi_1 \vee \phi_2)$. If $\delta \in d_1$ then $\delta \in \gamma(\phi_1) \subseteq \gamma(\phi_1 \vee \phi_2)$. If $\delta \in d_2$ then $\delta \in \gamma(\phi_2) \subseteq \gamma(\phi_1 \vee \phi_2)$. □

Since we have correct abstractions of all three semantical operators used in the concrete semantics of Sect. 3, we can define abstract counterparts of the interpretations (Definition 5), which are now maps from blocks to Boolean formulas, and of the denotation of an instruction (Definition 6). The abstract semantics is then computed as the fixpoint of the abstract counterpart of the T_P operator of Definition 7. Differently from the concrete semantics, which is not finitely computable, the number of Boolean formulas over a given set of variables is finite (modulo equivalence) and hence the abstract fixpoint is reached in a finite number of iterations. The result, as standard in denotational semantics, is an interpretation (the abstract semantics) that describes how the nullness of the variables evolves if the program is started from each given block of code and is executed until the next `return` bytecode of the method where the block occurs. However, this is not the information one usually needs in static analysis. Typically, one wants information at internal program points, before the methods end with a `return`. This is the case for nullness analysis, for instance, since nullness information is important before every bytecode that dereferences its receiver, in order to check if an exception might ever be thrown there. This problem is solved with a program transformation, applied before the analysis is performed, called *magic-sets transformation*, which yields a new program whose denotational semantics provides information at internal points of the original, untransformed program. This technique is traditional in logic programming and has been recently defined for Java bytecode in [25]. In conclusion, for each `getfield`, `putfield` and `call` bytecode *op* in P that dereferences a stack element s_k (for full Java bytecode, also before each `arraylength`, `throw`, `arrayload`, `arraystore`, `monitorenter` and `monitorexit` [21]), the magic-sets transformation gives us a formula ψ_{op} which holds just before *op*. If ψ_{op} entails $\neg\hat{s}_k$ then *op* is safe, since a non-`null` value is dereferenced by the bytecode.

Figure 5 shows how the null-pointer analysis of this section is performed. The program is analysed by computing

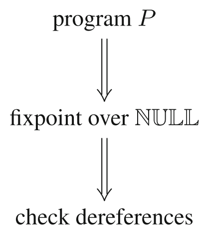

Fig. 5 The null-pointer analysis described in Sect. 4

the abstract semantics through the same fixpoint computation described in Sect. 3. This semantics is then used to check for safe dereferences, as we said above.

5 Oracle semantics for always non-`null` fields

The analysis of Sect. 4 never assumes that fields hold a non-`null` value. This is apparent from the approximation for `getfield` in Fig. 4, which does not constrain variable $\hat{s}_{j-1}$. This means that nothing is known about the nullness of the top of the output stack, i.e. of the value read from the field. This hypothesis is conservative but too strong: the resulting analysis can never be precise enough to verify real software. We show here how we overcome this limitation by identifying non-`null` fields, that is, fields that *always* hold a non-`null` value after the object they belong to has been constructed.

We start with the notion of *candidate* field, which has reference type and is always initialised by the constructors before it is read. We observe that 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 [21]. We do not consider any field as candidate when the constructor builds an instance of `java.lang.Throwable`: this is because, as Laurent Hubert has correctly observed in a personal communication, in that case the constructor might throw the partially constructed object itself, although this seems in contrast with the statement in [21] that if a constructor throws an exception then the partially constructed object cannot be used.

Definition 14 (Candidate Field) A field $\kappa.f:t$ is *candidate* if

1. $t \in \mathbb{K}$;
2. $\kappa \not\leq$ `java.lang.Throwable`;
3. 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;
4. for every execution path x in every constructor of κ, if x contains a `getfield` $\kappa.f:t$ then it also contains a previous `putfield` $\kappa.f:t$ over the created object.

For instance, fields `f` and `g` in Fig. 1 are candidate; fields `head` and `tail` in Fig. 2 are candidate; but only `k` is candidate in

```
public class C {
  private Object h, k;

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

since field `h` is read before it is initialised.

In order to compute an underapproximation of the set of candidate fields, we use a preliminary definite aliasing analysis to check if the `putfield` bytecodes work over the created object; namely, we check if their receiver is a definite alias of local variable 0. We consider no field as candidate when a constructor contains a (legal but rather unusual) `store 0` t bytecode, which might swap the receiver of the constructor (held in local 0), hence making the use of the aliasing analysis unreliable. After the aliasing analysis has been computed, the fact of being candidate for a field is just a syntactical property of the code, that we check through the graph algorithm in Fig. 6, which takes into account helper functions for better precision (as `helper()` in Fig. 1).

Let us discuss the algorithm in Fig. 6. It defines a function `candidates(c)` that yields the set of candidate fields w.r.t. a given constructor `c`. If class κ has more constructors `c1`...`cn`, then one takes the intersection of `candidates(c1)`,..., `candidates(cn)` as a set of candidate fields from class κ. The algorithm computes, for every block b, sets $b.w$ and $b.r$, initially empty. The former is an underapproximation of the set of fields of local variable 0 that are definitely written in every execution path starting at b and leading to a `return`; the latter is an overapproximation of the set of fields that may be read in some execution path starting at x before being written as a field of local variable 0. The algorithm uses a working set of blocks, those reachable from b by following helper functions also. The working set is analysed until it is empty. Every time that a block b is extracted from the working set, we compute the union of the fields that might be read by its successors and the intersection of the fields that are definitely written, inside local variable 0, by its successors that can lead to a `return`. Those sets are added to $b.r$ and to $b.w$, respectively. Then the instructions inside b are considered backwards. A `getfield` reads a field while a `putfield` writes a field of local variable 0 when the aliasing information proves the definite aliasing between the receiver and local variable 0. A `call` bytecode over a definite alias of local variable 0 leads to a helper function and in that case we compute the fields r that are read by some called method and the fields w that are written by all called methods. Set r is removed from $b.w$ and added to

Fig. 6 An algorithm to compute the candidate fields

```
Set candidates(Constructor c) {
  // all blocks reachable from the first block of c are added
  // to a workset. Helper functions are also included
  Set ws = reachable(c.firstBlock(),new Set());
  for (Block b: ws) { b.w = new Set(); b.r = new Set(); }
  // we process the workset
  while (!ws.isEmpty()) {
    remove some b from ws;
    let b1..bn be the successors of b;
    let r1..rm be those that lead to a return in the graph where b occurs
    // a field is read if it is read by some path
    b.r = b1.r union ... union bn.r;
    // a field is written if it is written by all paths leading to a return
    if (m > 0) b.w = r1.w intersect ... intersect rm.w;
    let ins1..insm be the bytecodes inside b
    // we process the bytecodes backwards
    for (int i = m; i > 0; i--) {
      if (insi is getfield f) {
        b.r.add(f); b.w.remove(f); // this field is read
      } else if (insi is a putfield f with receiver alias of local 0) {
        b.r.remove(f); b.w.add(f); // this field is written
      } else if (insi is a call M with receiver alias of local 0) {
        let M1...Mn be the methods that might be called here;
        // we continue inside the helper function(s)
        Set r = union Mi.firstBlock().r over i
        Set w = intersection Mi.firstBlock().w over i
        b.w.removeAll(r); b.w.addAll(w);
        b.r.removeAll(w); b.r.addAll(r);
      } else if (insi is a call M) {
        // this is not a helper function
        let M1...Mn be the methods that might be called here;
        Set r = reference fields read by some Mi or by any method that it calls;
        b.w.removeAll(r); b.r.addAll(r);
      } else if (insi is store 0 t) b.w = new Set();

      if (b.w or b.r changed during this iteration)
        add all predecessors of b to ws;
    }
  }
  return c.firstBlock().w;
}

// yields the set of blocks reachable from b. Helper functions are included
Set reachable(Block b, Set result) {
  if (!result.contains(b)) {
    result.add(b);
    for (Block f: b.successors()) reachable(f,result);
    if (b contains a call M bytecode whose receiver is an alias of local 0)
      reachable(M.firstBlock(),result);
  }
  return result;
}
```

$b.r$ and set w is added to $b.w$ and removed from $b.r$. If the `call` leads to a function that might not be a helper function, then we conservatively add to $b.r$ all fields read by the function(s) that it calls. A `store` 0 t bytecode resets the set of fields that are definitely written inside an alias of local 0, since after this instruction there is no guarantee that local 0 actually contains a reference to the object being initialised.

Every time that the approximation of a block b changes, its predecessors are added to the working set. If b is the beginning of a helper function, by *predecessors* we mean the blocks where a `call` to the helper function occurs.

Being candidate does not guarantee that the field never contains `null`: to that purpose, we have to check the values that are written, at run-time, inside the field.

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

It follows that, when P reads a non-`null` field, it does not find `null`. Being non-`null` is a semantical property, since we need to know which values flow inside the field. Let us hence define an *oracle*, telling us if a field is non-`null`. Later, we will show how such an oracle can be actually computed.

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`.

Example 11 Sets {`f`,`g`}, {`g`}, {`f`} and ∅ and correct oracles for the program in Fig. 1. Sets {`head`} and ∅ are correct oracles for the program in Fig. 2. However, sets {`tail`} and {`head`, `tail`} are made of candidates fields of the program in Fig. 2, but they are not correct since `null` is written inside `tail` by that program.

By using an oracle $O \in \mathbb{O}$, we can redefine the approximation of `getfield` so that it assumes that the fields in O

never hold `null`. Namely, we redefine

$$(\mathit{getfield}\ \kappa.f:t)_O^{\mathbb{NULL}} = \begin{cases} U \wedge \neg\check{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{e}) \\ \quad \wedge(\neg\hat{e} \rightarrow \neg\hat{s}_{j-1}) & \text{if } \kappa.f:t \in O \\ U \wedge \neg\check{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{e}) & \text{if } \kappa.f:t \notin O, \end{cases} \quad (1)$$

i.e. if $\kappa.f:t \in O$ and if no exception is thrown by the `getfield`, then the top of the output stack is non-`null` ($\neg\hat{e} \rightarrow \neg\hat{s}_{j-1}$); otherwise, nothing is said about the nullness of the top of the output stack. This redefinition induces a null-pointer analysis, parameterised w.r.t. O, which is correct if O is correct, but which may be incorrect otherwise; moreover, the larger the correct set O, the more precise is the induced analysis.

Proposition 4 (Correctness of the Oracle Semantics) *If $O \in \mathbb{O}$ is correct, then the* `null`*-pointer analysis parameterised w.r.t. O is correct.*

Proof We have already proved that the standard semantics, without oracle, is correct (Propositions 2 and 3). Hence it is enough to prove that the definition of $(\mathit{getfield}\ \kappa.f:t)_O^{\mathbb{NULL}}$ is correct:

$$(\mathit{getfield}\ \kappa.f:t) \in \gamma((\mathit{getfield}\ \kappa.f:t)_O^{\mathbb{NULL}})$$
$$\Leftrightarrow (\mathit{getfield}\ \kappa.f:t) \in \gamma\begin{pmatrix} U \wedge \neg\check{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{e}) \\ \wedge(\neg\hat{e} \rightarrow \neg\hat{s}_{j-1}) \end{pmatrix}$$
$$\Leftrightarrow (\mathit{getfield}\ \kappa.f:t) \in \gamma\begin{pmatrix} (\mathit{getfield}\ \kappa.f:t)^{\mathbb{NULL}} \\ \wedge(\neg\hat{e} \rightarrow \neg\hat{s}_{j-1}) \end{pmatrix}$$
$$(\text{Lemma 1}) \Leftrightarrow (\mathit{getfield}\ \kappa.f:t) \in \gamma((\mathit{getfield}\kappa.f:t)^{\mathbb{NULL}}) \cap\gamma(\neg\hat{e} \rightarrow \neg\hat{s}_{j-1})$$
$$(\text{Proposition 2}) \Leftrightarrow (\mathit{getfield}\ \kappa.f:t) \in \gamma(\neg\hat{e} \rightarrow \neg\hat{s}_{j-1}).$$

Let hence σ be such that $\sigma' = (\mathit{getfield}\ \kappa.f:t)(\sigma)$ is defined. We have (Sect. 3) $\sigma = \langle l \,\|\, rec :: s \,\|\, \mu\rangle$. If $rec = \texttt{null}$, we have $\sigma' \in \underline{\Xi}$, so that $\hat{e} \in \widehat{nullness}(\sigma')$. Hence $\check{nullness}(\sigma) \cup \widehat{nullness}(\sigma') \models (\neg\hat{e} \rightarrow \neg\hat{s}_{j-1})$. If instead $rec \neq \texttt{null}$, we have $\sigma' = \langle l \,\|\, \mu(rec).f :: s \,\|\, \mu\rangle$. Since $\kappa.f:t \in O$ and O is correct, the field $\kappa.f:t$ is non-`null`, i.e. it is candidate and `null` is never written inside it. Moreover, there must be a `putfield` $\kappa.f:t$ bytecode that is executed by the constructor that has initialised the object $\mu(rec)$, before the execution of this `getfield` $\kappa.f:t$. We conclude that a non-`null` value is already written inside $\mu(rec).f$, i.e. the top of the stack of σ' is non-`null`. Thus $\hat{s}_{j-1} \notin \widehat{nullness}(\sigma')$ and, also in this case, $\check{nullness}(\sigma) \cup \widehat{nullness}(\sigma') \models (\neg\hat{e} \rightarrow \neg\hat{s}_{j-1})$. □

The problem is now that of finding a correct $O \in \mathbb{O}$. The obvious choice $O = \emptyset$ is correct but leads us back to the analysis of Sect. 4. Proposition 5 will help us to make a better choice by *refining* a given oracle into a *better* one, which does not contain some fields that are not proved non-`null`.

Its correctness is based on the following lemma, which states the correctness of the parameterised semantics w.r.t. a non-standard semantics of the Java bytecode.

Lemma 3 *Let $O \in \mathbb{O}$ and define a non-standard semantics for* `getfield`*, which never reads* `null` *from a field in O:*

$$(\mathit{getfield}\ \kappa.f:t)_O = \lambda\langle l \,\|\, rec::s \,\|\, \mu\rangle \cdot \begin{cases} \langle l \,\|\, \mu(rec).f :: s \,\|\, \mu\rangle \\ \textit{if } rec \neq \texttt{null}, \\ (\mu(rec).f \neq \texttt{null} \textit{ or } \kappa.f:t \notin O) \\ \\ \langle l \,\|\, \ell :: s \,\|\, \mu[\ell := o]\rangle \\ \textit{if } rec \neq \texttt{null}, \\ \mu(rec).f = \texttt{null} \textit{ and } \kappa.f:t \in O \\ \\ \underline{\langle l \,\|\, \ell \,\|\, \mu[\ell \mapsto npe]\rangle} \\ \textit{otherwise} \end{cases}$$

where $\ell \in \mathbb{L}$ is fresh and o is an object of class t with fields initialised to default values. Then $(\mathit{getfield}\ \kappa.f:t)_O^{\mathbb{NULL}}$ is correct w.r.t. $(\mathit{getfield}\ \kappa.f:t)_O$, i.e.

$$(\mathit{getfield}\ \kappa.f:t)_O \in \gamma((\mathit{getfield}\ \kappa.f:t)_O^{\mathbb{NULL}}).$$

Proof If $\kappa.f:t \notin O$, we have $(\mathit{getfield}\ \kappa.f:t)_O = \mathit{getfield}\ \kappa.f:t$ and $(\mathit{getfield}\ \kappa.f:t)_O^{\mathbb{NULL}} = (\mathit{getfield}\ \kappa.f:t)^{\mathbb{NULL}}$ and the result follows from Proposition 2. Let instead $\kappa.f:t \in O$. By Lemma 2, we have $(\mathit{getfield}\ \kappa.f:t)_O \in \gamma(U \wedge \neg\check{e})$. Moreover, $(\mathit{getfield}\ \kappa.f:t)_O \in \gamma(\check{s}_{j-1} \leftrightarrow \hat{e})$, as can be proved identically as in Proposition 2. Take σ such that $\sigma' = (\mathit{getfield}\ \kappa.f:t)_O(\sigma)$ is defined. We have $\sigma = \langle l \,\|\, rec :: s \,\|\, \mu\rangle$. If $rec = \texttt{null}$ then $\sigma' \in \underline{\Xi}$, so that $\hat{e} \in \widehat{nullness}(\sigma')$ and $\check{nullness}(\sigma) \cup \widehat{nullness}(\sigma') \models \neg\hat{e} \rightarrow \neg\hat{s}_{j-1}$. If instead $rec \neq \texttt{null}$ then either $\sigma' = \langle l \,\|\, \mu(rec).f :: s \,\|\, \mu\rangle$ with $\mu(rec).f \neq \texttt{null}$ or $\sigma' = \langle l \,\|\, \ell :: s \,\|\, \mu[\ell := o]\rangle$ and in both cases we have $\hat{s}_{j-1} \notin \widehat{nullness}(\sigma')$. Then, also in this case, we have $\check{nullness}(\sigma) \cup \widehat{nullness}(\sigma') \models \neg\hat{e} \rightarrow \neg\hat{s}_{j-1}$. We conclude that $(\mathit{getfield}\ \kappa.f:t)_O \in \gamma(\neg\hat{e} \rightarrow \neg\hat{s}_{j-1})$. By Lemma 1 we conclude that $(\mathit{getfield}\ \kappa.f:t)_O \in \gamma(U \wedge \neg\check{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{e}) \wedge (\neg\hat{e} \rightarrow \neg\hat{s}_{j-1})) = \gamma((\mathit{getfield}\ \kappa.f:t)_O^{\mathbb{NULL}})$. □

We can now prove a result that lets us refine, iteratively, any (possibly incorrect) oracle into a correct oracle.

Proposition 5 (Oracle Refinement) *Define $F_P : \mathbb{O} \rightarrow \mathbb{O}$ as*

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

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

Proof Let $O \in \mathbb{O}$ be a fixpoint of F_P and assume, by contradiction, that O is not correct. Hence $I = \{\kappa.f:t \in O \mid \kappa.f:t$ is not non-`null`$\}$ is not empty. Since the fields in I are not non-`null`, by definition there is a finite execution x of P that leads to a `putfield` $\kappa.f:t$ that writes `null` into a field $\kappa.f:t \in I$. We can assume, without loss of generality, that `null` has never been written before, during x, inside a field in I (i.e. we stop x when, for the first time, `null` is written inside some field $\kappa.f:t \in I$). Consider a `getfield` $\kappa'.f':t'$ bytecode executed during x. If $\kappa'.f':t' \in O \setminus I$ then $\kappa'.f':t'$ is non-`null` and the `getfield` bytecode pushes a non-`null` value on top of the output stack. If otherwise $\kappa'.f':t' \in I$ then $\kappa'.f':t'$ is candidate and, by Definition 14, a `putfield` $\kappa'.f':t'$ must have written a value v inside field $\kappa'.f':t'$ of the same object accessed by the `getfield`. By the hypothesis about x, we conclude that $v \neq$ `null`, i.e. that also in this case the `getfield` bytecode pushes a non-`null` value on top of the output stack. In conclusion, the `getfield` bytecodes in x accessing a field in O never find `null` inside the field that they read, i.e. they behave accordingly to the non-standard semantics of `getfield` defined in Lemma 3. This means that the execution x is also a non-standard execution that uses the semantics of `getfield` defined in Lemma 3. Since our null-pointer semantics, parameterised w.r.t. O, is correct w.r.t. the concrete semantics that uses that non-standard semantics for `getfield` (Lemma 3) we conclude that it cannot prove that `null` is never written inside $\kappa.f:t$, since x writes `null` into $\kappa.f:t$. Then $\kappa.f:t \notin F_P(O)$ and $O \neq F_P(O)$, a contradiction. We conclude that O must be correct. □

Let us discuss why Proposition 5 can be used to compute a correct oracle by refining any (possibly incorrect) oracle O. By computing $F_P(O)$, one applies our `null`-pointer analysis parameterised w.r.t. O and checks in which fields of O the program writes 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 the worst case, it converges to $\emptyset$, which is always correct). In words, one starts with the optimistic hypothesis O_0 that all candidate fields are non-`null` and iteratively removes 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 of the analysis is correct (Proposition 4).

Example 12 Take $O_0 = \{$`f`, `g`$\}$ in the program in Fig. 1; we have $F_P(O_0) = O_0$ so O_0 is correct. In the program in Fig. 2, take $O_0 = \{$`head`, `tail`$\}$; we have $O_1 = F_P(O_0) = \{$`head`$\}$ and $F_P(O_1) = O_1$, so that O_1 is correct. For the class `C` above, take $O_0 = \{$`k`$\}$ and $F_P(O_0) = \emptyset$ is correct.

This iterative null-pointer analysis might seem prohibitively expensive. An upper bound to the number of needed iterations is indeed the possibly large number of candidate fields of P. However, in practice, no more than four iterations are used even for the largest programs of Sect. 7. Moreover, the first iteration might be expensive but an extensive use of caching makes the subsequent iterations quicker than the first one. Furthermore, preliminary computations, such as the construction of the magic-sets and of the strongly connected components of blocks, need to be performed only before the first iteration and are recycled for the subsequent iterations.

Static fields are accommodated in our framework. A candidate static field is defined as in Definition 14 by using `putstatic` and `getstatic` instead of `putfield` and `getfield` and by considering that there is only one constructor for the static class information, called ⟨`clinit`⟩. Aliasing is not used since there is no receiver object during the execution of that static constructor.

Figure 7 shows how the null-pointer analysis of this section is performed. Initially, a definite aliasing analysis is performed, whose results are useful to compute a set of candidate fields. Then the program is analysed by an iterated computation of the abstract semantics over $\mathbb{NULL}$ through the same fixpoint computation described in Sect. 3. The result of the last iteration is correct and is finally used to check for safe dereferences.

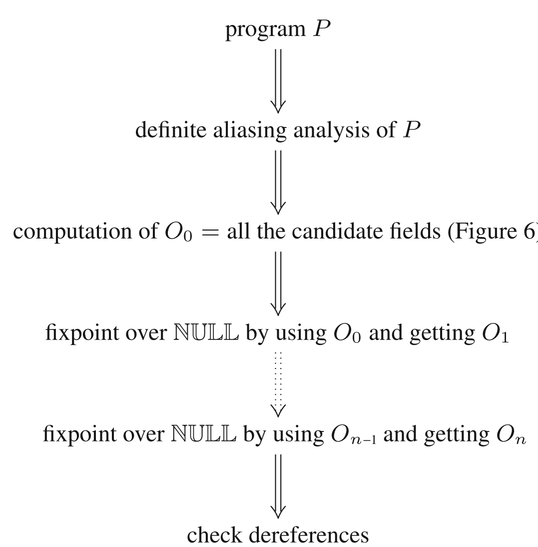

Fig. 7 The null-pointer analysis described in Sect. 5, by using an oracle of non-`null` fields

6 Dealing with locally non-**null** fields

The oracle computed for the program in Fig. 2, by following the theory of Sect. 5, is {head} (Example 12). This is because both fields head and tail are initialised in the only constructor of class List and they are hence both candidate fields (Definition 14). However, the analysis proves that the constructor always writes a non-null value inside head, but fails to prove the same for tail. This is completely correct, since there are cases when null is actually written inside tail by that constructor. For instance, this happens for the construction of the tails of l1 and l2 inside method main(). Since tail does not belong to the fixpoint oracle, the analysis assumes that it might contain a null value, which in turn leads to false alarms whenever the value of tail is dereferenced, as in the recursive call inside iter(). As we have said in Sect. 1, we can get rid of those spurious alarms by observing that the non-nullness of field tail has been explicitly checked before the recursive call so that it cannot hold null there.

It is important to note that *local reasonings* about the non-nullness of some fields at specific program points, based on explicit non-nullness checks, are only correct for a mono-threaded program. In a multi-threaded program, it is possible instead that the field gets reset to null by another thread, between the check and the dereference. It is possible to restrict our reasonings to those contexts when such situations do not occur, by using a preliminary analysis, but this is outside the scope of this paper, where we assume to deal with mono-threaded programs.

Even in a mono-threaded environment, local reasonings on the non-nullness of some field can be tricky. Consider for instance the fragment of code

```
if (a.tail != null) a.tail.iter();
```

This is safe, since the non-nullness of a.tail has been explicitly tested before the dereference in the call to iter(). However, the following fragment might not be safe

```
if (a.tail != null) { a = b; a.tail.iter(); }
```

since the value of a changed between the test and the dereference. Hence our analysis must be able to track those program variables that are modified in a piece of code (as we will see in Definition 19). Similarly, the fragment

```
if (a.tail != null) { b.tail = null;
a.tail.iter(); }
```

might not be safe since a and b may be aliases so that an update to field tail of b may also affect field tail of a. Hence our analysis must be able to track those fields that are modified in a piece of code (Definition 19).

In this section we define a simple static analysis which lets us prove that the first fragment is safe, while it considers the last two fragments as potentially unsafe, regardless of any possible aliasing information. The goal is to devise a simple, quick yet precise analysis which captures the most frequent cases, instead of a complex analysis that is precise also in subtle but rather unfrequent situations.

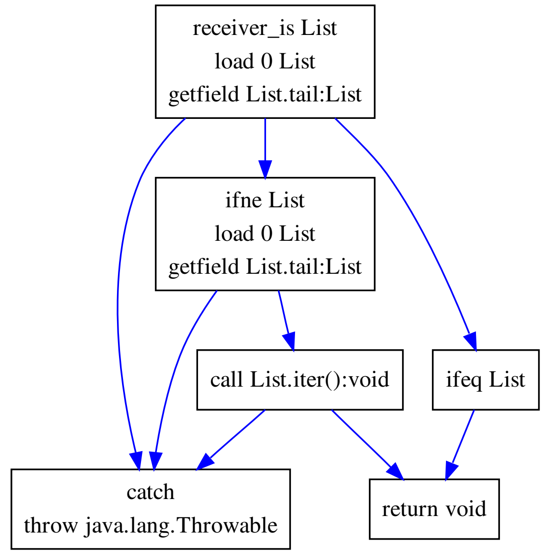

Fig. 8 The blocks of method iter() (Fig. 2)

Let us consider, in Fig. 8, the bytecode for the method iter() in Fig. 2. At the ifne List bytecode, the top of the stack contains, definitely, an alias of field tail of local variable 0, that we write as 0.tail. Hence, if its semantics is defined, we conclude that 0.tail is non-null (Sect. 3). At the second, lower getfield List.tail : List, the top of the stack is a definite alias of local variable 0. Hence, if that bytecode does not throw any exception, it loads 0.tail on top of the stack. Since, between the ifne and the getfield, nor local 0 nor any field named tail is modified, we conclude that the second getfield loads a non-null value on top of the stack. It follows that the call List.iter() : void bytecode cannot dereference null. In order to automate such reasonings, we need to know if some stack element is a definite alias of a local variable or of a field of some local variable. We also need to compute a set of fields of local variables which definitely hold a non-null value at selected program points. Namely, we need such sets before the getfield bytecodes, so that we can test if they load a non-null value on top of the stack. For instance, in Fig. 8, we want to compute a set containing 0.tail just before the second, lower getfield List.tail : List.

As it can be understood from the previous paragraph, an ingredient of this analysis is a definite aliasing analysis between stack elements and local variables or fields of local variables. There are plenty of aliasing analyses for imperative programs. Most of them, however, provide information about *possible* aliasing, while we need information about *definite* aliasing here. Others do not consider aliasing between local variables and fields. We have chosen to use the definite aliasing analysis described in [2], with some little corrections, which provides definite aliasing information also between local variables and fields. We stress, however, the fact that others, more precise analyses could be used as well, which would lead to more precise `null`-pointer analyses.

We formalise now the static analysis that computes a set of non-`null` fields. It is an abstract interpretation of the semantics in Sect. 3. It will be built exactly as in Sect. 4, by defining abstract bytecodes and abstract versions of the operators ; , $\cup$ and *extend*. First, we need to modify the semantics of Sect. 3, since we assume to analyse a program which has been already decorated with definite aliasing information. Hence we consider each bytecode `ins` as already decorated with some definite aliasing information *alias*: if that information does not hold, then the semantics of the bytecode is undefined. Namely, we define

$$ins_{alias}(\sigma) = \begin{cases} ins(\sigma) & \text{if } \sigma \text{ satisfies } alias \\ undefined & \text{otherwise.} \end{cases} \quad (2)$$

This redefinition does not affect the execution of the programs since, from the correctness of the definite aliasing analysis, it follows that *alias* always holds when $\texttt{ins}_{alias}$ is executed. Moreover, the abstractions of the bytecodes in Sect. 4 are still correct, since from (2) it follows that when $ins_{alias}(\sigma)$ is defined then also $ins(\sigma)$ is defined, so every formula ϕ which correctly approximates the nullness behaviour of *ins* also approximates the nullness behaviour of ins_{alias} (Definition 10). Equation (2) is however important when it comes to prove the correctness of the analysis that we are going to define.

Definition 17 introduces a *path*, that is, a sequence of field dereferences, starting from a local variable, that lead to a value. In the following, for simplicity, we write f for a field $\kappa.f : t \in \mathbb{F}$. The full field signature, however, is used in the actual implementation, to allow for more fields sharing the same identifier. We only consider fields of reference type, since in Java fields of primitive type cannot hold `null` and primitive values have no fields. Paths belong to a set $\mathbb{P}$. In order to fix an upper bound on the local variables used in the path, we also define a set $\mathbb{P}_i$.

Definition 17 A *path* is $k.f$ where $k \geq 0$ and $f \in \mathbb{F}$ has type in $\mathbb{K}$, or $p.f$ where p is a path and $f \in \mathbb{F}$ has type in $\mathbb{K}$. The set of all paths is written $\mathbb{P}$. The *starting local variable* of a path $p \in \mathbb{P}$ is

$$local(p) = \begin{cases} k & \text{if } p = k.f, \\ local(p') & \text{if } p = p'.f. \end{cases}$$

We also define $\mathbb{P}_i = \{p \in \mathbb{P} \mid local(p) < i\}$. Given $\sigma \in \Sigma_{i,j}$ and $p \in \mathbb{P}_i$, we define the *value of p in σ* as

$$\sigma(p) = \begin{cases} \mu(l_k).f & \text{if } p = k.f, l_k \in \mathbb{L} \text{ and } \mu(l_k) \text{ has a field } f \\ \mu(\sigma(p')).f & \text{if } p = p'.f, \sigma(p') \in \mathbb{L} \text{ and } \mu(\sigma(p')) \text{ has a field } f \\ undefined & \text{otherwise,} \end{cases}$$

where $\sigma = \langle l \,\|\, s \,\|\, \mu\rangle$ or $\sigma = \underline{\langle l \,\|\, s \,\|\, \mu\rangle}$.

Example 13 In the program in Fig. 2 we have $0.\texttt{tail} \in \mathbb{P}_1$ as well as $0.\texttt{tail} \in \mathbb{P}_2$. Let $\sigma = \langle \varepsilon \,\|\, [\ell] \,\|\, [\ell \to o, \ell' = o'] \rangle \in \Sigma$, where o and o' are objects of class `List`, $o.\texttt{head} = \ell$, $o.\texttt{tail} = \ell'$, $o'.\texttt{head} = \texttt{null}$ and $o'.\texttt{tail} = \texttt{null}$. Then $\sigma(0.\texttt{tail}) = \mu(\ell).\texttt{tail} = o.\texttt{tail} = \ell'$.

We can define now a map that *extracts* the set of *non-`null` paths*, i.e. paths that are non-`null` in a given state. Note that this set may well be infinite.

Definition 18 (Non-`null` Paths Extractor) Let $\sigma \in \Sigma_{i,j}$. We define the *non-`null` paths extractor*

$$nnpaths(\sigma) = \{p \in \mathbb{P} \mid \sigma(p) \in \mathbb{L}\}.$$

Example 14 Let σ be as in Example 13. We have

$$nnpaths(\sigma) = \left\{ \begin{array}{l} 0.\texttt{head}, 0.\texttt{tail}, 0.\texttt{head.head}, 0.\texttt{head.tail}, \\ 0.\texttt{head.head.head}, 0.\texttt{head.head.tail}, \ldots \end{array} \right\}.$$

Note that this is an infinite set.

The elements of the abstract domain $\mathbb{PATH}$ represent sets of denotations. They contain sets of non-`null` paths, which are guaranteed to hold a non-`null` value at the end of those denotations. It is important, for better precision, to distinguish normal and exceptional inputs and normal and exceptional outputs. Consider for instance a method of class `List`

```
void expand(List l) {
 // assume that this.tail might be null
   at this point
 new Object();
 this.tail.head = l;
}
```

After its execution, we expect that $0.\texttt{tail}$ does not hold `null`, where 0 is the local variable holding `this`, since otherwise an exception would be thrown by the assignment `this.tail.head = l`. However, this is not completely true: the method might stop and throw an exception because (for instance) of lack of memory for the `new` statement, in which case field `this.tail` might hold `null`. Hence we should rather say that $0.\texttt{tail}$ does not hold `null` *in the normal output states of the method*. There are other situations when we want to distinguish between normal and exceptional states:

```
static void expand(List l) {
  // assume that l.tail might be null
```

```
    at this point
  try {
    new Object();
    if (l.tail != null) l.tail.hashCode();
  }
  catch (Throwable t) {
    l.tail.toString();
  }
}
```

In this example, we do not want a warning about the possible nullness of the receiver of the call to `hashCode()`, since that statement is protected by an explicit check about the nullness of `l.tail`. Instead, we do expect a warning about the possible nullness of the receiver of the call to `toString()` since, there, `l.tail` might hold `null`. This means that we want to execute the exception handler from the exceptional states generated by the body of the `try` statement, rather than from the normal states: there are such exceptional states where `l.tail` might hold `null`.

These arguments let us conclude that it is important to keep distinct the set of non-`null` paths in the normal output states from those in the exceptional output states, both under the hypotheses that the input state is normal or exceptional itself. This leads to four sets of non-`null` paths:

1. the set *NN* of paths that are non-`null` in the normal output states of the denotations if the input state is normal;
2. the set *NE* of paths that are non-`null` in the exceptional output states of the denotations if the input state is normal;
3. the set *EN* of paths that are non-`null` in the normal output states of the denotations if the input state is exceptional;
4. the set *EE* of paths that are non-`null` in the exceptional output states of the denotations if the input state is exceptional.

As we have observed above, our abstract domain should also include a set of local variables and a set of fields of reference type that might be modified from the input to the output of the denotations. In principle, the same partition in four sets could be required for those sets. However, this would make the abstract domain clumsy and does not seem to increase the precision very much, hence we have chosen to use a unique set of local variables L and a unique set of fields F, for all the four situations above.

Definition 19 ($\mathbb{PATH}$ Abstract Domain) Let $i_1, j_1, i_2, j_2 \in \mathbb{N}$. The *abstract domain for non-`null` paths* is

$$\mathbb{PATH}_{i_1,j_1\to i_2,j_2} = \left\{ \langle NN \| NE \| EN \| EE \| L \| F\rangle \;\middle|\; \begin{array}{l} NN, NE, EN, EE \subseteq \mathbb{P}_{i_2} \\ L \subseteq \{0,\ldots,i_1-1\} \\ F \subseteq \{\kappa.f : t \in \mathbb{F} \mid t \in \mathbb{K}\} \end{array} \right\}.$$

Its elements are ordered as $\langle NN_1 \| NE_1 \| EN_1 \| EE_1 \| L_1 \| F_1\rangle \le \langle NN_2 \| NE_2 \| EN_2 \| EE_2 \| L_2 \| F_2\rangle$ if and only if $NN_1 \supseteq NN_2$, $NE_1 \supseteq NE_2$, $EN_1 \supseteq EN_2$, $EE_1 \supseteq EE_2$, $L_1 \subseteq L_2$ and $F_1 \subseteq F_2$. That ordering relation makes $\mathbb{PATH}_{i_1,j_1\to i_2,j_2}$ a complete lattice. Let $I \subseteq \mathbb{N}$ and $\{\langle NN_i \| NE_i \| EN_i \| EE_i \| L_i \| F_i\rangle\}_{i\in I} \subseteq \mathbb{PATH}_{i_1,j_1\to i_2,j_2}$. Their greatest lower bound $\sqcap$ is

$$\langle \cup_{i\in I} NN_i \| \cup_{i\in I} NE_i \| \cup_{i\in I} EN_i \| \cup_{i\in I} EE_i \| \cap_{i\in I} L_i \| \cap_{i\in I} F_i\rangle.$$

The elements of this abstract domain will typically be written as *path*. It is important to remark that Definition 19 requires the starting local variable of the paths to belong to the final state, which has i_2 local variables. The possibly modified local variables must instead belong to the i_1 local variables of the initial state (and hence, by Definition 4, also to the i_1 lowest local variables of the final state).

Example 15 An element of $\mathbb{PATH}_{1,0\to 1,1}$ is $path = \langle\{0.\mathtt{tail}\} \| \emptyset \| \mathbb{P}_1 \| \mathbb{P}_1 \| \emptyset \| \emptyset\rangle$. This might be the approximation of the behaviour of the sequential execution of the topmost two blocks in Fig. 8. That is, at the end of every execution of those blocks, if the input and output states are normal then `0.tail` is non-`null` (it has been checked by the `ifne` bytecode); if the input state is normal and the output state is exceptional then no path is known to be non-`null`. If the input state is exceptional, then every path is vacuously non-`null` in the output state, since there is no such output state (`receiver_is` is only defined on normal input states). Moreover, those blocks do not modify any local variable nor field.

The number j_1 and j_2 of the stack elements is not used in Definition 19, but it is essential for a formal definition of the concretisation map.

Definition 20 (Concretisation Map) We define the *concretisation map*

$$\gamma : \mathbb{PATH}_{i_1,j_1\to i_2,j_2} \to \wp(\Delta_{i_1,j_1\to i_2,j_2})$$

such that, given $path = \langle NN \| NE \| EN \| EE \| L \| F\rangle \in \mathbb{PATH}_{i_1,j_1\to i_2,j_2}$:

$$\gamma(path) = \left\{ \delta \in \Delta_{i_1,j_1\to i_2,j_2} \;\middle|\; \begin{array}{l} \text{for all } \sigma \in \Sigma_{i_1,j_1} \text{ s.t. } \delta(\sigma) \text{ is defined} \\ \text{letting } \sigma = \langle l \| s \| \mu\rangle \text{ and } \delta(\sigma) = \langle l' \| s' \| \mu'\rangle \\ \text{(both states are possibly underlined)} \\ 1.\ \sigma \in \Xi,\ \delta(\sigma) \in \Xi \Rightarrow NN \subseteq nnpaths(\delta(\sigma)) \\ 2.\ \sigma \in \Xi,\ \delta(\sigma) \in \underline{\Xi} \Rightarrow NE \subseteq nnpaths(\delta(\sigma)) \\ 3.\ \sigma \in \underline{\Xi},\ \delta(\sigma) \in \Xi \Rightarrow EN \subseteq nnpaths(\delta(\sigma)) \\ 4.\ \sigma \in \underline{\Xi},\ \delta(\sigma) \in \underline{\Xi} \Rightarrow EE \subseteq nnpaths(\delta(\sigma)) \\ 5.\ L \supseteq \{0 \le k < i_1 \mid l_k \ne l'_k\} \\ 6.\ F \supseteq \left\{ \kappa.f : t \;\middle|\; \begin{array}{l} t \in \mathbb{K},\ \ell \in dom(\mu) \\ \mu(\ell).f \ne \mu'(\ell).f \end{array} \right\} \end{array} \right\}.$$

Definition 20 uses different directions of approximation for the components of *path*: the paths *must* be non-`null` at the end of the denotations (for the right kind of normal/exceptional states combination) while the local variables in L and the fields in F should include all the local variables and fields that *may* be modified by the denotations (w.r.t. both normal and exceptional states).

Example 16 The denotation δ representing the sequential execution of the topmost two blocks in Fig. 8 belongs to $\gamma(path)$, where $path$ is defined in Example 15. This is because δ does not modify any local variable nor field, is only defined on input normal states and when the output state is normal then it must be the case that field `tail` of local variable 0 holds a non-`null` value there.

Lemma 4 *The map γ of Definition 20 is co-additive.*

Proof Let $i_1, j_1, i_2, j_2 \in \mathbb{N}$, $I \subseteq \mathbb{N}$ and $path_i = \langle NN_i \parallel NE_i \parallel EN_i \parallel EE_i \parallel L_i \parallel F_i \rangle \in \mathbb{PATH}_{i_1,j_1\to i_2,j_2}$ for all $i \in I$. We prove that $\gamma(\cap_{i\in I} path_i) = \cap_{i\in I}\gamma(path_i)$. By Definition 19, $\gamma(\cap_{i\in I} path_i)$ is

$$
\begin{aligned}
&\gamma(\langle \cup_{i\in I} NN_i \parallel \cup_{i\in I} NE_i \parallel \cup_{i\in I} EN_i \parallel \cup_{i\in I} EE_i \parallel \cap_{i\in I} L_i \parallel \cap_{i\in I} F_i\rangle)\\
&= \left\{ \delta \in \Delta_{i_1,j_1\to i_2,j_2} \;\middle|\; \begin{array}{l}
\text{for all } \sigma \in \Sigma_{i_1,j_1} \text{ s.t. } \delta(\sigma) \text{is defined}\\
\text{letting } \sigma = \langle l \parallel s \parallel \mu\rangle \text{ and } \delta(\sigma) = \langle l' \parallel s' \parallel \mu'\rangle\\
\text{(both states are possibly underlined)}\\
1.\ \ \sigma \in \Xi,\ \delta(\sigma) \in \Xi \Rightarrow \cup_{i\in I} NN_i \subseteq nnpaths(\delta(\sigma))\\
2.\ \ \sigma \in \Xi,\ \delta(\sigma) \in \underline{\Xi} \Rightarrow \cup_{i\in I} NE_i \subseteq nnpaths(\delta(\sigma))\\
3.\ \ \sigma \in \underline{\Xi},\ \delta(\sigma) \in \Xi \Rightarrow \cup_{i\in I} EN_i \subseteq nnpaths(\delta(\sigma))\\
4.\ \ \sigma \in \underline{\Xi},\ \delta(\sigma) \in \underline{\Xi} \Rightarrow \cup_{i\in I} EE_i \subseteq nnpaths(\delta(\sigma))\\
5.\ \ \cap_{i\in I} L_i \supseteq \{0 \le k < i_1 \mid l_k \ne l'_k\}\\
6.\ \ \cap_{i\in I} F_i \supseteq \left\{ \kappa.f : t \;\middle|\; \begin{array}{l} t \in \mathbb{K},\ \ell \in dom(\mu)\\ \mu(\ell).f \ne \mu'(\ell).f \end{array}\right\}
\end{array}\right\}\\
&= \cap_{i\in I} \left\{ \delta \in \Delta_{i_1,j_1\to i_2,j_2} \;\middle|\; \begin{array}{l}
\text{for all } \sigma \in \Sigma_{i_1,j_1} \text{ s.t. } \delta(\sigma) \text{ is defined}\\
\text{letting } \sigma = \langle l \parallel s \parallel \mu\rangle \text{ and } \delta(\sigma) = \langle l' \parallel s' \parallel \mu'\rangle\\
\text{(both states are possibly underlined)}\\
1.\ \sigma \in \Xi,\ \delta(\sigma) \in \Xi \Rightarrow NN_i \subseteq nnpaths(\delta(\sigma))\\
2.\ \sigma \in \Xi,\ \delta(\sigma) \in \underline{\Xi} \Rightarrow NE_i \subseteq nnpaths(\delta(\sigma))\\
3.\ \sigma \in \underline{\Xi},\ \delta(\sigma) \in \Xi \Rightarrow EN_i \subseteq nnpaths(\delta(\sigma))\\
4.\ \sigma \in \underline{\Xi},\ \delta(\sigma) \in \underline{\Xi} \Rightarrow EE_i \subseteq nnpaths(\delta(\sigma))\\
5.\ L_i \supseteq \{0 \le k < i_1 \mid l_k \ne l'_k\}\\
6.\ F_i \supseteq \left\{ \kappa.f : t \;\middle|\; \begin{array}{l} t \in \mathbb{K},\ \ell \in dom(\mu)\\ \mu(\ell).f \ne \mu'(\ell).f \end{array}\right\}
\end{array}\right\}\\
&= \cap_{i\in I}\gamma(path_i).
\end{aligned}
$$

□

The map γ is co-additive (Lemma 4). By a general result of abstract interpretation [9], we have the thesis. □

Figure 9 defines abstractions over the $\mathbb{PATH}$ domain for each bytecode of our language. Let us comment these definitions. In our language, only `store` modifies a local variable and only `putfield` modifies a field. Hence, in all other cases, we use $\emptyset$ for the last two components of the approximations. The definition of the first four components is somehow more complex. We note that the set $\mathbb{P}_i$ of all paths over i local variables is the most precise approximation available for each of these four components. We use $\mathbb{P}_i$ when we know that a component represents an impossible behaviour for a bytecode. Namely, for those bytecodes that never throw any exception, we use $\mathbb{P}_i$ for both the *NE* and *EE* components: since the set of output exceptional states is empty in that case, we can approximate this empty set however we want. By using $\mathbb{P}_i$, we pick up the best possible approximation. This is the case, for instance, of `store`. For those bytecodes that are only defined on normal input states, every approximation is correct for *EN* and *EE* and by using $\mathbb{P}_i$ we pick up the best possible approximation. This is the case of every bytecode except for `catch`. The latter is only defined on exceptional input states. Hence, for `catch`, every approximation is correct for *NN* and *NE* and by using $\mathbb{P}_i$ we pick up the best possible approximation.

Proposition 6 *$\mathbb{PATH}_{i_1,j_1\to i_2,j_2}$ is an abstract interpretation of $\wp(\Delta_{i_1,j_1\to i_2,j_2})$ with γ as concretisation map.*

Proof The domain $\mathbb{PATH}_{i_1,j_1\to i_2,j_2}$ is a complete lattice w.r.t. $\le$ with $\cap$ as greatest lower bound operator (Definition 19). The domain $\wp(\Delta_{i_1,j_1\to i_2,j_2})$ is a complete lattice w.r.t. set inclusion with $\cap$ as greatest lower bound operator.

$$
\begin{aligned}
(store\ k\ t_{alias})^{\mathbb{PATH}} &= \langle \emptyset \,\|\, \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \{k\} \,\|\, \emptyset \rangle \\
(ifne\ t_{alias})^{\mathbb{PATH}} &= \langle \{p \mid p \in \mathbb{P}_i \text{ and } p \text{ is alias of } s_{j-1}\} \,\|\, \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \emptyset \,\|\, \emptyset \rangle \\
(new\ \kappa_{alias})^{\mathbb{PATH}} &= \langle \emptyset \,\|\, \emptyset \,\|\, \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \emptyset \,\|\, \emptyset \rangle \\
(getfield\ \kappa.f : t_{alias})^{\mathbb{PATH}} &= \langle \{p \mid p \in \mathbb{P}_i \text{ and } p \text{ is alias of } s_{j-1}\} \,\|\, \emptyset \,\|\, \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \emptyset \,\|\, \emptyset \rangle \\
(putfield\ \kappa.f : t_{alias})^{\mathbb{PATH}} &= \langle \left\{ p \,\middle|\, \begin{array}{l} p \in \mathbb{P}_i,\ p \text{ is alias of } s_{j-2} \\ f \text{ does not occur in } p \end{array} \right\} \,\|\, \emptyset \,\|\, \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \emptyset \,\|\, \{f\} \rangle \\
(throw\ \kappa_{alias})^{\mathbb{PATH}} &= \langle \mathbb{P}_i \,\|\, \emptyset \,\|\, \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \emptyset \,\|\, \emptyset \rangle \\
(catch_{alias})^{\mathbb{PATH}} &= \langle \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \emptyset \,\|\, \mathbb{P}_i \,\|\, \emptyset \,\|\, \emptyset \rangle \\
(exception\_is\ K_{alias})^{\mathbb{PATH}} &= \langle \{p \mid p \in \mathbb{P}_i \text{ and } p \text{ is alias of } s_0\} \,\|\, \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \emptyset \,\|\, \emptyset \rangle \\
(receiver\_is\ K_{alias})^{\mathbb{PATH}} &= \langle \{p \mid p \in \mathbb{P}_i \text{ and } p \text{ is alias of } l_0\} \,\|\, \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \emptyset \,\|\, \emptyset \rangle \\
(ins_{alias})^{\mathbb{PATH}} &= \langle \emptyset \,\|\, \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \mathbb{P}_i \,\|\, \emptyset \,\|\, \emptyset \rangle \quad \text{for all other bytecodes } \mathtt{ins}
\end{aligned}
$$

Fig. 9 Bytecode abstractions for non-`null` paths in a program point with i local variables and j stack elements

Let use hence read some representative approximations in Fig. 9.

`ifne` The first component says that if we run `ifne` from a normal input state and we get a normal output state then in that output state every path p which is a definite alias of the checked value s_{j-1} on top of the input stack will be non-`null`. This agrees with the definition of *ifne* (Sect. 3). Since this bytecode is only defined on normal input states and only yields normal output states, the subsequent three components are $\mathbb{P}_i$. Moreover, this bytecode does not modify any local variable nor field hence the last two components are $\emptyset$. Bytecodes `exception_is` and `receiver_is` are approximated similarly, by only changing the stack element or local variable that contain the value which must not be `null` according to the semantics of the bytecode.

`getfield` The first component, again, says that if we run `getfield` from a normal input state and get a normal output state then, in the latter, all paths that are definitely alias of the receiver of the `getfield` must be non-`null`. However, it is interesting to observe that we use $\emptyset$ as second component *NE* here. This is because this bytecode might throw an exception when the receiver is `null`. Hence, there are cases when we run that bytecode from a normal state and get an exceptional state. In the latter, there is no way to conclude that the receiver was non-`null` and we hence use $\emptyset$ as approximation. Since `getfield` is only defined on normal input states, the third and fourth components are $\mathbb{P}_i$. Since it does not modify any local variable nor field, the last two components are $\emptyset$. The case of `putfield` is similar, but since that bytecode modifies a field f, we only consider as definitely non-`null` only those paths that are definitely alias of the receiver *and* where f does not occur. This is because the field update might write `null` in a path where f occurs, as in

```
b.f.g = exp1;
a.f = exp2;
```

After the first field update we are sure that `b.f` is not `null`, but after the second assignment this is not sure anymore, since `exp2` might hold `null` and `a` and `b` might be aliases. Also, note that after the assignment

```
a.f.f = exp
```

we do not assume that `a.f` is non-`null`, since it might be the case that `exp` holds `null` and `a.f` is an alias of `a`, so that we end up writing `null` into `a.f`;

`throw` This bytecode always throws an exception. The latter might be the value *top* on top of the input stack or a new `NullPointerException`, when *top* = `null` (Sect. 3). It follows that it is impossible to run `throw` from a normal or exceptional state and get a normal state. Hence, the first and third components are $\mathbb{P}_i$. It is possible, instead, to run `throw` from a normal state and get an exceptional state. As we said above, in that case, it is equally possible that the top of the input stack was `null` as well as that it was non-`null`. It follows that we cannot guarantee that the paths that are aliases of that value are definitely non-`null` as we did in the case of `ifne`. We use the less precise, but always correct, approximation $\emptyset$ as second component, instead. Moreover, `throw` is not defined on input exceptional states, so the fourth component is $\mathbb{P}_i$. Since that bytecode does not modify any local variable nor field, the last two components are $\emptyset$.

`catch` This bytecode is only defined on exceptional input states and always yields a normal output state (Sect. 3). Hence the first, second and fourth components are $\mathbb{P}_i$. If we run `catch` form an exceptional state we get a normal state where no local variable nor field has been modified. Hence we have no way to prove that some specific field is non-`null` and the third component is the always correct approximation $\emptyset$, as well as the last two components.

Example 17 Let us compute the approximations over $\mathbb{PATH}$ of the bytecodes occurring in the topmost two blocks in Fig. 8. We assume that the aliasing analysis has been able to conclude that the top of the stack is a definite alias of `0.tail` at the beginning of the `ifne` bytecode there. Observe that there is only one local variable in those blocks, hence $\mathbb{P}_i$ in Fig. 9 stands for $\mathbb{P}_1$ in this case.

$$(receiver\_is\ List)^{\mathbb{PATH}} = \langle \emptyset \,\|\, \mathbb{P}_1 \,\|\, \mathbb{P}_1 \,\|\, \mathbb{P}_1 \,\|\, \emptyset \,\|\, \emptyset \rangle$$

$$(load\ 0\ List)^{\mathbb{PATH}} = \langle \emptyset \,\|\, \mathbb{P}_1 \,\|\, \mathbb{P}_1 \,\|\, \mathbb{P}_1 \,\|\, \emptyset \,\|\, \emptyset \rangle$$

$$(getfield\ List.tail : List)^{\mathbb{PATH}} = \langle \emptyset \,\|\, \emptyset \,\|\, \mathbb{P}_1 \,\|\, \mathbb{P}_1 \,\|\, \emptyset \,\|\, \emptyset \rangle$$

$(\textit{ifne List})^{\mathbb{PATH}} = \langle \{0.\texttt{tail}\} \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle$

$(\textit{load 0 List})^{\mathbb{PATH}} = \langle \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle$

$(\textit{getfield List.tail} : \textit{List})^{\mathbb{PATH}} = \langle \emptyset \parallel \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle.$

The approximations in Fig. 9 are correct. In order to prove it (Proposition 7) we need the following result.

Lemma 5 *Let* ins *be a bytecode instruction such that* $ins : \mathbb{PATH}_{i_1,j_1 \to i_2,j_2}$. *Let* $L = \{0,\ldots,i_1-1\}$ *and* $F = \{\kappa.f : t \in \mathbb{F} \mid t \in \mathbb{K}\}$. *Let* $\sigma \in \Sigma_{i_1,j_1}$ *be such that* $ins(\sigma) \in \Sigma_{i_2,j_2}$ *is defined,* $\sigma = \langle l \parallel s \parallel \mu \rangle$ *and* $ins(\sigma) = \langle l' \parallel s' \parallel \mu' \rangle$ *(both states are possibly underlined). Then:*

1. *if* $ins(\sigma) \in \underline{\Xi}$ *whenever* $\sigma \in \Xi$, *we have* $ins \in \gamma(\langle \mathbb{P}_i \parallel \emptyset \parallel \emptyset \parallel \emptyset \parallel L \parallel F \rangle)$;
2. *if* $ins(\sigma) \in \Xi$ *whenever* $\sigma \in \Xi$, *we have* $ins \in \gamma(\langle \emptyset \parallel \mathbb{P}_i \parallel \emptyset \parallel \emptyset \parallel L \parallel F \rangle)$;
3. *if* $ins(\sigma) \in \underline{\Xi}$ *whenever* $\sigma \in \underline{\Xi}$, *we have* $ins \in \gamma(\langle \emptyset \parallel \emptyset \parallel \mathbb{P}_i \parallel \emptyset \parallel L \parallel F \rangle)$;
4. *if* $ins(\sigma) \in \Xi$ *whenever* $\sigma \in \underline{\Xi}$, *we have* $ins \in \gamma(\langle \emptyset \parallel \emptyset \parallel \emptyset \parallel \mathbb{P}_i \parallel L \parallel F \rangle)$;
5. *if* $l_i = l'_i$ *with* $i \in L$ *in any such* σ, *then* $ins \in \gamma(\langle \emptyset \parallel \emptyset \parallel \emptyset \parallel \emptyset \parallel \emptyset \parallel F \rangle)$;
6. *if* $\mu(\ell).f = \mu'(\ell).f$ *for every* $\ell \in dom(\mu), \kappa.f : t$ *with* $t \in \mathbb{K}$ *in any such* σ, *then* $ins \in \gamma(\langle \emptyset \parallel \emptyset \parallel \emptyset \parallel \emptyset \parallel L \parallel \emptyset \rangle)$.

Proof Let us prove point 1. Since $ins(\sigma) \in \underline{\Xi}$ whenever $\sigma \in \Xi$ and $ins(\sigma)$ is defined, the precondition of constraint 1 in Definition 20 is always false. Hence that constraint holds. Constraints 2, 3 and 4 of the same definition hold since $\emptyset$ is included in any other set. Constraint 5 holds since we have chosen L as the set of all input local variables, hence including any other set of input local variables. Constraint 6 holds since we have chosen F as the set of all fields of reference type, hence including any other set of fields of reference type. Points 2, 3 and 4 are proved similarly. Consider point 5. By choosing $\emptyset$ for the first four sets, we satisfy the first four constraints in Definition 20; by choosing F as the set of all fields of reference type, we satisfy the sixth constraint in the same definition. For the fifth, by the hypothesis that it is always the case that $l_k = l'_k$, we conclude that $\emptyset \supseteq \{0 \le k < i_1 \mid l_k \ne l'_k\}$. The proof of point 6 is similar to that of point 5. □

Proposition 7 (Correctness of the Abstract Bytecodes) *The approximations in Fig. 9 are correct w.r.t. the denotations of Sect. 3, i.e. for all bytecode* ins_{alias} *we have* $ins_{alias} \in \gamma((ins_{alias})^{\mathbb{PATH}})$.

Proof We consider each bytecode $\texttt{ins}_{alias}$ such that $ins_{alias} : \mathbb{PATH}_{i,j \to i',j'}$. For simplicity, in the following we do not write $alias$. Let $\sigma \in \Sigma_{i,j}$ be such that $\sigma' = ins(\sigma) \in \Sigma_{i',j'}$ is defined. Let $\sigma = \langle l \parallel s \parallel \mu \rangle$ and $\sigma' = \langle l' \parallel s' \parallel \mu' \rangle$ (both states are possibly underlined). Let $L = \{h \mid 0 \le h < i\}$ and $F = \{\kappa.f : t \in \mathbb{F} \mid t \in \mathbb{K}\}$.

`store` $k\ t$

From Sect. 3, we know that in this case $l_h \ne l'_h$ for every $0 \le h < i, h \ne k$. Hence $\{k\} \supseteq \{0 \le h < i \mid l_h \ne l'_h\}$. We conclude that

$$store\ k\ t \in \gamma(\langle \emptyset \parallel \emptyset \parallel \emptyset \parallel \emptyset \parallel \{k\} \parallel F \rangle). \tag{3}$$

We know that *store k t* is only defined on normal input states, always yields normal output states and does not modify any field. Hence points 2, 3, 4 and 6 of Lemma 5 hold (for points 3 and 4, observe that it is never the case that $(store\ k\ t)(\sigma)$ is defined when $\sigma \in \underline{\Xi}$). We conclude that

$$store\ k\ t \in \gamma(\langle \emptyset \parallel \mathbb{P}_i \parallel \emptyset \parallel \emptyset \parallel L \parallel F \rangle), \tag{4}$$

$$store\ k\ t \in \gamma(\langle \emptyset \parallel \emptyset \parallel \mathbb{P}_i \parallel \emptyset \parallel L \parallel F \rangle), \tag{5}$$

$$store\ k\ t \in \gamma(\langle \emptyset \parallel \emptyset \parallel \emptyset \parallel \mathbb{P}_i \parallel L \parallel F \rangle), \tag{6}$$

$$store\ k\ t \in \gamma(\langle \emptyset \parallel \emptyset \parallel \emptyset \parallel \emptyset \parallel L \parallel \emptyset \rangle). \tag{7}$$

From (3)–(7) and by Lemma 4 we conclude that

$$\begin{aligned} store\ k\ t &\in \gamma(\langle \emptyset \parallel \emptyset \parallel \emptyset \parallel \emptyset \parallel \{k\} \parallel F \rangle) \\ &\cap \gamma(\langle \emptyset \parallel \mathbb{P}_i \parallel \emptyset \parallel \emptyset \parallel L \parallel F \rangle) \\ &\cap \gamma(\langle \emptyset \parallel \emptyset \parallel \mathbb{P}_i \parallel \emptyset \parallel L \parallel F \rangle) \cap \gamma(\langle \emptyset \parallel \emptyset \parallel \emptyset \parallel \mathbb{P}_i \parallel L \parallel F \rangle) \\ &\cap \gamma(\langle \emptyset \parallel \emptyset \parallel \emptyset \parallel \emptyset \parallel L \parallel \emptyset \rangle) \\ &= \gamma(\langle \emptyset \parallel \mathbb{P}_i \parallel \mathbb{P}_i \parallel \mathbb{P}_i \parallel \{k\} \parallel \emptyset \rangle) \\ &= store\ k\ t^{\mathbb{PATH}}. \end{aligned}$$

`ifne` t

We only prove the correctness of the first component in Fig. 9. For the other components, the reasoning is similar to that used for *store k t*. Let hence $\sigma \in \Sigma_{i,j}$. From Sect. 3 we know that if $(ifne\ t)(\sigma)$ is defined then $\sigma = \langle l \parallel top :: s \parallel \mu \rangle \in \Xi_{i,j}$, $top \ne 0$, $top \ne \texttt{null}$, $\sigma' = (ifne\ t)(\sigma) = \langle l \parallel s \parallel \mu \rangle \in \Xi_{i',j'}$ and σ satisfies the aliasing information $alias$ at the beginning of this bytecode (Eq. 2). Let $p \in \mathbb{P}_i$ be a definite alias of s_{j-1} at the beginning of this bytecode, i.e. a definite alias of *top*. By definition of alias, we have $top \in \mathbb{L}$ and $\sigma(p) = top$. Since $local(p) < i$ and σ and σ' agree on the lower i local variables and have the same memory, it follows that $\sigma'(p) = top$. Hence $p \in nnpaths(\sigma')$ (Definition 18). By Definition 20 we conclude that

$ifne\ t \in \gamma(\langle\{p \mid p \in \mathbb{P}_i and\ p\ is\ an\ alias\ of\ s_{j-1}\}$
$\parallel \emptyset \parallel \emptyset \parallel \emptyset \parallel L \parallel F\rangle)$.

The proofs for `exception_is` K and `receiver_is` K are similar.

`new` κ

The choice of $\emptyset$ as second component is always safe (a better choice cannot be done here since this bytecode might throw an exception). For the other components, the reasoning is similar to that used for *store k t*.

`getfield` $\kappa.f:t$

We only prove the correctness of the first component in Fig. 9. For the other components, the reasoning is similar to that used for *store k t*. Let hence $\sigma \in \Sigma_{i,j}$. From Sect. 3 we know that if $(getfield\ \kappa.f : t)(\sigma)$ is defined and $(getfield\ \kappa.f : t)(\sigma) \in \Xi_{i',j'}$ then $\sigma = \langle l \parallel top :: s \parallel \mu\rangle \in \Xi_{i',j'}, top \in \mathbb{L}, \sigma' = (getfield\ \kappa.f : t)(\sigma) = \langle l \parallel \mu(top).f :: s \parallel \mu\rangle \in \Xi_{i',j'}$ and σ satisfies the aliasing information $alias$ at the beginning of this bytecode (Eq. 2). We can hence proceed as in the case of $ifne\ t$.

`putfield` $\kappa.f:t$

Let $\sigma \in \Sigma_{i,j}$ be such that $\sigma' = (putfield\ \kappa.f : t)(\sigma)$ is defined. From Sect. 3 we know that $\sigma = \langle l \parallel top :: rec :: s \parallel \mu\rangle$ and $\sigma' = \langle l \parallel s \parallel \mu[\mu(rec).f := top]\rangle$ (when $rec \in \mathbb{L}$) or $\sigma' = \underline{\langle l \parallel \ell \parallel \mu[\ell := npe]\rangle}$ (otherwise) where $\ell \in \mathbb{L}$ is fresh and *npe* is an exception object. In both cases, this bytecode does not modify any field except field $\kappa.f:t$ of $\mu(rec)$. Hence $\{f\} \supseteq \{\kappa.g:d \mid d \in \mathbb{K},\ \ell' \in dom(\mu)$ and $\mu(\ell').g \neq \mu'(\ell').g\}$ where $\mu' = \mu[\mu(rec).f := top]$ or $\mu' = \mu[\ell := npe]$. We conclude that the sixth component $\{f\}$ is correct. Consider the first component now, for which we must check the case when $\sigma' \in \Xi_{i',j'}$ and hence $\sigma' = \langle l \parallel s \parallel \mu[\mu(rec).f := top]\rangle$ and $rec \in \mathbb{L}$. Let $p \in \mathbb{P}_i$ be a definite alias of s_{j-2} at the beginning of this bytecode, i.e. a definite alias of rec. Assume that f does not occur in p. Since σ satisfies the aliasing information $alias$ at the beginning of this bytecode (Eq. 2) we have $\sigma(p) = rec \in \mathbb{L}$. Since $local(p) < i$, σ and σ' agree on the lower i local variables, μ and μ' only differ for field f of $\mu(rec)$ and that field does not occur in p, we conclude that $\sigma'(p) = rec \in \mathbb{L}$ as well. Hence $p \in nnpaths(\sigma')$ (Definition 18). By Definition 20 we conclude that

$$putfield\ \kappa.f:t \in \gamma(\langle\left\{p \,\middle|\, \begin{array}{l} p \in \mathbb{P}_i,\ \ p \text{ is an alias of } s_{j-2} \\ f \text{ does not occur in } p \end{array}\right\}$$
$$\parallel \emptyset \parallel \emptyset \parallel \emptyset \parallel L \parallel F\rangle).$$

For the other components, the reasoning is similar to that for `store` $k\ t$ above.

`throw` κ

The proof is similar to that of `store` $k\ t$ above by observing that this bytecode is only defined on normal states, always yields an exceptional state and does not modify any local variable nor field.

`catch`

The proof is similar to that of `store` $k\ t$ above by observing that this bytecode is only defined on exceptional states, always yields a normal state and does not modify any local variable nor field.

other bytecodes `ins`

The proof is similar to that of `store` $k\ t$ above by observing that all the remaining bytecodes are only defined on normal states, always yield normal states and do not modify any local variable nor field. □

We define now the abstract counterpart $;^{\mathbb{PATH}}$ of the composition of denotations ;. The idea is that in $path_1 ;^{\mathbb{PATH}} path_2$ the normal outputs represented by $path_1$ must be matched with the normal inputs represented by $path_2$ and similarly for the exceptional ones. In order to match two sets of non-`null` fields, assume that we know that the paths in a set P_1 are definitely non-`null` at the end of every execution of a piece of code c_1 and that the paths in another set P_2 are definitely non-`null` at the end of every execution of another piece of code c_2. We want to compute a set of paths that are definitely non-`null` at the end of every execution of the compound piece of code $c_1; c_2$. A possible, correct answer is P_2, since if those paths are non-`null` at the end of every execution of c_2 then they are non-`null` at the end of every execution of $c_1; c_2$. Instead, it is incorrect, in general, to assume that the paths in P_1 are definitely non-`null` at the end of every execution of $c_1; c_2$, since this is only true after c_1, but c_2 might modify some local variable or field, possibly making those paths `null`. It follows that a correct set of paths, larger than P_2, which are definitely non-`null` after every execution of $c_1; c_2$ is

$\{p \in P_1 \mid c_2$ does not modify $local(p)$
nor any field occurring in $p\} \cup P_2$.

This is formalised below.

Definition 21 (Sequential Composition of Paths) Let $L \subset \mathbb{N}$ be a set of local variables and $F \subseteq \mathbb{F}$ a set of fields. Let $p \in \mathbb{P}$. We define

$$affected(k.f, L, F) \Leftrightarrow k \in L\ or\ \ f \in F$$
$$affected(p.f, L, F) \Leftrightarrow affected(p, L, F)\ or f \in F.$$

Let $P_1, P_2 \subseteq \mathbb{P}$. We define the *sequential composition of* P_1 *and* P_2 *under* L *and* F as

$$P_1 \bullet_{L,F} P_2 = \{p \in P_1 \mid \neg affected(p, L, F)\} \cup P_2.$$

We can define now the sequential composition of two elements of $\mathbb{PATH}$.

Definition 22 (Abstract Sequential Composition) Let $path_i = \langle NN_i \parallel NE_i \parallel EN_i \parallel EE_i \parallel L_i \parallel F_i \rangle \in \mathbb{PATH}$ for $i = 1, 2$. Their *sequential composition* is

$$path_1 ;^{\mathbb{PATH}} path_2 = \langle NN \parallel NE \parallel EN \parallel EE \parallel L \parallel F \rangle$$

where

$$NN = NN_1 \bullet_{L_2,F_2} NN_2 \cap NE_1 \bullet_{L_2,F_2} EN_2,$$

$$NE = NN_1 \bullet_{L_2,F_2} NE_2 \cap NE_1 \bullet_{L_2,F_2} EE_2,$$

$$EN = EN_1 \bullet_{L_2,F_2} NN_2 \cap EE_1 \bullet_{L_2,F_2} EN_2,$$

$$EE = EE_1 \bullet_{L_2,F_2} EE_2 \cap EN_1 \bullet_{L_2,F_2} NE_2,$$

$$L = L_1 \cup L_2,$$

$$F = F_1 \cup F_2.$$

Definition 22 states that, in the denotations represented by $path_1 ;^{\mathbb{PATH}} path_2$, the set of paths NN that are definitely non-`null` in the normal output states when the computation begins from a normal input state is computed by composing a normal input to normal output behaviour allowed by $path_1$ followed by a normal input to normal output behaviour allowed by $path_1$. However, there is also the possibility of composing a normal input to exceptional output behaviour from $path_1$ followed by an exceptional input to normal output behaviour from $path_2$. Hence one considers the intersection of the paths that are definitely non-`null` in both situations. The definitions of NE, EN and EE are similar. The set of local variables that might be modified in the denotations represented by $path_1 ;^{\mathbb{PATH}} path_2$ are those that might be modified in those represented by $path_1$ or by $path_2$. Similarly for the set of fields that might be modified.

Example 18 Consider the approximations in Example 17. The abstract sequential composition of the first two approximations is

$$\begin{aligned}
&\langle \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle ;^{\mathbb{PATH}} \langle \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle \\
&\quad = \langle \emptyset \bullet_{\emptyset,\emptyset} \emptyset \cap \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \parallel \emptyset \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \cap \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \parallel \mathbb{P}_1 \\
&\qquad \bullet_{\emptyset,\emptyset} \emptyset \cap \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \parallel \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \cap \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \parallel \\
&\qquad \emptyset \cup \emptyset \parallel \emptyset \cup \emptyset \rangle \\
&\quad = \langle \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle.
\end{aligned}$$

The abstract sequential composition of this result with the third approximation is

$$\begin{aligned}
&\langle \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle ;^{\mathbb{PATH}} \langle \emptyset \parallel \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle \\
&\quad = \langle \emptyset \bullet_{\emptyset,\emptyset} \emptyset \cap \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \parallel \emptyset \bullet_{\emptyset,\emptyset} \emptyset \cap \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \parallel \\
&\qquad \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \emptyset \cap \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \parallel \\
&\qquad \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \cap \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \emptyset \parallel \emptyset \cup \emptyset \parallel \emptyset \cup \emptyset \rangle \\
&\quad = \langle \emptyset \parallel \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle.
\end{aligned}$$

The abstract sequential composition of this result with the fourth approximation is

$$\begin{aligned}
&\langle \emptyset \parallel \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle ;^{\mathbb{PATH}} \langle \{\texttt{0.tail}\} \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle \\
&\quad = \langle \emptyset \bullet_{\emptyset,\emptyset} \{\texttt{0.tail}\} \cap \emptyset \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \parallel \emptyset \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \cap \emptyset \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \parallel \\
&\qquad \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \{\texttt{0.tail}\} \cap \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \parallel \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \cap \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \emptyset \\
&\qquad \parallel \emptyset \cup \emptyset \parallel \emptyset \cup \emptyset \rangle \\
&\quad = \langle \{\texttt{0.tail}\} \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle.
\end{aligned}$$

The latter approximation means that, after the execution of the `ifne` bytecode, the analysis concludes that `0.tail` holds a definitely non-`null` value. By composing the latter result with the fifth approximation, one gets

$$\begin{aligned}
&\langle \{\texttt{0.tail}\} \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle ;^{\mathbb{PATH}} \langle \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle \\
&\quad = \langle \{\texttt{0.tail}\} \bullet_{\emptyset,\emptyset} \emptyset \cap \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \parallel \{\texttt{0.tail}\} \\
&\qquad \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \cap \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \parallel \\
&\qquad \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \emptyset \cap \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \parallel \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \\
&\qquad \mathbb{P}_1 \cap \mathbb{P}_1 \bullet_{\emptyset,\emptyset} \mathbb{P}_1 \parallel \emptyset \cup \emptyset \parallel \emptyset \cup \emptyset \rangle \\
&\quad = \langle \{\texttt{0.tail}\} \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle.
\end{aligned}$$

This means that our analysis concludes that, just before the execution of the second `getfield` from the top in Fig. 8, field `tail` of local variable 0 holds a non-`null` value. By composing the latter result with the last approximation, one gets

$$\langle \{\texttt{0.tail}\} \parallel \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset \rangle,$$

which approximates the behaviour of every sequential execution of the topmost two blocks in Fig. 8 (see also Example 15).

Example 19 Let

$$\begin{aligned}
path_1 &= \langle \{\texttt{0.tail}\} \parallel \{\texttt{1.tail}, \texttt{1.tail.head}\} \\
&\qquad \parallel \mathbb{P}_2 \parallel \emptyset \parallel \emptyset \parallel \emptyset \rangle \\
path_2 &= \langle \{\texttt{0.head}\} \parallel \emptyset \parallel \mathbb{P}_2 \parallel \mathbb{P}_2 \parallel \{0\} \parallel \{\texttt{head}\} \rangle.
\end{aligned}$$

Let $\bullet$ stand for $\bullet_{\{0\},\{\mathtt{head}\}}$. We have

$$\begin{aligned} path_1;^{\mathbb{PATH}} path_2 = \langle &\{0.\mathtt{tail}\} \bullet \{0.\mathtt{head}\} \cap \{1.\mathtt{tail}, \\ &1.\mathtt{tail.head}\} \bullet \mathbb{P}_2 \parallel \\ &\{0.\mathtt{tail}\} \bullet \emptyset \cap \{1.\mathtt{tail}, 1.\mathtt{tail.head}\} \bullet \mathbb{P}_2 \parallel \\ &\mathbb{P}_2 \bullet \{0.\mathtt{head}\} \cap \emptyset \bullet \mathbb{P}_2 \parallel \emptyset \bullet \mathbb{P}_2 \cap \mathbb{P}_2 \bullet \emptyset \parallel \{0\} \parallel \{\mathtt{head}\}\rangle \\ = \langle &\{0.\mathtt{head}\} \parallel \emptyset \parallel \\ &\{0.\mathtt{head}, 1.\mathtt{tail}, 1.\mathtt{tail.tail}, 1.\mathtt{tail.tail}, \\ &\mathtt{tail}, \ldots\} \parallel \\ &\{1.\mathtt{tail}, 1.\mathtt{tail.tail}, 1.\mathtt{tail.tail}, \mathtt{tail}, \ldots\} \parallel \\ &\{0\} \parallel \{\mathtt{head}\}\rangle. \end{aligned}$$

For the definition of the abstract counterpart of *extend*, we assume that some aliasing information is available. This is the aliasing information at the program point where the method call modelled through *extend* occurs (Definition 6).

Definition 23 (Abstract *extend*) Let $i, j \in \mathbb{N}$, *alias* be some definite aliasing information and $M = \kappa.m(t_1, \ldots, t_n) : t$ with $j = b + n + 1$ and $b \geq 0$. Define $(extend^{i,j}_{M,alias})^{\mathbb{PATH}} : \mathbb{PATH}_{n+1,0\to i',r} \to \mathbb{PATH}_{i,j\to i,b+r}$ with $r = 0$ if $t = \mathtt{void}$ and $r = 1$ otherwise, such that, for every $path = \langle NN \parallel NE \parallel EN \parallel EE \parallel L \parallel F\rangle \in \mathbb{PATH}_{n+1,0\to i',r}$, the element $(extend^{i,j}_{M,alias})^{\mathbb{PATH}}(path)$ is

$$\langle NN_{b,L,alias} \cup \left\{ p \;\middle|\; \begin{array}{l} p \in \mathbb{P}_i,\ p \text{ is an alias of } s_b \\ \text{no } f \in F \text{ occurs in } p \end{array} \right\} \parallel \emptyset \parallel \mathbb{P}_i \parallel \mathbb{P}_i \parallel \emptyset \parallel F\rangle,$$

where

$$NN_{b,L,alias} = \left\{ p[l \to k] \;\middle|\; \begin{array}{l} p \in NN,\ l = local(p),\ l \notin L \\ s_{b+l} \text{ is a definite alias of } l_k \end{array} \right\}.$$

Let us discuss Definition 23. The element *path* approximates the denotations of the called method. Since the result of *extend* is only defined on normal input states (Sect. 3) the set of normal or exceptional states resulting from an input exceptional state is empty and we pick up the best possible approximation $\mathbb{P}_i$ as third and fourth component. Moreover, a method call does not change any local variable of the caller, so we use $\emptyset$ as fifth component. Instead, a field is modified during a method call if and only if the callee modifies the field, hence we keep the same approximation F as sixth component. For the first component, consider the set NN of paths p that are definitely non-`null` at the end of the method if we start its execution from a normal input state and obtain a normal output state. Let l be their starting local variable. We can guarantee that a path $p[l \to k]$ (we replace the starting local l with k) holds a non-`null` value at the end of the method call if the stack element s_{b+l} used to hold the actual argument of the call is definitely an alias of local variable l_k of the caller. In other words, we are performing parameter passing here and propagation of non-`null` paths along aliased parameters. The requirement $l \notin L$ is needed since we must be sure that local variable l is not modified inside the method, or otherwise its final value might not have anything to do with the initial actual parameter s_{b+l}. Beyond parameter passing, we state that all paths that are definitely alias of the receiver s_b of the call must be non-`null` if an exception is not thrown and if no field possibly modified during the method call occurs in those paths. For *NE*, we cannot guarantee any path to be non-`null` since it is possible that an exception is thrown when the receiver is `null`, the callee is not executed and no path is hence non-`null`.

Example 20 Assume that the approximation of the execution of the `iter()` method in Fig. 2 is

$$path = \langle \emptyset \parallel \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset\rangle.$$

Assume that the aliasing analysis *alias* guarantees that, at the recursive call inside method `iter()` in Fig. 2, the top of the stack s_0 is a definite alias of $0.\mathtt{tail}$. We have $b = 0$, $n = 1$ and $r = 0$. Hence

$$(extend^{1,1}_{\mathtt{iter()},alias})^{\mathbb{PATH}}(path) = \langle \{0.\mathtt{tail}\} \parallel \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset\rangle.$$

Assume now to add the method

```
void m() throws Exception {
  if (this.tail == null) throw new Exception();
}
```

to class `List`. Since, when that method returns normally, field `this.tail` must be non-`null`, it is possible to verify that its approximation, as computed by our analysis, is

$$path' = \langle \{0.\mathtt{tail}\} \parallel \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset\rangle.$$

Hence the approximation of a call to `m()`, in a program point with one local variable, one stack element (the receiver) and with the same aliasing information *alias* is such that $b = 0$, $n = 1$ and $r = 0$ and

$$\begin{aligned} &(extend^{1,1}_{\mathtt{m()},alias})^{\mathbb{PATH}}(path') \\ &\quad = \langle \{0.\mathtt{tail}, 0.\mathtt{tail.tail}\} \parallel \emptyset \parallel \mathbb{P}_1 \parallel \mathbb{P}_1 \parallel \emptyset \parallel \emptyset\rangle. \end{aligned}$$

The last semantical operator is $\cup^{\mathbb{PATH}}$.

Definition 24 The operator $\cup^{\mathbb{PATH}} : \mathbb{PATH}^2 \to \mathbb{PATH}$ is the least upper bound operator over $\mathbb{PATH}$. Namely, given $path_i = \langle NN_i \parallel NE_i \parallel EN_i \parallel EE_i \parallel L_i \parallel F_i\rangle \in \mathbb{PATH}$ for $i = 1, 2$, we define $path_1 \cup^{\mathbb{PATH}} path_2$ as

$$\langle NN_1 \cap NN_2 \parallel NE_1 \cap NE_2 \parallel EN_1 \cap EN_2 \parallel EE_1 \cap EE_2 \parallel L_1 \cup L_2 \parallel F_1 \cup F_2\rangle.$$

Proposition 8 (Correctness of the Abstract Operators) *The operators* $;^{\mathbb{PATH}}$, $extend^{\mathbb{PATH}}$ *and* $\cup^{\mathbb{PATH}}$ *are correct.*

Proof For $;^{\mathbb{PATH}}$, let $\delta \in \gamma(path_1); \gamma(path_2)$. We prove $\delta \in \gamma(path_1 ;^{\mathbb{PATH}} path_2)$ by proving that the six points of Definition 20 hold for δ. Let hence σ and $\delta(\sigma)$ be as in the hypotheses of Definition 20: $\sigma = \langle l \parallel s \parallel \mu\rangle$ and $\delta(\sigma) = \langle l' \parallel s' \parallel \mu'\rangle$ (possibly underlined). From the definition of δ, it must be the case that $\delta = \delta_1; \delta_2$ for suitable $\delta_1 \in \gamma(path_1)$ and $\delta_2 \in \gamma(path_2)$. Let $\delta_1(\sigma) = \langle l'' \parallel s'' \parallel \mu''\rangle$ (possibly underlined). Assume that $\sigma \in \Xi$ and $\delta(\sigma) \in \Xi$. Hence $\delta_1(\sigma) \in \Xi$ and $\delta_2(\delta_1(\sigma)) \in \Xi$, or $\delta_1(\sigma) \in \underline{\Xi}$ and $\delta_2(\delta_1(\sigma)) \in \Xi$:

- $\delta_1(\sigma) \in \Xi$ and $\delta_2(\delta_1(\sigma)) \in \Xi$: from Definition 20, $NN_1 \subseteq nnpaths(\delta_1(\sigma))$ and $NN_2 \subseteq nnpaths(\delta_2(\delta_1(\sigma)))$. Let $p \in NN_1 \bullet_{L_2,F_2} NN_2$. By Definition 21, either $p \in NN_2$ and so $\delta_2(\delta_1(\sigma))(p) \in \mathbb{L}$, or $p \in NN_1$, $\neg affected(p, L_2, F_2)$ and $\delta_1(\sigma)(p) \in \mathbb{L}$. In the latter case, since $local(p) < i_1 \le i_2$, $local(p) \notin L_2$ and no field in F_2 occurs in p, we conclude (Definition 20) that local variable $local(p)$ has given the same value in $\delta_1(\sigma)$ and $\delta_2(\delta_1(\sigma))$ and all fields in p have the same values in $\delta_1(\sigma)$ and $\delta_2(\delta_1(\sigma))$. It follows that $\delta_2(\delta_1(\sigma))(p) = \delta_1(\sigma)(p) \in \mathbb{L}$. In both cases, hence, we have $p \in nnpaths(\delta_2(\delta_1(\sigma)))$. By the genericity of p, we have $NN_1 \bullet_{L_2,F_2} NN_2 \subseteq nnpaths(\delta_2(\delta_1(\sigma)))$;
- $\delta_1(\sigma) \in \underline{\Xi}$ and $\delta_2(\delta_1(\sigma)) \in \Xi$: from Definition 20, $NE_1 \subseteq nnpaths(\delta_1(\sigma))$ and $EN_2 \subseteq nnpaths(\delta_2(\delta_1(\sigma)))$. By reasoning as in the case above, we conclude that $NE_1 \bullet_{L_2,F_2} EN_2 \subseteq nnpaths(\delta_2(\delta_1(\sigma)))$.

We conclude that in both cases we have $NN_1 \bullet_{L_2,F_2} NN_2 \cap NE_1 \bullet_{L_2,F_2} EN_2 \subseteq nnpaths(\delta_2(\delta_1(\sigma)))$, which satisfies point 1 of Definition 20. One can prove similarly points 2, 3 and 4 of the same definition. For point 5, assume that $l_k \ne l'_k$ for some $0 \le k < i_1$. Hence $l_k \ne l''_k$ or $l''_k \ne l'_k$ or both and, by Definition 20, $k \in L_1$ or $k \in L_2$ or both. In all cases, we have $k \in L_1 \cup L_2$. By the genericity of k, we conclude that $L_1 \cup L_2 \supseteq \{0 \le k < i_1 \mid l_k \ne l'_k\}$ and point 5 of Definition 20 holds for δ. Consider point 6 now. Assume that $\mu(\ell).f \ne \mu'(\ell).f$ for some $\ell \in dom(\mu)$, $\kappa.f : t \in \mathbb{F}$ with $t \in \mathbb{K}$. Since $dom(\mu) \subseteq dom(\mu'') \subseteq dom(\mu')$ (Definition 4), either $\mu(\ell).f \ne \mu''(\ell).f$ or $\mu''(\ell).f \ne \mu'(\ell)$ or both. Hence, by Definition 20, either $f \in F_1$ or $f \in F_2$ or both. In all cases, we have $f \in F_1 \cup F_2$. By the genericity of f, we conclude that $F_1 \cup F_2 \supseteq \{\kappa.f : t \mid t \in \mathbb{K},\ \ell \in dom(\mu)$ and $\mu(\ell).f \ne \mu'(\ell).f\}$ and point 6 of Definition 20 holds for δ.

For $extend^{\mathbb{PATH}}$, let $\delta \in \gamma(path)$. In the hypotheses of Definition 23, we prove that $(\delta' = extend^{i,j}_{M,alias}(\delta)) \in \gamma((extend^{i,j}_{M,alias})^{\mathbb{PATH}}(path))$. Let $\sigma \in \Sigma_{i,j}$ be such that $\delta'(\sigma)$ is defined. From Sect. 3, $\sigma' = \delta(\langle [v_0, \ldots, v_n] \parallel \varepsilon \parallel \mu\rangle) = \langle l' \parallel top \parallel \mu'\rangle$ (possibly underlined), $\sigma = \langle l \parallel v_n :: \cdots :: v_0 :: s \parallel \mu\rangle$ and

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

Hence δ' is never defined on input exceptional states and does not modify any local variable, and by Lemma 5, points 3, 4 and 5, we conclude that

$$\delta' \in \gamma(\langle \emptyset \parallel \emptyset \parallel \mathbb{P}_i \parallel \emptyset \parallel L \parallel F'\rangle), \tag{8}$$

$$\delta' \in \gamma(\langle \emptyset \parallel \emptyset \parallel \emptyset \parallel \mathbb{P}_i \parallel L \parallel F'\rangle), \tag{9}$$

$$\delta' \in \gamma(\langle \emptyset \parallel \emptyset \parallel \emptyset \parallel \emptyset \parallel \emptyset \parallel F'\rangle), \tag{10}$$

where $L = \{0, \ldots, i-1\}$ and $F' = \{\kappa.f : t \in \mathbb{F} \mid t \in \mathbb{K}\}$. Moreover, since $\sigma' = \delta(\langle [v_0, \ldots, v_n] \parallel \varepsilon \parallel \mu\rangle) = \langle l' \parallel top \parallel \mu'\rangle$ and $\delta \in \gamma(path)$, we conclude that $F \supseteq \{\kappa.f : t \mid t \in \mathbb{K},\ \ell \in dom(\mu)$ and $\mu(\ell).f \ne \mu'(\ell).f\}$ (Definition 20). Since the memory of $\delta'(\sigma)$ is μ' or $\mu[\ell := npe]$ with ℓ fresh, we have

$$\delta' \in \gamma(\langle \emptyset \parallel \emptyset \parallel \emptyset \parallel \emptyset \parallel L \parallel F\rangle). \tag{11}$$

Assume that $\sigma \in \Xi$ and $\delta'(\sigma) \in \Xi$. From the definition of δ' above, it can only be the case that $\delta'(\sigma) = \langle l \parallel top :: s \parallel \mu'\rangle$, $v_0 \in \mathbb{L}$ and $\sigma' \in \Xi$. From $\delta \in \gamma(path)$ and Definition 20 we have $NN \subseteq nnpaths(\delta)$. Let $p = y.f_1.\ldots.f_x \in NN$, $y = local(p)$, $y \notin L$ and s_{b+y} (i.e. v_y) be a definite alias of l_k. Since $y \notin L$, from $\delta \in \gamma(path)$ and Definition 20 we have $v_y = l'_y$, i.e. l'_y is a definite alias of l_k. Moreover, $\delta'(\sigma)$ and σ' have the same memory μ'. From Definition 17 it follows that $\delta'(\sigma)(k.f_1.\ldots.f_n) = \sigma'(p) \in \mathbb{L}$, i.e. $\delta'(\sigma)(p[l \to k]) \in \mathbb{L}$. This means that $p[l \to k] \in nnpaths(\delta')$. Since p is arbitrary, we have $NN_{b,L,alias} \subseteq nnpaths(\delta')$. Furthermore, we have seen that $v_0 \in \mathbb{L}$, i.e. $s_b \in \mathbb{L}$. Let hence $p \in \mathbb{P}_i$ be a definite alias of s_b in σ and let no $f \in F$ occur in p. Hence $\sigma(p) = s_b \in \mathbb{L}$. Since $\delta \in \gamma(path)$ and from the hypothesis about p and F, we know that $\mu(\ell).g = \mu'(\ell).g$ for every $\ell \in dom(\mu)$ and $g \notin F$. Hence $\sigma(p) = \delta'(\sigma)(p) \in \mathbb{L}$. Then $p \in nnpaths(\delta')$. We conclude that

$$\delta' \in \gamma\left(\left\langle NN_{b,L,alias} \cup \left\{ p \;\middle|\; \begin{array}{l} p \in \mathbb{P}_i,\ p \text{ is an alias of } s_b \\ \text{no } f \in F \text{ occurs in } p \end{array} \right\} \parallel \emptyset \parallel \emptyset \parallel \emptyset \parallel L \parallel F'\right\rangle\right). \tag{12}$$

Finally, from (8), (9), (10), (11) and (12) and Lemma 4 we have

$$\delta' \in \gamma\left(\left\langle NN_{b,L,alias} \cup \left\{ p \;\middle|\; \begin{array}{l} p \in \mathbb{P}_i,\ p \text{ is an alias of } s_b \\ \text{no } f \in F \text{ occurs in } p \end{array} \right\} \parallel \emptyset \parallel \mathbb{P}_i \parallel \mathbb{P}_i \parallel \emptyset \parallel F\right\rangle\right)$$
$$= \gamma((extend^{i,j}_{M,alias})^{\mathbb{PATH}}(path)).$$

For $\cup^{\mathbb{PATH}}$, in the hypotheses of Definition 24, we have

$$\gamma(\langle NN_1 \cap NN_2 \parallel NE_1 \cap NE_2 \parallel EN_1 \cap EN_2 \parallel EE_1 \cap EE_2 \parallel L_1 \cup L_2 \parallel F_1 \cup F_2 \rangle)$$

$$= \left\{ \delta \in \Delta_{i_1,j_1 \to i_2,j_2} \;\middle|\; \begin{array}{l} \text{for all } \sigma \in \Sigma_{i_1,j_1} \text{ s.t. } \delta(\sigma) \text{ is defined} \\ \text{letting } \sigma = \langle l \parallel s \parallel \mu \rangle \text{ and } \delta(\sigma) = \langle l' \parallel s' \parallel \mu' \rangle \\ \text{(both states are possibly underlined)} \\ 1.\ \sigma \in \Xi,\ \delta(\sigma) \in \Xi \Rightarrow NN_1 \cap NN_2 \subseteq \mathit{nnpaths}(\delta(\sigma)) \\ 2.\ \sigma \in \Xi,\ \delta(\sigma) \in \underline{\Xi} \Rightarrow NE_1 \cap NE_2 \subseteq \mathit{nnpaths}(\delta(\sigma)) \\ 3.\ \sigma \in \underline{\Xi},\ \delta(\sigma) \in \Xi \Rightarrow EN_1 \cap EN_2 \subseteq \mathit{nnpaths}(\delta(\sigma)) \\ 4.\ \sigma \in \underline{\Xi},\ \delta(\sigma) \in \underline{\Xi} \Rightarrow EE_1 \cap EE_2 \subseteq \mathit{nnpaths}(\delta(\sigma)) \\ 5.\ L_1 \cup L_2 \supseteq \{0 \le k < i_1 \mid l_k \ne l'_k\} \\ 6.\ F_1 \cup F_2 \supseteq \left\{ \kappa.f : t \;\middle|\; \begin{array}{l} t \in \mathbb{K},\ \ell \in \mathit{dom}(\mu) \\ \mu(\ell).f \ne \mu'(\ell).f \end{array} \right\} \end{array} \right\}$$

which includes

$$\left\{ \delta \in \Delta_{i_1,j_1 \to i_2,j_2} \;\middle|\; \begin{array}{l} \text{for all } \sigma \in \Sigma_{i_1,j_1} \text{ s.t. } \delta(\sigma) \text{ is defined} \\ \text{letting } \sigma = \langle l \parallel s \parallel \mu \rangle \text{ and } \delta(\sigma) = \langle l' \parallel s' \parallel \mu' \rangle \\ \text{(both states are possibly underlined)} \\ 1.\ \sigma \in \Xi,\ \delta(\sigma) \in \Xi \Rightarrow NN_1 \subseteq \mathit{nnpaths}(\delta(\sigma)) \\ 2.\ \sigma \in \Xi,\ \delta(\sigma) \in \underline{\Xi} \Rightarrow NE_1 \subseteq \mathit{nnpaths}(\delta(\sigma)) \\ 3.\ \sigma \in \underline{\Xi},\ \delta(\sigma) \in \Xi \Rightarrow EN_1 \subseteq \mathit{nnpaths}(\delta(\sigma)) \\ 4.\ \sigma \in \underline{\Xi},\ \delta(\sigma) \in \underline{\Xi} \Rightarrow EE_1 \subseteq \mathit{nnpaths}(\delta(\sigma)) \\ 5.\ L_1 \supseteq \{0 \le k < i_1 \mid l_k \ne l'_k\} \\ 6.\ F_1 \supseteq \left\{ \kappa.f : t \;\middle|\; \begin{array}{l} t \in \mathbb{K},\ \ell \in \mathit{dom}(\mu) \\ \mu(\ell).f \ne \mu'(\ell).f \end{array} \right\} \end{array} \right\} \cup$$

$$\left\{ \delta \in \Delta_{i_1,j_1 \to i_2,j_2} \;\middle|\; \begin{array}{l} \text{for all } \sigma \in \Sigma_{i_1,j_1} \text{ s.t. } \delta(\sigma) \text{ is defined} \\ \text{letting } \sigma = \langle l \parallel s \parallel \mu \rangle \text{ and } \delta(\sigma) = \langle l' \parallel s' \parallel \mu' \rangle \\ \text{(both states are possibly underlined)} \\ 1.\ \sigma \in \Xi,\ \delta(\sigma) \in \Xi \Rightarrow NN_2 \subseteq \mathit{nnpaths}(\delta(\sigma)) \\ 2.\ \sigma \in \Xi,\ \delta(\sigma) \in \underline{\Xi} \Rightarrow NE_2 \subseteq \mathit{nnpaths}(\delta(\sigma)) \\ 3.\ \sigma \in \underline{\Xi},\ \delta(\sigma) \in \Xi \Rightarrow EN_2 \subseteq \mathit{nnpaths}(\delta(\sigma)) \\ 4.\ \sigma \in \underline{\Xi},\ \delta(\sigma) \in \underline{\Xi} \Rightarrow EE_2 \subseteq \mathit{nnpaths}(\delta(\sigma)) \\ 5.\ L_2 \supseteq \{0 \le k < i_1 \mid l_k \ne l'_k\} \\ 6.\ F_2 \supseteq \left\{ \kappa.f : t \;\middle|\; \begin{array}{l} t \in \mathbb{K},\ \ell \in \mathit{dom}(\mu) \\ \mu(\ell).f \ne \mu'(\ell).f \end{array} \right\} \end{array} \right\}$$

that is $\gamma(\langle NN_1 \parallel NE_1 \parallel EN_1 \parallel EE_1 \parallel L_1 \parallel F_1 \rangle) \cup \gamma(\langle NN_2 \parallel NE_2 \parallel EN_2 \parallel EE_2 \parallel L_2 \parallel F_2 \rangle)$. □

The analysis of this section is not necessarily finite since the abstract domain of Definition 19 has infinite height (paths can be arbitrarily deep, as in Example 14). In order to keep the analysis finite by reaching the abstract fixpoint in a finite number of iterations, we fix a maximal depth k for the paths. Longer paths are not approximated, i.e. they are always considered to be potentially `null`. In our experiments we have used $k = 5$ and verified that smaller constants yield very precise results as well, since, in most cases, programmers do not test long paths.

Figure 10 shows how we exploit the analysis of this section. A definite aliasing analysis is performed first to compute a set of candidate fields and support the analysis over $\mathbb{PATH}$ through the fixpoint computation of Sect. 3. Then the program is analysed by an iterated static analysis over $\mathbb{NULL}$. During these iterations, the preliminary information on locally non-`null` fields computed by the analysis over $\mathbb{PATH}$ is used to improve the precision of the `getfield`'s. Namely, we can assume, at the time the oracle-based analysis is performed, those bytecodes decorated with some static information B about the *paths* that are definitely non-`null`. Since the analysis over $\mathbb{PATH}$ is correct (Propositions 7 and 8) we can assume that B holds there, or otherwise their semantics is undefined. This does not change the concrete semantics of the programs nor the correctness of the static analysis over $\mathbb{NULL}$, as we have already observed for

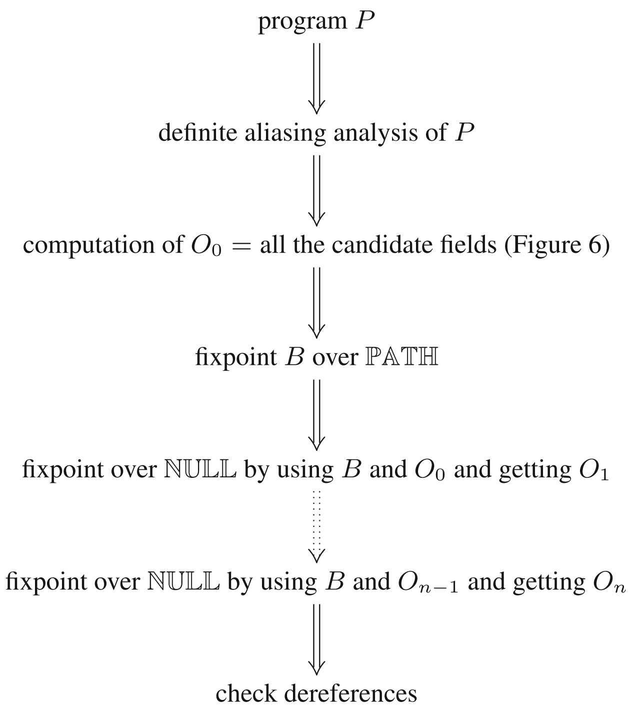

Fig. 10 The null-pointer analysis exploiting our supporting non-`null` paths analysis

$\mathbb{PATH}$ when we have decorated the bytecodes with aliasing information (Eq. 2). Then we improve the definition of the approximation of *getfield* from Sect. 5 by redefining:

$$(getfield\ \kappa.f:t)^{\mathbb{NULL}}_{O,alias,paths} = \begin{cases} U \wedge \neg\check{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{e}) \wedge (\neg\hat{e} \rightarrow \neg\hat{s}_{j-1}) \\ \quad \text{if } \kappa.f:t \in O \text{ or} \\ \quad (s_{j-1} \text{ is a definite alias of } p \text{ and } p.f \in paths) \\ U \wedge \neg\check{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{e}) \\ \quad \text{otherwise,} \end{cases} \tag{13}$$

where p can be a path or just a local variable number.

Example 21 Consider the second `getfield` bytecode from the top in Fig. 8. Assume that the aliasing analysis has been able to conclude that the top of the stack is a definite alias of local variable 0 there. Since our static analysis over $\mathbb{PATH}$ concludes that path `0.tail` holds a non-`null` value there (Example 18), we have `0.tail` $\in$ *paths* and hence

$$(getfield\ \kappa.f:t)^{\mathbb{NULL}}_{O,alias,paths} = U \wedge \neg\check{e}$$
$$\wedge(\check{s}_{j-1} \leftrightarrow \hat{e}) \wedge (\neg\hat{e} \rightarrow \neg\hat{s}_{j-1}).$$

That is, if no exception is thrown then a non-`null` value is loaded on top of the stack by that bytecode, which is enough to conclude that the subsequent recursive method call to `iter()` cannot throw a `NullPointerException`.

The result of the last iteration of the oracle-based null-pointer analysis with this redefinition is correct and is finally used to check for safe dereferences.

Proposition 9 *The redefinition above of the semantics of* `getfield` *leads to an oracle-based null-pointer analysis whose iterations converge to a correct analysis.*

Proof Let us define a non-standard semantics $(getfield\ \kappa.f:t)_{O,alias,paths}$ for `getfield` as

$$\lambda \underbrace{\langle l \,\|\, rec :: s \,\|\, \mu\rangle}_{\sigma} . \begin{cases} \langle l \,\|\, \mu(rec).f :: s \,\|\, \mu\rangle \\ \text{if } nnpaths(\sigma) \supseteq paths, \sigma \text{ satisfies } alias, \\ rec \neq \texttt{null} \text{ and } (\mu(rec).f \neq \texttt{null} \text{ or } \kappa.f:t \notin O) \\[1ex] \langle l \,\|\, \ell :: s \,\|\, \mu[\ell := o]\rangle \\ \text{if } nnpaths(\sigma) \supseteq paths, \sigma \text{ satisfies } alias, \\ rec \neq \texttt{null}, \mu(rec).f = \texttt{null} \text{ and } \kappa.f:t \in O \\[1ex] undefined \\ \text{if } nnpaths(\sigma) \not\supseteq paths \text{ or } \sigma \text{ does not satisfy } alias, \\[1ex] \underline{\langle l \,\|\, \ell \,\|\, \mu[\ell \mapsto npe]\rangle} \\ \text{otherwise,} \end{cases}$$

where $\ell \in \mathbb{L}$ is fresh and o is an object of class t with fields initialised to default values. This definition coincides with that used in Lemma 3 but requires that the static information *alias* and *paths* hold at the input state σ. We prove that $(getfield\ \kappa.f:t)^{\mathbb{NULL}}_{O,alias,paths}$ is correct w.r.t. $(getfield\ \kappa.f:t)_{O,alias,paths}$, i.e.

$$(getfield\ \kappa.f:t)_{O,alias,paths} \in \gamma((getfield\ \kappa.f:t)^{\mathbb{NULL}}_{O,alias,paths})$$

from which the correctness of the last iteration of the oracle-based null-pointer analysis follows as in Proposition 5. Note that this non-standard semantics is less defined than that of Lemma 3, which does not require any aliasing nor non-`null` paths information to hold. Hence the approximation (1) is correct for this redefinition also. Comparing (13) to (1), one observes that the only difference is that when s_{j-1} is a definite alias of p (according to *alias*) and $p.f \in paths$ then the formula $\neg\hat{e} \rightarrow \neg\hat{s}_{j-1}$ is added to the approximation. Assume hence that those conditions hold. We prove that $\delta = (getfield\ \kappa.f:t)_{O,alias,paths}$ is such that $\delta \in \gamma(\neg\hat{e} \rightarrow \neg\hat{s}_{j-1})$ where γ is the concretisation map of Definition 10. This will entail the thesis. Let $\sigma \in \Sigma_{i,j}$ be such that $\delta(\sigma)$ is defined. We have $\sigma = \langle l \,\|\, rec :: s \,\|\, \mu\rangle \in \Xi$. If $\delta(\sigma) \in \underline{\Xi}$ then $\hat{e} \in \widehat{nullness}(\delta(\sigma))$ and $\widecheck{nullness}(\sigma) \cup \widehat{nullness}(\delta(\sigma)) \models \neg\hat{e} \rightarrow \neg\hat{s}_{j-1}$. If $\delta(\sigma) \in \Xi$, instead, from the definition of δ we conclude that $rec \neq \texttt{null}$, i.e. $s_{j-1} \neq \texttt{null}$, that *alias* holds for σ and that $nnpaths(\sigma) \supseteq paths$. Since s_{j-1} is a definite alias of p and $p.f \in paths$, we conclude that

Fig. 11 Time in seconds, number of analysed methods, number *fs* of reference fields proved non-`null` and of `getfields`, `putfields` and `calls` proved safe. The last row reports the average difference of each column between the two analyses. Only `java.lang.*` and `java.util.*` library classes are included but their dereferences are not counted. These benchmarks come from http://sourceforge.net, but for `Kitten` and `Julia`, that are our own code

	Our analysis from Sections 4 and 5					
program	size	time	fs	get's	put's	calls
MyChatClient	189	0.58	7	100.00%	100.00%	82.79%
JLex	502	2.22	50	71.07%	71.76%	54.01%
CaffeineMark	723	2.35	3	100.00%	100.00%	100.00%
JavaCup	1029	8.87	45	47.78%	96.04%	93.92%
EveryonesJ’Editor	1151	7.60	289	99.57%	100.00%	86.00%
JavaCC	1618	17.42	161	92.14%	96.16%	76.78%
EBookME	1900	14.08	105	93.48%	100.00%	94.16%
JUnitCore	2030	7.02	75	99.54%	98.23%	92.63%
Pizza	2708	25.48	300	86.50%	93.50%	73.21%
Jess	3432	24.54	97	97.60%	99.58%	76.38%
Kitten	3925	25.76	300	82.11%	97.01%	88.67%
JEdit	4710	47.64	379	96.47%	97.90%	88.59%
Julia	5154	44.29	650	98.06%	98.33%	98.47%

	The analysis in [19]				
program	time	fs	get's	put's	calls
MyChatClient	-	-	-	-	-
JLex	3.52	44	71.69%	64.18%	48.32%
CaffeineMark	3.80	1	98.55%	100.00%	63.08%
JavaCup	5.52	31	47.44%	96.04%	86.17%
EveryonesJavaEditor	-	-	-	-	-
JavaCC	8.07	58	92.55%	95.80%	71.58%
EBookME	-	-	-	-	-
JUnitCore	6.10	65	99.04%	100.00%	70.01%
Pizza	10.00	86	79.77%	89.49%	75.19%
Jess	-	-	-	-	-
Kitten	-	-	-	-	-
JEdit	-	-	-	-	-
Julia	-	-	-	-	-
	-41.59%	*-55.05%*	*-1.61%*	*-1.84%*	*-13.50%*

$\sigma(p) = s_{j-1} \neq$ `null` (if p is a local variable number, then here by $\sigma(p)$ we mean the value of that local variable in σ) and $\sigma(p.f) = \mu(\sigma(p)).f \in \mathbb{L}$. Hence $\mu(rec).f = \mu(s_{j-1}).f = \mu(\sigma(p)).f \in \mathbb{L}$. By the definition of δ we conclude that $\delta(\sigma)$ has the non-`null` value $\mu(rec).f$ on top of the stack, so $\hat{s}_{j-1} \notin \widehat{nullness}(\delta(\sigma))$ and hence, also in this case, $\check{nullness}(\sigma) \cup \widehat{nullness}(\delta(\sigma)) \models \neg\hat{e} \rightarrow \neg\hat{s}_{j-1}$. From Definition 10 we have $\delta \in \gamma(\neg\hat{e} \rightarrow \neg\hat{s}_{j-1})$. □

7 Experiments

We have implemented our analyses in Java inside our JULIA generic analyser, which can be used at the address http://julia.scienze.univr.it. The experiments have been performed on a quad-core Intel Xeon 64 bits machine running at 2.66 GHz, with 4 gigabytes of RAM, Linux 2.6.27 and Sun jdk 1.6. For the first analysis of Sects. 4 and 5, we have coded Boolean formulas as binary decision diagrams [5] with the BUDDY library (http://sourceforge.net/projects/buddy). For the second analysis of Sect. 6, we have coded sets of paths by using bitmaps.

Figure 11 compares our analysis from Sects. 4 and 5 with the implementation NIT of [19] (we thank Laurent Hubert for his help with the use of NIT). This comparison is important since NIT is the only other null-pointer analysis of Java programs of high precision running without manual annotations and reporting statistics about its precision w.r.t. the number of safe dereferences. We have included library methods in the `java.lang.*` and `java.util.*` classes and have approximated the other methods with a *worst-case assumption*, i.e. by assuming them to return a possibly `null` value. That figure shows that our analysis scales to programs of more than 5000 methods and still works in reasonable time. It is possible to include all library methods in the

Fig. 12 Time and precision of the nullness analysis from Sect. 6. The captions have the same meaning as in Fig. 11. The last row reports the average increase of each column w.r.t. the same column in Fig. 11 for the analysis from Sects. 4 and 5

	Our analysis from Section 6				
program	time	fs	`get's`	`put's`	`calls`
`MyChatClient`	0.64	7	100.00%	100.00%	82.79%
`JLex`	4.89	50	90.09%	88.47%	58.82%
`CaffeineMark`	3.33	3	100.00%	100.00%	100.00%
`JavaCup`	11.74	45	48.22%	98.30%	94.17%
`EveryonesJ'Editor`	10.76	291	99.57%	100.00%	86.64%
`JavaCC`	27.32	162	94.48%	98.26%	77.66%
`EBookME`	20.49	107	93.48%	100.00%	96.46%
`JUnitCore`	18.46	77	99.54%	98.23%	93.38%
`Pizza`	56.97	302	88.10%	94.47%	75.04%
`Jess`	46.29	98	97.69%	100.00%	78.96%
`Kitten`	45.54	300	82.67%	97.45%	89.74%
`JEdit`	98.27	383	97.98%	98.84%	91.40%
`Julia`	85.73	650	98.36%	98.55%	98.73%
	+88.08%	*+0.56%*	*+2.22%*	*+1.92%*	*+1.64%*

analysis and get slightly more precise results for longer analysis times. The good performance over `CaffeineMark` is consequence of the fact that program mostly performs numerical calculations and is little object-oriented. Note that we only count dereferences inside the analysed application, not inside the included libraries. Our analysis, coded in Java, is about 41.59% slower than the natively compiled OCaml of NIT. The latter did not manage to analyse seven benchmarks for some error in the application extraction. W.r.t. precision, we observe that NIT uses more abstract values than the domain in [19] and assumes that, after a dereference, the receiver is non-`null`, which is not the case in [19]. Hence NIT yields more precise results than its theoretical definition. You can see from Fig. 11 that NIT finds 55.05% less non-`null` fields than our analysis. Also for the number of `getfields`, `putfields` and instance `calls` proved safe, JULIA is on the average more precise. This is particularly the case for the dereferences at method calls, where NIT is 13.50% less precise. Better precision means fewer false alarms to the user of the tool; moreover, subsequent static analyses will enjoy a simpler control-flow since it can be simplified by removing more useless nullness checks. These results show that our analysis is more precise than that in [19] for a small extra cost.

Figure 12 shows the results of performing the same analyses in Fig. 11 with the help of the preliminary analysis on locally non-`null` paths defined in Sect. 6, by following the picture in Fig. 10. You can see that the results are more precise than those in Fig. 11. The times required for the analyses are, however, larger than the corresponding times in Fig. 11. Nevertheless, the extra precision induced by the analysis in Sect. 6 can still be exploited without making the total cost of the analysis explode.

```
public class Test {
    // fields
    private @NonNull Test g;
    private @NonNull java.lang.Object f;

    // constructors
    public @Raw Test(@NonNull java.lang.Object l1);
    public @Raw Test(@Nullable Test l1);

    // methods
    private static @PolyNull java.lang.Object foo(@PolyNull Test l0);
    private void @Raw helper(@NonNull Test l1);
    public static void main(@NonNull java.lang.String[] l0);
}
```

Fig. 13 The nullness annotations built by our tool for the program in Fig. 1

Our analyses can be used to build automatically nullness annotations for the method and field signatures in the program. For instance, Fig. 13 shows the annotations built by the analysis of Sect. 4 for the program in Fig. 1. Since our tool works at bytecode level, the parameter names are not available and we use local variable numbers `l`i instead. If the program is compiled with debugging information, the argument names are available instead. The annotations in Fig. 1 follow the syntax specified in [24], which itself is an extension of the generic syntax for type annotations defined for Java [4]. Namely, `@NonNull` stands for a type that does not allow `null` among its values, while `@Nullable` allows `null`. The annotation `@Raw` stands for a type whose values, if non-`null`, are objects whose `@NonNull` fields are exceptionally allowed to hold `null`: this annotation is normally used for the constructors and for the helper functions called by the constructors to help building the objects, as `helper()` in Fig. 1. When we put `@Raw` before the name of the constructor or helper function (as in Fig. 13), `@Raw` refers to the receiver (local variable 0) of that constructor or method. The `@PolyNull` annotation allows a limited form of type polymorphism. For instance, in Fig. 13, the annotation for `foo()` can be seen as a shorthand for the two (syntactically not allowed) annotations:

```
private static @Nullable java.lang.Object
 foo(@Nullable Test l0);
private static @NonNull java.lang.Object
 foo(@NonNull Test l0);
```

In other words, this `@PolyNull` annotation guarantees that the return value of `foo()` is non-`null` if and only if its argument is non-`null`.

These nullness annotations are built by our tool in the following way:

– the magic-sets transformation (see end of Sect. 4) is instructed to provide information before each method call, so that information about the nullness of the parameters can be collected;
– the same magic-sets transformation is instructed to provide information before each `return` bytecode as well, so that information about the nullness of the return value can be collected;
– if, at the previous point, the return value of some method M has not been proved to be `@NonNull`, we use the less precise annotation `@PolyNull` for the return value of M and for some of its formal parameters $p_1, \ldots, p_n$ when the Boolean formula computed as denotation of the body of the method during the analysis in Sect. 4 entails the formula $(\neg\check{l}_{p_1} \wedge \cdots \wedge \neg\check{l}_{p_n}) \rightarrow \neg\hat{s}_0$. That is, when the analysis proves that the return value $\hat{s}_0$ is non-`null` whenever the local variables holding the formal parameters are non-`null`;
– the algorithm in Fig. 6 identifies the *helper functions* called by the constructors: we put `@Raw` before those helper functions and before any constructor;
– a field is decorated with `@NonNull` if and only if it belongs to the fixpoint oracle computed as in Sect. 5, otherwise it is decorated with `@Nullable`.

The correctness of the resulting annotations follows from the correctness of our static analyses and of the magic-sets transformation. Our tool can also be used to *check* already existing nullness annotations: although this possibility has not been implemented, it is enough to perform the static analysis of the program and then compare the results with the annotation provided.

Other tools are able to infer or verify nullness annotations. The higher precision of our analyses means that our tool generates more precise annotations and verifies more annotations than others. We have experimented with the tool DAIKON [12], which is able to infer nullness annotations by collecting and analysing execution traces of the program. It does not work for the program in Fig. 1 since that program throws a `NullPointerException` at run-time, which aborts the collection of the execution traces. Even by modifying the program so that it does not throw any exception anymore (i.e. by removing the `if (args.length > 0)` conditional), the tool is not able to infer the `@PolyNull` annotation for method `foo()` since `p` is potentially `null` at method call time and hence the tool assumes that it is still potentially `null` when the `return p` occurs. That is, the tool does not exploit the information provided by the guard of the conditional. Moreover, it does not infer that field `g` is non-`null`. It annotates instead field `f` as non-`null`, but there is no guarantee that the `@NonNull` annotations generated by DAIKON are correct: they are only *likely* true [12], so they must be subsequently checked through some type-checking engine, such as the Checker Framework [24]. It is currently under investigation if situations like this are frequent in practice or if the extra precision of our analyses is mostly important for reducing the number of false alarms when proving dereferences safe, rather than for inferring more precise nullness annotations.

8 Conclusions and ongoing work

The importance of this work is the formal development of two static analyses for null-pointer analysis of Java programs from a theoretically clean framework of denotational semantics for the Java bytecode. The results have been proved to be fast, scalable and more precise than those of other tools. The analyses can already be used to generate nullness annotations for the programs and to simplify the code by removing spurious nullness checks. In this context, our current effort is to be able to generate nullness annotations also for generic types, in the form of `List<@NonNull C>`.

It is well true, however, that the precision of the results in Fig. 12 can be improved. This is mostly important if the analysis has to be used to signal to the user where a possible `NullPointerException` can be generated at run-time. In that case, only a very small set of warnings can be accepted, since a large number of false alarms will likely induce the user to give up using the tool. From this point of view, the precision of our analyses must still be improved. To that purpose, a simple idea is that of including *all* library methods in the analysis, instead of only those of some extensively used libraries such as the `java.lang.*` and `java.util.*` classes. We have experimentally verified that this provides slight gains in precision but often makes the analysis of large programs run out of memory on our machine. There are however other ways of increasing the precision of the results. We have already implemented the following improvements:

Information about field assignments. After a statement such as `o.f=exp`, field `f` of `o` is definitely non-`null` if `exp` is definitely non-`null`. This is not captured by the analysis in Sect. 6, which instead in that situation assumes that all paths where `f` occurs can potentially hold `null` (see the approximation of `putfield` in Fig. 9 and the effects of a field update under composition in Definition 21). In order to improve this situation, we need to know if `exp` can only yield a non-`null` value. To that purpose we can use our same null-pointer analysis of Sect. 4, but there is a problem here, since the latter is performed *after* the preliminary local non-nullness analysis for the paths (Fig. 10). We have hence introduced a new oracle, the set of `putfield`'s which definitely assign a non-`null` value to their field, initially including all `putfield`'s, and have performed the analysis of Sect. 6 with the help of that oracle. We compute an iteration of the analysis from Sect. 6, an iteration of the analysis from Sect. 4, we refine the oracle and then we go back again to the analysis from Sect. 6, until both oracles (that for the `putfield`'s and that for the non-`null` fields of Sect. 5) stabilise.

More flexible local non-nullness information for the fields. A drawback of the abstract domain from Sect. 6 is that it only considers paths of field dereferences from local variables which are aliases of stack elements. This is important for a fast analysis but reduces its precision when, for instance, variables are copied, such as in `a=b`, by using temporary variables on the stack. We have hence built a better analysis for locally non-`null` fields, taking into account how values flow across local variables, stack elements, fields and arrays, and which is kept efficient by using a constraint-based approach instead of a denotational approach. This means, however, that it is not context-sensitive. Hence we keep open the possibility of coupling this new analysis with that in Sect. 6.

Better information at field updates. The analysis in Sect. 6 assumes, conservatively, that a path is potentially `null` as soon as a field in the path gets assigned (Definition 21). This is relatively pessimistic, since a field update might keep a path non-`null` when it is performed on an object which does not belong to the path. To collect such information, we perform a preliminary possible *sharing* analysis [26] among data structure and a preliminary *creation point* analysis [16] of reference values. Those analyses have non-trivial costs but happen to increase significantly the precision of the local non-nullness information about the fields.

The optimisations above increase the precision of the analysis by around 3% w.r.t. the results in Fig. 12. The current analyser, exploiting the latest results, is available at http://julia.scienze.univr.it.

There are other possibilities for optimisation though, not yet implemented:

Still better information at field updates. Despite our efforts, there are still situations when a field update erases too many locally non-`null` paths. We believe that an important, currently missing information is when a field is definitely non-initialised at a given program point. In that case, it must hold `null` and an update to that field cannot make any path non-`null` since a `null` field cannot be used in a non-`null` path (Definition 17). How to compute such information is however unclear to us.

Information on full *arrays.* Our analysis currently assumes that the values read from an array are always potentially non-`null` (we have not described the bytecodes dealing with arrays in this paper). This is correct but too conservative. A first improvement will be to spot *full* arrays, i.e. arrays that, by definition, have all their elements set to non-`null` values: this is the case of the `args` parameter passed to the `main` method of a program and of the $i-1$ lowest dimensions of a i-dimensional array created by the `multianewarray` bytecode. A second improvement could underapproximate the elements of an array which are already initialised to non-`null` values. This seems a complex task to us, since it will have to interact with approximations of numerical variables.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles Techniques and Tools. Addison-Wesley, Reading (1986)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Dealing with numeric fields in termination analysis of Java-like languages. In: Huisman, M. (ed.) Proceedings of the 10th Workshop on Formal Techniques for Java-like Programs (FTfJP'08), July 2008. http://clip.dia.fi.upm.es/~samir/home/viewpost.php?post=Publications
3. Armstrong, T., Marriott, J., Schachte, P., Søndergaard, H.: Two classes of Boolean functions for dependency analysis. Sci. Comput. Program. **31**(1), 3–45 (1998)
4. Bloch, J.: Jsr 175: A Metadata Facility for the Java Programming Language (2004). http://jcp.org/en/jsr/detail?id=175
5. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
6. Chalin, P., James, P.R.: Non-null references by default in Java: alleviating the nullity annotation burden. In: Ernst, E. (ed.) Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07). Lecture Notes in Computer Science, Berlin, Germany, July–August 2007, vol. 4609, pp. 227–247. Springer, Berlin (2007)
7. Cielecki, M., Fulara, J., Jakubczyk, K., Jancewicz, Ł.: Propagation of JML non-null annotations in Java programs. In: Gitzel, R., Aleksy, M., Schader, M. (eds.) Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java (PPPJ'06), Mannheim, Germany, August–September 2006, pp. 135–140. ACM, New York (2006)
8. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the 2nd International Symposium on Programming, Paris, France, April 1976, pp. 106–130. Dunod, Paris (1976)
9. 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 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77), pp. 238–252 (1977)

10. Ekman T., Hedin, G.: The jastadd extensible Java compiler. In: Ernst, E. (ed.) Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07). Lecture Notes in Computer Science, Berlin, Germany, July–August 2007, vol. 4609, pp. 1–18. Springer, Berlin (2007)
11. Engelen, A.F.M.: Nullness Analysis of Java Source Code. PhD thesis, University of Nijmegen, Department of Computer Science (2006)
12. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
13. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: Crocker, R., Steel, G.L. Jr. (eds.) Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03), Anaheim, CA, USA, October 2003, pp. 302–312. ACM, New York (2003)
14. Fähndrich, M., Xia, S.: Establishing object invariants with delayed types. In: Gabriel, R.P., Bacon, D.F., Videira Lopes, C., Steele, G.L. Jr. (eds.) Proceedings of the 2007 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07), Montreal, Quebec, Canada, October 2007, pp. 337–350. ACM, New York (2007)
15. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) Proceedings of the 2001 International Symposium of Formal Methods Europe (FME'01). Lecture Notes in Computer Science, Berlin, Germany, March 2001, vol. 2021, pp. 500–517. Springer, Berlin (2001)
16. Hill, P.M., Spoto, F.: Deriving escape analysis by abstract interpretation. High. Order Symb. Comput. **19**(4), 415–463 (2006)
17. Hovemeyer, D., Pugh, W.: Finding more null pointer bugs, but not too many. In: Das, M., Grossman, D. (eds.) Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'07), San Diego, CA, USA, June 2007, pp. 9–14. ACM, New York (2007)
18. Hovemeyer, D., Spacco, J., Pugh, W.: Evaluating and tuning a static analysis to find null pointer bugs. In: Ernst, M., Jensen, T.P. (eds.) Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE'05), Lisbon, Portugal, September 2005, pp. 13–19. ACM, New York (2005)
19. Hubert, L., Jensen, T., Pichardie, D.: Semantic foundations and inference of non-null annotations. In: Barthe, G., de Boer, F.S. (eds.) Proceedings of the 10th International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08). Lecture Notes in Computer Science, vol. 5051, pp. 132–149. Springer, Berlin (2008)
20. Leino, K.R.M., Saxe, J.B., Stata, R.: ESC/Java User's Manual. Compaq Systems Research Center, technical note 2000-002 edition, October (2000)
21. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley, Reading (1999)
22. Male, C., Pearce, D.J., Potanin, A., Dymnikov, C.: Java bytecode verification for `@NonNull` types. In: Hendren, L. (ed.) Proceedings of the 17th Int. Conference on Compiler Construction (CC'2008). Lecture Notes in Computer Science, Budapest, Hungary, March–April 2008, vol. 4959, pp. 229–244. Springer, Budapest (2008)
23. Palsberg, J., Schwartzbach, M.I.: Object-oriented type inference. In: Proceedings of OOPSLA'91. ACM SIGPLAN Notices, vol. 26(11), pp. 146–161. ACM, New York (1991)
24. Papi, M.M., Ali, M., Correa, T.L., Perkins, J.H., Ernst, M.D.: Practical pluggable types for Java. In: Ryder, B.G., Zeller, A. (eds.) Proceedings of the ACM/SIGSOFT 2008 International Symposium on Software Testing and Analysis (ISSTA'08), Seattle, WA, USA, July 2008, pp. 201–212. ACM, New York (2008)
25. Payet, É, Spoto, F.: Magic-sets transformation for the analysis of Java bytecode. In: Nielson, H.R., Filé, G. (eds.) Proceedings of the 14th International Static Analysis Symposium (SAS'07). Lecture Notes in Computer Science, Kongens Lyngby, Denmark, August 2007, vol. 4634, pp. 452–467. Springer, Berlin (2007)
26. Secci S., Spoto, F.: Pair-sharing analysis of object-oriented programs. In: Hankin, C. (ed.) Proceedings of Static Analysis Symposium (SAS). Lecture Notes in Computer Science, London, UK, September 2005, vol. 3672, pp. 320–335. Springer, Berlin (2005)
27. Spoto, F.: Nullness Analysis in Boolean form. In: Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM'08), Cape Town, South Africa, November 2008, pp. 21–30. IEEE Press, New York (2008)

Author Biography

Fausto Spoto is Associate Professor in Computer Science at the University of Verona, Italy. He took a PhD in computer science in Pisa in 2000. His main interests are related to the static analysis of computer programs, for automated verification and optimisation of software.