Inference of field initialization

Technical Report UW-CSE-10-02-01
University of Washington Department of Computer Science & Engineering
February 6, 2010

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

Michael D. Ernst
Computer Science & Engineering
University of Washington, USA
mernst@uw.edu

Abstract

A *raw* object is partially initialized, with only some of its fields set to legal values. A raw object may violate its object invariants, such as that a given field is non-`null`. Programs often need to manipulate partially-initialized objects, but they must do so with care. Furthermore, analyses must be aware of rawness. For instance, software verification cannot depend on object invariants for raw objects.

We present a static analysis that infers a safe over-approximation of the program variables, fields, or array elements that, at run-time, might hold non-fully initialized objects. Our formalization is flow-sensitive and considers the exception flow in the analyzed programs. We have proved the analysis to be sound.

We have also implemented our analysis, in a tool called JULIA that computes both nullness and rawness information. We have evaluated JULIA on over 50K lines of code. We have compared its output to manually-written nullness and rawness information, and to an independently-written type-checking tool that checks nullness and rawness. JULIA's output is accurate and, we believe, useful both to programmers and to static analyses.

1. Introduction

Object-oriented programming languages, such as Java, allow the definition of *instance variables* or *fields*, that hold the state of the objects. Those fields are normally (but not necessarily) initialized inside the *constructors* defined in their class. Before that initialization, in Java, they hold the default value for their static type. For fields of reference type, the default value is `null`.

Many type-checking systems and static verification tools aim to prove that some *object invariant*, *inv*, holds for the objects of a given class. This means that *inv* must be true immediately after the object construction and must remain true every time a (public) method is entered or exited. A typical object invariant states that some field *f* is non-`null`. This means that all constructors must initialize *f* to a non-`null` value and that *f* will never be reset to `null` later. This is an important piece of information, since it guarantees that no dereference of *f* will ever throw a `NullPointerException`. This conclusion is *not* true *inside* the constructors. Before *f* is initialized, it still holds the default value `null`. Our work focuses on initialization properties, rather than all ways any invariant can be violated, such as by setting a value to a non-`null` value that is inconsistent with its formal specification.

Since, in general, object invariants do not hold for partially-initialized ("raw" [9]) objects, it is important to identify those *sites* — program fields, parameters, return values, or array elements — that might hold a raw object at run-time. These sites are not just the `this` variable inside the constructors: that variable can be passed to methods and stored in fields or arrays. Moreover, a raw variable loses its rawness as soon as all its fields have been initialized (or some relevant subset of its fields, see Section 5.4). Hence `this` might be non-raw inside a constructor, from a given program point onwards.

```
... // many non-null fields defined here

public OptionsDialog(Frame owner) {
  super(owner, "Options");
  // initializes a non-null field
  this.owner = owner;
  // initializes the remaining non-null fields
  setup();
  // 'this' is non-raw here
  pack();
  ...
}
```

Figure 1: A snippet of code from the JFlex program.

Figure 1 contains a snippet of code from class `OptionsDialog` of the JFlex scanner generator, one of the applications that we analyze in Section 6. Method `setup()` is called by the constructor of class `OptionsDialog` to help it build the object: this helper method initializes most of the fields of the object.

The JULIA tool performs our rawness analysis and infers that the receiver of `setup()` is raw, and all other references are non-raw, such as the receiver of `pack()`. Furthermore, JULIA infers that many fields are non-`null`. Without an inference of object initialization, a tool would either be unsound, or would be forced to conclude that all fields are possibly `null`. JULIA warns about any possibly-erroneous dereference; these include possibly-`null` fields wherever used in the program (unless the field can be proved to be non-`null` at that point), and non-`null` fields when used before the object is adequately initialized.

In principle, it would be correct to annotate all receivers, fields, parameters, and return values as `@Raw`: that would be a sound over-approximation of the set of raw sites. However, this would not be useful, and would hobble follow-on analyses [20] and human understanding. We aim for a precise analysis that infers as few rawness annotations as possible, only where needed.

We have extended the JULIA tool [16], which already contains a highly precise nullness analysis [22], to implement our rawness analysis. JULIA infers precise nullness and rawness annotations for non-trivial software, automatically, in a few minutes. We have also integrated JULIA with toolsets for working with annotations and pluggable type systems [2, 4]. This enables us to compare our rawness analysis with pluggable type-checking for a nullness and rawness type system. The type system is weaker, but compositional and with easier-to-understand results. JULIA's results are correct by construction (the analyses in the JULIA tool are formally correct), but the type-checker still issues some warnings while type-checking them, because of the different perspective of the two tools. JULIA is based on flow and context-sensitive static analyses and abstract interpretation, while the checker framework is based on type-checking, augmented by flow-sensitivity and some other enhancements [20]. Nevertheless, we have achieved a good degree of integration of the two tools and Section 6 discusses the remaining differences and evaluates the quality of the automatic annotation *w.r.t.* that of a previous manual annotation that has been verified by type-checking.

Our work is relevant beyond the scope of nullness analysis. Our rawness analysis is not coupled to nullness analysis and is computed independently. It can be used in other contexts, whenever it is necessary to prove that some fields are definitely initialized at some program points. Moreover, the constraint-based analysis of Java bytecode in Section 5, and the structure of the correctness proof in Appendix A, can be used for other properties than rawness. Namely, JULIA embeds a constraint-based analysis, following the same scheme, that determines when arrays and collection classes are *full*, that is, contain non-`null` elements only (see Section 3).

This paper makes the following contributions:

- It defines and proves correct a rawness analysis for Java bytecode that is completely independent from any other analysis (such as nullness).
- It provides definitions and proofs that explicitly consider the exceptional flows in the program.
- We have implemented our rawness analysis, demonstrating its practicality. We have also integrated it with other nullness and rawness tools, permitting comparisons and increasing confidence in both toolsets.
- Experiments with our implementation demonstrate its precision. The experiments also yield insight into the algorithm's strengths and weaknesses and the properties of real code.

To the best of our knowledge, the first two points above are novel. For example, NIT's rawness inference is coupled to nullness and its theory does not consider exceptional flows [15]. Being independent from nullness analysis simplifies the formalization and the proof of correctness, and permits applicability to other problem domains. The last point above has never been investigated before. For example, NIT does not dump the rawness information that it computes, so its evaluation and its use for type-checking is impossible. See Section 2 for more comparisons with related work.

The rest of the paper is organized as follows. Section 2 overviews the most closely-related work. Section 3 reviews the nullness analysis implemented by the JULIA tool. Section 4 presents an operational semantics for Java bytecode, which is the concrete semantics of our abstract interpretation for rawness. Section 5 defines this constraint-based abstract interpretation. Section 6 reports on our experiments: we compared JULIA's output to manual annotations and to an independently-implemented type-checker. The proof of correctness is found in Appendix A. The JULIA tool is available for use through its web interface: `http://julia.scienze.univr.it`.

2. Related work

Fähndrich and Leino [9] check object initialization using a type qualifier (called "raw") that indicates how many fields are initialized. On exiting a constructor, the type is non-raw for that class and all of its superclasses, but still raw for any subtypes whose constructor has not yet been exited. In a raw type, all fields declared in that type are assumed to be possibly `null`, and the type checker enforces that these not-yet-initialized fields are not used. The nullness and rawness type-checker that we used in our experiments is a re-implementation of this algorithm, with enhancements. Delayed types [10] specify *when* fields can be assumed to have been initialized; by contrast, rawness specifies *where* fields can be assumed to have been initialized.

The most closely related work is NIT and JASTADD. NIT [15] is a nullness inference tool that, in parallel, also infers rawness. Unlike our work, their formalization and proofs do not consider exceptional flows. The NIT tool does not output any of the rawness annotations that it infers. JASTADD [6] infers nullness along with a coarser variant of rawness, in which each object is fully initialized or fully uninitialized, without reference to how many constructors have been exited. The rawness analysis of JASTADD is informally presented and is not proved correct. Rawness increased the percentage of references that JASTADD reports as safe from 69% to 71%, for three packages in the JDK. In our experiments, JULIA reported over 98% of references to be safe. Like JULIA, NIT and JASTADD can produce an annotation file that can be inserted into Java source code or class files [2]. JACK [19] requires annotated method signatures, then does a flow-sensitive, alias-sensitive flow analysis to determine nullness and rawness types for local variables. It operates on bytecode. Like JASTADD, it infers the coarse version of rawness. Unlike JULIA, none of these tools' rawness analysis seems to have been evaluated and compared to manually-identified correct annotations.

Several other nullness inference tools for Java exist, but unlike JULIA they do not infer rawness annotations. DAIKON [8] runs the program and soundly outputs `@Nullable` for variables that were ever observed to be `null`. It would be unsound to report `@NonNull` for values that were never observed to be `null`. DAIKON can produce an annotation file. HOUDINI [11] inserts `@NonNull` at every possible location, then runs a static checker. Whenever the static checker issues a warning, HOUDINI removes the relevant annotation. HOUDINI iterates this process until it reaches a fixed point. HOUDINI is neither sound nor complete. INAPA [7] is based on similar principles to HOUDINI. FINDBUGS [14, 13] finds null pointer dereferences by using an imprecise analysis that produces many false warnings, but then prioritizing and filtering aggressively so that few false warnings are reported to a user. It attempts to infer programmer intent (*w.r.t.* nullness) based on code patterns. It is neither sound nor complete.

Our rawness analysis is a constraint-based abstract interpretation [5] of a concrete operational semantics for Java bytecode, presented in Section 4. Other operational semantics for Java bytecode are available, such as that of Freund and Mitchell [12]. Here we follow our formalization in [21], which is also the basis of the JULIA analyser [16], and hence we match theory with implementation. Our formalization is indebted to [17], where Java and Java bytecode are mathematically formalized and the compilation of Java into bytecode and its type-safeness are machine-proved. Our formalization of the state of the JVM (Definition 2 in Section 4.2) is similar to theirs, as well as our formalization of heap and objects.

3. Nullness Inference

The definition of our rawness analysis does not use any previous nullness analysis. Nevertheless, we use it here after a nullness analysis is performed to project its results (sets of initialized fields) over those fields that are deemed non-`null` by the nullness analysis, in order to infer rawness annotations that are useful for nullness type-checking (see Subsection 5.4). The exact nullness analysis which is used is not important here. We briefly describe the one implemented in the JULIA analysis tool.

In order to achieve a very high level of precision for nullness analysis, JULIA uses a mix of different techniques: static analyses based on denotational abstract interpretation and constraint-based abstract interpretation. Both come in different flavors, for inferring properties of the local variables, fields, arrays of references, and collections.

The kernel of the nullness analysis of JULIA is a denotational, bottom-up abstract interpretation of the bytecode, which builds logical formulas whose models over-approximate the nullness behaviors of the variables in scope in a piece of code. This analysis has been proved correct [25]. Since the number of variables, at a given program point, is necessarily finite, the Boolean formulas are finite and relatively small, so they can be efficiently represented by *binary decision diagrams* [3]. The drawback of this approach is that it infers nothing about the nullness of the fields of the objects. Its precision is consequently unsatisfactory.

We now list three improvements, each of which increases precision but increases the computational cost. The overall system achieves high precision (see our results, presented in Section 6). A separate paper [22] presents all these techniques in detail and compares them *w.r.t.* precision and cost of the resulting nullness analyses.

(1) An *optimistic* approach [25] considers a field f, initialised by all constructors, to hold a non-`null` value (*globally non-`null`*) unless a counter-example is found, that is, an assignment of a possibly `null` value to f. The resulting precision is slightly better than that of NIT [15], although it is in general slower.

(2) *Local non-nullness* [23] recognizes fields that are definitely non-`null` at specific program points, because they have just been assigned a non-`null` value or have been checked for non-nullness. Local non-nullness achieves a form of flow-sensitivity. The analysis is implemented as a denotational bottom-up abstract interpretation of the code.

(3) A constraint-based static analysis uses the same technique that we describe in Section 5, to identify *full* arrays and collection objects (hashsets, hashmaps, linked lists, etc.), whose elements are all non-`null` at some program points.

The output of a nullness analysis indicates, for every variable and every program point (where the variable is in scope), whether the variable may be `null` or is definitely non-`null`. This output can be used by a lint-like bug detection tool that informs the user of places where the `null` value might be dereferenced. (Most of the warnings will be false alarms, but a few might be actual bugs.) The tool output can also be inserted in the program for documentation and debugging, or can be exported to another tool to aid in further analysis. JULIA can produce `null`-dereference warnings directly, and can output nullness annotations to program source or other tools by using the Annotation File Utilities [2].

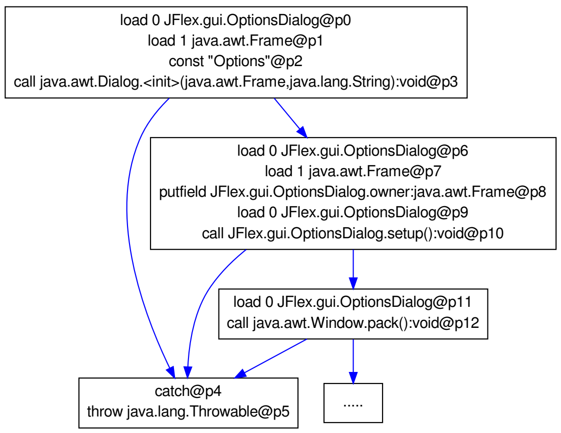
4. Operational Semantics

We describe here an operational semantics of the Java bytecode, that we abstract into our rawness analysis in Section 5 and prove correct in Appendix A.

4.1 Syntax

For simplicity of presentation, our formalism assumes `int` to be the only primitive type and *classes* to be the only reference types; we only allow *instance* fields and methods. Our implementation handles full sequential Java bytecode and all Java types. In particular, multithreading is not handled and the analysis of multithreaded applications might yield incorrect results, because JULIA assumes that immediately after a field is checked, it still has the same value.

We assume the Java bytecode is preprocessed into a control flow graph. This same representation is used in [21, 25, 24]; a similar representation is also chosen in [1], although, there, Prolog clauses encode the graph, while we work directly on the graph itself. A



```
load 0 JFlex.gui.OptionsDialog@p0
load 1 java.awt.Frame@p1
const "Options"@p2
call java.awt.Dialog.<init>(java.awt.Frame,java.lang.String):void@p3

load 0 JFlex.gui.OptionsDialog@p6
load 1 java.awt.Frame@p7
putfield JFlex.gui.OptionsDialog.owner:java.awt.Frame@p8
load 0 JFlex.gui.OptionsDialog@p9
call JFlex.gui.OptionsDialog.setup():void@p10

load 0 JFlex.gui.OptionsDialog@p11
call java.awt.Window.pack():void@p12

catch@p4
throw java.lang.Throwable@p5

.....
```

Figure 2: The blocks of code for the constructor in Figure 1.

control flow graph is a directed graph of *basic blocks*. All jumps are from the end of one basic block, to the beginning of another basic block. We graphically write

$$\boxed{\begin{array}{c}\texttt{ins@}p\\ \textit{rest}\end{array}}\begin{array}{l}\rightarrow b_1\\ \cdots\\ \rightarrow b_m\end{array}$$

for a block of code starting with a bytecode instruction `ins` at program point p, possibly followed by more bytecodes *rest* and linked to m subsequent blocks $b_1, \ldots, b_m$. For instance, `load 1 java.awt.Frame@p1` stands for an instance of the bytecode `load 1 java.awt.Frame` (which loads on the stack an object of class `java.awt.Frame`) occurring at program point `p1`. In most cases, the program point p is irrelevant, so we write just `ins`. Bytecodes have explicit, inferred types.

Exception handlers start with a `catch` bytecode. A conditional bytecode, a virtual method call, or the selection of an exception handler, on the basis of the run-time type of the receiver or exception, is translated into a block linked to many subsequent blocks. Each subsequent block starts with a *filtering* bytecode, such as `exception_is[_not]` in the case of exceptional handlers, that specifies when that continuation is taken. They are not needed in Figure 1 since a *default handler* is used there: *any* kind of exception is caught and thrown back to the caller.

4.2 Semantics

Our operational semantics keeps a *state*, providing values for the variables of the program. An *activation stack* of states is used to model the method call mechanism, exactly as in an actual implementation of the JVM.

Definition 1. (Classes) The set of *classes* $\mathbb{K}$ in program P is partially ordered *w.r.t.* $\leq$, which expresses the subclass relationship. A *type* is an element of $\mathbb{T} = \mathbb{K} \cup \{\texttt{int}\}$. A class $\kappa \in \mathbb{K}$ has *instance fields* $\kappa.f : t$ (field f of type $t \in \mathbb{T}$ defined in class κ), where κ and t are often omitted, and *instance methods* $\kappa.m(\vec{\tau}) : t$ (method m with arguments of type $\vec{\tau} \subseteq \mathbb{T}$, returning a value of type $t \in \mathbb{T} \cup \{\texttt{void}\}$, defined in class κ), where κ, $\vec{\tau}$, and t are often omitted. Constructors are seen as methods named `init` and returning $\langle\texttt{void}\rangle$. □

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 an infinite 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*, or *heap*, which binds locations to *objects*. The empty stack is written ε. An object o belongs to class $o.\kappa \in \mathbb{K}$ (is an *instance* of $o.\kappa$) and maps identifiers (the fields f of class $o.\kappa$ and of its superclasses) into $o.f$, which can be a value or `uninit`. 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. If v is a value or `uninit`, we say that *v has type t* in a state $\langle l \,\|\, s \,\|\, \mu \rangle$ if $v \in \mathbb{Z} \cup \{\texttt{uninit}\}$ and $t = \texttt{int}$, or $v \in \{\texttt{null}, \texttt{uninit}\}$ and $t \in \mathbb{K}$, or $v \in \mathbb{L}$, $t \in \mathbb{K}$ and $\mu(v).\kappa \le t$. $\square$

Compared to [21], Definition 2 allows fields to hold `uninit`. As will be clear from the formal semantics, this value is a special case of `null` or 0 that allows us to distinguish a field holding `null` or 0 because it has not been initialized yet, from a field already initialized, possibly to `null` or 0.

Example 1. A possible state at the beginning of the constructor in Figure 2 is $\sigma = \langle [\ell, \ell'] \,\|\, \varepsilon \,\|\, \mu \rangle$, where $\mu(\ell)(\texttt{owner}) = \texttt{uninit}$. Location ℓ contains the *receiver* $\mu(\ell)$ of the constructor, *i.e.*, `this`, whose fields are not initialized at the beginning of the constructor. Location ℓ' contains an object of class `java.awt.Frame`, the explicit argument of the constructor. $\square$

The JVM supports exceptions. Hence we distinguish *normal* states Ξ arising during the normal execution of a piece of code, from *exceptional* states $\underline{\Xi}$ arising *just after* a bytecode that throws an exception. States in $\underline{\Xi}$ have always a stack of height 1 containing a location (bound to the thrown exception object). We write them underlined in order to distinguish them from the normal states.

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

When we denote a state by σ, we do not specify if it is normal or exceptional. If we want to stress that we deal with a normal or with an exceptional state, then we write $\langle l \,\|\, s \,\|\, \mu \rangle$ in the first case and $\underline{\langle l \,\|\, s \,\|\, \mu \rangle}$ in the second.

Example 2. A state σ at the beginning of the block in Figure 2 containing `catch@p4` might be an exceptional state arising when the call `setup()` aborts because of an `OutOfMemoryError` (the code of that method contains many `new` statements). In that case, we would have $\sigma = \underline{\langle [\ell, \ell'] \,\|\, \ell'' \,\|\, \mu' \rangle}$, where ℓ and ℓ' are as in Example 1, $\mu'(\ell)(\texttt{owner}) = \ell''' \in \mathbb{L}$ (field `owner` of `this` has been already initialized at that point), $\mu(\ell'').\kappa = \texttt{OutOfMemoryError}$ and $\mu(\ell'')$ has no `uninit` fields. $\square$

The semantics of a bytecode `ins`@p is a partial map $ins : \Sigma_{i_1,j_1} \to \Sigma_{i_2,j_2}$ from an *initial* to a *final* state. The indices i_1, j_1, i_2, j_2 depend on p. The number and type of local variables and stack elements at each p are statically known [18]. In the following we silently assume that the bytecodes are run in a program point with i local variables and j stack elements and that the semantics of the bytecodes is undefined for input states of wrong sizes or types. These assumptions are required by [18] and must hold for legal Java bytecode.

4.3 Basic instructions

Bytecode `const` v pushes $v \in \mathbb{Z} \cup \mathbb{L} \cup \{\texttt{null}\}$ on the stack. When $v \in \mathbb{L}$, location v must be already allocated in the memory and hold an object of a very restricted set of classes, with all fields already initialized [18]. Formally, the semantics of this bytecode is

$$const\ v = \lambda \langle l \,\|\, s \,\|\, \mu \rangle . \begin{cases} \langle l \,\|\, v :: s \,\|\, \mu \rangle & \text{if } v \notin \mathbb{L} \text{ or} \\ & (\mu(v) \text{ is defined and has} \\ & \text{no } \texttt{uninit} \text{ field}) \\ undefined & \text{otherwise.} \end{cases}$$

The λ-notation defines a partial map, because of the *undefined* case. Namely, this bytecode is undefined when it tries to push a location which is not already in memory or has an uninitialized field. Since, above, $\langle l \,\|\, s \,\|\, \mu \rangle$ (where s might be ε) is not underlined, the map is also undefined on exceptional states, *i.e.*, the bytecode is executed when the JVM is not in an exceptional state. This is the case for *all* bytecodes but `catch`, which starts the exceptional handlers from an exceptional state.

Bytecode `dup` t duplicates the top of the stack, of type t:

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

Bytecode `load` $i\ t$ pushes on the stack the value of local variable number i, which must exist and have type t:

$$load\ i\ t = \lambda \langle l \,\|\, s \,\|\, \mu \rangle . \langle l \,\|\, l[i] :: s \,\|\, \mu \rangle .$$

Conversely, bytecode `store` $i\ t$ pops the top of the stack of type t and writes it in local variable i:

$$store\ i\ t = \lambda \langle l \,\|\, top :: s \,\|\, \mu \rangle . \langle l[i \mapsto top] \,\|\, s \,\|\, \mu \rangle .$$

If l contains less than $i+1$ variables, the resulting set of local variables gets expanded. The semantics of a conditional bytecode is undefined when its condition is false. For instance, `if_ne` t checks if the top of the stack, which must have type t, is not 0 when $t = \texttt{int}$ or is not `null` otherwise:

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

The *undefined* case corresponds to the fact that the JVM does not continue the execution of the code if the condition is false. Note that, in our formalization, conditional bytecodes are used in complementary pairs (for instance, `if_ne` and `if_eq`), at the beginning of the two branches of a condition, so that only one of them is defined for each given state.

4.4 Object-manipulating instructions

Some bytecodes create or access objects in memory. Bytecode `new` κ pushes on the stack a reference to a new object o of class κ, with reference fields initialized to `uninit`, that is, $o(\kappa'.f) = \texttt{uninit}$ for every field $\kappa'.f : t$ with $t \in \mathbb{K}$ and $\kappa \le \kappa'$. Its semantics, *new* κ, is

$$\lambda \langle l \,\|\, s \,\|\, \mu \rangle . \begin{cases} \langle l \,\|\, \ell :: s \,\|\, \mu[\ell \mapsto o] \rangle & \text{if there is enough memory,} \\ \underline{\langle l \,\|\, \ell \,\|\, \mu[\ell \mapsto oome] \rangle} & \text{otherwise} \end{cases}$$

with $\ell \in \mathbb{L}$ fresh and *oome* new instance of `OutOfMemoryError`. This is the first example of a bytecode that can throw an exception. Note that the initial value of the fields is fixed to `uninit` rather than to `null` or 0, as it would be in a standard semantics [21].

The semantics *getfield* $\kappa.f : t$ of bytecode `getfield` $\kappa.f : t$ reads the field $\kappa.f : t$ of a receiver object *rec* popped from the stack, of type κ. It *interprets* `uninit` as `null` or 0 before pushing it on the stack, since the value `uninit` is not allowed on the stack (Definition 2). Hence *getfield* $\kappa.f : t$ is defined as

$$\lambda \langle l \,\|\, rec :: s \,\|\, \mu \rangle . \begin{cases} \langle l \,\|\, \mu(rec).f :: s \,\|\, \mu \rangle & \text{if } rec \neq \texttt{null}, \mu(rec).f \neq \texttt{uninit} \\ \langle l \,\|\, \texttt{null} :: s \,\|\, \mu \rangle & \text{if } rec \neq \texttt{null}, \mu(rec).f = \texttt{uninit} \\ & \quad t \in \mathbb{K} \\ \langle l \,\|\, 0 :: s \,\|\, \mu \rangle & \text{if } rec \neq \texttt{null}, \mu(rec).f = \texttt{uninit} \\ & \quad t = \texttt{int} \\ \underline{\langle l \,\|\, \ell \,\|\, \mu[\ell \mapsto npe] \rangle} & \text{otherwise} \end{cases}$$

with $\ell \in \mathbb{L}$ fresh and *npe* a new instance of `NullPointerException`. The bytecode `putfield` $\kappa.f : t$ writes the top of the stack, of type t, inside field $\kappa.f : t$ of the object pointed to by a value *rec* below the top of the stack, of type κ (ℓ and *npe* are as before). Its semantics *putfield* $\kappa.f : t$ is

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

Note that this bytecode might only remove `uninit` from the approximation of field f, since the value *top* on the stack is not allowed to be `uninit` (Definition 2).

4.5 Exception-handling instructions

Bytecode `throw` κ throws, explicitly, the object pointed by the top of the stack, of type $\kappa \leq$`Throwable` (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. It takes an exceptional state and transforms it into a normal state, subsequently used by the handler:

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

where $top \in \mathbb{L}$ has type `Throwable`. Note that `catch` is undefined on all normal states. After `catch`, bytecode `exception_is` K selects the appropriate exception handler on the basis of the run-time class of *top*. Namely, it filters those states whose top of the stack is an instance of a class in $K \subseteq \mathbb{K}$. Its semantics *exception_is* K is

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

Bytecode `exception_is_not` K is just a shortcut for the bytecode `exception_is` H, where H is the set of exception classes that are not an instance of some class in K.

4.6 Method calls and return

When a caller transfers the control to a callee $\kappa.m(\vec{\tau}) : t$, the JVM performs an operation *makescope* $\kappa.m(\vec{\tau}) : t$ which copies the topmost stack elements into the corresponding local variables and clears the stack.

Definition 4. Let $\kappa.m(\vec{\tau}) : t$ be a method or constructor and π be the number of stack elements needed to hold its actual parameters, including the implicit parameter `this`. We define $(makescope\ \kappa.m(\vec{\tau}) : t) : \Sigma \to \Sigma$ as

$$\lambda\langle l \,\|\, v_{\pi-1} :: \cdots :: v_1 :: rec :: s \,\|\, \mu\rangle.\langle [rec, v_1, \ldots, v_{\pi-1}] \,\|\, \varepsilon \,\|\, \mu\rangle$$

provided $rec \neq \texttt{null}$ and the look-up of $m(\vec{\tau}) : t$ from the class $\mu(rec).\kappa$ leads to $\kappa.m(\vec{\tau}) : t$. We let it be undefined otherwise. □

This formalizes the fact that the ith local variable of the callee is a copy of the element located $(\pi - 1) - i$ positions down the top of the stack of the caller.

Bytecode `return` t terminates a method and clears its operand stack. If $t \neq \texttt{void}$, the return value is the only element left on the final stack:

$$return\ \texttt{void} = \lambda\langle l \,\|\, s \,\|\, \mu\rangle.\langle l \,\|\, \varepsilon \,\|\, \mu\rangle$$
$$return\ t = \lambda\langle l \,\|\, top :: s \,\|\, \mu\rangle.\langle l \,\|\, top \,\|\, \mu\rangle, \text{ where } t \neq \texttt{void}.$$

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

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

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

$$\frac{}{\langle \boxed{} \,\|\, \langle l \,\|\, top \,\|\, \mu\rangle\rangle :: \langle b \,\|\, \langle l' \,\|\, s' \,\|\, \mu'\rangle\rangle :: a \Rightarrow \langle b \,\|\, \langle l' \,\|\, top :: s' \,\|\, \mu\rangle\rangle :: a} \quad (4)$$

$$\frac{}{\langle \boxed{} \,\|\, \underline{\langle l \,\|\, e \,\|\, \mu\rangle}\rangle :: \langle b \,\|\, \langle l' \,\|\, s' \,\|\, \mu'\rangle\rangle :: a \Rightarrow \langle b \,\|\, \underline{\langle l' \,\|\, e \,\|\, \mu\rangle}\rangle :: a} \quad (5)$$

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

Figure 3: The transition rules of our operational semantics for Java bytecode (Section 4.7).

4.7 The transition rules

We can now define the operational semantics of our language.

Definition 5. A *configuration* is a pair $\langle b \,\|\, \sigma\rangle$ of a block b and a state σ. It represents the fact that the JVM is going to execute b in state σ. An *activation stack* is a stack $c_1 :: c_2 :: \cdots :: c_n$ of configurations, where c_1 is the topmost, *current* or *active* configuration. □

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

Definition 6. The (small step) operational semantics of a Java bytecode program P is a relation $a' \Rightarrow_P a''$ (P is usually omitted) providing the immediate successor activation stack a'' of an activation stack a'. It is defined by the rules in Figure 3. □

Rule 1 executes an instruction `ins`, different from `call`, by using its semantics *ins*. The JVM then moves forward to run the rest of the instructions. Rule 2 calls a method on a non-`null` receiver. It looks up the correct implementation $\kappa_i.m(\vec{\tau}) : t$ of the method, by using the look-up procedures of the language, and finds the block b where that implementation starts. It then builds its initial state σ', by using *makescope*, and creates a new current configuration containing b and σ'. It pops the actual arguments from the old current configuration and the `call` from the instructions still to be executed at return time. Since a method call can actually call many implementations, depending on the run-time class of the receiver, this rule is apparently non-deterministic. However, only one thread of execution will continue, since we assume that the method look-up rules of the language are deterministic (as in Java bytecode). Control returns to the caller by rule 4, which rehabilitates the configuration of the caller but forces the memory to be that at the end of the execution of the callee. The return value of the callee is pushed on the stack of the caller. This rule is executed if the state reached at the end of the caller is a normal state. If it is an exceptional state, rule 5 is executed instead, which propagates the exception back to the caller. If a method call occurs on a `null` receiver,

$$\langle\boxed{\begin{array}{c}\texttt{load 0 OD@p0}\\ \texttt{load 1 java.awt.Frame@p1}\\ \texttt{const "Options"@p2}\\ \texttt{call java.awt.Dialog.}\langle\texttt{init}\rangle\ldots\texttt{@p3}\end{array}}\begin{array}{l}\rightarrow b_4\\ \rightarrow b_6\end{array}\ \|\langle[\ell,\ell']\,\|\,\varepsilon\,\|\,\mu\rangle\rangle::a$$

$$\overset{(1)}{\Rightarrow}\langle\boxed{\begin{array}{c}\texttt{load 1 java.awt.Frame@p1}\\ \texttt{const "Options"@p2}\\ \texttt{call java.awt.Dialog.}\langle\texttt{init}\rangle\ldots\texttt{@p3}\end{array}}\begin{array}{l}\rightarrow b_4\\ \rightarrow b_6\end{array}\ \|\langle[\ell,\ell']\,\|\,\ell\,\|\,\mu\rangle\rangle::a$$

$$\overset{(1)}{\Rightarrow}\langle\boxed{\begin{array}{c}\texttt{const "Options"@p2}\\ \texttt{call java.awt.Dialog.}\langle\texttt{init}\rangle\ldots\texttt{@p3}\end{array}}\begin{array}{l}\rightarrow b_4\\ \rightarrow b_6\end{array}\ \|\langle[\ell,\ell']\,\|\,\ell'::\ell\,\|\,\mu\rangle\rangle::a$$

$$\overset{(1)}{\Rightarrow}\langle\boxed{\texttt{call java.awt.Dialog.}\langle\texttt{init}\rangle\ldots\texttt{@p3}}\begin{array}{l}\rightarrow b_4\\ \rightarrow b_6\end{array}\ \|\langle[\ell,\ell']\,\|\,\ell''::\ell'::\ell\,\|\,\mu\rangle\rangle::a$$

$$\overset{(2)}{\Rightarrow}\langle \mathit{first}\,\|\langle[\ell,\ell',\ell'']\,\|\,\varepsilon\,\|\,\mu\rangle\rangle::\langle\boxed{}\begin{array}{l}\rightarrow b_4\\ \rightarrow b_6\end{array}\ \|\langle[\ell,\ell']\,\|\,\varepsilon\,\|\,\mu\rangle\rangle::a$$

$$\overset{(\ldots)}{\Rightarrow}\langle\boxed{}\,\|\langle[\ell,\ell',\ell'']\,\|\,\varepsilon\,\|\,\mu'\rangle\rangle::\langle\boxed{}\begin{array}{l}\rightarrow b_4\\ \rightarrow b_6\end{array}\ \|\langle[\ell,\ell']\,\|\,\varepsilon\,\|\,\mu\rangle\rangle::a$$

$$\overset{(4)}{\Rightarrow}\langle\boxed{}\begin{array}{l}\rightarrow b_4\\ \rightarrow b_6\end{array}\ \|\langle[\ell,\ell']\,\|\,\varepsilon\,\|\,\mu'\rangle\rangle::a$$

$$\overset{(6)}{\Rightarrow}\langle\boxed{\begin{array}{c}\texttt{load 0 OD@p6}\\ \texttt{load 1 java.awt.Frame@p7}\\ \texttt{putfield OD.owner}\ldots\texttt{@p8}\\ \texttt{load 0 OD@p9}\\ \texttt{call OD.setup() : void@p10}\end{array}}\begin{array}{l}\rightarrow b_4\\ \rightarrow b_{11}\end{array}\ \|\langle[\ell,\ell']\,\|\,\varepsilon\,\|\,\mu'\rangle\rangle::a$$

$$\overset{(1)}{\Rightarrow}\langle\boxed{\begin{array}{c}\texttt{load 1 java.awt.Frame@p7}\\ \texttt{putfield OD.owner}\ldots\texttt{@p8}\\ \texttt{load 0 OD@p9}\\ \texttt{call OD.setup() : void@p10}\end{array}}\begin{array}{l}\rightarrow b_4\\ \rightarrow b_{11}\end{array}\ \|\langle[\ell,\ell']\,\|\,\ell\,\|\,\mu'\rangle\rangle::a$$

$$\overset{(1)}{\Rightarrow}\langle\boxed{\begin{array}{c}\texttt{putfield OD.owner}\ldots\texttt{@p8}\\ \texttt{load 0 OD@p9}\\ \texttt{call OD.setup() : void@p10}\end{array}}\begin{array}{l}\rightarrow b_4\\ \rightarrow b_{11}\end{array}\ \|\langle[\ell,\ell']\,\|\,\ell'::\ell\,\|\,\mu'\rangle\rangle::a$$

$$\overset{(1)}{\Rightarrow}\langle\boxed{\begin{array}{c}\texttt{load 0 OD@p9}\\ \texttt{call OD.setup() : void@p10}\end{array}}\begin{array}{l}\rightarrow b_4\\ \rightarrow b_{11}\end{array}\ \|\langle[\ell,\ell']\,\|\,\varepsilon\,\|\,\underbrace{\mu'[\mu(\ell).\texttt{owner}\mapsto\ell']}_{\mu''}\rangle\rangle::a$$

$$\overset{(1)}{\Rightarrow}\langle\boxed{\texttt{call OD.setup() : void@p10}}\begin{array}{l}\rightarrow b_4\\ \rightarrow b_{11}\end{array}\ \|\langle[\ell,\ell']\,\|\,\ell\,\|\,\mu''\rangle\rangle::a$$

Figure 4: A partial execution according to the semantics of Figure 3. b_x is the block in Figure 2 starting at px. `OD` stands for `JFlex.gui.OptionsDialog`. Location ℓ'' points to the string `"Options"` from the constant pool. *first* is the first block of the constructor of `java.awt.Dialog`. $\overset{(\ldots)}{\Rightarrow}$ is a complete execution of the latter and μ' is the memory at its end.

rule 3 creates a new state whose stack contains only a reference to a `NullPointerException`. No actual call happens in this case. Rule 6 applies when all instructions inside a block have been executed; it runs one of its immediate successors, if any. This rule is normally deterministic, since if a block of our formalization of the Java bytecode has two or more immediate successors then they start with mutually exclusive conditional instructions and only one thread of control is actually followed.

From now on, when we use the notation $\Rightarrow$, we often specify the rule in Figure 3 which is used at each derivation step; for instance, we write $\overset{(1)}{\Rightarrow}$ for a derivation step through rule (1).

Example 3. Consider the state $\langle[\ell,\ell']\,\|\,\varepsilon\,\|\,\mu\rangle$ from Example 1 at `p0` in Figure 2. The operational semantics can proceed from `p0` as in Figure 4. $\square$

5. Rawness Analysis

JULIA collects rawness information with a constraint-based *rawness analysis* which is performed at the end and independently from the nullness analysis. The rawness analysis builds a constraint: a graph whose nodes contain sets of fields that have not been initialized yet. There is a node for each local variable at a given program point and for each method parameter and return value. There is also a node for each field and another for the values stored into arrays; differently from local variables, fields and arrays have flow-insensitive approximations. Assignments and parameter passing are modeled as directed arcs $a \rightarrow b$ in the graph. They represent the inclusion of the content of a in the content of b.

When an object is created by a `new` κ bytecode, all fields of κ and of its superclasses are used as an approximation for the newly created object, left on top of the stack, since its has no initialized field at that point. Then arcs are built to link subsequent program points. For instance, the arc l_0@`p0`$\rightarrow$ l_0@`p1` is built for the program in Figure 1. But the arc l_0@`p0`$\rightarrow$ s_0@`p1` is also built, since local variable 0 at `p0` is loaded on the stack as its only element s_0 at `p1`. Arcs are built for all possible flows of control induced by loops, conditionals and exceptions. For instance, in Figure 1, both arcs l_0@`p3`$\rightarrow$ l_0@`p4` and l_0@`p3`$\rightarrow$ l_0@`p6` are built. Assignments to fields give rise to arrows from the rightvalue to the node for the field. For instance, the assignment at `p8` induces the arc: s_1@`p8`$\rightarrow$`owner`. In order to model the fact that field `owner` is now initialized, a *filtering* arc is built also, for each definite alias of the receiver. For instance, the same assignment at `p8` introduces the arc l_0@`p8`$\overset{\texttt{owner}}{\rightarrow}$ l_0@`p9` *instead* of l_0@`p8`$\rightarrow$ l_0@`p9`. The former arc states that all fields in l_0@`p8` are included in l_0@`p9`, but `owner`. The definite aliasing information needed here is the same used, for instance, in our nullness analysis (see [25]), so we can recycle it.

When all arcs have been built for the whole program, the constraint is solved through a least fixpoint calculation. The approximation of each variable is an over-approximation of the set of its fields that might not have been initialized yet. For instance, in Figure 1, the approximation computed for l_0@`p0` is the set $S = \{\texttt{owner},\ldots\}$ of all fields of class `OptionsDialog` and of its superclasses, and that for s_0@`p10` is $S\setminus\{\texttt{owner}\}$, since the non-`null` field `owner` has been already initialized at `p10`. The approximation computed for s_0@`p12` is $\emptyset$, since all fields of class `OptionsDialog` and of its superclasses have been already initialized at `p12`.

5.1 The abstraction map for rawness

In order to formalize our rawness analysis and prove its correctness, we define now the abstraction map from states to rawness.

Definition 7. (Rawness Abstraction) Let $\sigma=\langle[v_0\ldots v_{i-1}]\,\|\,w_{j-1}::\cdots::w_0\,\|\,\mu\rangle$ be a state (possibly underlined) with i local variables and j stack elements. Its *rawness abstraction* $\alpha(\sigma)$ maps the symbols $\{l_0\ldots l_{i-1}, s_0\ldots s_{j-1}, f_1,\ldots,f_n\}$, where $f_1,\ldots,f_n$ are all the fields in P, into sets of *uninitialized fields*, that is, fields that have not been initialized yet for each local variable, stack element or field:

$$\alpha(\sigma)(l_k)=\begin{cases}\emptyset & \text{if } v_k\in\mathbb{Z}\cup\{\texttt{null}\}\\ \{f\mid\mu(v_k).f=\texttt{uninit}\} & \text{if } v_k\in\mathbb{L}\end{cases}$$

$$\alpha(\sigma)(s_k)=\begin{cases}\emptyset & \text{if } w_k\in\mathbb{Z}\cup\{\texttt{null}\}\\ \{f\mid\mu(w_k).f=\texttt{uninit}\} & \text{if } w_k\in\mathbb{L}\end{cases}$$

$$\alpha(\sigma)(f_k)=\{f\mid\text{there exists }\ell\in\mathbb{L}\text{ s.t. }\mu(\mu(\ell).f_k).f=\texttt{uninit}\}.$$

$\square$

By Definition 7, the rawness of a stack element or local variable is the set of fields still bound to `uninit` in the object they hold. The rawness of a field f_k, instead, includes a field f if there is an object $\mu(\ell)$ at ℓ with field f_k bound to a location $\ell'=\mu(\ell).f_k$ that holds an object $\mu(\ell')$ whose field f is still `uninit`. Note that we silently assume that $\mu(\mu(\ell).f_k).f$ is well defined, *i.e.*, that all its components are defined. Instance fields are *flattened* by this abstraction, *i.e.*, they are treated as static fields and we cannot distinguish fields with the same name but in different objects. This abstraction is necessary to get a finite static analysis, since the number of objects in memory is potentially unbound.

Example 4. Consider the state $\sigma = \langle [\ell, \ell'] \,\|\, \varepsilon \,\|\, \mu \rangle$ from Example 1. Its abstraction is such that $\mathtt{owner} \in \alpha(\sigma)(l_0)$. □

5.2 The abstract constraint

A program P induces a constraint, which is a graph whose nodes contain a set of uninitialized fields. There are many kinds of nodes:

- $l_k @ p$ stands for the kth local variable ($k \geq 0$) at program point p
- $s_k @ p$ stands for the kth stack element ($k \geq 0$) at program point p
- $f @ ew$ stands for field f, at any program point
- $return @ m$ stands for the return value of method m
- $exception @ m$ stands for any exception thrown by method m
- $l_k @ \, end\ of\ m$ stands for the kth local variable ($k \geq 0$) at the end of every normal execution of method m
- $\{\kappa_1.f_1, \ldots, \kappa_n.f_n\}$ stands for a node containing those uninitialized fields

Arcs are directed: $n_1 \rightarrow n_2$ states that all the fields in n_1 are also in n_2. The *filtering* arc $n_1 \xrightarrow{f} n_2$ states that all the fields in n_1 *except* f are also in n_2. Arcs are built for each pair of subsequent bytecode instructions:

Definition 8. Let $\mathtt{ins}_p @ p$ and $\mathtt{ins}_q @ q$ be two bytecodes. Let i_p and j_p be the number of local variables and stack elements at the beginning of the execution of $\mathtt{ins}_p$, respectively. Let $U_{i_p,j_p} = \{l_k @ p \rightarrow l_k @ q \mid 0 \leq k < i_p\} \cup \{s_k @ p \rightarrow s_k @ q \mid 0 \leq k < j_p\}$. We define the constraints $con(\mathtt{ins}_p @ p, \mathtt{ins}_q @ q)$ as follows. If $\mathtt{ins}_q$ is not a `catch` then:

$$
\begin{aligned}
con(\mathtt{const}\ v, \mathtt{ins}_q) &= con(\mathtt{catch}, \mathtt{ins}_q) \\
&= con(\mathtt{exception\_is[\_not]}\ K, \mathtt{ins}_q) = U_{i_p,j_p} \\
con(\mathtt{dup}\ t, \mathtt{ins}_q) &= U_{i_p,j_p} \cup \{s_{j_p-1} @ p \rightarrow s_{j_p} @ q\} \\
con(\mathtt{load}\ x\ t, \mathtt{ins}_q) &= U_{i_p,j_p} \cup \{l_x @ p \rightarrow s_{j_p} @ q\} \\
con(\mathtt{store}\ x\ t, \mathtt{ins}_q) &= \{l_k @ p \rightarrow l_k @ q \mid 0 \leq k < i_p, k \neq x\} \\
&\quad \cup \{s_k @ p \rightarrow s_k @ q \mid 0 \leq k < j_p - 1\} \cup \{s_{j_p-1} @ p \rightarrow l_x @ q\} \\
con(\mathtt{if\_ne}\ t, \mathtt{ins}_q) &= U_{i_p,j_p-2} \\
con(\mathtt{new}\ \kappa, \mathtt{ins}_q) &= U_{i_p,j_p} \cup \{\{\kappa'.f : t \mid t \in \mathbb{K} \text{ and } \kappa \leq \kappa'\} \rightarrow s_{j_p} @ q\} \\
con(\mathtt{getfield}\ f, \mathtt{ins}_q) &= U_{i_p,j_p-1} \cup \{f @ ew \rightarrow s_{j_p-1} @ q\} \\
con(\mathtt{putfield}\ f, \mathtt{ins}_q) &= \{l_k @ p \rightarrow l_k @ q \mid 0 \leq k < i_p, (l_k, s_{j_p-2}) \notin alias_p\} \\
&\quad \cup \{s_k @ p \rightarrow s_k @ q \mid 0 \leq k < j_p - 2, (s_k, s_{j_p-2}) \notin alias_p\} \\
&\quad \cup \{l_k @ p \xrightarrow{f} l_k @ q \mid 0 \leq k < i_p, (l_k, s_{j_p-2}) \in alias_p\} \\
&\quad \cup \{s_k @ p \xrightarrow{f} s_k @ q \mid 0 \leq k < j_p - 2, (s_k, s_{j_p-2}) \in alias_p\} \\
&\quad \cup \{s_{j_p-1} @ p \rightarrow f @ ew\}
\end{aligned}
$$

$$
\begin{aligned}
con(\mathtt{call}\ m_1 \ldots m_n, \mathtt{ins}_q) &= \cup_{k=1}^{n} \cup_{u=0}^{\pi-1} \{s_{j_p-u-1} @ p \rightarrow l_{\pi-u-1} @ first(m_k)\} \\
&\quad \cup \{return @ m_k \rightarrow s_{j_p-\pi} @ q \mid 1 \leq k \leq n\} \\
&\quad \cup \left\{ l_k @ p \rightarrow l_k @ q \;\middle|\; \begin{array}{l} 1 \leq k < i_p \text{ and if } (l_k, s_{j_p-u-1}) \in alias_p \\ \text{for some } 0 \leq u < \pi \text{ then at least an } m_h \\ \text{contains a } \mathtt{store}\ l_{\pi-u-1}\ t \end{array} \right\} \\
&\quad \cup \left\{ l_{\pi-u-1} @ \, end\ of\ m_w \rightarrow l_k @ q \;\middle|\; \begin{array}{l} 1 \leq k < i_p,\ 1 \leq w \leq n, \\ (l_k, s_{j_p-u-1}) \in alias_p \\ \text{for some } 0 \leq u < \pi \text{ and no } m_h \\ \text{contains a } \mathtt{store}\ l_{\pi-u-1}\ t \end{array} \right\} \\
&\quad \cup \left\{ s_k @ p \rightarrow s_k @ q \;\middle|\; \begin{array}{l} 1 \leq k < j_p - \pi \text{ and if } (s_k, s_{j_p-u-1}) \in alias_p \\ \text{for some } 0 \leq u < \pi \text{ then at least an } m_h \\ \text{contains a } \mathtt{store}\ l_{\pi-u-1}\ t \end{array} \right\} \\
&\quad \cup \left\{ l_{\pi-u-1} @ \, end\ of\ m_w \rightarrow s_k @ q \;\middle|\; \begin{array}{l} 1 \leq k < j_p - \pi,\ 1 \leq w \leq n, \\ (s_k, s_{j_p-u-1}) \in alias_p \\ \text{for some } 0 \leq u < \pi \text{ and no } m_h \\ \text{contains a } \mathtt{store}\ l_{\pi-u-1}\ t \end{array} \right\}.
\end{aligned}
$$

If, instead, $\mathtt{ins}_q$ is a `catch`, we define

$$
\begin{aligned}
con(\mathtt{throw}\ \kappa, \mathtt{catch}) &= \{l_k @ p \rightarrow l_k @ q \mid 0 \leq k < i_p\} \\
&\quad \cup \{s_{j_p-1} @ p \rightarrow s_0 @ q\} \\
con(\mathtt{call}\ m_1 \ldots m_n, \mathtt{catch}) &= \cup_{k=1}^{n} \cup_{u=0}^{\pi-1} \{s_{j_p-u-1} @ p \rightarrow l_{\pi-u-1} @ first(m_k)\} \\
&\quad \cup \{exception @ m_k \rightarrow s_0 @ q \mid 1 \leq k \leq n\} \cup \{l_k @ p \rightarrow l_k @ q \mid 1 \leq k < i_p\} \\
con(\mathtt{ins}_p, \mathtt{catch}) &= U_{i_p,0}, \text{ where } \mathtt{ins}_p \text{ is not a } \mathtt{throw} \text{ nor a } \mathtt{call}.
\end{aligned}
$$

Moreover, if p is a program point inside method m, we define the constraints

$$
\begin{aligned}
final\_con(\mathtt{throw}\ \kappa) &= \{s_{j_p-1} @ p \rightarrow exception @ m\} \\
final\_con(\mathtt{return\ void}) &= \{l_k @ p \rightarrow l_k @ \, end\ of\ m \mid 0 \leq k < i_p\} \\
final\_con(\mathtt{return}\ t) &= \{l_k @ p \rightarrow l_k @ \, end\ of\ m \mid 0 \leq k < i_p\} \\
&\quad \cup \{s_{j_p-1} @ p \rightarrow return @ m\}
\end{aligned}
$$

where $t \neq \mathtt{void}$. □

Definition 8 has two cases. The first is when $\mathtt{ins}_q$ is not a `catch`. This means that the normal output state of $\mathtt{ins}_p$ flows to the beginning of $\mathtt{ins}_q$. If $\mathtt{ins}_p$ is a `const`, the sets of uninitialized fields for local variables and stack elements do not change. This is also the case for `catch`, `exception_is` and `exception_is_not`. Hence we build a constraint U_{i_p,j_p} that states this fact. For `dup`, we also build an arc saying that the set of uninitialized fields for the new top of the stack ($s_{j_p} @ q$) contains all the uninitialized fields of the old top of the stack ($s_{j_p-1} @ p$). Similar constraints are built for `load` and `store`: the latter keeps the approximation of the local variables unchanged but for l_x that gets the approximation of the old top of the stack. If $\mathtt{ins}_p$ is an `if_ne`, two elements are removed from the stack. If it is a $\mathtt{new}\ \kappa$, the new top of the stack contains all fields defined in κ or in a superclass κ' of κ, since they are not yet initialized. Bytecodes `getfield` and `putfield` create arcs from and to the node $f @ ew$ for the accessed field f. The latter bytecode modifies the rawness of every definite alias of its receiver s_{j_p-2}, since field f is being initialized. The constraints generated when $\mathtt{ins}_p$ is a `call` are the most complex. They link the actual arguments (the topmost stack elements of the caller) to the formal arguments (the lowest local variables at the first bytecode $first(m_k)$ of the initial block of each callee m_k). Moreover, the local variables l_k of the caller and its stack elements s_k that are not actual arguments might keep their approximation or can see it improved when they are a definite alias of an actual argument s_{j_p-u-1} and the corresponding formal argument $l_{\pi-u-1}$ is not updated inside the callee. If this is the case, then the final approximation for $l_{\pi-u-1}$ inside the callee can be used as approximation for l_k (respectively, s_k) after the call. This situation is important since it allows *helper functions* to improve the rawness approximation for the variables of the caller. This is the case, for instance, of `setup()` in Figure 1, whose code initializes tens of fields of an `OptionsDialog`.

The second case of Definition 8 is when $\mathtt{ins}_q @ q$ is a `catch`. This means that $\mathtt{ins}_p$ is executed, it throws an exception e which is caught by $\mathtt{ins}_q$ and stored as $s_0 @ q$. In any case, the original rawness approximation for the local variables remains correct. If $\mathtt{ins}_p$ is not a `call` nor a `throw`, then e is an internal exception [18] without uninitialized fields, so we can use $U_{i_p,0}$. Otherwise, e might be the top of the stack (s_{j_p-1}@ p) for `throw` or an exception thrown by the called method(s) for `call`. If $\texttt{ins}_p$ is a `call`, we also link the actual arguments to the formal ones.

The function *final_con* generates constraints for the final bytecode of a block with no successors. That bytecode can only be a `throw` or a `return` inside some method m. In the first case the top of the stack (s_{j_p-1}@ p) is linked to the exception thrown by m. In the second case it is linked to the return value of the method, if any, and the local variables are linked to the approximation of the local variables at the end of m.

Example 5. Consider $\texttt{ins}_p = \texttt{load 1 java.awt.Frame@p1}$ and $\texttt{ins}_q = \texttt{const "Options"@p2}$ from Figure 1. At `p1` we have $i_p = 2$ local variables and $j_p = 1$ stack elements. Thus $con(\texttt{ins}_p, \texttt{ins}_q) = \{l_0@\,\texttt{p1} \rightarrow l_0@\,\texttt{p2}, l_1@\,\texttt{p1} \rightarrow l_1@\,\texttt{p2}, l_1@\,\texttt{p1} \rightarrow s_1@\,\texttt{p2}, s_0@\,\texttt{p1} \rightarrow s_0@\,\texttt{p2}\}$. □

Example 6. Let $\texttt{ins}_p = \texttt{call java.awt.Dialog.}\langle\texttt{init}\rangle\ldots\texttt{@p3}$ and $\texttt{ins}_q = \texttt{load 0 JFlex.gui.OptionsDialog@p6}$ from Figure 1. At `p3` we have $i_p = 2$ local variables and $j_p = 3$ stack elements. Our aliasing analysis computes $alias_{\texttt{p3}} = \{(l_0, s_0), (l_1, s_1)\}$. This `call` has $n = 1$ targets and $\pi = 3$ parameters (including the implicit `this` parameter). Let *first* be the first bytecode in the initial block of the constructor m of `java.awt.Dialog`, whose code does not contain any `store` 0 nor any `store` 1. Hence $con(\texttt{ins}_p, \texttt{ins}_q) = \{s_0@\,\texttt{p3} \rightarrow l_0@\mathit{first}, s_1@\,\texttt{p3} \rightarrow l_1@\mathit{first}, s_2@\,\texttt{p3} \rightarrow l_2@\mathit{first}, l_0@\,\mathit{end\ of\ m} \rightarrow l_0@\,\texttt{p6}, l_1@\,\texttt{end of}\ m \rightarrow l_1@\,\texttt{p6}\}$. □

We can define the constraints induced by the whole program.

Definition 9. Let $\boxed{\begin{matrix}\texttt{ins}_1\\ \cdots\\ \texttt{ins}_n\end{matrix}} \begin{matrix}\rightarrow b_1\\ \cdots\\ \rightarrow b_m\end{matrix}$ be a block. If $m > 0$, its induced constraints are $\cup_{k=1}^{n-1} con(\texttt{ins}_k, \texttt{ins}_{k+1}) \cup \cup_{h=1}^{m} con(\texttt{ins}_n, \mathit{first}(b_h))$, where $\mathit{first}(b_h)$ is the first instruction in b_h. If $m = 0$, they are $\cup_{k=1}^{n-1} con(\texttt{ins}_k, \texttt{ins}_{k+1}) \cup \mathit{final\_con}(\texttt{ins}_n)$. Those induced by a program P are the union of the constraints induced by each block of P. □

5.3 Correctness of the analysis

Once the constraints for P have been built, they can be *solved, i.e.*, a least solution can be found, satisfying the inclusions represented by their arcs. This is possible since arcs (normal and filtering) stand for monotonic functions from the approximation of their source to that of their sink. Hence a least solution exists, is unique, and can be computed, for instance, with an iterated fixpoint calculation from the empty approximation for each node.

Definition 10. The *solution* of a constraint G is the least assignment $\mathcal{S}$ of sets of fields to nodes, such that $\mathcal{S}(\{f_1, \ldots, f_n\}) = \{f_1, \ldots, f_n\}$ for every node $\{f_1, \ldots, f_n\} \in G$, $\mathcal{S}(n_1) \subseteq \mathcal{S}(n_2)$ for every $n_1 \rightarrow n_2 \in G$ and $\mathcal{S}(n_1) \setminus \{f\} \subseteq \mathcal{S}(n_2)$ for every $n_1 \xrightarrow{f} n_2 \in G$. □

Example 7. The solution of the constraints generated for the program in Figure 1 is such that $\mathcal{S}(l_0@\texttt{p0}) = \{\texttt{owner}, \ldots\}$ contains all the fields defined in `OptionsDialog` and in its superclasses. Moreover, $\texttt{owner} \notin \mathcal{S}(s_0@\texttt{p10}) \neq \emptyset$ and $\mathcal{S}(s_0@\texttt{p12}) = \emptyset$ (that is, all fields of `OptionsDialog` and of its superclasses have been definitely assigned when calling `pack()`). □

We can now provide the correctness result for our analysis. It states that the abstraction of all the states generated during the execution of P according to our operational semantics is over-approximated by the solution of the constraint generated for P. The hypothesis of this proposition guarantees that the considered execution is feasible*, i.e.*, it did not hang the Java Virtual Machine.

PROPOSITION 1. *Let* $\langle b_{\mathit{first}(\texttt{main})} \| \varsigma \rangle \Rightarrow^* \langle \boxed{\begin{matrix}\texttt{ins@}\,p\\ \mathit{rest}\end{matrix}} \begin{matrix}\rightarrow b_1\\ \cdots\\ \rightarrow b_m\end{matrix} \| \sigma \rangle :: a$ *be any execution of our operational semantics, from method* `main` *and an initial state* ς *whose objects in memory have no uninitialized fields, with* $\texttt{ins}(\sigma)$ *defined when* `ins` *is not a* `call`*, or with* $\sigma \in \Xi$ *with at least* π *stack elements when* `ins` *is a* `call` *with* π *parameters. Let there be i local variables and j stack elements at p. Then for every* $0 \leq k < i$ *we have* $\alpha(\sigma)(l_k) \subseteq \mathcal{S}(l_k@\,p)$*, for every* $0 \leq k < j$ *we have* $\alpha(\sigma)(s_k) \subseteq \mathcal{S}(s_k@\,p)$ *and for every field* f_k *we have* $\alpha(\sigma)(f_k) \subseteq \mathcal{S}(f_k@\,ew)$*.* □

Note that, in Java bytecode, method `main` receives an array of strings as parameter and those strings have no uninitialized fields. Hence the hypothesis on ς is sensible.

Proposition 1 is proved in Appendix A.

5.4 Building the `@Raw` annotations

Our rawness analysis can be used whenever one wants to know if some field of a given variable is definitely initialized at a given program point. As we said in Section 3, this is the case for nullness analysis, since existing type-checkers for nullness allow the specification of a set of *non-*`null` *fields*, which are always initialized by all constructors of their defining class and are always assigned a non-`null` value. Hence they do hold a non-`null` value, but only *after* their first initialization. Where this is not yet the case, rawness must be specified.

Given a set of non-`null` fields NN, our analysis infers a superset of the variables v at a given program point p and a superset of the fields f of the program that should be typed as `@Raw`. It is enough to check if $\mathcal{S}(v@\,p) \cap NN \neq \emptyset$ or $\mathcal{S}(f@\,ew) \cap NN \neq \emptyset$, respectively. Similarly for the formal parameters of the methods and for their return value. It is also possible to derive class-specific rawness, that is, to determine if all non-`null` fields defined in a given class κ have been initialized or not. Namely, a superset of the variables v at a given program point p and a superset of the fields f of the program that should be typed as `@Raw(`κ`)` can be determined by checking if $\mathcal{S}(v@\,p) \cap \{\kappa.g \mid \kappa.g \in NN\} \neq \emptyset$ or $\mathcal{S}(f@\,ew) \cap \{\kappa.g \mid \kappa.g \in NN\} \neq \emptyset$, respectively.

6. Experimental results

We have implemented the analysis that is defined in Section 5 and proved correct in Appendix A. This section describes experiments that assess the effectiveness of the analysis. The key question is whether JULIA's output is correct.

6.1 Subject programs and analysis output

We applied the inference tool, JULIA, to three programs. JFlex 1.4.3 is a scanner generator (`http://jflex.de/`). Plume is library of utility programs and data structures (`http://code.google.com/p/plume-lib/`, downloaded on Feb. 3, 2010). The Annotation File Utilities (AFU) 3.0 are tools for reading/writing Java annotations [2].

Figure 5 lists the sizes of the programs, the analysis time, and raw data about Julia's output.

JULIA's scalability depends on the size of the reachable code in an application, rather than on the lines of source code. JULIA starts its analysis at all entry points to the program, and then proceeds to discover and analyze all reachable code in the program. It treats as entry points: (1) any `public static void main(String[])`

	size	reachable program & libraries			analysis time (sec.)		dereferences	inferred annotations	
program	(lines)	methods	lines	bytecodes	nullness	rawness	safe / all (%)	`@NonNull`	`@Raw`
`JFlex`	14987	3885	39097	390315	282	3	8584 / 8751 (98.1)	572 / 741 (77.2)	6 / 1109 (0.5)
`AFU`	13892	4246	38415	412332	346	3	5054 / 5143 (98.3)	642 / 854 (75.2)	15 / 1124 (1.3)
`plume`	19652	5400	49921	511277	660	4	8481 / 8613 (98.5)	617 / 915 (67.4)	3 / 1118 (0.3)
`plume` progs	6167	5400	49921	511277	660	4	6048 / 6148 (98.4)	235 / 300 (78.4)	3 / 339 (0.9)

Figure 5: Experimental results. "Lines" is counted with the cloc program (`http://cloc.sourceforge.net/`). Size is computed separately for the application as downloaded, and for the reachable, analyzed portion of the program, including any reachable libraries but not counting unreachable methods in the program or the libraries. Analysis time and results are for the most precise nullness analysis currently available in JULIA. Dereferences are counted only in the the reachable application code (which does not include libraries). Safe dereferences are those that JULIA can guarantee will never throw a null pointer exception at run time. In the "Inferred annotation" columns, the denominator is the total number of sites at which the annotation could possibly be written, in fields and method signatures of the reachable application code. The percentage of inferred annotations is also given. The last row reports the analysis of `plume`, as in the previous line, but with statistics projected over the 10 classes that have a `main()` method; see Section 6.2.2.

method, and (2) any `public static void test*()` method in a class that extends `TestCase`, to handle JUnit tests.

The rawness analysis is fast — just a few seconds. The nullness analysis runs as long as 11 minutes on our subject programs. Most of the nullness analysis runtime is due to the aliasing and shape analysis that it calls as a subroutine.

Recall from Section 1 the two primary uses for a nullness/rawness inference: to indicate locations where a null pointer exception may be thrown, or to provide annotations for a human or a follow-on analysis. The last two groups of columns in figure 5 address these two uses.

A "dereference" is any location at which a variable must be non-`null` to avoid throwing a null pointer exception. These include field and method dereferences, array accesses, array length expressions, `throw` statements, and synchronization operations. JULIA proves that over 98% of the dereferences in each application program are safe — that is, these locations can never throw a null pointer exception at run time. This fact can aid in optimization and reasoning. For comparison, these numbers are around 80% in the case of NIT (see [23] for a comparison).

Figure 5 indicates the number of annotations inferred, and the maximum number of sites at which the annotation could possibly be inferred. For `@NonNull`, the sites include fields, method formal parameters, and method return types. A single type may have multiple sites; for example, up to three `@NonNull` annotations could be placed on `Map<String,Object>`. Receivers and constructor results are *not* counted as sites, because they are trivially non-`null`. Primitive and void types are never counted, because they cannot be `null`. The sites for `@Raw` are the same as those for `@NonNull`, plus receivers. Constructor results are trivially non-raw.

JULIA annotates a significant amount of the program, lessening the programmer burden. (Either `@NonNull` annotations, or a smaller but still significant number of `@Nullable` annotations, are automatically inserted into the program source code.) `@Raw` is inferred for as much as 1.3% of all references in the program.

Our experiments count annotations on fields and in method signatures. These are the places that annotations are most useful. It would also be possible to infer types for local variables or for expressions within a method body, but our experiments ignore this possibility, and JULIA does not perform such output.

Depending on the program, JULIA proves that 67–78% of all references can be marked as `@NonNull`. Furthermore, only a very few references are marked as `@Raw`.

Previous evaluations of nullness inference tools have generally reported numbers like these, quantifying the tool's output. A problem with such numbers is that they do not indicate whether the tool's output is *correct* or *useful*. Section 6.2 addresses the correctness question by comparing JULIA's output to manual annotations of nullness and rawness. Section 6.3 addresses usefulness of both the nullness and rawness annotations, for one particular use: proving a program is free of null pointer dereferences, or fixing any such problems in the program.

6.2 Comparison to human-written annotations

This section compares JULIA's annotations to a correct set of annotations written by a human.

As part of a different project, plume was previously annotated with nullness and rawness annotations.[1] The manual annotations use `@NonNull` as the default (except for local variables, which default to `@Nullable`), and so only write an annotation for `@Nullable` references. This leads to fewer annotations overall. Plume has 508 nullness or rawness annotations on 312 distinct lines, plus another 36 warning suppression annotations.

The manual annotations were checked by a pluggable type-checker built upon the Checker Framework [20]. The type-checker verified both the correctness of the annotations, and that there are no in plume. The outcome of this process is a guarantee that the annotations are correct, and that plume has no null pointer dereferences.[2] This gives perspective on the 132 (= 8613−8481) possibly-unsafe dereferences that Julia reports in plume: they are probably all false warnings.

To gain perspective on these false warnings and on JULIA's strengths and weaknesses, both of this paper's authors examined differences between the manual annotations and JULIA's output. We examined rawness annotations in all of plume, and nullness annotations in a subset of plume.

6.2.1 *Rawness comparison for all of plume*

We examined all rawness differences, everywhere in plume. Plume as downloaded contains 7 instances of `@Raw`, and JULIA's output contains 3 instances of `@Raw`. One `@Raw` annotation is in both sets, so there are 8 differences to examine.

Three manual `@Raw` annotations, on method receivers, are not inferred by JULIA and are extraneous — they are weaknesses in the manual annotation. The program type-checks with these annotations, but it also type-checks without them, so there is no need for them. They were inserted at a time when the type-checker did need them, because it was unable to infer that an object is initialized before its constructor exits. JULIA and the current type-checker can make such inferences. These annotations have since been removed from plume.

[1]Except for this section, all of our experiments use a version of plume from which all nullness/rawness annotations have been removed.

[2]The guarantee is modulo the fact that when the programmer annotated the program, the programmer also suppressed some type-checking warnings. The programmer only did so when manual reasoning indicated the warning was a false warning, but the programmer may have made mistakes during this manual reasoning.

Inconse-quential			T-C artifact		Unanalyzed code			Manual				Julia						
								error		weakness		err	weakness					
21			40		69			8		18		2	48					
IM	WS	LA	SW	BO	DC	UC	UL	SU	ME	UN	MW	JE	PR	RE	SI	MA	IT	JW
13	4	4	18	22	32	34	3	6	2	16	2	2	20	8	6	4	4	6

Figure 6: Number of lines of differences between manual annotations and JULIA output, classified according to Section 6.2.2. In general, each difference results in two lines of diff output. The table classifies the same differences twice: once into coarser and once into finer-grained categories.

Three manual `@Raw` annotations on parameters of type `Object` are not inferred by JULIA, because JULIA does more detailed analysis than the type-checker. In particular, JULIA recognizes that any value passed to the given methods is initialized up to the `@Object` constructor, even if it is not be fully initialized. The type-checker requires these parameters to be marked `@Raw` because they are not fully initialized.

Finally, two `@Raw` annotations inferred by JULIA are a result of imprecision in JULIA's analysis.

6.2.2 *Full comparison for programs in plume*

We examined all differences between the manual annotations and JULIA's output, for a subset of plume. For our subset, we chose all the programs in plume: each class that contains a main method. There are 10 such classes (out of 44), and they contain about 1/4 of plume's lines of code. Running the `diff` program on these classes yields 206 lines of differences (compared to 1450 lines of differences for all of plume). The last line of Figure 5 provides more measurements.

Usually, there are 2 lines of diff output per difference: one line in the diff output shows the old code, and one shows the new code. In some cases, such as import statements and warning suppression, there are more or fewer. To permit counting without fear of ambiguity, we always use number of lines of diff output.

We classified each of the 206 lines of diff output according to the following categories:

Inconsequential differences

IM (import statement) JULIA added an import statement that is redundant with an existing one.

WS (whitespace) The difference is in whitespace or the order of modifiers.

LA (`@LazyNonNull`) The manual annotation uses `@LazyNonNull`, which means that the variable may start out as `null`, but once set to a non-`null` value, it may continue to be reassigned but is never again assigned to `null`. JULIA uses `@Nullable`, which is equivalent from JULIA's point of view.

Type-checker artifacts

SW (`@SuppressWarnings`) The difference is the removal of a `@SuppressWarnings` annotation that was present in the original program, in a method body. It was placed there to overcome a limitation of the type-checker. JULIA may or may not have a similar limitation; if it does, that will be reflected in a difference in some field or method signature.

BO (inside method body) The difference is an annotation within a method body (other than `@SuppressWarnings`), which our experiment ignores but the type-checker needs.

Unanalyzed code

DC (dead code) A method is never executed, so JULIA does not bother to annotate it. For example, this applies to old code that is no longer used, and to code used for debugging. As another example that alone accounts for 25 of the 32 lines of diff, an API required a `Reader` object, but did not use it, so plume created a `DummyReader` class and passed that to the API.

UC (unreachable code) JULIA cannot find a path that executes a given method, because no such path exists in plume, though a path may exist in client code. The prime cause for this is the fact that one of the selected classes, EntryReader, is really a library, not a program. Its main method is just a simple usage example that does not fully exercise the class.

UL (unanalyzed library) JULIA cannot find a path to a method because the path starts in a library that JULIA did not analyze. For example, a class's `boolean start(RootDoc doc)` method is called reflectively when Javadoc executes, but JULIA does not add Javadoc's main method to the set of entry points for every program.

Errors in manual annotations

SU (static initialization unchecked) The type checker verifies that instance fields are properly initialized by the time the constructor exits, but does not do a similar check for static fields, so a static field marked as `@NonNull` may contain `null`. The plume authors have subsequently verified these errors and corrected them by changing the annotation to `@Nullable`.

ME (manual annotation error, other) A constructor argument was improperly marked as `@NonNull` that should have been `@Nullable`. This annotation has also now been fixed. Overall, JULIA did not reveal any null pointer errors, only the 4 (= 8/2) incorrect annotations.

Weaknesses in manual annotations

UN (unannotated code) The developers did not consider this code worth annotating: for example, the code is undocumented, is used for testing, or is under development. JULIA's inference results would make the annotation task much easier.

MW (manual annotation weakness, other) The programmer wrote `@Nullable` on a method return value that is actually always non-`null`. The annotation type-checks, but the more precise annotation is better.

Errors in Julia output

JE (errors in JULIA annotations) JULIA uses multiple annotations for arrays of references and for generic classes of the standard Java library — one annotation for the container and one for the element type. However, JULIA does not yet do so for user-defined generic classes such as `Pair<S,T>`.

Weaknesses in JULIA output

PR (property) Method `System.getProperty()` always returns non-`null` for certain properties, such as `"java.class.path"` and `"line.separator"` — unless the program does something perverse like `System.getProperties().remove("java.class.path");`, which is not the case here. But JULIA does not infer this fact.

RE (regular expressions) A call to `Matcher.group()` returns non-`null` if it is guarded by a call to `Matcher.matches()`, *and* the string representation contains parentheses at the top level. This is beyond JULIA's reasoning capabilities.

SI (static initialization) A static field was initialized to non-`null` at the beginning of `main()`, and the class was never used outside the dynamic scope of the `main()` method. JULIA cannot guarantee that fact about its use, and so marked the field as `@Nullable`. The field was not set in the static initializer because the initializer has no access to the arguments to `main`.

MA (map) A map was queried via `get()`, using a value that came from an iterator that was indirectly related to the map. Thus, `get()` returned a map value and not `null`, but JULIA did not recognize this fact.

IT (iterator) In a class that implements `java.util.Iterator`, a superclass constructor somewhere in a library uses a (different) iterator. JULIA was not able to establish that the two iterators are not aliased.

JW (JULIA weaknesses, other) JULIA cannot establish that the result of `TimeZone.getDisplayName()` is non-`null`. (Nor can we, but the Javadoc implies that it is.) As a more complicated example, `CalendarBuilder.build()` sets a calendar value to `null`, apparently to satisfy Java's definite assignment check, but then calls `parser.parse()` which is guaranteed (by the structure of its input) to eventually either throw an exception or make a callback that will set the variable that `CalendarBuilder.build()` eventually returns.

Suppose that a programmer wants to use a type-checker to verify that an unannotated version of the 10 plume programs has no null pointer errors. Further suppose that the libraries those programs use are already annotated. (The type-checker comes with an annotated version of the JDK and some other libraries.) The programmer can start with JULIA's output, then edit approximately 75 (= (40+69+2+48)/2) lines. This modest cost demonstrates that JULIA's output is accurate and can be useful to programmers.

6.3 Type-checking of inferred annotations

JULIA infers most of the necessary nullness and rawness annotations, which is a great help to a programmer who would otherwise be forced to write them all from scratch.

Section 6.2 compared JULIA's annotations to a correct set of annotations. Another comparison would be to count the number of type-checking errors that occur when type-checking a program with and without JULIA's annotations. We used the Nullness Checker that is distributed with the Checker Framework [20, 4]. This type-checker requires the programmer to annotate the program, then verifies the correctness of the annotations and the absence of null pointer errors. It shares no code with the JULIA inference tool, making it an effective cross-validation for the correctness of both tools.

Unfortunately, the number of type-checking errors is not a good metric of annotation quality. In general, adding a correct, necessary annotation may increase or decrease the number of compiler warnings, depending on how the variable is used and the correctness and verifiability of the surrounding annotations. For instance, for all of plume (not just the programs as evaluated in Section 6.2), here is the number of type-checking warnings:

Warnings	Annotations
365	no annotations
408	JULIA nullness annotations only
411	JULIA nullness & rawness annotations

As the annotations get more complete and closer to what a programmer needs in order to verify that the program has no null pointer errors, the number of warnings increases! Therefore, we do not believe this is a useful metric, for evaluating annotations or suggesting fixes to them.

7. Conclusion

We have defined a new analysis for computing field initialization ("rawness"), proved it correct, and implemented it. Our experiments compare it to human-provided and machine-checked correct annotations, and these experiments confirm the accuracy of the analysis.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of Java bytecode. In *ESOP*, pages 157–172, Mar. 2007.

[2] Annotation File Utilities website. `http://types.cs.washington.edu/annotation-file-utilities/`, Feb. 3, 2010.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[4] Checker Framework website. `http://types.cs.washington.edu/checker-framework/`, Feb. 3, 2010.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[6] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *J. Object Tech.*, 6(9):455–475, Oct. 2007.

[7] Arnout F. M. Engelen. Nullness analysis of Java source code. Master's thesis, University of Nijmegen Dept. of Computer Science, Aug. 10 2006.

[8] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

[9] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, pages 302–312, Nov. 2003.

[10] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350, Oct. 2007.

[11] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 2(4):97–108, Feb. 2001.

[12] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. *ACM TOPLAS*, 21(6):1196–1250, Nov. 1999.

[13] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *PASTE*, pages 9–14, June 2007.

[14] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE*, pages 13–19, Sep. 2005.

[15] Laurent Hubert, Thomas Jensen, and David Pichardie. Semantic foundations and inference of non-null annotations. In *FMOODS*, pages 132–149, 2008.

[16] Web interface to the JULIA analyzer. `http://julia.scienze.univr.it`.

[17] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM TOPLAS*, 28(4):619–695, 2006.

[18] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, 2nd edition, 1999.

[19] Chris Male and David J. Pearce. Non-null type inference with type aliasing for Java. `http://www.mcs.vuw.ac.nz/~djp/files/MP07.pdf`, Aug. 20, 2007.

[20] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, July 2008.

[21] É. Payet and F. Spoto. Magic-sets transformation for the analysis of Java bytecode. In *SAS*, pages 452–467, Aug. 2007.

[22] F. Spoto. The nullness analyser of JULIA. Submitted for publication, 2010.

[23] F. Spoto. Precise null-pointer analysis. *Software and Systems Modeling*, 2010. To appear.

[24] F. Spoto, F. Mesnard, and É. Payet. A termination analyser for Java bytecode based on path-length. *ACM TOPLAS*, 2010. To appear.

[25] Fausto Spoto. Nullness analysis in boolean form. In *SEFM*, Nov. 2008.

A. Proofs

The following lemma states the correctness of the solution of the constraints generated for a program P, *w.r.t.* the execution of a single bytecode. It will be used in the proof of Proposition 1 in

Section A.1.

LEMMA 1. *Consider a block* $\boxed{\begin{array}{c}\texttt{ins}_1\\ \vdots\\ \texttt{ins}_n\end{array}}\begin{array}{l}\to b_1\\ \cdots\\ \to b_m\end{array}$ *in the program. Consider the pairs of bytecodes* $\langle \texttt{ins}_k, \texttt{ins}_{k+1}\rangle$ *for* $k = 1,\ldots,n-1$ *where* $\texttt{ins}_k$ *is not a* `call`*; and the pairs* $\langle \texttt{ins}_n, \mathit{first}(b_h)\rangle$ *for* $h = 1,\ldots,m$ *where* $\texttt{ins}_n$ *is not a* `call`*. Name those pairs, generically,* $\langle \texttt{ins}_p@\,p, \texttt{ins}_q@\,q\rangle$*. Let* σ_p *be such that both* $\sigma_q = ins_p(\sigma_p)$ *and* $ins_q(\sigma_q)$ *are defined. Let there be* i_p *local variables and* j_p *stack elements at* p *and* i_q *local variables and* j_q *stack elements at* q*. If* $\alpha(\sigma_p)(l_k) \subseteq \mathcal{S}(l_k@\,p)$ *for every* $0 \le k < i_p$*,* $\alpha(\sigma_p)(s_k) \subseteq \mathcal{S}(s_k@\,p)$ *for every* $0 \le k < j_p$ *and* $\alpha(\sigma_p)(f_k) \subseteq \mathcal{S}(f_k@\,ew)$ *for all fields* f_k*, then we have* $\alpha(\sigma_q)(l_k) \subseteq \mathcal{S}(l_k@\,q)$ *for every* $0 \le k < i_q$*,* $\alpha(\sigma_q)(s_k) \subseteq \mathcal{S}(s_k@\,q)$ *for every* $0 \le k < j_q$ *and* $\alpha(\sigma_q)(f_k) \subseteq \mathcal{S}(f_k@\,ew)$ *for all fields* f_k*.* $\square$

Proof. The pairs $\texttt{ins}_p@\,p$ and $\texttt{ins}_q@\,q$ are exactly those considered in Definition 8. We consider the most significant cases, from which the others can be derived similarly.

$\texttt{ins}_p$ is a `const` v and $\texttt{ins}_q$ is not a `catch`

(The case when $\texttt{ins}_q$ is a `catch` does not exist, since `const` v never throws any exception.) By Definition 8 there are arcs U_{i_p,j_p} in the constraints generated for program P. From the definition of *const v* (Section 4), we have $\sigma_p = \langle l \,\|\, s \,\|\, \mu\rangle$ and $\sigma_q = \langle l \,\|\, v :: s \,\|\, \mu\rangle$. Hence, by definition of solution (Definition 10), we have $\alpha(\sigma_q)(l_k) = \alpha(\sigma_p)(l_k) \subseteq \mathcal{S}(l_k@\,p) \subseteq \mathcal{S}(l_k@\,q)$ for every $0 \le k < i_p = i_q$. Similarly, $\alpha(\sigma_q)(s_k) = \alpha(\sigma_p)(s_k) \subseteq \mathcal{S}(s_k@\,p) \subseteq \mathcal{S}(s_k@\,q)$ for every $0 \le k < j_p = j_q - 1$. Moreover, when $v \in \mathbb{Z} \cup \{\texttt{null}\}$, we have $\alpha(\sigma_q)(s_{j_q-1}) = \emptyset \subseteq \mathcal{S}(s_{j_q-1}@\,q)$. The same is true when $v \in \mathbb{L}$ since, in that case, $\mu(v)$ cannot have any `uninit` fields because *const v* is defined (Section 4). Since σ_p and σ_q have the same memory μ_p, we also have $\alpha(\sigma_q)(f_k) = \alpha(\sigma_p)(f_k) \subseteq \mathcal{S}(f_k@\,ew)$ for all fields f_k.

$\texttt{ins}_p$ is a `catch` v and $\texttt{ins}_q$ is not a `catch`

(The case when $\texttt{ins}_q$ is a `catch` does not exist, since `catch` never throws any exception.) By Definition 8 there are arcs U_{i_p,j_p} in the constraints generated for program P. From the definition of *catch* (Section 4), we have $\sigma_p = \langle l \,\|\, top \,\|\, \mu\rangle$ and $\sigma_q = \langle l \,\|\, top \,\|\, \mu\rangle$. Hence, by definition of solution (Definition 10), we conclude that $\alpha(\sigma_q)(l_k) = \alpha(\sigma_p)(l_k) \subseteq \mathcal{S}(l_k@\,p) \subseteq \mathcal{S}(l_k@\,q)$ for every $0 \le k < i_p = i_q$. Similarly, $\alpha(\sigma_q)(s_k) = \alpha(\sigma_p)(s_k) \subseteq \mathcal{S}(s_k@\,p) \subseteq \mathcal{S}(s_k@\,q)$ for every $0 \le k < j_p = j_q$. Since σ_p and σ_q have the same memory μ_p, we also have $\alpha(\sigma_q)(f_k) = \alpha(\sigma_p)(f_k) \subseteq \mathcal{S}(f_k@\,ew)$ for all fields f_k.

$\texttt{ins}_p$ is a `store` $x\ t$ and $\texttt{ins}_q$ is not a `catch`

(The case when $\texttt{ins}_q$ is a `catch` does not exist, since `store` $x\ t$ never throws any exception.) From the definition of *store x t* (Section 4), we have $\sigma_p = \langle l \,\|\, top :: s \,\|\, \mu\rangle$ and $\sigma_q = \langle l[i \mapsto top] \,\|\, s \,\|\, \mu\rangle$. By Definition 8 the constraints generated for program P include the arcs $l_k@\,p \to l_k@\,q$ for every $0 \le k < i_p$ with $k \ne x$, and arcs $s_k@\,p \to s_k@\,q$ for every $0 \le k < j_p - 1$. Hence, by definition of solution (Definition 10) we have $\alpha(\sigma_q)(l_k) = \alpha(\sigma_p)(l_k) \subseteq \mathcal{S}(l_k@\,p) \subseteq \mathcal{S}(l_k@\,q)$ for every $0 \le k < i_p = i_q$, $k \ne x$. Similarly, $\alpha(\sigma_q)(s_k) = \alpha(\sigma_p)(s_k) \subseteq \mathcal{S}(s_k@\,p) \subseteq \mathcal{S}(s_k@\,q)$ for every $0 \le k < j_p - 1 = j_q$. Moreover, there is an arc $s_{j_p-1}@\,p \to l_x@\,q$, so that $\alpha(\sigma_q)(l_x) = \alpha(\sigma_p)(s_{j_p-1}) \subseteq \mathcal{S}(s_{j_p-1}@\,p) \subseteq \mathcal{S}(l_x@\,q)$. Since σ_p and σ_q have the same memory μ_p, also $\alpha(\sigma_q)(f_k) = \alpha(\sigma_p)(f_k) \subseteq \mathcal{S}(f_k@\,ew)$ for all fields f_k.

$\texttt{ins}_p$ is a `new` κ and $\texttt{ins}_q$ is not a `catch`

By Definition 8 there are arcs U_{i_p,j_p} in the constraints generated for program P. From the definition of *new* κ (Section 4), we have $\sigma_p = \langle l \,\|\, s \,\|\, \mu\rangle$ and $\sigma_q = \langle l \,\|\, \ell :: s \,\|\, \mu[\ell \mapsto o]\rangle$ (the other case of the definition of *new* κ cannot be used here since $\texttt{ins}_q$ is not a `catch` and hence it is only defined on non-exceptional states). We know that o is an object of class κ whose reference fields are all bound to `uninit`. Moreover, σ_p and σ_q have the same memory μ_p, but for a fresh location ℓ (hence not yet reachable from the fields). Hence, by definition of solution (Definition 10), we conclude that $\alpha(\sigma_q)(l_k) = \alpha(\sigma_p)(l_k) \subseteq \mathcal{S}(l_k@\,p) \subseteq \mathcal{S}(l_k@\,q)$ for every $0 \le k < i_p = i_q$. Similarly, $\alpha(\sigma_q)(s_k) = \alpha(\sigma_p)(s_k) \subseteq \mathcal{S}(s_k@\,p) \subseteq \mathcal{S}(s_k@\,q)$ for every $0 \le k < j_p = j_q - 1$. By definition of o and since there is an arc $\{\kappa'.f : t \mid t \in \mathbb{K} \text{ and } \kappa \le \kappa'\} \to s_{j_q-1}@\,q$, we have $\alpha(\sigma_q)(s_{j_q-1}) = \{\kappa'.f : t \mid t \in \mathbb{K} \text{ and } \kappa \le \kappa'\} \subseteq \mathcal{S}(s_{j_q-1}@\,q)$. We also have $\alpha(\sigma_q)(f_k) = \alpha(\sigma_p)(f_k) \subseteq \mathcal{S}(f_k@\,ew)$ for all fields f_k.

$\texttt{ins}_p$ is a `new` κ and $\texttt{ins}_q$ is a `catch`

This case exemplifies the situation when $\texttt{ins}_q$ is a `catch`. Similar cases can be proved in the same way. By Definition 8 there are arcs $U_{i_p,0}$ in the constraints generated for program P. From the definition of *new* κ (Section 4), we have $\sigma_p = \langle l \,\|\, s \,\|\, \mu\rangle$ and $\sigma_q = \underline{\langle l \,\|\, \ell \,\|\, \mu[\ell \mapsto oome]\rangle}$ (the other case of the definition of *new* κ cannot be used here since $\texttt{ins}_q$ is a `catch` and hence it is only defined on exceptional states). We know that *oome* is an object of class `OutOfMemoryError` without `uninit` fields. Moreover, σ_p and σ_q have the same memory μ_p, but for a fresh location ℓ (hence not yet reachable from the fields). Hence, by definition of solution (Definition 10), we have $\alpha(\sigma_q)(l_k) = \alpha(\sigma_p)(l_k) \subseteq \mathcal{S}(l_k@\,p) \subseteq \mathcal{S}(l_k@\,q)$ for every $0 \le k < i_p = i_q$. Moreover, $\alpha(\sigma_q)(s_0) = \emptyset \subseteq \mathcal{S}(s_0@\,q)$. We also have $\alpha(\sigma_q)(f_k) = \alpha(\sigma_p)(f_k) \subseteq \mathcal{S}(f_k@\,ew)$ for all fields f_k.

$\texttt{ins}_p$ is a `getfield` f and $\texttt{ins}_q$ is not a `catch`

By Definition 8 there are arcs U_{i_p,j_p-1} in the constraints generated for program P. From the definition of *getfield f* (Section 4), we have $\sigma_p = \langle l \,\|\, rec :: s \,\|\, \mu\rangle$ and $\sigma_q = \langle l \,\|\, x :: s \,\|\, \mu\rangle$ with $x \in \mathbb{Z} \cup \{\texttt{null}, \mu_p(rec).f\}$ (the last case of the definition of *getfield f* cannot be used here since $\texttt{ins}_q$ is not a `catch` and hence it is only defined on non-exceptional states). Hence, by definition of solution (Definition 10), we have $\alpha(\sigma_q)(l_k) = \alpha(\sigma_p)(l_k) \subseteq \mathcal{S}(l_k@\,p) \subseteq \mathcal{S}(l_k@\,q)$ for every $0 \le k < i_p = i_q$. Similarly we conclude that $\alpha(\sigma_q)(s_k) = \alpha(\sigma_p)(s_k) \subseteq \mathcal{S}(s_k@\,p) \subseteq \mathcal{S}(s_k@\,q)$ for every $0 \le k < j_p - 1 = j_q - 1$. Moreover, we have $\alpha(\sigma_q)(s_{j_q-1}) = \emptyset$ when $x \in \mathbb{Z} \cup \{\texttt{null}\}$ and (Definition 7) $\alpha(\sigma_q)(s_{j_q-1}) = \{g \mid \mu_p(x).g = \texttt{uninit}\}$ when $x = \mu_p(rec).f \in \mathbb{L}$. In the latter case, $\alpha(\sigma_q)(s_{j_q-1}) \subseteq \{g \mid$ there exists $\ell \in \mathbb{L}$ such that $\mu_p(\mu_p(\ell).f).g = \texttt{uninit}\} = \alpha(\sigma_q)(f)$. Since σ_p and σ_q have the same memory μ_p, we conclude that $\alpha(\sigma_q)(s_{j_q-1}) \subseteq \alpha(\sigma_q)(f) = \alpha(\sigma_p)(f) \subseteq \mathcal{S}(f@\,ew)$. By Definition 8, the constraints for program P include the arc $f@\,ew \to s_{j_p-1}@\,q$. Since $j_p = j_q$ and by definition of solution (Definition 10), we have $\alpha(\sigma_q)(s_{j_q-1}) \subseteq \mathcal{S}(f@\,ew) \subseteq \mathcal{S}(s_{j_q-1}@\,q)$. Since σ_p and σ_q have the same memory μ_p, we conclude that $\alpha(\sigma_q)(f_k) = \alpha(\sigma_p)(f_k) \subseteq \mathcal{S}(f_k@\,ew)$ for all fields f_k.

$\texttt{ins}_p$ is a `putfield` f and $\texttt{ins}_q$ is not a `catch`

From the definition of *putfield f* (Section 4), $\sigma_p = \langle l \,\|\, top :: rec :: s \,\|\, \mu\rangle$ and

$$\sigma_q = \langle l \,\|\, s \,\|\, \underbrace{\mu[\mu(rec).f \mapsto top]}_{\mu_q}\rangle$$

(the last case of the definition of *putfield f* cannot be used here since $\texttt{ins}_q$ is not a `catch` and hence it is only defined on non-exceptional states). By Definition 8 there are arcs $l_k@\,p \to l_k@\,q$

in the constraints generated for program P, for every $0 \le k < i_p$ with $(l_k, s_{j_p-2}) \notin alias_p$, and arcs $l_k@\,p \xrightarrow{f} l_k@\,q$ for every $0 \le k < i_p$ with $(l_k, s_{j_p-2}) \in alias_p$. In the former case, since the fields of an object can only be initialized during the execution of this bytecode, by Definition 10 we conclude that $\alpha(\sigma_q)(l_k) \subseteq \alpha(\sigma_p)(l_k) \subseteq \mathcal{S}(l_k@\,p) \subseteq \mathcal{S}(l_k@\,q)$. In the latter case, by Definition 7 we have $\alpha(\sigma_q)(l_k) = \emptyset$ if $l[k] \notin \mathbb{L}$, and $\alpha(\sigma_q)(l_k) = \{g \mid \mu_q(l[k]).g = \texttt{uninit}\}$ if $l[k] \in \mathbb{L}$. Since we are assuming that $(l_k, s_{j_p-2}) \in alias_p$, we know that $l[k] = rec$. Since $\mu_q(rec).f = top \neq \texttt{uninit}$ (the value $\texttt{uninit}$ is only allowed as the value of a field, never as a value on the stack), we conclude that $f \notin \alpha(\sigma_q)(l_k) = \{g \mid \mu_q(rec).g = \texttt{uninit}\}$. Since the fields of an object can only be initialized during the execution of this bytecode, by Definition 10 we conclude that, in this case, $\alpha(\sigma_q)(l_k) \subseteq \alpha(\sigma_p)(l_k) \setminus \{f\} \subseteq \mathcal{S}(l_k@\,p) \setminus \{f\} \subseteq \mathcal{S}(l_k@\,q)$. The same proof can be applied to the stack elements. Given a field $f_k \neq f$, we have that μ_p and μ_q agree on the values of field f_k of their objects. Hence $\alpha(\sigma_q)(f_k) = \alpha(\sigma_p)(f_k) \subseteq \mathcal{S}(f_k@\,ew)$. For field f, instead, we have (Definition 7) $\alpha(\sigma_q)(f) = \{g \mid \text{there exists } \ell \in \mathbb{L} \text{ such that } \mu_q(\mu_q(\ell).f).g = \texttt{uninit}\}$. Since $\mu_q(\ell).f \in \{\mu_p(\ell).f, top\}$ and $\mu_q(\ell) = \mu_p(\ell)$ for all locations ℓ where $\mu_p(\ell)$ (or, equivalently, $\mu_q(\ell)$) is defined, we have $\alpha(\sigma_q)(f) \subseteq \{g \mid \text{there exists } \ell \in \mathbb{L} \text{ s.t. } \mu_q(\mu_p(\ell).f).g = \texttt{uninit}\} \cup \{g \mid \mu_p(top).g = \texttt{uninit}\} = \{g \mid \text{there exists } \ell \in \mathbb{L} \text{ s.t. } \mu_p(\mu_p(\ell).f).g = \texttt{uninit}\} \cup \alpha(\sigma_p)(s_{j_p-1}) = \alpha(\sigma_p)(f) \cup \alpha(\sigma_p)(s_{j_p-1})$. By Definition 8, we know that there is an arc $s_{j_p-1}@\,p \to f@\,ew$ among the arcs built for P. Thus $\alpha(\sigma_q)(f) \subseteq \alpha(\sigma_p)(f) \cup \alpha(\sigma_p)(s_{j_p-1}) \subseteq \mathcal{S}(f@\,ew)$. $\square$

A.1 Proof of Proposition 1

Proposition 1, defined in Section 5.3, states that the analysis is safe: it never reports as non-raw a reference that may not be fully initialized. We now prove this proposition.

Proof *of Proposition 1*

We observe that the blocks in the configurations of an activation stack, but the topmost, cannot be both empty and with no successors. This is because configurations are only stacked by rule (2) at page 5 and if *rest* is empty there, then $m \ge 1$ or otherwise the code ends with a `call` bytecode and no `return`, which is illegal in Java bytecode [18].

We proceed by induction on the length n of the execution

$$\langle b_{\mathit{first}(\texttt{main})} \,\|\, \varsigma\rangle \Rightarrow^* \langle \boxed{\begin{array}{c}\texttt{ins}@\,p\\ rest\end{array}} \begin{smallmatrix}\to b_1\\ \vdots\\ \to b_m\end{smallmatrix} \,\|\, \sigma\rangle :: a.$$

If $n = 0$, the execution is just $\langle b_{\mathit{first}(\texttt{main})} \,\|\, \varsigma\rangle$ and $\sigma = \varsigma$. Since the objects in the memory of ς have no uninitialized fields, we have $\alpha(\sigma)(l_k) = \emptyset$ for every $0 \le k < i$, $\alpha(\sigma)(s_k) = \emptyset$ for every $0 \le k < j$ and $\alpha(\sigma)(f_k) = \emptyset$ for all fields f_k, so that the thesis holds.

Assume now that the thesis holds for any such execution of length $n' \le n$. Consider an execution

$$\langle b_{\mathit{first}(\texttt{main})} \,\|\, \varsigma\rangle \Rightarrow^{n+1} \langle \underbrace{\boxed{\begin{array}{c}\texttt{ins}_q@\,q\\ rest_q\end{array}}}_{b_q} \begin{smallmatrix}\to b_1\\ \vdots\\ \to b_m\end{smallmatrix} \,\|\, \sigma_q\rangle :: a_q \tag{7}$$

with $ins_q(\sigma_q)$ defined. This execution must have the form

$$\begin{aligned}\langle b_{\mathit{first}(\texttt{main})} \,\|\, \varsigma\rangle &\Rightarrow^{n_p} \langle \underbrace{\boxed{\begin{array}{c}\texttt{ins}_p@\,p\\ rest_p\end{array}}}_{b_p} \begin{smallmatrix}\to b'_1\\ \vdots\\ \to b'_{m'}\end{smallmatrix} \,\|\, \sigma_p\rangle :: a_p\\ &\Rightarrow^{n+1-n_p} \langle b_q \,\|\, \sigma_q\rangle :: a_q\end{aligned} \tag{8}$$

with $0 \le n_p \le n$, that is, it must have a strict prefix of length n_p whose final activation stack has a topmost configuration with a non-empty block b_p. This is because, for instance, $b_{\mathit{first}(\texttt{main})}$ is a non-empty block (the `main` method must contain at least a `return`). Let hence such n_p be maximal, i_p be the number of local variables at p, j_p the number of stack elements at p and similarly i_q and j_q at q. By inductive hypothesis we know that

$$\begin{aligned}\alpha(\sigma_p)(l_k) &\subseteq \mathcal{S}(l_k@\,p) \quad \text{for all } 0 \le k < i_p\\ \alpha(\sigma_p)(s_k) &\subseteq \mathcal{S}(s_k@\,p) \quad \text{for all } 0 \le k < j_p\\ \alpha(\sigma_p)(f_k) &\subseteq \mathcal{S}(f_k@\,ew) \quad \text{for all fields } f_k.\end{aligned} \tag{9}$$

We will prove that (9) holds for q instead of p also, which completes the proof by induction. We distinguish on the basis of the rule of the operational semantics that is applied at the beginning of the derivation $\Rightarrow^{n+1-n_p}$ in Equation (8).

Rule (1). Then $ins_p(\sigma_p)$ is defined and $\texttt{ins}_p$ is not a `call`.

case a: $\texttt{ins}_p$ is not a `return` nor a `throw`

If $rest_p$ is non-empty then, by the maximality of n_p, (8) must be

$$\begin{aligned}\langle b_{\mathit{first}(\texttt{main})} \,\|\, \varsigma\rangle &\Rightarrow^{n_p} \langle \underbrace{\boxed{\begin{array}{c}\texttt{ins}_p@\,p\\ \texttt{ins}_q@\,q\\ rest_q\end{array}}}_{b_p} \begin{smallmatrix}\to b_1\\ \vdots\\ \to b_m\end{smallmatrix} \,\|\, \sigma_p\rangle :: a_p\\ &\overset{(1)}{\Rightarrow} \langle \underbrace{\boxed{\begin{array}{c}\texttt{ins}_q@\,q\\ rest_q\end{array}}}_{b_q} \begin{smallmatrix}\to b_1\\ \vdots\\ \to b_m\end{smallmatrix} \,\|\, \underbrace{ins_p(\sigma_p)}_{\sigma_q}\rangle :: \underbrace{a_p}_{a_q}.\end{aligned}$$

Otherwise $m' \ge 1$ (legal Java bytecode can only end with a `return` or a `throw`) and, by the maximality of n_p, it must be $b_q = b'_h$ for a suitable $1 \le h \le m'$, so that (8) must have the form

$$\begin{aligned}\langle b_{\mathit{first}(\texttt{main})} \,\|\, \varsigma\rangle &\Rightarrow^{n_p} \langle \underbrace{\boxed{\texttt{ins}_p@\,p}}_{b_p} \begin{smallmatrix}\to b'_1\\ \vdots\\ \to b'_{m'}\end{smallmatrix} \,\|\, \sigma_p\rangle :: a_p\\ &\overset{(1)}{\Rightarrow} \langle \boxed{} \begin{smallmatrix}\to b'_1\\ \vdots\\ \to b'_{m'}\end{smallmatrix} \,\|\, ins_p(\sigma_p)\rangle :: a_p\\ &\overset{(6)}{\Rightarrow} \langle b_q \,\|\, \underbrace{ins_p(\sigma_p)}_{\sigma_q}\rangle :: \underbrace{a_p}_{a_q}.\end{aligned}$$

By the inductive hypothesis (9) for n_p and Lemma 1, we conclude that (9) holds also with q instead of p.

case b: $\texttt{ins}_p$ is a $\texttt{return}\ t$

We show the case when $t \neq \texttt{void}$, since the other is simpler (there is no return value to consider). Then $rest_p$ is empty and $m' = 0$ (no code is executed after a `return` in legal Java bytecode, but the method terminates) and since $ins_p(\sigma_p) \in \Xi$ (definition of *return t*), (8) must be in one of these two forms, depending on the emptiness of block b in (4):

$$\begin{aligned}&\langle b_{\mathit{first}(\texttt{main})} \,\|\, \varsigma\rangle\\ &\Rightarrow^{n_p} \langle \underbrace{\boxed{\texttt{return}\ t}}_{b_p} \,\|\, \underbrace{\langle l_p \,\|\, top :: s_p \,\|\, \mu_p\rangle}_{\sigma_p}\rangle :: \underbrace{\overbrace{\langle b_q \,\|\, \langle l_c \,\|\, s_c \,\|\, \mu_c\rangle\rangle}^{call-time} :: a_q}_{a_p}\\ &\overset{(1)}{\Rightarrow} \langle \boxed{} \,\|\, \langle l_p \,\|\, top \,\|\, \mu_p\rangle\rangle :: a_p \overset{(4)}{\Rightarrow} \langle b_q \,\|\, \underbrace{\langle l_c \,\|\, top :: s_c \,\|\, \mu_p\rangle}_{\sigma_q}\rangle :: a_q\end{aligned} \tag{10}$$

or

$$\langle b_{\mathit{first}(\texttt{main})} \,\|\, \varsigma\rangle$$

$$\Rightarrow^{n_p} \langle \underbrace{\boxed{\texttt{return}\ t}}_{b_p} \| \underbrace{\langle l_p \| top :: s_p \| \mu_p\rangle}_{\sigma_p}\rangle :: \underbrace{\overbrace{\langle \boxed{} \begin{smallmatrix}\to b'_1\\ \vdots\\ \to b'_{m'}\end{smallmatrix} \| \langle l_c \| s_c \| \mu_c\rangle\rangle}^{call-time} :: a_q}_{a_p}$$

$$\overset{(1)}{\Rightarrow} \langle \boxed{} \| \langle l_p \| top \| \mu_p\rangle\rangle :: a_p \overset{(4)}{\Rightarrow} \langle \boxed{} \begin{smallmatrix}\to b'_1\\ \vdots\\ \to b'_{m'}\end{smallmatrix} \| \langle l_c \| top :: s_c \| \mu_p\rangle\rangle :: a_q$$

$$\overset{(6)}{\Rightarrow} \langle b_q \| \underbrace{\langle l_c \| top :: s_c \| \mu_p\rangle}_{\sigma_q}\rangle :: a_q$$

where, in the latter case, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \le h \le m'$. We only prove the case for (10), the other being similar. Consider configuration *call − time*. Since only rule (2) can stack configurations, it was on top when a `call` was executed and (10) must have the form

$$\langle b_{first(\texttt{main})} \| \varsigma\rangle$$

$$\Rightarrow^{n_c} \langle \boxed{\begin{array}{c}\texttt{call}\ \kappa_1.m\ldots\kappa_n.m@c\\ \texttt{ins}_q@q\\ rest_q\end{array}} \begin{smallmatrix}\to b'_1\\ \vdots\\ \to b'_{m'}\end{smallmatrix} \| \underbrace{\langle l_c \| v_{\pi-1} :: \cdots :: v_1 :: rec :: s_c \| \mu_c\rangle}_{\sigma_c}\rangle :: a_q$$

$$\overset{(2)}{\Rightarrow} \langle first(\kappa_w.m) \| \langle [rec :: v_1 :: \cdots :: v_{\pi-1}] \| \varepsilon \| \mu_c\rangle\rangle :: a_p$$

$$\Rightarrow^{n_p-n_c-1} \langle b_p \| \sigma_p\rangle :: a_p \overset{(1)}{\Rightarrow} \langle \boxed{} \| \langle l_p \| top \| \mu_p\rangle\rangle :: a_p \overset{(4)}{\Rightarrow} \langle b_q \| \sigma_q\rangle :: a_q$$

for a suitable $1 \le w \le n$, where the rules in the portion $\Rightarrow^{n_p-n_c-1}$ never make the stack lower than at the beginning of that portion. By inductive hypothesis for n_p and n_c, we know that (9) holds for both p and c. Since it holds for p and the memory in σ_q is μ_p, we conclude that

$$\alpha(\sigma_q)(f_k) \subseteq \mathcal{S}(f_k@ew) \quad \text{for all fields } f_k.$$

Since it holds for c, since the fields of an object can only be initialized during the execution of the program (Section 4) and since $i_c = i_q$, we conclude that

$$\alpha(\sigma_q)(l_k) \subseteq \alpha(\sigma_c)(l_k) \subseteq \mathcal{S}(l_k@c) \quad \text{for all } 0 \le k < i_q$$
$$\alpha(\sigma_q)(s_k) \subseteq \alpha(\sigma_c)(s_k) \subseteq \mathcal{S}(s_k@c) \quad \text{for all } 0 \le k < j_q - 1.$$

By Definition 8, we have $\mathcal{S}(l_k@c) \subseteq \mathcal{S}(l_k@q)$ for all those l_k for which an arc $l_k@c \to l_k@q$ is built (similarly for s_k). For them we have $\alpha(\sigma_q)(l_k) \subseteq \mathcal{S}(l_k@q)$ and $\alpha(\sigma_q)(s_k) \subseteq \mathcal{S}(s_k@q)$. For the l_k that, instead, at call-time, are definite alias of an actual parameter s_{j_c-u-1} of the `call` and such that the corresponding formal parameter $l_{\pi-u-1}$ is never modified inside $\kappa_w.m$, by inductive hypothesis for n_p we know that

$$\alpha(\sigma_q)(l_k) = \alpha(\sigma_p)(l_{\pi-u-1}) \subseteq \mathcal{S}(l_{\pi-u-1}@p)$$

(similarly for s_k). By Definition 8, there is an arc of the form $l_{\pi-u-1}@\,end\ of\ \kappa_w.m \to l_k@q$ among the constraints for program P. Since $\texttt{ins}_p$ is a `return` t, there is also an arc $l_{\pi-u-1}@p \to l_{\pi-u-1}@\,end\ of\ \kappa_w.m$. Hence

$$\alpha(\sigma_q)(l_k) \subseteq \mathcal{S}(l_{\pi-u-1}@p) \subseteq \mathcal{S}(l_{\pi-u-1}@\,end\ of\ \kappa_w.m) \subseteq \mathcal{S}(l_k@q)$$

and similarly for s_k with $0 \le k < s_{j_q-1}$. It remains to prove the same result for s_{j_q-1}, that is, for the returned value. By inductive hypothesis for n_p we know that $\alpha(\sigma_p)(s_{j_p-1}) \subseteq \mathcal{S}(s_{j_p-1}@p)$. By Definition 8, there are arcs $s_{j_p-1}@p \to return@\kappa_w.m$ and $return@\kappa_w.m \to s_{j_c-\pi}@q$. Moreover, $j_c = j_q - 1 + \pi$ so that $j_c - \pi = j_q - 1$. We conclude that

$$\alpha(\sigma_q)(s_{j_q-1}) = \alpha(\sigma_p)(s_{j_p-1}) \subseteq \mathcal{S}(s_{j_p-1}@p)$$
$$\subseteq \mathcal{S}(return@\kappa_w.m) \subseteq \mathcal{S}(s_{j_c-\pi}@q) = \mathcal{S}(s_{j_q-1}@q).$$

case c: $\texttt{ins}_p$ is a `throw` κ

If $rest_p$ is empty and $m' > 0$, the execution (8) must have the form

$$\langle b_{first(\texttt{main})} \| \varsigma\rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\texttt{throw}\ \kappa}}_{b_p} \begin{smallmatrix}\to b'_1\\ \vdots\\ \to b'_{m'}\end{smallmatrix} \| \underbrace{\langle l_p \| e :: s_p \| \mu_p\rangle}_{\sigma_p}\rangle :: a_p$$

$$\overset{(1)}{\Rightarrow} \langle \boxed{} \begin{smallmatrix}\to b'_1\\ \vdots\\ \to b'_{m'}\end{smallmatrix} \| \underbrace{\langle l_p \| e \| \mu_p\rangle}_{\sigma_q}\rangle :: a_p \overset{(6)}{\Rightarrow} \langle b_q \| \sigma_q\rangle :: \underbrace{a_p}_{a_q}$$

where, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \le h \le m'$. If $rest_p$ is non-empty, the execution (8) must have the form

$$\langle b_{first(\texttt{main})} \| \varsigma\rangle \Rightarrow^{n_p} \langle \underbrace{\boxed{\begin{array}{c}\texttt{throw}\ \kappa\\ \texttt{catch}\\ rest_q\end{array}}}_{b_p} \begin{smallmatrix}\to b_1\\ \vdots\\ \to b_m\end{smallmatrix} \| \underbrace{\langle l_p \| e :: s_p \| \mu_p\rangle}_{\sigma_p}\rangle :: a_p$$

$$\overset{(1)}{\Rightarrow} \langle \underbrace{\boxed{\begin{array}{c}\texttt{catch}\\ rest_q\end{array}}}_{b_q} \begin{smallmatrix}\to b_1\\ \vdots\\ \to b_m\end{smallmatrix} \| \underbrace{\langle l_p \| e \| \mu_p\rangle}_{\sigma_q}\rangle :: \underbrace{a_p}_{a_q}$$

since `catch` is the only bytecode whose semantics can be defined on the exceptional state σ_q. In both these cases, by inductive hypothesis for n_p and Lemma 1, we have the thesis.

If $rest_p$ is empty and $m' = 0$, the execution (8) must have one of these two forms, depending on the emptiness of block b in (5):

$$\langle b_{first(\texttt{main})} \| \varsigma\rangle$$

$$\Rightarrow^{n_p} \langle \underbrace{\boxed{\texttt{throw}\ \kappa}}_{b_p} \| \underbrace{\langle l_p \| e :: s_p \| \mu_p\rangle}_{\sigma_p}\rangle :: \underbrace{\overbrace{\langle b_q \| \langle l_q \| s_q \| \mu_q\rangle\rangle}^{call-time} :: a_q}_{a_p}$$

$$\overset{(1)}{\Rightarrow} \langle \boxed{} \| \underline{\langle l_p \| e \| \mu_p\rangle}\rangle :: a_p \overset{(5)}{\Rightarrow} \langle b_q \| \underbrace{\langle l_q \| e \| \mu_p\rangle}_{\sigma_q}\rangle :: a_q \qquad (11)$$

or

$$\langle b_{first(\texttt{main})} \| \varsigma\rangle$$

$$\Rightarrow^{n_p} \langle \underbrace{\boxed{\texttt{throw}\ \kappa}}_{b_p} \| \underbrace{\langle l_p \| e :: s_p \| \mu_p\rangle}_{\sigma_p}\rangle :: \underbrace{\overbrace{\langle \boxed{} \begin{smallmatrix}\to b'_1\\ \vdots\\ \to b'_{m'}\end{smallmatrix} \| \langle l_q \| s_q \| \mu_q\rangle\rangle}^{call-time} :: a_q}_{a_p}$$

$$\overset{(1)}{\Rightarrow} \langle \boxed{} \| \underline{\langle l_p \| e \| \mu_p\rangle}\rangle :: a_p \overset{(5)}{\Rightarrow} \langle \boxed{} \begin{smallmatrix}\to b'_1\\ \vdots\\ \to b'_{m'}\end{smallmatrix} \| \underbrace{\langle l_q \| e \| \mu_p\rangle}_{\sigma_q}\rangle :: a_q$$

$$\overset{(6)}{\Rightarrow} \langle b_q \| \underbrace{\langle l_q \| e \| \mu_p\rangle}_{\sigma_q}\rangle :: a_q$$

where, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \le h \le m'$. We only prove the case (11), the other being similar. Consider configuration *call − time*. Since only rule (2) can stack configurations, it was on top when a `call` was executed and (11) must have the form

$$\langle b_{first(\texttt{main})} \| \varsigma\rangle$$

$$\Rightarrow^{n_c} \langle \boxed{\begin{array}{c}\texttt{call}\ \kappa_1.m\ldots\kappa_n.m@c\\ \texttt{ins}_q@q\\ rest_q\end{array}} \begin{smallmatrix}\to b'_1\\ \vdots\\ \to b'_{m'}\end{smallmatrix} \| \underbrace{\langle l_q \| v_{\pi-1} :: \cdots :: v_1 :: rec :: s_q \| \mu_q\rangle}_{\sigma_c}\rangle :: a_q$$

$$\overset{(2)}{\Rightarrow} \langle \mathit{first}(\kappa_w.m) \,\|\, \langle [rec::v_1::\cdots::v_{\pi-1}] \| \varepsilon \| \mu_q \rangle\rangle :: \langle b_q \| \langle l_q \| s_q \| \mu_q \rangle\rangle :: a_q$$

$$\Rightarrow^{n_p - n_c - 1} \langle b_p \,\|\, \sigma_p \rangle :: a_p \overset{(1)}{\Rightarrow} \langle \boxed{} \| \underline{\langle l_p \,\|\, e \,\|\, \mu_p \rangle} \rangle :: a_p \overset{(5)}{\Rightarrow} \langle b_q \,\|\, \sigma_q \rangle :: a_q.$$

Since $ins_q(\sigma_q)$ is defined and $\sigma_q \in \underline{\Xi}$, the only possibility is that $\texttt{ins}_q$ is a `catch` (it is the only bytecode defined on exceptional states). By Definition 8, the constraints for program P must include the arcs $l_k@\,c \to l_k@\,q$. By inductive hypothesis (9) for n_c, we know that $\alpha(\sigma_c)(l_k) \subseteq \mathcal{S}(l_k@\,c)$ for $0 \le k < i_c = i_q$. Since the fields of an object can only be initialized during the execution of the program, by inductive hypothesis for n_c we conclude that $\alpha(\sigma_q)(l_k) \subseteq \alpha(\sigma_c)(l_k) \subseteq \mathcal{S}(l_k@\,c) \subseteq \mathcal{S}(l_k@\,q)$ (because of the arc $l_k@\,c \to l_k@\,q$). Consider now $s_0 = e$, the only stack element of σ_q. We have $\alpha(\sigma_q)(s_0) = \alpha(\sigma_p)(s_{j_p-1}) \subseteq \mathcal{S}(s_{j_p-1}@\,p)$ (by inductive hypothesis for n_p). By Definition 8, there are arcs $s_{j_p-1}@\,p \to \mathit{exception}@\,\kappa_w.m$ and $\mathit{exception}@\,\kappa_w.m \to s_0@\,q$ in the constraints for program P. Then $\alpha(\sigma_q)(s_0) \subseteq \mathcal{S}(s_{j_p-1}@\,p) \subseteq \mathcal{S}(\mathit{exception}@\,\kappa_w.m) \subseteq \mathcal{S}(s_0@\,q)$. Finally, by inductive hypothesis for n_p we know that $\alpha(\sigma_p)(f_k) \subseteq \mathcal{S}(f_k@\,ew)$ for all fields f_k. Since the memory of σ_q is still μ_p, we conclude that $\alpha(\sigma_q)(f_k) \subseteq \mathcal{S}(f_k@\,ew)$ for all fields f_k.

Rule (2)**.** By definition of *makescope*, (8) must have the form

$$\langle b_{\mathit{first}(\texttt{main})} \,\|\, \varsigma \rangle \Rightarrow^{n_p}$$

$$\langle \underbrace{\boxed{\begin{array}{c}\texttt{call}\ \kappa_1.m \ldots \kappa_n.m@\,p \\ rest_p\end{array}}}_{b_p} \begin{array}{l}\to b'_1 \\ \vdots \\ \to b'_{m'}\end{array} \,\|\, \underbrace{\langle l_p \,\|\, v_{\pi-1} :: \cdots :: v_1 :: rec :: s \,\|\, \mu_p \rangle}_{\sigma_p} \rangle :: a_p$$

$$\overset{(2)}{\Rightarrow} \langle \underbrace{\mathit{first}(\kappa_i.m)}_{b_q} \,\|\, \underbrace{\langle [rec :: v_1 :: \cdots :: v_{\pi-1}] \,\|\, \varepsilon \,\|\, \mu_p \rangle}_{\sigma_q} \rangle :: a_q.$$

Since σ_p and σ_q have the same memory μ_p, for the shape of their stack and local variable array we have $\alpha(\sigma_q)(l_{\pi-u-1}) = \alpha(\sigma_p)(s_{j_p-u-1})$ for every $0 \le u < \pi$. By Definition 8, the constraints for program P include the arcs $s_{j_p-u-1}@\,p \to l_{\pi-u-1}@\,q$, since q is the program point at the beginning of block $\mathit{first}(\kappa_i.m)$. By inductive hypothesis for n_p we conclude that

$$\alpha(\sigma_q)(l_{\pi-u-1}) = \alpha(\sigma_p)(s_{j_p-u-1}) \subseteq \mathcal{S}(s_{j_p-u-1}@p) \subseteq \mathcal{S}(l_{\pi-u-1}@q)$$

for every $0 \le u < \pi$. Since $i_q = \pi$, we conclude that

$$\alpha(\sigma_q)(l_k) \subseteq \mathcal{S}(l_k@\,q)$$

for every $0 \le k < i_q$. The same result for s_k is vacuously true since $j_q = 0$. For the fields, we observe that σ_p and σ_q have the same memory μ_p. From the inductive hypothesis (9) for n_p, we conclude that $\alpha(\sigma_q)(f_k) = \alpha(\sigma_p)(f_k) \subseteq \mathcal{S}(f_k@\,ew)$ for all fields f_k.

Rule (3)**.** Then (8) must have the form

$$\langle b_{\mathit{first}(\texttt{main})} \,\|\, \varsigma \rangle \Rightarrow^{n_p}$$

$$\langle \underbrace{\boxed{\begin{array}{c}\texttt{call}\ \kappa_1.m \ldots \kappa_n.m@\,p \\ rest_p\end{array}}}_{b_p} \begin{array}{l}\to b'_1 \\ \vdots \\ \to b'_{m'}\end{array} \,\|\, \underbrace{\langle l_p \| v_{\pi-1} :: \cdots :: v_1 :: \texttt{null} :: s \| \mu_p \rangle}_{\sigma_p} \rangle :: a_p$$

$$\overset{(3)}{\Rightarrow} \langle \underbrace{\boxed{rest_p} \begin{array}{l}\to b'_1 \\ \vdots \\ \to b'_{m'}\end{array}}_{b_q} \,\|\, \underbrace{\underline{\langle l_p \,\|\, \ell \,\|\, \mu_p[\ell \mapsto npe] \rangle}}_{\sigma_q} \rangle :: a_q$$

when $rest_p$ is non-empty, while otherwise it has the form

$$\langle b_{\mathit{first}(\texttt{main})} \,\|\, \varsigma \rangle \Rightarrow^{n_p}$$

$$\langle \underbrace{\boxed{\texttt{call}\ \kappa_1.m \ldots \kappa_n.m@\,p}}_{b_p} \begin{array}{l}\to b'_1 \\ \vdots \\ \to b'_{m'}\end{array} \,\|\, \underbrace{\langle l_p \| v_{\pi-1} :: \cdots :: v_1 :: \texttt{null} :: s \| \mu_p \rangle}_{\sigma_p} \rangle :: a_p$$

$$\overset{(3)}{\Rightarrow} \langle \boxed{} \begin{array}{l}\to b'_1 \\ \vdots \\ \to b'_{m'}\end{array} \,\|\, \underbrace{\underline{\langle l_p \,\|\, \ell \,\|\, \mu_p[\ell \mapsto npe] \rangle}}_{\sigma_q} \rangle :: a_q \overset{(6)}{\Rightarrow} \langle b_q \,\|\, \sigma_q \rangle :: a_q$$

where, by maximality of n_p, we have $b_q = b'_h$ for a suitable $1 \le h \le m'$. In both cases, we know that ℓ is fresh and npe is a `NullPointerException` object with no uninitialized fields. Moreover, $\sigma_q \in \underline{\Xi}$ and hence $\texttt{ins}_q$ must be a `catch` (it is the only bytecode that is defined on an exceptional state). By Definition 8, there are arcs $l_k@\,p \to l_k@\,q$ for $0 \le k < i_p = i_q$. We also know that σ_p and σ_q have the same memory μ_p. By inductive hypothesis (9) for n_p, we have

$$\alpha(\sigma_q)(l_k@\,q) = \alpha(\sigma_p)(l_k@\,p) \subseteq \mathcal{S}(l_k@\,p) \subseteq \mathcal{S}(l_k@\,q).$$

The state σ_q has only one stack element ℓ. Since npe has no uninitialized fields, we have $\alpha(\sigma_q)(s_0) = \emptyset \subseteq \mathcal{S}(s_0@\,q)$. Since σ_p and σ_q have the same memory μ_p and by inductive hypothesis for n_p, we have $\alpha(\sigma_q)(f_k) = \alpha(\sigma_p)(f_k) \subseteq \mathcal{S}(f_k@\,ew)$ for all fields f_k.

Rules (4)**,** (5) **and** (6)**.** They cannot be applied since b_p is non-empty.

$\square$