Inference of Field Initialization

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 fields set to legal values. It may violate its object invariants, such as that a given field is non-`null`. Programs often manipulate partially-initialized objects, but they must do so with care. Furthermore, analyses must be aware of field initialization. For instance, proving the absence of null pointer dereferences or of division by zero, or proving that object invariants are satisfied, requires information about initialization.

We present a static analysis that infers a safe over-approximation of the program variables, fields, and array elements that, at run time, might hold raw objects. Our formalization is flow-sensitive and interprocedural, and it considers the exception flow in the analyzed program. We have proved the analysis sound and implemented it in a tool called Julia that computes initialization and nullness information. We have evaluated Julia on over 160K lines of code. We have compared its output to manually-written initialization and nullness information, and to an independently-written type-checking tool that checks initialization and nullness. Julia's output is accurate and useful both to programmers and to static analyses.

Categories and Subject Descriptors: F.3.1 - Logics and Meanings of Programs - Specifying and Verifying and Reasoning about Programs - Mechanical Verification
General Terms: Verification, Theory
Keywords: static analysis, abstract interpretation, initialization

1. INTRODUCTION

Modern programming languages such as Java require the initialization of local variables before their use. By contrast, *fields* of objects hold a default value, which is `null` for fields of reference type in Java. Hence, it is difficult to prove invariants involving fields. Suppose that all assignments to a field f write a value with some property p. It is still possible that a value read from f does not satisfy p, unless one can prove that f is always initialized before being read. We call an object *raw* [8] when its fields are not all initialized yet. Hence it is important to know which variables might hold raw objects.

Initialization analysis soundly over-approximates the set of local variables, fields, parameters, return values, and array elements that might hold a raw value at run time. These sites are not just the `this` variable inside the constructors, since it 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 them, see Section 4.5). Hence `this` might be non-raw inside a constructor, from a given program point onwards.

This work was supported by NSF grant CNS-0855252.

ICSE '11, May 21–28, 2011, Honolulu, Hawaii, USA


```
... // many fields defined here

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

Figure 1: A snippet of code from the JFlex program, showing an example where a variable becomes non-raw after all its fields get initialized.

Initialization analysis has many applications in static analysis of software. Our original motivation was type-checking of nullness annotations. For example, the Checker Framework [18] allows fields to be marked as `@NonNull`, but that information is valid only after those fields have been initialized, that is, the containing object is *cooked* [3]. Initialization analysis is important even for a field that is intended to be able to hold `null`, because there are usually object invariants that relate field values to one another. Common examples include that exactly one out of two fields should be `null`, or that one field's value is meaningful only if another field is non-`null`. Initialization analysis can also prevent a programmer from violating object invariants by forgetting to initialize a field, and can ensure that a programmer explicitly initializes fields to `null` when appropriate, which is good style. Another example is the zeroness analysis of software, where one wants to guarantee that divisions never occur over a divisor equal to 0. Since 0 is the default value for integer fields in Java, proving that all assignments to some field f write a non-zero value does *not* entail that a division by $o.f$ (field f of object o) is valid. Instead, o must be proven to be non-raw. As another example, Boolean fields of an object are often used as flags. Since `false` is the default value for Boolean fields, forgetting their initialization leads to programming errors, when the default value is misread as the value of a flag.

Figure 1 contains a snippet of code from class `OptionsDialog` of the JFlex scanner generator, one of the applications analyzed in Section 5. The helper method `setup()` is called by the constructor of class `OptionsDialog` to help it build the object: `setup()` initializes most of the fields. Our initialization analysis infers that the receiver of `setup()` is raw and that all other references are non-raw, such as the receiver of `pack()`. Furthermore, our Julia tool infers that many fields are non-`null`. Without an inference of object initialization, a nullness inference tool would either be unsound, or would be forced to conclude that all fields are possibly `null`.

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 and human understanding. The goal of this paper is to build a sound and precise analysis that infers as few rawness annotations as possible.

Our algorithm builds a constraint graph. Each node represents a value, along with the set of fields known to be uninitialized. The

edges represent data- and control-flow. A fixpoint calculation solves the constraints, computing the smallest sound approximation to the uninitialized fields for each value.

Initialization analysis is useful primarily to support other analyses and to aid human understanding. We have evaluated our initialization analysis in the context of nullness inference and checking. (We emphasize that initialization analysis has many applications beyond this one.) We chose this domain since it is very important, it has been extensively studied, and mature and robust tools are available. State-of-the-art tools include Julia [21] for nullness inference and the Checker Framework [18] for nullness checking. In particular, the nullness analysis of Julia is provably correct and very precise [21]. Julia determines if a collection or array is *full* (only contains non-`null` elements), if a field is always non-`null`, and if an expression is non-`null` locally at a given program point, because it has been assigned a non-`null` value or has been explicitly compared with `null`.

Julia's nullness inference uses a notion of globally non-`null` fields, that are always initialized to a non-`null` value, in all constructors, before being read [22]. In this sense, it performs a reasoning that is related to initialization analysis. However, this was deeply embedded in and specific to the nullness inference, and its results were not explicitly represented as initialization analysis. Our contribution is to create, formalize, and evaluate a new, independent, reusable initialization analysis.

The present work does not make Julia's nullness inference any more accurate or sound, since our initialization analysis is performed after our nullness analysis and does not influence its results; it just provides added information that is needed for external checking. Without information about object initialization, a follow-on analysis, such as nullness checking or one of the other applications described above, would be much less useful. The nullness checker would either be unsound (invalidating any guarantee it might hope to provide) or would be so conservative that it would issue many spurious warnings (making the tool unusable in practice). Another alternative would be for the checker to do initialization inference itself, but that is slow and non-modular, and inference is not in the spirit of a checking tool.

We have implemented our initialization analysis in Julia, resulting in a single tool that computes sound and precise rawness and nullness annotations. We use the Checker Framework to validate Julia's output. Since those two tools use different algorithms and share no code, this provides confidence in the correctness of our initialization analysis and its implementation in Julia. No previous initialization inference has externally validated its output quality.

Currently, the implementation of our initialization analysis is not sound for multi-threaded applications, because Julia assumes that immediately after a field is checked, its value still satisfies the check. This can be solved by applying a worst-case assumption to any field that is not processed in a thread-local way. We do not know of any initialization analysis that copes with this problem.

This paper makes the following contributions:

1. We formally define and prove correct an initialization analysis for Java bytecode, independent from any other analysis (such as nullness).
2. We explicitly consider the exceptional control flows in the program in our definitions and proofs.
3. We implemented our initialization analysis in a scalable and robust tool.
4. We evaluated the correctness and precision of our implementation experimentally. Julia outperformed other initialization inference tools and discovered errors in manual annotations. An independently-implemented type checker verified the correctness of Julia's output.

To the best of our knowledge, all of these points are novel.

2. RELATED WORK

We describe here the most closely-related work.

Fähndrich and Leino [8] 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, or *cooked*, for that class and all of its superclasses, but is 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 possibly-uninitialized fields are not used. The initialization and nullness type-checker that we used in our experiments [18] is a re-implementation of this algorithm, with enhancements. In OIGJ [25], initialization can continue until an object's owner's constructor (not just the object's constructor) completes. Delayed types [9] specify *when* fields can be assumed to have been initialized; by contrast, initialization specifies *where* fields can be assumed to have been initialized.

The most closely related work is Nit, JastAdd, and Jack. Nit [14] infers nullness and, in parallel, also initialization. Unlike our work, its formalization and proofs do not consider exceptional flows. Initialization analysis is mixed with nullness analysis, thus making formalization and proofs more complex. Moreover, it does not report initialization of specific references, but only statistics about the amount of initialized fields. JastAdd [5] infers nullness along with a coarser variant of initialization, where each object is fully initialized or fully uninitialized, without reference to how many constructors have been exited. Its initialization analysis is presented informally and is not proved correct. The latter increased the percentage of references that JastAdd reports as safe from 69% to 71%, for three packages in the JDK. In our experiments, Julia typically reported over 98% of references to be safe, independently from initialization, since its nullness analysis does not depend from initialization analysis. Like Julia, Nit and JastAdd produce an annotation file that can be inserted into Java source code or class files [2]. But, both Nit and JastAdd crashed when run on most of our benchmarks. Furthermore, these tools cannot formally verify the generated annotations, whose correctness follows from the correctness of the theory and implementation of the tool. Jack [17] requires annotated method signatures, then does a flow-sensitive, alias-sensitive flow analysis to determine initialization and nullness types for local variables. It operates on bytecode. Like JastAdd, it infers the coarse version of initialization. Unlike Julia, none of these tools' initialization analysis seems to have been evaluated and compared to manually-identified correct annotations.

Several other nullness inference tools for Java exist. Unlike Julia, they do not infer initialization annotations. Daikon [7] runs the program and soundly outputs `@Nullable` for variables that were ever observed to be `null`. Daikon can produce an annotation file. Houdini [10] inserts `@NonNull` at every possible site, then runs a static checker. Whenever the static checker issues a warning, Houdini removes the relevant annotation. It iterates this process until it reaches a fixed point. Houdini is neither sound nor complete. Inapa [6] is based on similar principles to Houdini. FindBugs [13, 12] finds null pointer dereferences by using an imprecise analysis that internally 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 initialization analysis is a constraint-based abstract interpretation [4] of a concrete operational semantics for Java bytecode, presented in Section 3. Other operational semantics for Java bytecode are available, such as that of Freund and Mitchell [11]. Here we use exactly our formalization in [19], which is also the basis of the Julia analyser, and hence we match theory with implementation. Our formalization is indebted to [15], where Java and Java bytecode are

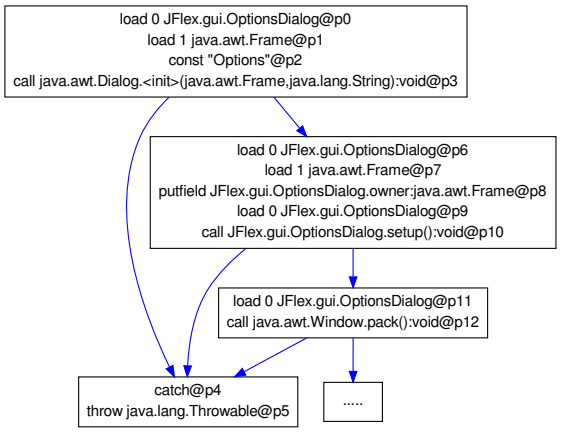


Figure 2: The blocks of code for the constructor in Figure 1.

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 3.2.1) is similar to theirs, as well as our formalization of heap and objects.

This paper is the first description of Julia's initialization analysis. An earlier version of Julia's nullness analysis [22] does no initialization analysis: the receiver of private methods are conservatively annotated with `@Raw`, but no variables, fields, or return values are annotated. That paper and this one share only a formalization of Java bytecode (Section 3); all other material in this paper is new. A tool paper about Julia's nullness analyzer [21] states the existence of an initialization analysis, without definitions, proofs, or experiments. It refers to this paper for details.

3. OPERATIONAL SEMANTICS

This section presents an operational semantics for Java bytecode [22]. Then, the initialization analysis of Section 4 defines an abstract interpretation that executes it on an abstract domain [4].

Bytecodes are the instruction set of the Java Virtual Machine, which has a stack-based architecture. While lower-level than the Java programming language, it does support high-level concepts such as objects, dispatching, and garbage collection. Our formalization is at the bytecode level for several reasons. First, bytecode is much simpler than a programming language: there are a relatively small number of bytecodes, compared to varieties of source statements, and bytecode lacks complexities like inner classes. Second, our implementation of initialization inference is at the bytecode level; as a result, our formalism and implementation are similar, and our proofs are more revealing about our implementation than they would be otherwise. We require a formalization, since one of our goals is to prove our analysis correct.

Our operational semantics is generally standard, adding `uninit` to previous formalisms. Its heart is Figures 3 and 4, and the text of this section primarily serves to introduce their terminology and comment on their rules.

3.1 Syntax

For simplicity of presentation, we assume that `int` is the only primitive type and *classes* are the only reference types, with only *instance* fields and methods. Our implementation handles all Java types and bytecodes. Our algorithm works on Java bytecodes that have been preprocessed into a control flow graph. This same representation is used in [19, 20, 24]; a similar representation is also used in [1],

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

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

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

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

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

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

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

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

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

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

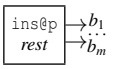
$$exception\_is\ K = \lambda\langle l \,\|\, top \,\|\, \mu\rangle. \begin{cases} \langle l \,\|\, top \,\|\, \mu\rangle & \text{if } top \in \mathbb{L} \text{ and } \mu(top).\kappa \in K \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

$$return\ \texttt{void} = \lambda\langle l \,\|\, s \,\|\, \mu\rangle.\langle l \,\|\, \varepsilon \,\|\, \mu\rangle$$

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

Figure 3: The bytecode semantics. Each instruction is modeled as a function that maps a pre-state to a post-state. $\ell \in \mathbb{L}$ is a fresh location. *oome* is a new instance of `OutOfMemoryError`. *npe* is a new instance of `NullPointerException`.

although, there, Prolog clauses encode the graph, while we work directly on it. A control flow graph is a directed graph of *basic blocks*. All jumps are from the end to the beginning of blocks. We graphically write



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 examples, see Figure 2. The program point p is often irrelevant, so we write just `ins`. Bytecodes operate on *variables*, which encompass both stack elements and local variables. A standard algorithm [16] infers their types.

An exception handler starts with a `catch` bytecode. A conditional, virtual method call, or selection of an exception handler is translated into a block linked to many subsequent blocks. Each of these subsequent blocks starts with a *filtering* bytecode, such as `exception_is[_not]` for exceptional handlers and `receiver_is` for virtual method calls, that specifies when that continuation is taken. They are not needed in Figure 1, where a *default handler* is used: *any* kind of exception is caught and thrown back to the caller.

3.2 Semantics

Our semantics keeps a *state* (Section 3.2.1) that maps program variables to values. An *activation stack* (Section 3.2.3) of states models the method call mechanism, exactly as in an actual implementation of the JVM [16].

3.2.1 State

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

Definition 2. (**State**) A *value* is an element of $\mathbb{Z} \cup \mathbb{L} \cup \{\texttt{null}\}$, where for simplicity we use $\mathbb{Z}$ instead of 32-bit two's-complement integers as in the actual JVM (this choice is irrelevant in this paper) and 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 is a stack of values (the *operand stack*) which grows leftwards, and μ is 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 identifier f (a field f of class $o.\kappa$ or 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`, then 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 $\mu(v).\kappa \leq t$. □

Compared to [19], Definition 2 lets fields hold `uninit`, a special case of `null` or 0 that lets us distinguish an uninitialized field, holding its default value, from a field already initialized to `null` or 0. States are type-correct, in the sense that each variable holds a value consistent with its declared, static type. This is expressed by the last sentence of Definition 2.

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 raw *receiver* $\mu(\ell)$ of the constructor, *i.e.*, `this`, whose fields are not initialized yet. Location ℓ' contains a non-raw object of class `java.awt.Frame`, the explicit argument of the constructor. □

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 $\underline{\Xi}$ always have 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 $\Sigma_{i,j} = \Xi_{i,j} \cup \underline{\Xi}_{i,1}$, the union of normal and exceptional states. □

When we denote a state by σ, we do not specify whether it is normal or exceptional. If we want to stress that fact, we write $\langle l \,\|\, s \,\|\, \mu \rangle$ for a normal state and $\underline{\langle l \,\|\, s \,\|\, \mu \rangle}$ for an exceptional state.

Example 2. A state σ at the beginning of the block in Figure 2 containing `catch@p4` might be an exceptional state arising when `setup()` aborts with an `OutOfMemoryError` (the code of `setup()` 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` is already initialized at `p4`), $\mu(\ell''').\kappa = \texttt{OutOfMemoryError}$ and $\mu(\ell'')$ has no `uninit` fields. □

3.2.2 Bytecodes

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, where 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 [16]. 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, as is required by [16] and as must hold for legal Java bytecode. Figure 3 defines the semantics of the bytecode. We discuss it below.

Basic instructions. Bytecode `const` v pushes $v \in \mathbb{Z} \cup \mathbb{L} \cup \{\texttt{null}\}$ on the stack. When $v \in \mathbb{L}$ (that is, a reference rather than a primitive is being pushed), 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 [16]. Figure 3 defines a partial map, because of the *undefined* case: *const* v is undefined when `const` v tries to push a location already used or referencing a non-fully initialized object. Since $\langle l \,\|\, s \,\|\, \mu \rangle$ (where s might be ε) is not underlined, *const* v is undefined on exceptional states as well, *i.e.*, `const` v is run only when the JVM is in a normal state. This is the case for *all* bytecodes but `catch`, which starts the exceptional handlers from an exceptional state, and which is undefined on all normal states.

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

Bytecode `load` i t pushes on the stack the value of local variable number i, which must exist and have type t. Conversely, bytecode `store` i t pops the top of the stack of type t and writes it in local variable i; if l contains less than $i+1$ variables, the set of local variables grows.

In our formalization, conditional bytecodes are used in complementary pairs (such as `if_ne` and `if_eq`), at the beginning of the two conditional branches. 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, of type t, is not 0 when $t = \texttt{int}$ or is not `null` otherwise; the *undefined* case means that the JVM does not continue the execution of the code if the condition is false.

Object-manipulating instructions. These 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`: $o.f = \texttt{uninit}$ for every field $\kappa'.f : t$ with $t \in \mathbb{K}$ and $\kappa \leq \kappa'$. 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 [19].

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).

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 the underlying value *rec*, of type κ. Its semantics might only remove `uninit` from the approximation of field f, since the value *top* on the stack cannot be `uninit` (Definition 2).

Exception-handling instructions. Bytecode `throw` κ throws the object pointed by the top of the stack, of type $\kappa \leq \texttt{Throwable}$.

Bytecode `catch` starts an exception handler. It takes an exceptional state and transforms it into a normal state, subsequently used by the handler. After `catch`, bytecode `exception_is` K can be used to select the appropriate handler on the basis of the run-time class of *top*: it filters those states whose top of the stack is an instance of a class in $K \subseteq \mathbb{K}$. `exception_is_not` K is shorthand for `exception_is` H, where H are the exception classes that are not instance of any class in K.

Method calls and return. When a caller transfers control to a callee $\kappa.m(\vec{\tau}) : t$, the JVM runs an operation *makescope* $\kappa.m(\vec{\tau}) : t$ that copies the topmost stack elements, which hold the actual arguments of the call, to local variables that correspond to the formal parameters of the callee, and clears the stack. For instance methods, `this` is a special argument held in local variable 0 of the callee.

Definition 4. (***makescope***) Let $\kappa.m(\vec{\tau}) : t$ be a method and π the number of stack elements holding its actual parameters, including the

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

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

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

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

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

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

Figure 4: The transition rules of our semantics (Section 3.2.3).

implicit parameter `this`. We define $(\mathit{makescope}\ \kappa.m(\vec{\tau}) : t) : \Sigma \to \Sigma$ as

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

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

That is, 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. Rule (2) in Figure 4 throws an exception when *rec* is `null`.

Bytecode `return` t terminates a method and clears its operand stack, leaving only the return value when $t \neq \texttt{void}$.

3.2.3 The Transition Rules

We now define the operational semantics of our language.

Definition 5. (**Configuration**) 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 $c_1 :: c_2 :: \dots :: c_n$ 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. (**Operational Semantics**) The (small step) operational semantics of a Java bytecode program P is a relation $a' \Rightarrow_P a''$ (P is usually omitted) providing the immediate successor activation stack a'' of an activation stack a'. It is defined by the rules in Figure 4. □

Rule 1 runs an instruction `ins`, different from `call`, by using its semantics *ins*. Then it moves forward to run the remaining instructions.

Rules 2 and 3 are for method calls. If a call occurs on a `null` receiver, rule 3 creates a new state whose stack contains only a reference to a `NullPointerException`. No actual call happens in this case. Instead, rule 2 calls a method on a non-`null` receiver. It looks up the correct implementation $\kappa_i.m(\vec{\tau}) : t$ by using the look-up rules of the language, 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. A method call might lead to many implementations, depending on the run-time class of the receiver, and this rule seems non-deterministic. However, only one thread of execution will continue, since we assume that the look-up rules are deterministic (as in Java bytecode).

Control returns to the caller by rules 4 and 5. If the callee ends in a normal state, rule 4 rehabilitates the caller configuration but keeps the memory at the end of the execution of the callee and pushes the return value on the stack of the caller. If the callee ends in an exceptional state, rule 5 propagates the exception back to the caller.

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

In the notation $\Rightarrow$, we often specify the rule in Figure 4 used; for instance, we write $\overset{(1)}{\Rightarrow}$ for a derivation step through rule (1).

$$\left\langle \boxed{\begin{array}{c}\texttt{load 0 OD@p0}\\ \texttt{load 1 java.awt.Frame@p1}\\ \texttt{const "Options"@p2}\\ \texttt{call java.awt.Dialog.⟨init⟩...@p3}\end{array}} \begin{array}{l}\to b_6^4\end{array} \| \langle [\ell, \ell'] \| \varepsilon \| \mu \rangle \right\rangle :: a$$

$$\overset{(1)}{\Rightarrow} \left\langle \boxed{\begin{array}{c}\texttt{load 1 java.awt.Frame@p1}\\ \texttt{const "Options"@p2}\\ \texttt{call java.awt.Dialog.⟨init⟩...@p3}\end{array}} \begin{array}{l}\to b_6^4\end{array} \| \langle [\ell, \ell'] \| \ell \| \mu \rangle \right\rangle :: a$$

$$\overset{(1)}{\Rightarrow} \left\langle \boxed{\begin{array}{c}\texttt{const "Options"@p2}\\ \texttt{call java.awt.Dialog.⟨init⟩...@p3}\end{array}} \begin{array}{l}\to b_6^4\end{array} \| \langle [\ell, \ell'] \| \ell' :: \ell \| \mu \rangle \right\rangle :: a$$

$$\overset{(1)}{\Rightarrow} \left\langle \boxed{\texttt{call java.awt.Dialog.⟨init⟩...@p3}} \begin{array}{l}\to b_6^4\end{array} \| \langle [\ell, \ell'] \| \ell'' :: \ell' :: \ell \| \mu \rangle \right\rangle :: a$$

$$\overset{(2)}{\Rightarrow} \langle \mathit{first} \| \langle [\ell, \ell', \ell''] \| \varepsilon \| \mu \rangle \rangle :: \left\langle \boxed{\ } \begin{array}{l}\to b_6^4\end{array} \| \langle [\ell, \ell'] \| \varepsilon \| \mu \rangle \right\rangle :: a$$

$$\Rightarrow \left\langle \boxed{\ } \| \langle [\ell, \ell', \ell''] \| \varepsilon \| \mu' \rangle \right\rangle :: \left\langle \boxed{\ } \begin{array}{l}\to b_6^4\end{array} \| \langle [\ell, \ell'] \| \varepsilon \| \mu \rangle \right\rangle :: a$$

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

$$\overset{(6)}{\Rightarrow} \left\langle \boxed{\begin{array}{c}\texttt{load 0 OD@p6}\\ \texttt{load 1 java.awt.Frame@p7}\\ \texttt{putfield OD.owner...@p8}\\ \texttt{load 0 OD@p9}\\ \texttt{call OD.setup():void@p10}\end{array}} \begin{array}{l}\to b_{11}^4\end{array} \| \langle [\ell, \ell'] \| \varepsilon \| \mu' \rangle \right\rangle :: a$$

$$\overset{(1)}{\Rightarrow} \left\langle \boxed{\begin{array}{c}\texttt{load 1 java.awt.Frame@p7}\\ \texttt{putfield OD.owner...@p8}\\ \texttt{load 0 OD@p9}\\ \texttt{call OD.setup():void@p10}\end{array}} \begin{array}{l}\to b_{11}^4\end{array} \| \langle [\ell, \ell'] \| \ell \| \mu' \rangle \right\rangle :: a$$

$$\overset{(1)}{\Rightarrow} \left\langle \boxed{\begin{array}{c}\texttt{putfield OD.owner...@p8}\\ \texttt{load 0 OD@p9}\\ \texttt{call OD.setup():void@p10}\end{array}} \begin{array}{l}\to b_{11}^4\end{array} \| \langle [\ell, \ell'] \| \ell' :: \ell \| \mu' \rangle \right\rangle :: a$$

$$\overset{(1)}{\Rightarrow} \left\langle \boxed{\begin{array}{c}\texttt{load 0 OD@p9}\\ \texttt{call OD.setup():void@p10}\end{array}} \begin{array}{l}\to b_{11}^4\end{array} \| \langle [\ell, \ell'] \| \varepsilon \| \underbrace{\mu'[\mu(\ell).\texttt{owner} \mapsto \ell']}_{\mu''} \rangle \right\rangle :: a$$

$$\overset{(1)}{\Rightarrow} \left\langle \boxed{\texttt{call OD.setup():void@p10}} \begin{array}{l}\to b_{11}^4\end{array} \| \langle [\ell, \ell'] \| \ell \| \mu'' \rangle \right\rangle :: a$$

Figure 5: A partial execution according to the semantics of Figure 4. `OD` stands for `JFlex.gui.OptionsDialog`. b_x is the block in Figure 2 starting at px. Location ℓ'' points to the string `"Options"` from the constant pool. *first* is the first block of the constructor of `java.awt.Dialog`. $\Rightarrow$ is a complete execution of the latter and μ' is the memory at its end.

Example 3. The operational semantics, starting from the state $\langle [\ell, \ell'] \| \varepsilon \| \mu \rangle$ in Example 1, can proceed from `p0` as in Figure 5.

The first steps push on the stack the actual arguments of the constructor of `java.awt.Dialog`, whose complete execution generates a new memory μ' at its end. The computation continues with that memory. □

4. INITIALIZATION ANALYSIS

We define here a constraint-based abstract interpretation of the concrete semantics of Section 3, that Julia uses to perform an *initialization analysis*, after and independently from its nullness analysis. This makes formalization and correctness proofs (see the technical report [23]) simpler and allows its use beyond nullness analysis.

Given a program P, the initialization analysis builds and then solves a constraint graph. Each node represents a set of non-initialized fields. Each directed edge, or arc, represents a relationship between those sets. The arc $n_1 \to n_2$ states that the fields in n_1 are also in n_2. The *filtering* arc $n_1 \xrightarrow{f} n_2$ states that fields in n_1 *except* f are also in n_2; it is used for a `putfield` bytecode that sets a field (possibly for the first time). When a `new` κ bytecode creates an object o, its uninitialized fields are all fields of κ and of its superclasses. Arcs are built to link subsequent program points, following all possible flows of control induced by loops, conditionals and exceptions.

Generally speaking, each node stands for a variable:

- 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 ("everywhere")
- *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 that explicit set of uninitialized fields.

A solution to the constraint graph is, for each node, a set of uninitialized fields. The best assignment is the one that has the smallest sets while being consistent with the constraints. We compute it with a least fixpoint calculation. For each variable, the result over-approximates the set of its possibly uninitialized fields.

Example 4. In Figure 2, the approximation computed for l_0@`p0` is $S = \{$`owner`$, \ldots\}$, all fields of class `OptionsDialog` and of its superclasses; that for s_0@`p10` is $S \setminus \{$`owner`$\}$, since field `owner` has been already initialized at `p10`. The approximation for s_0@`p12` is $\emptyset$, since all fields of class `OptionsDialog` and of its superclasses have been already initialized at `p12`. □

4.1 The Abstraction Map for Initialization

In order to formalize our initialization analysis and prove it correct, we now define the abstraction map from states to initialization information. For a given state, it maps each reference, stack element, and field to a set of possibly-uninitialized fields.

Definition 7. (**Initialization Abstraction**) Let $\sigma = \langle [v_0 \ldots v_{i-1}] \| w_{j-1} :: \cdots :: w_0 \| \mu \rangle$ be a state (possibly exceptional, that is, underlined) with i local variables and j stack elements. Its *initialization 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, i.e.*, fields that have not been initialized yet for each local variable, stack element or field:

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

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

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

□

By Definition 7, the initialization information of a stack element or local variable is the set of fields bound to `uninit` in the object they hold. The initialization information of a field f_k, instead, includes a field f if there is any 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 holds `uninit`. 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.*, treated as static fields: we cannot distinguish fields with the same name but in different objects. This abstraction is necessary to get a finite analysis, since the number of objects in memory is potentially unbounded.

Example 5. Consider the state $\sigma = \langle [\ell, \ell'] \| \varepsilon \| \mu \rangle$ from Example 1. Its abstraction is such that `owner`$\in \alpha(\sigma)(l_0)$. □

4.2 The Abstract Constraint Graph

Each pair of adjacent bytecode instructions in the control flow graph gives rise to a set of constraint arcs. This section defines those arcs.

Definition 8. (**Bytecode constraints**) Let $\texttt{ins}_p$*@ p* and $\texttt{ins}_q$*@ q* be two bytecodes. Let i_p and j_p be the number of local variables and stack elements at the beginning of $\texttt{ins}_p$, respectively. As a shorthand, we will use $U_{i_p,j_p} = \{l_k@\, p \to l_k@\, q \mid 0 \leq k < i_p\} \cup \{s_k@\, p \to s_k@\, q \mid 0 \leq k < j_p\}$, which indicates that there is no change in the given locals or stack elements.

There are three cases: edges to a bytecode other than `catch`; edges to `catch`; and constraints for the last bytecode of a method.

Case 1: edges to a bytecode other than `catch`:

$$\begin{aligned}
con(\texttt{const } v, \texttt{ins}_q) &= U_{i_p,j_p} \\
con(\texttt{catch}, \texttt{ins}_q) &= U_{i_p,j_p} \\
con(\texttt{exception_is[_not] } K, \texttt{ins}_q) &= U_{i_p,j_p} \\
con(\texttt{dup } t, \texttt{ins}_q) &= U_{i_p,j_p} \cup \{s_{j_p-1}@\, p \to s_{j_p}@\, q\} \\
con(\texttt{load } x\, t, \texttt{ins}_q) &= U_{i_p,j_p} \cup \{l_x@\, p \to s_{j_p}@\, q\} \\
con(\texttt{store } x\, t, \texttt{ins}_q) &= \{l_k@\, p \to l_k@\, q \mid 0 \leq k < i_p, k \neq x\} \\
&\quad \cup \{s_k@\, p \to s_k@\, q \mid 0 \leq k < j_p - 1\} \\
&\quad \cup \{s_{j_p-1}@\, p \to l_x@\, q\} \\
con(\texttt{if_ne } t, \texttt{ins}_q) &= U_{i_p,j_p-2} \\
con(\texttt{new } \kappa, \texttt{ins}_q) &= U_{i_p,j_p} \cup \{\{\kappa'.f : t \mid t \in \mathbb{K} \text{ and } \kappa \leq \kappa'\} \to s_{j_p}@\, q\} \\
con(\texttt{getfield } f, \texttt{ins}_q) &= U_{i_p,j_p-1} \cup \{f@\, ew \to s_{j_p-1}@\, q\} \\
con(\texttt{putfield } f, \texttt{ins}_q) &= \{l_k@\, p \to l_k@\, q \mid 0 \leq k < i_p, (l_k, s_{j_p-2}) \notin alias_p\} \\
&\quad \cup \{s_k@\, p \to 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 \to f@\, ew\} \\
con(\texttt{call } m_1 \ldots m_n, \texttt{ins}_q) &= \cup_{k=1}^{n} \cup_{u=0}^{\pi-1} \{s_{j_p-u-1}@\, p \to l_{\pi-u-1}@first(m_k)\} \\
&\quad \cup \{return@\, m_k \to s_{j_p-\pi}@\, q \mid 1 \leq k \leq n\} \\
&\quad \cup \left\{ l_k@\, p \to 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 } \texttt{store } l_{\pi-u-1}\ t \end{array} \right\} \\
&\quad \cup \left\{ l_{\pi-u-1}@\, end\ of\ m_w \to 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 } \texttt{store } l_{\pi-u-1}\ t \end{array} \right\} \\
&\quad \cup \left\{ s_k@\, p \to 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 } \texttt{store } l_{\pi-u-1}\ t \end{array} \right\} \\
&\quad \cup \left\{ l_{\pi-u-1}@\, end\ of\ m_w \to 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 } \texttt{store } l_{\pi-u-1}\ t \end{array} \right\}
\end{aligned}$$

Case 2: edges to `catch`:

$con(\texttt{throw}\ \kappa, \texttt{catch}) = \{l_k@p \to l_k@q \mid 0 \le k < i_p\} \cup \{s_{j_p-1}@p \to s_0@q\}$

$con(\texttt{call}\ m_1 \ldots m_n, \texttt{catch}) = \cup_{k=1}^{n} \cup_{u=0}^{\pi-1} \{s_{j_p-u-1}@p \to l_{\pi-u-1}@first(m_k)\}$
$\cup \{exception@m_k \to s_0@q \mid 1 \le k \le n\} \cup \{l_k@p \to l_k@q \mid 1 \le k < i_p\}$

$con(\texttt{ins}_p, \texttt{catch}) = U_{i_p,0}$, where $\texttt{ins}_p$ is neither a `throw` nor a `call`.

Case 3: constraints for the last bytecode of a method:
If the program point p belongs to method m, we define

$$final\_con(\texttt{throw}\ \kappa) = \{s_{j_p-1}@p \to exception@m\}$$
$$final\_con(\texttt{return void}) = \{l_k@p \to l_k@end\ of\ m \mid 0 \le k < i_p\}$$
$$final\_con(\texttt{return}\ t) = \{l_k@p \to l_k@end\ of\ m \mid 0 \le k < i_p\} \cup \{s_{j_p-1}@p \to return@m\}$$

where $t \neq \texttt{void}$. □

The first case of Definition 8 is when $\texttt{ins}_q$ is not a `catch`: the normal output state of $\texttt{ins}_p$ flows to the beginning of $\texttt{ins}_q$. If $\texttt{ins}_p$ is a `const`, the sets of uninitialized fields for local variables and stack elements do not change. This happens also for `catch`, `exception_is` and `exception_is_not`.

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 the uninitialized fields of the old top of the stack ($s_{j_p-1}@p$). Similar constraints are built for `load` and `store`.

If $\texttt{ins}_p$ is an `if_ne`, two elements are removed from the stack. If it is a `new` κ, 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; `putfield` modifies the initialization information of every definite alias of its receiver s_{j_p-2}, since it initializes f. In our implementation, we reuse here the definite aliasing analysis of [20].

The constraints generated for `call` are the most complex. They link the actual arguments (at the top of the stack of the caller) to the formal ones (the lowest local variables at the first bytecode $first(m_k)$ of each callee m_k). 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 callee does not update the corresponding formal argument $l_{\pi-u-1}$. In that case, the final approximation for $l_{\pi-u-1}$ inside the callee is used to approximate l_k (respectively, s_k) after the call. This is important to let *helper functions* improve the initialization approximation for the variables of the caller, as is the case for `setup()` in Figure 1, whose code initializes tens of fields of an `OptionsDialog`.

The second case of Definition 8 is when $\texttt{ins}_q@q$ is a `catch`: the execution of $\texttt{ins}_p$ throws an exception e, caught by $\texttt{ins}_q$ and stored as $s_0@q$. The initialization approximation for the local variables does not change. If $\texttt{ins}_p$ is neither a `call` nor a `throw`, then e is an internal exception [16] without uninitialized fields and we can use $U_{i_p,0}$. Otherwise, e is the top of the stack ($s_{j_p-1}@p$) for `throw` or an exception thrown by the called method(s) for `call`. In the second case, we link actual to formal arguments.

Function *final_con* generates constraints for the last bytecodes of a method *m, i.e.*, a `throw` or a `return`: the top of the stack ($s_{j_p-1}@p$) is linked to the exception thrown by m or to its return value, if any, respectively. For `return`, local variables are linked to the approximation of the local variables at the end of m.

Example 6. Consider $\texttt{ins}_p$ =`load 1 java.awt.Frame@p1` and $\texttt{ins}_q$ =`const "Options"@p2` from Figure 1. At `p1` we have $i_p = 2$ local variables and $j_p = 1$ stack elements. Thus $con(\texttt{ins}_p, \texttt{ins}_q) = \{l_0@\texttt{p1} \to l_0@\texttt{p2}, l_1@\texttt{p1} \to l_1@\texttt{p2}, l_1@\texttt{p1} \to s_1@\texttt{p2}, s_0@\texttt{p1} \to s_0@\texttt{p2}\}$. □

Example 7. Let $\texttt{ins}_p$ =`call java.awt.Dialog.⟨init⟩...@p3` and $\texttt{ins}_q$ =`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 `this`). Let *first* be the first bytecode of the constructor m of `java.awt.Dialog`, whose code does not contain any `store 0` nor any `store 1`. Hence $con(\texttt{ins}_p, \texttt{ins}_q) = \{s_0@\texttt{p3} \to l_0@first, s_1@\texttt{p3} \to l_1@first, s_2@\texttt{p3} \to l_2@first, l_0@end\ of\ m \to l_0@\texttt{p6}, l_1@end\ of\ m \to l_1@\texttt{p6}\}$. □

We now define the constraints induced by the whole program. They are the union of the constraints generated for each pair of adjacent instructions, possibly in two consecutive blocks.

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

4.3 Constraint Solving

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

Definition 10. (**Constraint solution**) 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 $n_1 \to n_2 \in G$ and $\mathcal{S}(n_1) \setminus \{f\} \subseteq \mathcal{S}(n_2)$ for every $n_1 \xrightarrow{f} n_2 \in G$. □

Example 8. The solution of the constraints for the program in Figure 1 is such that $\mathcal{S}(l_0@\texttt{p0}) = \{\texttt{owner}, \ldots\}$ contains the fields of `OptionsDialog` and of its superclasses. Moreover, $\texttt{owner} \notin \mathcal{S}(s_0@\texttt{p10}) \neq \emptyset$ and $\mathcal{S}(s_0@\texttt{p12}) = \emptyset$: all fields of `OptionsDialog` and its superclasses are initialized when `pack()` is called. □

4.4 Correctness of the Analysis

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 constraints 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* $\langle b_{first(\texttt{main})} \| \varsigma \rangle \Rightarrow^* \left\langle \boxed{\begin{matrix}ins@p \\ rest\end{matrix}} \begin{matrix}\to b_1 \\ \cdots \\ \to b_m\end{matrix} \right\| \sigma \rangle :: a$ *be any execution of our operational semantics, from method* `main` *and an initial state* ς *whose objects in memory have no uninitialized fields, with* $ins(\sigma)$ *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 $0 \le k < i$: $\alpha(\sigma)(l_k) \subseteq \mathcal{S}(l_k@p)$.
For every $0 \le k < j$: $\alpha(\sigma)(s_k) \subseteq \mathcal{S}(s_k@p)$.
For every field f_k: $\alpha(\sigma)(f_k) \subseteq \mathcal{S}(f_k@ew)$. □

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. The proof of Proposition 1 is in [23].

4.5 Building the `@Raw` Annotations

Our initialization analysis computes the fields of a given variable that are definitely initialized at a given program point. But type-checkers require a more abstract information, that is, the indication, by `@Raw`, of which variables might hold a raw value, with no reference to the specific uninitialized fields.

Given a set of non-`null` fields NN and a class κ, let us define $NN_\kappa = \{\kappa'.g \in NN \mid \kappa' \geq \kappa\}$, that is, the set of non-`null` fields defined in κ or in one of its superclasses. We can infer a superset of the variables v, of type κ, at a given program point p and a superset of the fields f, of type κ, that should be typed as `@Raw`, by checking if $\mathcal{S}(v@\,p) \cap NN_\kappa \neq \emptyset$ or $\mathcal{S}(f@\,ew) \cap NN_\kappa \neq \emptyset$, respectively. Similarly for the formal parameters of the methods and for their return value.

5. EXPERIMENTAL RESULTS

We have implemented our analysis in Section 4 in the Julia tool. It can be used through the web interface `http://julia.scienze.univr.it`. This section describes experiments that assess its effectiveness and compare it with other tools and with manual annotations.

Section 5.1 gives statistics about Julia's output. Section 5.2 compares Julia with other inference tools. Section 5.3 compares Julia's output to manual annotations of initialization and nullness and to a type checker.

As explained in the introduction, an initialization analysis is useful in other contexts besides nullness analysis. However, nullness analysis is a familiar and well-developed topic, and was our original motivation, so we use it as an illustration of the benefits of initialization analysis.

5.1 Quantitative Results

We present results for four programs. The Annotation File Utilities (AFU) 3.0 are tools for reading/writing Java annotations [2]. JFlex 1.4.3 is a scanner generator (`http://jflex.de/`). Plume is a library of utility programs and data structures (`http://code.google.com/p/plume-lib/`, downloaded on Feb. 3, 2010). Daikon 4.6.4 is an invariant generator (`http://pag.csail.mit.edu/daikon/`). Figure 6 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 application code, 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[])` method, and (2) any `public static void test*()` method in a class that extends `TestCase`, to handle JUnit tests.

The column *total time* in Figure 6 reports the full analysis time, including nullness, initialization, and all supporting analyses, on a quadcore Intel machine running at 2.66Ghz with 8 gigabytes of RAM. The initialization analysis is fast (see the *init. time* column) — just a few seconds. Most of the time is spent for the supporting aliasing and heap analysis.

One use of initialization inference is to support nullness inference. Nullness inference may be used 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 6 address these two uses and are described in the following two paragraphs.

A "dereference" is any location where 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. In each application, Julia proves over 94% of the dereferences 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 [22].

Figure 6 indicates the number of annotations inferred, and the maximum number of sites where they could possibly be inferred. For `@NonNull`, these include fields, method formal parameters and return types. A single type may have multiple sites: up to three `@NonNull` annotations could be placed on `Map<String,Object>`. Receivers and constructor results are *not* counted as sites, since they are trivially non-`null`. Primitive and void types are not counted, since they cannot be `null`. The sites for `@Raw` are the same as those for `@NonNull`, plus receivers. 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 source code.) `@Raw` is inferred for as much as 0.9% of all sites.

5.2 Comparison with Other Inference Tools

Two other tools that aim to infer initialization are Nit and JastAdd. This section compares these tools to Julia.

Nit crashes when run on any of our subject programs. We managed to make it work on part of the Annotation File Utilities, starting from the two entry points in `annotations.io.classfile.[ClassFileWriter|ClassFileReader]` but not from the main entry point in `annotator.Main`; we call this "AFU light". To permit a direct comparison, we also ran Julia from those two entry points only, as reported in Figure 7. Julia (correctly) considers fewer methods reachable than Nit. This makes the comparison harder, since the two tools analyze different amounts of code. However, it is undeniable that Nit is much faster, but the quality of its analysis is much worse. Nit proves fewer (a smaller proportion of) dereferences safe and generates fewer `@NonNull` and more `@Raw` annotations. Remember the precision is proportional to the number of the `@NonNull` annotations and inversely proportional to that of the `@Raw` annotations. Nit's 63 `@Raw` annotations in Figure 7 are actually an underestimate, because they do not include receivers, return values, or any references in inner types (in collections, maps...). By contrast, Julia considers all sites for rawness and still reports only 10 `@Raw` annotations. Another concern is soundness. A spot-check of Nit's results revealed errors: `@NonNull` annotations on references that could be null, or lack of `@Raw` annotations where they were needed. We are not aware of errors in the theory underlying Nit, but this incorrect output is nonetheless a cause for concern.

JastAdd crashes when run on `AFU`, `plume`, or `Daikon`, apparently because it mishandles overloaded methods. It works fine and fast for `JFlex`, reporting 14 rawness annotations, more than the 3 reported by Julia, and 389 (non-)nullness annotations, fewer than the 591 reported by Julia. We could not get JastAdd to print statistics on safe dereferences. JastAdd's imprecise nullness analysis never marks static fields as `@NonNull`, which may cause it to output fewer `@Raw` annotations than ideal. This illustrates one reason that an initialization analysis should be evaluated along with a precise client analysis. Another imprecision in JastAdd, that causes it to output spurious `@Raw` annotations not reported by Julia, is that it considers as raw every variable where the receiver of a constructor might propagate, even after all its fields have been initialized. Since JastAdd is faster, could we reduce overall runtime by using it to provide a first, coarser approximation of initialization analysis, that Julia would improve? This does not seem practical because of JastAdd's instability and its lack of a formal proof of correctness.

5.3 Comparison to Human-Written Annotations

Julia is more precise than a state-of-the-art type-checker for initialization and nullness. Furthermore, Julia's results pointed out errors in the type-checker and in manually-written annotations.

program	size (lines)	reachable program & libraries methods	lines	bytecodes	time (sec.) total	init.	dereferences safe / all (%)	inferred annotations @NonNull	@Raw
AFU	13892	4342	42342	435617	209	2	5071 / 5143 (98.6)	649 / 854 (76.0)	10 / 1124 (0.9)
JFlex	14987	3858	41134	385612	118	2	8624 / 8753 (98.5)	591 / 741 (79.8)	3 / 1109 (0.3)
plume	19652	5391	53403	518166	321	2	8360 / 8457 (98.8)	675 / 912 (74.0)	1 / 1118 (0.1)
Daikon	112077	11481	189223	1526231	2151	10	70747/75062 (94.3)	7145/10435 (68.5)	97/15153 (0.6)
plume progs	6167	5391	53403	518166	321	2	2470 / 2499 (98.8)	221 / 277 (80.1)	1 / 316 (0.3)

Figure 6: Experimental results. "Lines" is counted with the cloc program (http://cloc.sourceforge.net/). Size is computed separately for the application as downloaded, and for its reachable, analyzed portion, including any reachable libraries but not counting unreachable program and library methods. Dereferences are counted only in the the reachable application code (not in the 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 most important statistic is the number of @Raw annotations in the last column; nullness information is provided for context. The "plume progs" row reports the analysis of plume, as in a previous line, but with statistics projected over the 10 classes that have a main() method; see Section 5.3.2.

program	size (lines)	reachable program & libraries methods	lines	bytecodes	time (sec.) total	init.	dereferences safe / all (%)	inferred annotations @NonNull	@Raw
AFU light w/ Julia	13892	2597	26164	234823	86	1	2683 / 2725 (98.5)	340 / 405 (83.9)	10 / 553 (1.8)
AFU light w/ Nit	13892	?	?	?	10	?	3145 / 3887 (80.9)	316 / 502 (63.0)	63 / 502 (12.5)
JFlex w/ Julia	14987	3858	41134	385612	118	2	8624 / 8753 (98.5)	591 / 741 (79.8)	3 / 1109 (0.3)
JFlex w/ JastAdd	14987	?	?	?	3	?	? / ? (?)	389 / ? (?)	14 / ? (?)

Figure 7: Comparison of three inference tools: Julia, Nit and JastAdd. A "?" entry means that the tool does not output information needed to compute that entry.

As discussed in Section 1, our analysis is proved formally correct, modulo threading and user-defined class type parameters. (We have not proved Julia's implementation correct.) However, the Checker Framework still issues some warnings while type-checking Julia's results, because of the different perspective of the two tools. Julia is based on flow- and context-sensitive static analyses and abstract interpretation; the Checker Framework is based on type-checking, augmented by local flow-sensitivity and other enhancements [18].

Plume comes with 508 nullness or rawness manual annotations on 312 distinct lines, plus another 36 warning suppression annotations.[1] The default is @NonNull and @NonRaw (except for local variables, which are subject to type inference), and so only @Nullable and/or @Raw references are marked, which leads to fewer annotations overall. These manual annotations were checked by a pluggable type-checker built upon the Checker Framework [18]. It verified both their correctness and that there are no possible null dereferences.[2] In contrast, Julia reports 97 (= 8457 − 8360) possibly-unsafe dereferences and 1 rawness annotation in plume. To gain perspective on these (probably false) warnings and on Julia's strengths and weaknesses, we examined differences between the manual and Julia's annotations. We examined rawness annotations in all of plume, rawness and nullness annotations in a subset of plume, and rawness annotations in all of Daikon.

5.3.1 Rawness Comparison for All of Plume

We examined all rawness differences in plume. Plume contains 7 @Raw annotations and 3 (rawness) warning suppressions. By contrast, Julia's output contains only 1 @Raw annotation. Julia's annotation is on the receiver of MultiVersionControl.parseArgs(), which also appears in the manual annotation.

The 6 differing annotations are all weaknesses in the manual annotation. In other words, Julia's output is correct, and all these variables always hold fully-initialized values. As a result of Julia's analysis, the plume authors removed these extraneous @Raw annotations from plume.

[1] Except for this section, all of our experiments use a version of plume from which all nullness/rawness annotations and warning suppressions have been removed. Therefore, Julia always starts from a clean slate without any programmer assistance.

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

Manual		Julia	
error	weakness	error	weakness
6	18	0	26

Figure 8: Number of lines of diff output between manual annotations and Julia output, classified according to Section 5.3.2. In general, each difference results in two lines of diff output.

The warning suppressions in the manual annotations are examples of places where Julia's analysis is more precise than that of the type checker.

As an example of an analysis difference that accounted for most of the annotation differences, Julia knows that every Object is non-@Raw (because it has no fields to initialize), and Julia knows when casting an object to a subtype may result in a @Raw type. This is because Julia records the specific set of possibly-initialized fields for each value (see Section 4). By contrast, the type system treats rawness as a binary property, and requires a variable to be marked as @Raw if any of its subclasses may not yet be done initializing. This difference is relevant at the call from MultiVersionControl.main() to the Options constructor, for example. The manual annotations require warning suppression for that call.

In another case, the plume developers had temporarily inserted spurious @Raw annotations to work around a different type checker limitation related to inferring that an object can be (partly) initialized before its (superclass) constructor exits. The plume authors forgot to remove the annotations when the type checker was improved. Julia's output reminded the plume authors to remove those temporary annotations. An example was in FileIOException.getLineNumber().

5.3.2 Full Comparison for Programs in Plume

We examined all differences between the manual annotations and Julia's output, for a subset of plume. We used only a subset because the manual reasoning is so arduous (and doubly so for libraries with potentially arbitrary calling patterns). 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 over 30% of plume's lines of code. The last line of Figure 6 provides more measurements. We expected to find rawness differences, but did not; we briefly discuss the results anyway, as they yield insights into the strengths and weaknesses of Julia and manual annotations.

To compare the manual annotations to Julia's inference output, we ran the `diff` program. Running `diff` on the plume programs yields 193 lines of diff output (compared to 1489 lines of diff output for all of plume). 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 or subjectivity, we always use number of lines of diff output.

A technical report [23] analyzes every difference in detail. 140 out of 193 lines of diff output are uninteresting, for example because they are due to whitespace, annotations within method bodies (Julia only annotates signatures), or dead code. Figure 8 classifies each remaining line according to the following four categories:

Errors in manual annotations: 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`. Overall, Julia did not reveal any null pointer errors, only the 3 incorrect annotations. The plume authors have subsequently corrected these 3 errors by changing the annotation to `@Nullable`.

Weaknesses in manual annotations: The plume authors skipped annotating a few classes, such as tests. Julia's inference results would make the annotation task much easier.

Weaknesses in Julia output: Julia does not reason about the structure of regexps, some complex inter-procedural control flow, etc. The type-checker also suffers these weaknesses. When the manual annotations use `@NonNull` in such a situation, they also must suppress warnings.

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. (Rawness annotations are also required for any such verification.) 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 13 (= 26/2) annotations. The programmer must also make some other changes to accommodate differences between the type-checker and Julia, such as suppress some false positive warnings. This modest cost suggests that Julia's output is accurate and can be useful to programmers.

5.3.3 Rawness Comparison for Daikon

Similarly to Section 5.3.1, we compared manual and Julia's inferred rawness annotations for Daikon. Wherever there was a difference, Julia's annotation was correct and the manual one was incorrect. This process also revealed an error in the nullness/initialization type-checker.

6. 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, machine-checked, correct annotations, and these experiments confirm the accuracy of the analysis. This shows that a precise initialization analysis can provide, automatically, annotations that can be manipulated or type-checked by other tools and have similar quality as those written by hand.

REFERENCES

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

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

[3] J. Boyland, W. Retert, and Y. Zhao. Comprehending annotations on object-oriented programs using fractional permissions. In *IWACO*, pages 1–11, 2009.

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

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

[6] A. F. M. Engelen. Nullness analysis of Java source code. Master's thesis, University of Nijmegen Dept. of Computer Science, 2006.

[7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Programming*, 69(1–3):35–45, 2007.

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

[9] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350, 2007.

[10] C. Flanagan, R. Joshi, and K. R. M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 2(4):97–108, 2001.

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

[12] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *PASTE*, pages 9–14, 2007.

[13] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE*, pages 13–19, 2005.

[14] L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *FMOODS*, pages 132–149, 2008.

[15] 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.

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

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

[18] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, 2008.

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

[20] F. Spoto. Nullness analysis in boolean form. In *SEFM*, 2008.

[21] F. Spoto. The nullness analyser of Julia. In *LPAR*, 2010.

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

[23] F. Spoto and M. D. Ernst. Inference of field initialization. Technical Report UW-CSE-10-02-01, U. Wash. Dept. of Comp. Sci. & Eng., Seattle, WA, USA, 2010.

[24] F. Spoto, F. Mesnard, and É. Payet. A termination analyser for Java bytecode based on path-length. *ACM TOPLAS*, 32(3), 2010.

[25] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and immutability in generic Java. In *OOPSLA*, 2010.