# Formalisation and Implementation of an Algorithm for Bytecode Verification of @NonNull Types

Chris Male, David J. Pearce, Alex Potanin and Constantine Dymnikov [a]

[a]*School of Engineering and Computer Science,*
*Victoria University of Wellington, NZ*
{*malechri,djp,alex,dymnikkost*}*@ecs.vuw.ac.nz*

**Abstract**

Java's annotation mechanism allows us to extend its type system with non-null types. Checking such types cannot be done using the existing bytecode verification algorithm. We extend this algorithm to verify non-null types using a novel technique that identifies aliasing relationships between local variables and stack locations in the JVM. We formalise this for a subset of Java Bytecode and report on experiences using our implementation.

## 1 Introduction

The `NullPointerException` is a common kind of error arising in Java programs which occurs when references holding `null` are dereferenced. When this happens in a running program, it almost always indicates some kind of failure has occurred [7]. Eliminating by hand every possible `NullPointerException` in a given program is both tedious and error prone [37,36]. Likewise, identifying the cause of such a failure once it occurs is time consuming [8].

In this paper, we present a system which can ensure a program never throws a `NullPointerException`. Our system operates as an extension to the Java bytecode verifier. There are many advantages to this, compared with a system operating on Java source code directly:

- Our system is *language agnostic*. That is, the analysis works regardless of what source language was used. This is particularly important given the wide variety of programming languages which compile to Java bytecode.

- Our system can provide a *strong guarantee*. That is, if an application passes our bytecode verification process, we have a high degree of certainty it will never throw a `NullPointerException` (modulo some limitations discussed

in §5). Such a guarantee can be exploited by the JVM to employ more aggressive optimisation. For example, knowledge that a given sequence of instructions cannot throw an exception can lead to greater instruction level parallelism [39]. Likewise, whilst null-checks can be implemented efficiently on some architectures, they may still impose overheads on others —- thus, eliminating them offers potential performance benefits [42].

- Our system is *straightforward to implement*. This contrasts with approaches operating at the source code level, which must address the complex challenge of parsing and analysing Java source code.

To implement our system, we exploit Java's annotation mechanism, which allows annotations on types at the bytecode level. Thus, the presence of a `@NonNull` annotation on a method parameter or field indicates it cannot hold `null`. Likewise, a method whose return type is annotated with `@NonNull` can never return `null`.

## 1.1 Type Aliasing

Java Bytecodes have access to a fixed size local variable array and stack [48]. These act much like machine registers, in that they have no fixed type associated with them; rather, they can have different types at different program points. To address this, the standard bytecode verifier automatically infers the types of local variables and stack locations at each point within the program. The following shows a simple program, and the inferred types that hold immediately before each instruction:

```
static int f(Integer);    locals     stack
 0:   aload_0             [Integer] []
 1:   ifnull 8            [Integer] [Integer]
 4:   aload_0             [Integer] []
 5:   invokevirtual ...   [Integer] [Integer]
 8:   return              [Integer] []
```

Here, there is one local variable at index 0. On method entry, this is initialised with the `Integer` parameter. The `aload_0` instruction loads the local variable at index 0 onto the stack and, hence, the `Integer` type is inferred for that stack location.

A bytecode verifier for non-null types must infer that the value loaded onto the stack immediately before the `invokevirtual` method call cannot be `null`, as this is the call's receiver. The challenge here is that `ifnull` compares the top of the stack against `null`, but then discards this value. Thus, the bytecode verifier must be aware that, at that exact moment, the top of the stack and local variable 0 are aliases. The algorithm used by the standard bytecode verifier is unable to do this. Therefore, we extend this algorithm to maintain information about such aliases, and we refer to this technique as *type aliasing*.

2

*1.2   Contributions*

This paper makes the following contributions:

(1) We formalise our non-null bytecode verifier for a subset of Java Bytecode.

(2) We detail an implementation of our system for Java Bytecode.

(3) We report on our experiences with using our system on real-world programs.

While considerable previous work on non-null types exists (e.g. [57,25,38,12,22]), none has directly addressed the problem of bytecode verification. While these existing techniques could be used for this purpose, they operate on higher-level program representations and must first translate bytecode into their representation. This introduces unnecessary overhead that is undesirable for the (performance critical) bytecode verifier. Our technique operates on bytecode directly, thus eliminating this inefficiency. Furthermore, no previous work has presented a formalised system for checking non-null types and demonstrated it on real-world benchmarks.

Finally, an earlier version of this paper was presented at the *Conference on Compiler Construction (CC'08)* [49]. We present here an extended version of this paper, which includes a much more complete formalisation, accompanying proofs and discussion of many implementation issues omitted previously.

## 2   Simple Java Virtual Machine (Sᴊᴠᴍ)

Before presenting the formalism of our non-null verification algorithm, we first discuss the simplified virtual machine (called the Sᴊᴠᴍ) which we target. Several existing formalisms of the proper JVM can be found in the literature (see e.g. [58,6,46]). In our case, we wish to elide details of the JVM which are not relevant as much as possible.

The Sᴊᴠᴍ is much simpler than the proper JVM, but retains those characteristics important to us. In particular, the main simplifications are:

- **Data Types**. The only data types used in the Sᴊᴠᴍ are the following:
$$T \ ::= \ C \mid \texttt{int} \mid null$$
Here, $C$ represents a class reference, and $null$ is the special type given to the null value. Thus, the Sᴊᴠᴍ only supports object references, and primitive integers (i.e. int). Likewise, the Sᴊᴠᴍ has no notion of generic types (although we return to discuss these in §5.6).

- **Constructors**. There is no notion of constructor in the SJVM. Instead, every field is required to have an appropriate initialiser which consists either of `null` or an explicit object creation (e.g. **new** `String()`). The reasons for this requirement will become more evident later on in §5.3, when we discuss the problems caused by constructors in Java.

- **Exceptions**. There is no notion of an exception in the SJVM. Instead, if the machine is unable to continue (e.g. because of a `null` dereference), then it is *stuck*. The lack of exceptions simplifies the analysis of intra-procedural control-flow. However, it is very straightforward to extend our formalism to deal with exceptional flow (see e.g. [16,50,43]), and our implementation handles this correctly.

- **Static Methods / Fields**. The SJVM does not support static methods or fields. While it is very easy to extend our formalism to static methods, the problem of static field initialisers is more challenging (see §5.3 for more on this).

- **Interfaces**. The SJVM does not support interfaces. This simplification ensures that the type hierarchy forms a complete lattice (see §3.1 for more on this).

- **Bytecodes**. The SJVM has a smaller bytecode instruction set than the JVM (see Figure 1 for the list of bytecodes supported in the SJVM). This helps keep our formalism compact. Furthermore, in most cases, it is straightforward to see how one would deal with those bytecodes not considered. Again, our implementation does support the full JVM bytecode instruction set (except `jsr` — see §5.1).

- **Local Variable Array / Stack**. In the JVM, each method has a fixed-size local variable array (for storing local variables) and a stack of known maximum depth (for storing temporary values). In contrast, the SJVM provides an infinite number of slots in the local variable array of a method, and imposes no limit on the maximum stack height.

A concrete state in the SJVM is a pair of the form $(\Sigma, \Pi)$, where $\Sigma$ models the program heap and $\Pi$ models the call-stack. Here, $\Pi$ is a stack of tuples of the form $(\delta^{\mathsf{M}}, \mathrm{pc}, \Gamma, \kappa)$, where $\delta^{\mathsf{M}}$ identifies a method, pc models its program counter, $\Gamma$ models its local variable array and stack and, finally, $\kappa$ models the method's stack pointer and identifies the first empty location on the stack. To manipulate $\Pi$, we pattern match using concatenation. Hence, if $\Pi = H :: T$, then $H$ is the head (i.e. topmost tuple) of $\Pi$, whilst $T$ is its tail (i.e. remaining tuples). The local variable array, $\Gamma$, is a map from locations to values. In particular, locations are labelled consecutively with integers starting from $0$, with the first $m$ locations representing the local array and the remainder representing the stack. A *value* is either an integer, `null`, or a valid object identifier $\rho$. A *method descriptor* uniquely identifies a method within the program, including its owning class, name, parameter and return types. Method descriptors have the form $\delta^{\mathsf{M}} = (O, N, (T_1, \ldots, T_n) \rightarrow T, m)$ where,

| Instruction | Effect on Stack / Description |
|---|---|
| `load` $i$ | $[\ldots] \Rightarrow [\ldots, ref]$ <br> Load reference from local variable $i$ onto stack |
| `store` $i$ | $[\ldots, ref] \Rightarrow [\ldots]$ <br> Pop reference off stack and store in local variable $i$ |
| `loadconst` $c$ | $[\ldots] \Rightarrow [\ldots, c]$ <br> Load constant $c$ onto stack (either `null` or an `int`). |
| `pop` | $[\ldots, val] \Rightarrow [\ldots]$ <br> Pop value off stack. |
| `getfield` $\delta^{\text{F}}$ | $[\ldots, ref] \Rightarrow [\ldots, value]$ <br> Load value from field identified by $\delta^{\text{F}}$ in object referenced by $ref$. |
| `putfield` $\delta^{\text{F}}$ | $[\ldots, ref, value] \Rightarrow [\ldots]$ <br> Pop value off stack and write to field identified by $\delta^{\text{F}}$ in object referenced by $ref$ |
| `invoke` $\delta^{\text{M}}$ | $[\ldots, ref, p_1, \ldots, p_n] \Rightarrow [\ldots, value]$ <br> Invoke method identified by $\delta^{\text{M}}$ on object referenced by $ref$. |
| `new` $T$ | $[\ldots] \Rightarrow [\ldots, ref]$ <br> Construct new object of type $T$ and place reference on stack. |
| `return` | $[\ldots, ref] \Rightarrow [\ldots]$ <br> Return from method using $ref$ as return value. |
| `ifceq` $dest$ | $[\ldots, v_1, v_2] \Rightarrow [\ldots]$ <br> Branch to $dest$ if $v_1$ and $v_2$ have the same value. |
| `goto` $dest$ | $[\ldots] \Rightarrow [\ldots]$ <br> Branch unconditionally to $dest$ |

Fig. 1. The SJVM bytecode instruction set.

$$\text{store } i : \Big(\Sigma, (\delta^{\text{M}}, \text{pc}, \Gamma, \kappa) {::} S\Big) \longrightarrow \Big(\Sigma, (\delta^{\text{M}}, \text{pc}{+}1, \Gamma[i {\mapsto} \Gamma(\kappa{-}1)], \kappa{-}1) {::} S\Big)$$

$$\text{load } i : \Big(\Sigma, (\delta^{\text{M}}, \text{pc}, \Gamma, \kappa) {::} S\Big) \longrightarrow \Big(\Sigma, (\delta^{\text{M}}, \text{pc}{+}1, \Gamma[\kappa {\mapsto} \Gamma_1(i)], \kappa{+}1) {::} S\Big)$$

$$\text{loadconst } c : \Big(\Sigma, (\delta^{\text{M}}, \text{pc}, \Gamma, \kappa) {::} S\Big) \longrightarrow \Big(\Sigma, (\delta^{\text{M}}, \text{pc}{+}1, \Gamma[\kappa {\mapsto} c], \kappa{+}1) {::} S\Big)$$

$$\text{pop} : \Big(\Sigma, (\delta^{\text{M}}, \text{pc}, \Gamma, \kappa) {::} S\Big) \longrightarrow \Sigma, (\delta^{\text{M}}, \text{pc}{+}1, \Gamma, \kappa{-}1) {::} S\Big)$$

$$\frac{(\rho, \Sigma_2) = \mathbf{init}(T, \Sigma_1)}{\text{new } T : \Big(\Sigma_1, (\delta^{\text{M}}, \text{pc}, \Gamma, \kappa) {::} S\Big) \longrightarrow \Big(\Sigma_2, (\delta^{\text{M}}, \text{pc}{+}1, \Gamma[\kappa \mapsto \rho], \kappa{+}1) {::} S\Big)}$$

$$\frac{\psi_1 = \Gamma_1(\kappa{-}1) \quad \psi_1 \neq \text{null} \quad \delta^{\text{F}} = (O, N, T) \quad \psi_2 = \mathbf{read}(N, \psi_1, \Sigma)}{\text{getfield } \delta^{\text{F}} : \Big(\Sigma, (\delta^{\text{M}}, \text{pc}, \Gamma, \kappa) {::} S\Big) \longrightarrow \Big(\Sigma, (\delta^{\text{M}}, \text{pc}{+}1, \Gamma[\kappa{-}1 \mapsto \psi_2], \kappa) {::} S\Big)}$$

$$\frac{\psi_1 = \Gamma(\kappa{-}1)) \quad \psi_2 = \Gamma(\kappa{-}2) \quad \psi_2 \neq \text{null} \quad \delta^{\text{F}} = (O, N, T) \quad \Sigma_2 = \mathbf{write}(\psi_1, N, \psi_2, \Sigma_1)}{\text{putfield } \delta^{\text{F}} : \Big(\Sigma_1, (\delta^{\text{M}}, \text{pc}, \Gamma, \kappa) {::} S\Big) \longrightarrow \Big(\Sigma_2, (\delta^{\text{M}}, \text{pc}{+}1, \Gamma, \kappa{-}2) {::} S\Big)}$$

$$\frac{\psi_0 = \Gamma_1(\kappa_1{-}(n+1)) \quad \psi_0 \neq \text{null} \quad \psi_1 = \Gamma_1(\kappa_1{-}n), \ldots, \psi_n = \Gamma_1(\kappa_1{-}1)}{\delta^{\text{M}}_2 = (O, M, (P_1, \ldots, P_n) \to T_r, \kappa_2) \quad \Gamma_2 = \{0 {\mapsto} \psi_0, 1 {\mapsto} \psi_1, \ldots, n {\mapsto} \psi_n\}}{\text{invoke } \delta^{\text{M}}_2 : \Big(\Sigma, (\delta^{\text{M}}_1, \text{pc}, \Gamma_1, \kappa_1) {::} S\Big) \longrightarrow \Big(\Sigma, (\delta^{\text{M}}_2, 0, \Gamma_2, \kappa_2) {::} (\delta^{\text{M}}_1, \text{pc}, \Gamma_1, \kappa_1 {-} (n{+}1)) {::} S\Big)}$$

$$\frac{\psi_0 = \Gamma_1(\kappa_1{-}1) \quad \Gamma_3 = \Gamma_2[\kappa_2 \mapsto \psi_0]}{\text{return} : \Big(\Sigma, (\delta^{\text{M}}_1, \text{pc}_1, \Gamma_1, \kappa_1) {::} (\delta^{\text{M}}_2, \text{pc}_2, \Gamma_2, \kappa_2) {::} S\Big) \longrightarrow \Big(\Sigma, (\delta^{\text{M}}_2, \text{pc}_2{+}1, \Gamma_3, \kappa_2{+}1) {::} S\Big)}$$

$$\frac{\psi_1 = \Gamma(\kappa_1{-}1) \quad \psi_2 = \Gamma(\kappa_1{-}2) \quad \psi_1 = \psi_2}{\text{ifceq } i : \Big(\Sigma, (\delta^{\text{M}}, \text{pc}, \Gamma, \kappa) {::} S\Big) \longrightarrow \Big(\Sigma, (\delta^{\text{M}}, \text{pc}{+}i, \Gamma, \kappa{-}2) {::} S\Big)}$$

$$\frac{\psi_1 = \Gamma(\kappa_1{-}1) \quad \psi_2 = \Gamma(\kappa_1{-}2) \quad \psi_1 \neq \psi_2}{\text{ifceq } i : \Big(\Sigma, (\delta^{\text{M}}, \text{pc}, \Gamma, \kappa) {::} S\Big) \longrightarrow \Big(\Sigma, (\delta^{\text{M}}, \text{pc}{+}1, \Gamma, \kappa{-}2) {::} S\Big)}$$

$$\text{goto } i : \Big(\Sigma, (\delta^{\text{M}}, \text{pc}, \Gamma, \kappa) {::} S\Big) \longrightarrow \Big(\Sigma, (\delta^{\text{M}}, \text{pc}{+}i, \Gamma, \kappa) {::} S\Big)$$

Fig. 2. Operational semantics for the SJVM bytecode instruction set.

as before, $m$ identifies the number of locations representing the local array. Similarly, a *field descriptor* $\delta^F = (O, N, T)$ identifies the owning class, field name and type.

The *program heap*, $\Sigma$, is a map from object identifiers, $\rho$, to object descriptors. An *object descriptor* is a map from field names to their values, where the special field $ gives the object's runtime type. For example, consider the following class:

```
class Tst { int x = 3;  Tst y = null; }
```

Then, $\{\$ \mapsto \text{Tst}, x \mapsto 3, y \mapsto \text{null}\}$ is the initial object descriptor for an instance of `Tst`. Method bodies are simply an array of bytecode instructions, which differs from the JVM where they are arrays of *bytes*. This is a useful simplification, since it allows us to ignore the fact that bytecodes come in varying lengths.

The operational semantics for the SJVM bytecodes are given as transitions of the form: $(\Sigma_1, \Pi_1) \longrightarrow (\Sigma_2, \Pi_2)$. Here, $\Sigma_2$ and $\Pi_2$ represent the (possibly updated) program heap and call stack. Figure 2 gives the operational semantics for all SJVM bytecodes. Here, $\Gamma[i \mapsto \psi]$ returns a new local array identical to $\Gamma$, except with location $i$ holding value $\psi$. Also, **read**$(N, \psi, \Sigma)$ returns the value of the field named $N$ in the object referred to by $\psi$ in the program heap $\Sigma$. Similarly, **write**$(\psi_1, N, \psi_2, \Sigma)$ returns a heap where field $N$ in the object referred to by $\psi_2$ now has value $\psi_1$. In both cases, the field must exist for the operation to succeed (i.e. if the field does not exist, the machine is stuck). Finally, the method **init**$(T, \Sigma_1)$ returns a pair $(\rho, \Sigma_2)$, where $\Sigma_1 \subset \Sigma_2$ (hence, all objects in $\Sigma_1$ appear unchanged in $\Sigma_2$) and $\rho \in dom(\Sigma_2)$ (hence $\rho$ refers to a fresh object). Furthermore, the object referred to by $\rho$ may itself contain references to fresh objects (for those fields initialised with object creations). As an example, consider the following bytecodes (which correspond with the given Java source):

```
Tst.m : (int)⟶Object          class Tst {
0:   load 0                      int x;
1:   getfield Tst.x              Object y;
2:   load 1
3:   ifceq 7                     Object m(int z) {
4:   load 0                       if(x != z) { y = null; }
5:   loadconst null              }
6:   putfield Tst.y             }
7:   return
```

Here, we can see that, on entry to `m(int)`, the **this** reference is held in location 0, whilst the first parameter is held in location 1. Then, the following is a valid execution trace for this method, given a program heap containing a single instance of `Tst` and a value of 4 passed in for parameter z:

$$\left( \left\{ \rho \mapsto \{ \$ \mapsto \mathtt{Tst}, x \mapsto 3, y \mapsto \mathtt{null} \} \right\}, (\delta^{\mathrm{M}}, 0, \{0 \mapsto \rho, 1 \mapsto 4\}, 2) :: \emptyset \right)$$

$$\hookrightarrow \left( \left\{ \rho \mapsto \{ \$ \mapsto \mathtt{Tst}, x \mapsto 3, y \mapsto \mathtt{null} \} \right\}, (\delta^{\mathrm{M}}, 1, \{0 \mapsto \rho, 1 \mapsto 4, 2 \mapsto \rho\}, 3) :: \emptyset \right)$$

$$\hookrightarrow \left( \left\{ \rho \mapsto \{ \$ \mapsto \mathtt{Tst}, x \mapsto 3, y \mapsto \mathtt{null} \} \right\}, (\delta^{\mathrm{M}}, 2, \{0 \mapsto \rho, 1 \mapsto 4, 2 \mapsto 3\}, 3) :: \emptyset) \right)$$

$$\hookrightarrow \left( \left\{ \rho \mapsto \{ \$ \mapsto \mathtt{Tst}, x \mapsto 3, y \mapsto \mathtt{null} \} \right\}, (\delta^{\mathrm{M}}, 3, \{0 \mapsto \rho, 1 \mapsto 4, 2 \mapsto 3, 3 \mapsto 4\}, 4) :: \emptyset) \right)$$

$$\hookrightarrow \left( \left\{ \rho \mapsto \{ \$ \mapsto \mathtt{Tst}, x \mapsto 3, y \mapsto \mathtt{null} \} \right\}, (\delta^{\mathrm{M}}, 4, \{0 \mapsto \rho, 1 \mapsto 4\}, 2) :: \emptyset) \right)$$

$$\hookrightarrow \left( \left\{ \rho \mapsto \{ \$ \mapsto \mathtt{Tst}, x \mapsto 3, y \mapsto \mathtt{null} \} \right\}, (\delta^{\mathrm{M}}, 5, \{0 \mapsto \rho, 1 \mapsto 4, 2 \mapsto \rho\}, 3) :: \emptyset \right)$$

$$\hookrightarrow \left( \left\{ \rho \mapsto \{ \$ \mapsto \mathtt{Tst}, x \mapsto 3, y \mapsto \mathtt{null} \} \right\}, (\delta^{\mathrm{M}}, 6, \{0 \mapsto \rho, 1 \mapsto 4, 2 \mapsto \rho, 3 \mapsto \mathtt{null}\}, 4) :: \emptyset) \right)$$

$$\hookrightarrow \left( \left\{ \rho \mapsto \{ \$ \mapsto \mathtt{Tst}, x \mapsto 3, y \mapsto \mathtt{null} \} \right\}, (\delta^{\mathrm{M}}, 7, \{0 \mapsto \rho, 1 \mapsto 4\}, 2) :: \emptyset) \right)$$

Considering this execution trace, we see that the stack pointer, $\kappa$, always refers to the first empty location on the stack.

In Figure 2, we have implicitly assumed that the bytecode being executed corresponds to that determined by the method and program counter for the topmost calling context. Also, there is no consideration of type safety — that is, the SJVM does not check whether, for example, a reference is provided as the receiver for a `getfield` bytecode. The reason for this is simply that, since this issue is orthogonal, we assume any program executing in the SJVM has already passed a bytecode verification stage equivalent to that found in the JVM.

We can now give a suitable definition for an execution trace of the SJVM:

**Definition 1** *An* execution trace $(\Sigma_1, \Pi_1) \rightsquigarrow (\Sigma_2, \Pi_2)$ *is a sequence of one or more state transitions, as defined by the rules of Figure 2, which transform an initial state* $(\Sigma_1, \Pi_1)$ *into a final state* $(\Sigma_2, \Pi_2)$.

One issue remaining here, is what a "terminating" execution trace looks like. In particular, the observant reader will notice the rule for `return` statements requires at least two stack frames. Thus, execution states with only one stack frame (such as for our example above) can never return. In some sense, this reflects the fact that we do not know what called the programs' `main` method — so we cannot return to it. From our perspective, we are not concerned with termination of execution traces, and we will simply not distinguish normal termination from being stuck.

## 3   Non-null Type Verification

We now present the formalisation of our non-null verification algorithm. The algorithm infers the nullness of local variables at each point within a method. We assume that method parameters, return types and fields are annotated with `@NonNull` (where appropriate) by the programmer. Our algorithm is intraprocedural; that is, it

concentrates on verifying each method in isolation, rather than the whole program together. The algorithm constructs an abstract representation of each method's execution; if this is possible, then the method is type safe and will never dereference `null`. The abstract representation of a method mirrors the control-flow graph (CFG): its nodes contain an abstract representation of the program store, called an *abstract store*, giving the types of local variables and stack locations at that point; its edges represent the effects of the instructions.

For simplicity, we formalise our algorithm for the SJVM bytecode instruction set, rather than for the full JVM. However, in practice, our implementation operates on the full JVM bytecode instruction set and supports almost all of its features, including Java Generics. Details of our implementation which are not considered by the formalisation can be found in §5.

## 3.1 Abstract Types

Our non-null verification algorithm uses *abstract types* to encode information about whether a given variable or field is allowed to hold `null`. These (roughly speaking) extend those types found in the SJVM, and allow references to be declared as non-null (in §5.6 we extend this to Java Generics). For example:

```
Vector v1;
@NonNull Vector v2;
```

Here, `v1` is a *nullable* reference (one that may be `null`), while `v2` is a *non-null* reference (one that may not be `null`). The abstract types used in our system are:

$$\alpha \; ::= \; @\texttt{NonNull} \mid \epsilon$$
$$\hat{T} \; ::= \; \alpha \, C \mid null \mid \bot$$

Here, the special *null* type is given to the `null` value, $\epsilon$ denotes the absence of a `@NonNull` annotation, $C$ denotes a class name (e.g. `Integer`) and $\bot$ is given to locations which hold no value (e.g. they are uninitialised, in dead code, etc). The reader may notice the absence of the `int` type and, in fact, our analysis effectively ignores this altogether since it has no bearing on the problem.

A formal definition of the subtype relation for our non-null types is given in Figure 3. An important property of our subtype relation is that it forms a *complete lattice* (i.e. that every pair of types $\hat{T}_1, \hat{T}_2$ has a unique least upper bound, $\hat{T}_1 \sqcup \hat{T}_2$, and a unique greatest lower bound, $\hat{T}_1 \sqcap \hat{T}_2$). This helps ensure termination of our algorithm. This property holds only because the SJVM does not support Java `interfaces` (see §5.2 for more on this).

9

$$\frac{}{\texttt{@NonNull} \leq \epsilon} \qquad\qquad\qquad \text{(S-NONNULL)}$$

$$\frac{C \texttt{ extends } B}{\alpha\,C\ \leq \alpha\,B} \qquad \frac{\alpha_1 \leq \alpha_2}{\alpha_1\,C\ \leq \alpha_2\,C} \qquad\qquad \text{(S-CLASSa, S-CLASSb)}$$

$$\frac{}{\bot \leq \alpha\,C} \qquad \frac{}{\bot \leq \mathit{null}} \qquad \frac{}{\mathit{null} \leq C} \qquad \text{(S-BOTa, S-BOTb, S-NULL)}$$

Fig. 3. Subtyping rules for the abstract types used in our system. We assume reflexivity and transitivity, that `java.lang.Object` is the class hierarchy root and, hence, is also $\top$.

Finally, since those types used by the SJVM do not include non-null information (recall §2), we need some way to identify those fields, parameters and return types which the programmer has annotated with `@NonNull`. Therefore, we employ two conversion maps, $\Delta_F$ and $\Delta_M$, for this purpose. Here, $\Delta_F$ accepts a field descriptor, and returns the (programmer declared) abstract type; likewise, $\Delta_M$ accepts a method descriptor, and returns abstract types for the parameters and return. This reflects what happens in the real JVM, where bytecodes do not themselves incorporate information about annotations; rather, the information is stored separately and must be looked up using the descriptor provided in the bytecode.

### 3.2 Abstract Store

Our system models the state of the SJVM before a given bytecode instruction in any possible execution trace using an *abstract store*. We formalise this as $(\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa})$, where $\hat{\Sigma}$ is the *abstract heap*, $\hat{\Gamma}$ is the *abstract location array* and $\hat{\kappa}$ is the *stack pointer* which identifies the first free location on the stack. Here, $\hat{\Gamma}$ maps *abstract locations* to *type references*. These abstract locations are labelled $0, \ldots, n-1$, with the first $m$ locations representing the local variable array, and the remainder representing the stack (hence, $n-m$ is the maximum stack size and $m \leq \hat{\kappa} \leq n$). A type reference is a reference to a *type object* which, in turn, can be thought of as an abstract type with identity. Thus, we can have two distinct type objects representing the same abstract type. This is analogous to the situation with normal (Java) objects, where distinct objects can (by coincidence) have the same values for their fields (where "distinct" generally means: reside at different memory locations). Furthermore, as with normal objects, we can have aliasing of type objects and this is crucial to our system. For example, in the following abstract store, locations $0$ and

2 are type aliases:

$$\hat{\Sigma} = \{r_1 \mapsto \texttt{@NonNull Integer}, r_2 \mapsto \texttt{String}\},$$

$$\hat{\Gamma} = \{0 \mapsto r_1, 1 \mapsto r_2, 2 \mapsto r_1\}, \hat{\kappa} = 3$$

Here, the abstract heap, $\hat{\Sigma}$, maps type references to types. Thus, $\hat{\Sigma}$ can be thought of as a very coarse abstraction of the SJVM's program heap. Obviously, it does not make sense for $\hat{\Gamma}$ to use references which don't exist in $\hat{\Sigma}$. This notion of a *well-formed* abstract store is stated formally as follows:

**Definition 2 (Well-formed Store)** *An abstract store* $(\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa})$ *is well-formed iff* $dom(\hat{\Gamma}) = \{0, \ldots, n-1\}$ *for some* $n$, $ran(\hat{\Gamma}) \subseteq dom(\hat{\Sigma})$ *and* $0 \le \hat{\kappa} \le n$.

*3.3 Abstract Semantics*

The effect of a bytecode instruction is given by its *abstract semantics*, which we describe using transition rules. These summarise the abstract store immediately after the instruction in terms of the abstract store immediately before it; any necessary constraints on the abstract store immediately before the instruction are also identified.

The abstract semantics for the SJVM bytecode instruction set are given in Figure 4. Here, $\hat{\Gamma}[r_1/r_2]$ generates an abstract store from $\hat{\Gamma}$ where all abstract locations holding $r_1$ now hold $r_2$. The helper function **thisMethT**() gives the type of the enclosing method. Also, recall from §3.1 that $\Delta_F$ and $\Delta_M$ respectively return the appropriate (programmer declared) abstract type(s) for a given field or method.

As with the operational semantics of SJVM bytecodes (recall Figure 2), our abstract semantics does not contain certain checks that are already performed during bytecode verification (for example, that an array is only indexed by an integer). Also, observe that for the `loadconst` bytecode, the type *null* is always placed on the stack, regardless of whether an integer or `null` constant is actually loaded. This simplification is useful since we have no explicit `int` type, and is safe since integer values are of no concern to our system.

A useful illustration of our semantics is the `putfield` bytecode. This requires the value to be assigned on top of the stack, followed by the object reference itself. Looking at the `putfield` rule, we see $\kappa$ decreases by two, indicating the net effect is two less elements on the stack (which corresponds with its operational semantics from Figure 2). The abstract value to be assigned is found in location $\kappa - 1$, which represents the top of the stack. The object reference comes after this on the stack (recall Figure 1). A constraint is given to ensure this is non-null, thus protecting against a dereference of `null`.

$$\texttt{store}\; i : \left(\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa}\right) \longrightarrow \hat{\Sigma}, \hat{\Gamma}[i \mapsto \hat{\Gamma}(\hat{\kappa}-1)], \hat{\kappa}-1$$

$$\texttt{load}\; i : \left(\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa}\right) \longrightarrow \left(\hat{\Sigma}, \hat{\Gamma}[\hat{\kappa} \mapsto \hat{\Gamma}(i)], \hat{\kappa}+1\right)$$

$$\frac{r \notin dom(\hat{\Sigma}_1) \quad \hat{\Sigma}_2 = \hat{\Sigma}_1 \cup \{r \mapsto null\}}{\texttt{loadconst}\; c : \left(\hat{\Sigma}_1, \hat{\Gamma}, \hat{\kappa}\right) \longrightarrow \left(\hat{\Sigma}_2, \hat{\Gamma}[\hat{\kappa} \mapsto r], \hat{\kappa}+1\right)}$$

$$\texttt{pop} : \left(\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa}\right) \longrightarrow \left(\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa}-1\right)$$

$$\frac{r \notin dom(\hat{\Sigma}_1) \quad \hat{\Sigma}_2 = \hat{\Sigma}_1 \cup \{r \mapsto \texttt{@NonNull}\, T\}}{\texttt{new}\; T : \left(\hat{\Sigma}_1, \hat{\Gamma}, \hat{\kappa}\right) \longrightarrow \left(\hat{\Sigma}_2, \hat{\Gamma}[\hat{\kappa} \mapsto r], \hat{\kappa}+1\right)}$$

$$\frac{\hat{\Sigma}_1(\hat{\Gamma}(\hat{\kappa}-1)) = \texttt{@NonNull}\, C \quad r \notin dom(\hat{\Sigma}) \quad \hat{T} = \Delta_F(\delta^{\mathsf{F}}) \quad \hat{\Sigma}_2 = \hat{\Sigma}_1 \cup \{r \mapsto \hat{T}\}}{\texttt{getfield}\; \delta^{\mathsf{F}} : \left(\hat{\Sigma}_1, \hat{\Gamma}, \hat{\kappa}\right) \longrightarrow \left(\hat{\Sigma}_2, \hat{\Gamma}[\hat{\kappa}-1 \mapsto r], \hat{\kappa}\right)}$$

$$\frac{\hat{\Sigma}(\hat{\Gamma}(\hat{\kappa}-1)) = \hat{T}_1 \quad \hat{\Sigma}(\hat{\Gamma}(\hat{\kappa}-2)) = \texttt{@NonNull}\, C \quad \hat{T}_2 = \Delta_F(\delta^{\mathsf{F}}) \quad \hat{T}_1 \leq \hat{T}_2}{\texttt{putfield}\; \delta^{\mathsf{F}} : \left(\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa}\right) \longrightarrow \left(\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa}-2\right)}$$

$$\frac{\begin{array}{c}\Delta_M(\delta^{\mathsf{M}}) = (\hat{T}_{P1}, \ldots, \hat{T}_{Pn}) \to \hat{T}_r) \\ \hat{\Sigma}_1(\hat{\Gamma}(\hat{\kappa}-n)), \ldots, \hat{\Sigma}_1(\hat{\Gamma}(\hat{\kappa}-1)) = \hat{T}_1, \ldots, \hat{T}_n \quad \hat{\Sigma}_1(\hat{\Gamma}(\hat{\kappa}-(n+1))) = \texttt{@NonNull}\, C \\ r \notin dom(\hat{\Sigma}_1) \quad \hat{\Sigma}_2 = \hat{\Sigma}_1 \cup \{r \mapsto \hat{T}_r\} \quad \hat{T}_1 \leq \hat{T}_{P1}, \ldots, \hat{T}_n \leq \hat{T}_{Pn}\end{array}}{\texttt{invoke}\; \delta^{\mathsf{M}} : \left(\hat{\Sigma}_1, \hat{\Gamma}, \hat{\kappa}\right) \longrightarrow \left(\hat{\Sigma}_2, \hat{\Gamma}[\hat{\kappa}-(n+1) \mapsto r], \hat{\kappa}-n\right)}$$

$$\frac{(\hat{T}_{P1}, \ldots, \hat{T}_{Pn}) \to \hat{T}_r = \text{thisMethT}() \quad \hat{\Sigma}(\hat{\Gamma}(\hat{\kappa}-1)) = \hat{T} \quad \hat{T} \leq \hat{T}_r}{\texttt{return} : \left(\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa}\right) \longrightarrow \left(\emptyset, \emptyset, 0\right)}$$

$$\frac{\begin{array}{c}r_1 = \hat{\Gamma}(\hat{\kappa}-2) \quad r_2 = \hat{\Gamma}(\hat{\kappa}-1) \\ \hat{\Sigma}_1(r_1) = \hat{T}_1 \quad \hat{\Sigma}_1(r_2) = \hat{T}_2 \quad r_3 \notin dom(\hat{\Sigma}_1) \quad \hat{\Sigma}_2 = \hat{\Sigma}_1 \cup \{r_3 \mapsto \hat{T}_1 \sqcap \hat{T}_2\}\end{array}}{\texttt{ifceq} : \left(\hat{\Sigma}_1, \hat{\Gamma}, \hat{\kappa}\right) \xrightarrow{true} \left(\hat{\Sigma}_2, \hat{\Gamma}[r_1/r_3, r_2/r_3], \hat{\kappa}-2\right)}$$

$$\frac{\begin{array}{c}r_1 = \hat{\Gamma}(\hat{\kappa}-2) \quad r_2 = \hat{\Gamma}(\hat{\kappa}-1) \\ \hat{\Sigma}_1(r_1) = \hat{T}_1 \quad \hat{\Sigma}_1(r_2) = \hat{T}_2 \quad r_3, r_4 \notin dom(\hat{\Sigma}) \quad \hat{\Sigma}_2 = \hat{\Sigma}_1 \cup \{r_3 \mapsto \hat{T}_1 - \hat{T}_2, r_4 \mapsto \hat{T}_2 - \hat{T}_1\}\end{array}}{\texttt{ifceq} : \left(\hat{\Sigma}_1, \hat{\Gamma}, \hat{\kappa}\right) \xrightarrow{false} \left(\hat{\Sigma}_2, \hat{\Gamma}[r_1/r_3, r_2/r_4], \hat{\kappa}-2\right)}$$

$$\texttt{goto} : \left(\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa}\right) \longrightarrow \left(\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa}\right)$$

Fig. 4. Abstract semantics for the SJVM bytecode instruction set.

Considering the remaining rules from Figure 4, the main interest lies with `ifceq`. There is one rule for each of the true/false branches. The true branch uses the greatest lower bound operator, $\hat{T}_1 \sqcap \hat{T}_2$ (recall §3.1). This creates a single type object which is substituted for both operands to create a type aliasing relationship. For the false branch, a special *difference* operator, $\hat{T}_1 - \hat{T}_2$, is employed which is similar to set difference. For example, the set of possible values for a variable `o` of type `Object` includes all instances of `Object` (and its subtypes), as well as `null`; so, if the condition `o!=null` is true, then `null` is removed from this set. Thus, it is defined as follows:

**Definition 3** $\hat{T}_1 - \hat{T}_2$ *is* `@NonNull` $\hat{T}$, *if* $\hat{T}_1 = \alpha \; \hat{T} \wedge \hat{T}_2 = null$, *and* $\hat{T}_1$ *otherwise.*

The semantics for the `return` bytecode indicate that: firstly, we always expect a return value (for simplicity); and, secondly, no bytecode can follow it in the CFG.

*3.4 An Illustrative Example*

Recall our non-null verification algorithm constructs an abstract representation of a method's execution. This corresponds to an annotated CFG whose nodes represent the bytecode instructions and edges the transitions described by our abstract semantics. Each node is associated with an abstract program store, $(\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa})$, giving the types of local variables immediately before that instruction. The idea is that, if this representation of a method can be constructed, such that all constraints implied by our abstract semantics are resolved, the method is type safe and cannot dereference `null`.

Figure 5 illustrates the bytecode instructions for a simple method and its corresponding abstract representation (note, in §3.5 we will detail the exact mechanism by which abstract stores are merged). When a method is called, the local variable array is initialised with the values of the incoming parameters, starting from $0$ and using as many as necessary; for instance methods, the first parameter is always the `this` reference. Thus, the first abstract location of the first store in Figure 5 has type `@NonNull Test`; the remainder have nullable type `Integer`, with each referring to a unique type object. Notice here, that we are conservatively assuming parameters are not aliased on entry. If we did not do this, the analysis might incorrectly retype a parameter location.

In Figure 5, the effect of each instruction is reflected in the changes between the abstract stores before and after it. Of note are the two `ifceq` instructions: the first establishes a type aliasing relationship between locations $1$ and $2$ (on the true branch); the second causes a retyping of location $1$ to `@NonNull Integer` (on the false branch) which also retypes location $2$ through type aliasing. Thus, at the `invoke` instruction, the top of the stack (which represents the receiver reference) holds `@NonNull Integer`, indicating it will not dereference `null`.

13

```
class Test {
 String f(Integer i, Integer j) {
  if(i==j && i!=null) {
   return j.toString();
  } else { return null;}
}}
```
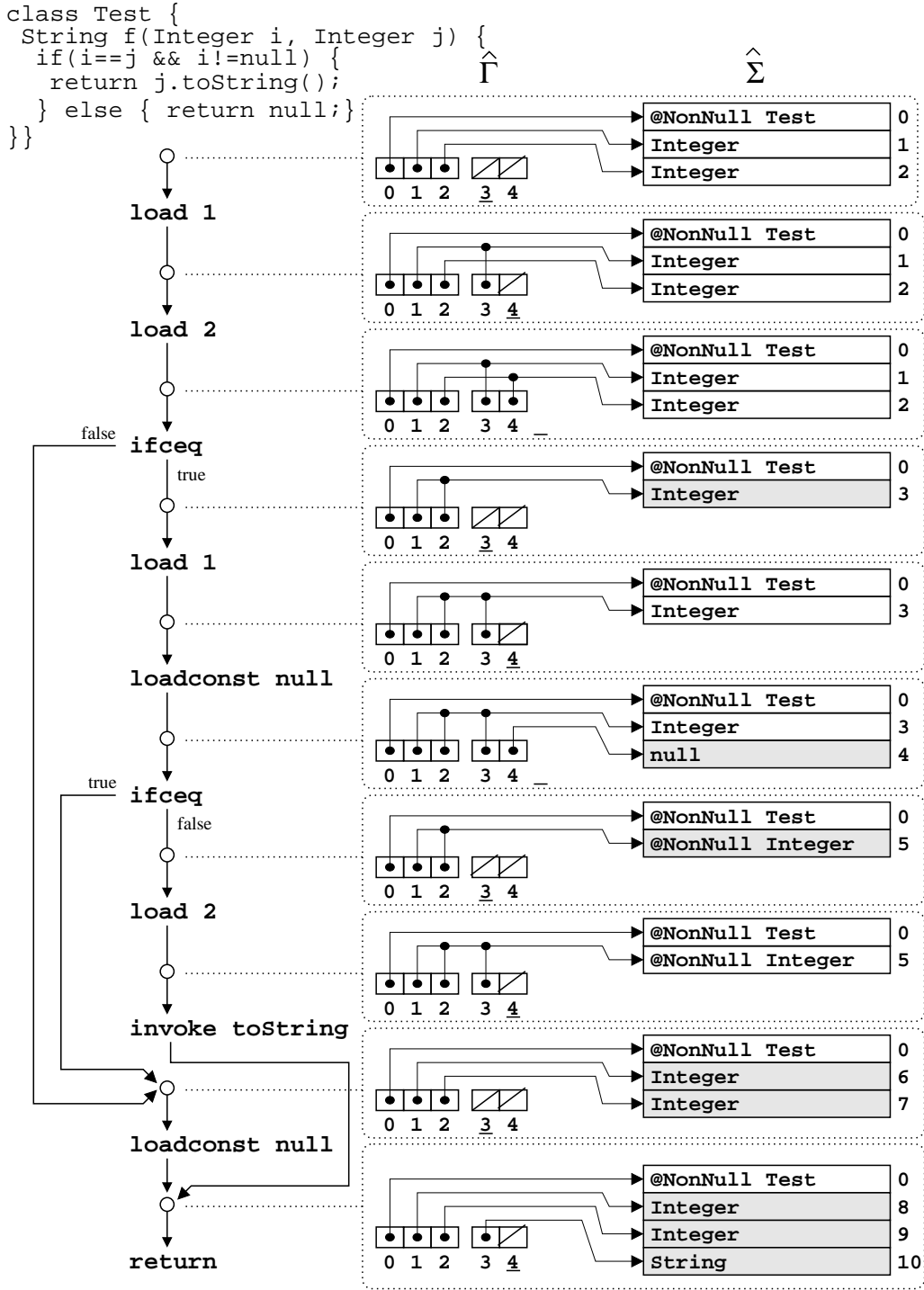


Fig. 5. SJVM bytecode representation of a simple Java Method (source given above) and the state of the abstract store, $(\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa})$, going into each instruction. The value of $\hat{\kappa}$ is indicated by the underlined abstract location; when the stack is full, this points past the last location. The type objects in $\hat{\Sigma}$ are given a unique identifier to help distinguish new objects from old ones; we assume unreferenced type objects are immediately garbage collected, which is reflected in the identifiers becoming non-contiguous. Type aliases are indicated by references which are "joined". For example, the second abstract store reflects the state immediately after the load 1 instruction, where locations 1 and 3 are type aliases. Finally, the mechanism by which abstract stores are merged together will be discussed in §3.5

14

We now consider what happens at join points in the CFG. The `return` instruction in Figure 5 is a good illustration, since two distinct paths reach it and each has its own abstract store. These must be combined to summarise all possible program stores at that point. In Figure 5, the store coming out of the `invoke` instruction has a type aliasing relationship, whereas that coming out of the `loadconst` instruction does not; also, in the former, location 2 has type `@NonNull Integer`, whilst the latter gives it nullable type `Integer`. This information must be combined conservatively. Since location 2 can hold `null` on at least one incoming path, it can clearly hold `null` at the join point. Hence, the least conservative type for location 2 is `Integer`. Likewise, if a type alias relationship does not hold on all incoming paths, we cannot assume it holds at the join.

We formalise this notion of conservatism as a subtype relation, $\leq$, over abstract stores. Here, $S_1 \leq S_2$ can be thought of as stating $S_1$ is *at least as precise* than $S_2$. More specifically, if a location in $S_1$ has type $\hat{T}_1$, and that same location in $S_2$ has type $\hat{T}_2$, then $\hat{T}_1$ must be *at least as precise* as $\hat{T}_2$ (i.e. $\hat{T}_1 \leq \hat{T}_2$). Likewise, if a type aliasing relationship between two locations is present in $S_2$, then it must also be present in $S_1$. This is stated formally as follows:

**Definition 4** *Let $S_1 = (\hat{\Sigma}_1, \hat{\Gamma}_1, \hat{\kappa})$, $S_2 = (\hat{\Sigma}_2, \hat{\Gamma}_2, \hat{\kappa})$ be well-formed abstract stores. Then $S_1 \leq S_2$ iff $\forall x, y \in \{0 \ldots (\hat{\kappa} - 1)\}[\hat{\Sigma}_1(\hat{\Gamma}_1(x)) \leq \hat{\Sigma}_2(\hat{\Gamma}_2(x)) \wedge (\hat{\Gamma}_2(x) = \hat{\Gamma}_2(y) \implies \hat{\Gamma}_1(x) = \hat{\Gamma}_1(y))]$.*

Note, Definition 4 requires $\hat{\kappa}$ be identical on each incoming store; this reflects a standard requirement of Java Bytecode. Now, to construct the abstract store at a join point, our verification system finds the least upper bound, $\sqcup$, of incoming abstract stores — this is the least conservative information obtainable. We formalise this as follows, where the *transfer function*, $f()$, is defined by the abstract semantics of Figure 4, $x$ identifies a node in the CFG, and the edge label, $l$, distinguishes the true/false branches for `ifceq`:

**Definition 5** *Let $G = (V, E)$ be the control-flow graph for a method $M$. Then, the dataflow equations for $M$ are given by $S_M(y) = \bigsqcup_{x \xrightarrow{l} y \in E} f(x, S_M(x), l)$.*

This states that the abstract store, $S_M(y)$, going into a location $y$ is the least-upper bound of the abstract stores coming out of each bytecode preceding it in the control-flow graph. And, furthermore, that the abstract store coming out of an instruction is determined by applying the transfer function to the store going into it. Note, this definition of an abstract store may be recursive (i.e. $S_M(y)$ may be defined in terms of itself); this occurs when there are loops in the control-flow graph.

Finally, our dataflow equations can be solved as usual by iterating them until a fixed

point is reached using a *worklist algorithm*. In doing this, our implementation currently pays no special attention to the order in which nodes in the control-flow graph are visited. More complex iteration strategies (see e.g. [35,9,26]) and optimisations (e.g. [27,24,54]) could be used here to provide a more efficient implementation.

## 4 Soundness

We now demonstrate our algorithm always *terminates* and is *correct*; that is, if a method passes our verification process, then it will never dereference `null`. As indicated already, the proofs which follow apply only to the formalisation of our algorithm for the SJVM. Unfortunately, we cannot provide as strong a guarantee for our actual implementation, primarily because it permits runtime casts (see §5.7 for more on this).

As stated previously, we assume that the SJVM program our verification algorithm operates on has already passed a bytecode verification stage equivalent to that found in the JVM. We introduce the following notion to emphasise this in our proofs:

**Definition 6** *An* SJVM *method is considered to be valid if it passes a bytecode verification phase equivalent to that of the standard JVM verification process [48].*

The consequences of Definition 6 include: all conventional types are used safely; stack sizes are always the same at join points; method and field lookups always resolve; etc.

### 4.1 Termination

Demonstrating termination amounts to showing the dataflow equations always have a *least fixed-point*. This requires that our subtyping relation is a *join-semilattice* (i.e. any two abstract stores always have a unique least upper bound) and that the transfer function, $f$, is monotonic. These are addressed by Lemmas 1 and 2.

Strictly speaking, Definition 4 does not define a join-semilattice over abstract stores, since two stores may not have a unique least upper bound. For example, consider:

$$S_1 = (\{r_1 \mapsto \texttt{Integer}, r_2 \mapsto \texttt{Float}\}, \{0 \mapsto r_1, 1 \mapsto r_1, 2 \mapsto r_2\}, 3)$$

$$S_2 = (\{r_1 \mapsto \texttt{Integer}, r_2 \mapsto \texttt{Float}\}, \{0 \mapsto r_2, 1 \mapsto r_2, 2 \mapsto r_1\}, 3)$$

Then, the following are minimal upper bounds of $S_1$ and $S_2$:

$$S_3 = (\{r_1 \mapsto \texttt{Number}, r_2 \mapsto \texttt{Number}\}, \{0 \mapsto r_1, 1 \mapsto r_1, 2 \mapsto r_2\}, 3)$$

$$S_4 = (\{r_1 \mapsto \texttt{Number}, r_2 \mapsto \texttt{Number}\}, \{0 \mapsto r_2, 1 \mapsto r_2, 2 \mapsto r_1\}, 3)$$

Here, $S_3 \leq S_4$, $S_4 \leq S_3$, $\{S_1, S_2\} \leq \{S_3, S_4\}$ and $\neg \exists S.[\{S_1, S_2\} \leq S \leq \{S_3, S_4\}]$. Hence, there is no unique least upper bound of $S_1$ and $S_2$. Such situations arise in our implementation because type objects are implemented as Java Objects and, hence, $r_1 \neq r_2$ corresponds to objects having different addresses. Now, while $S_3$ and $S_4$ are distinct, they are also *equivalent*:

**Definition 7** *Let $S_1 = (\hat{\Sigma}_1, \hat{\Gamma}_1, \hat{\kappa})$, $S_2 = (\hat{\Sigma}_2, \hat{\Gamma}_2, \hat{\kappa})$, then $S_1$ and $S_2$ are equivalent, written $S_1 \equiv S_2$, iff $S_1 \leq S_2$ and $S_1 \geq S_2$.*

An interesting observation from Definition 7 is that our subtype operator for stores is not a partial order (since this requires anti-symmetry); rather, it is a *preorder*.

**Lemma 1 (Store Subtyping)** *Let $S_1 = (\hat{\Sigma}_1, \hat{\Gamma}_1, \hat{\kappa})$, $S_2 = (\hat{\Sigma}_2, \hat{\Gamma}_2, \hat{\kappa})$ with $dom(\hat{\Gamma}_1) = dom(\hat{\Gamma}_2)$. If $U$ is the set of minimal upper bounds of $S_1$ and $S_2$, then $U \neq \emptyset$ and $\forall x, y \in U.[x \equiv y]$.*

**Proof 1** *Firstly, $U \neq \emptyset$ since $(\{r_1 \mapsto \texttt{Object}, \ldots, r_n \mapsto \texttt{Object}\}, \{1 \mapsto r_1, \ldots, n \mapsto r_n\}, \hat{\kappa})$ is an upper bound for any store where $dom(\hat{\Gamma}) = \{1, \ldots, n\}$. Now, suppose for contradiction that we have two $u_1, u_2 \in U$, where $u_1 \not\equiv u_2$. Then, by Definition 7, either $u_1 \not\leq u_2$ and/or $u_2 \not\leq u_1$. Now, if $u_1 \leq u_2$ we have a contradiction since $u_2$ is not a minimal upper bound and, similarly, if $u_2 \leq u_1$. Thus, $u_1 \not\leq u_2$ and $u_2 \not\leq u_1$ must hold. Suppose $u_1 = (\hat{\Sigma}_{u1}, \hat{\Gamma}_{u1}, \hat{\kappa})$ and $u_2 = (\hat{\Sigma}_{u2}, \hat{\Gamma}_{u2}, \hat{\kappa})$, then following Definition 4 we obtain the following by pushing in negations (and assuming $x, y \in \{0 \ldots (\hat{\kappa}-1)\}$):*

$$\exists x, y \left[ \hat{\Sigma}_{u1}(\hat{\Gamma}_1(x)) \not\leq \hat{\Sigma}_{u2}(\hat{\Gamma}_{u2}(x)) \vee (\hat{\Gamma}_{u2}(x) = \hat{\Gamma}_{u2}(y) \wedge \hat{\Gamma}_{u1}(x) \neq \hat{\Gamma}_{u1}(y)) \right]$$

$$\wedge$$

$$\exists x, y \left[ \hat{\Sigma}_{u2}(\hat{\Gamma}_1(x)) \not\leq \hat{\Sigma}_{u1}(\hat{\Gamma}_{u1}(x)) \vee (\hat{\Gamma}_{u1}(x) = \hat{\Gamma}_{u1}(y) \wedge \hat{\Gamma}_{u2}(x) \neq \hat{\Gamma}_{u2}(y)) \right]$$

*Let $tl$ denote the top-left disjunct, $tr$ denoting the top-right disjunct and so on. Then, there are four cases to consider: $tl \wedge bl$, $tr \wedge br$, $tl \wedge br$ and $tr \wedge bl$ (in fact, since the last two are symmetric we only consider one of them).*

**i)** $\exists x [\hat{\Sigma}_{u1}(\hat{\Gamma}_{u1}(x)) \not\leq \hat{\Sigma}_{u2}(\hat{\Gamma}_{u2}(x))]$ *and* $\exists y [\hat{\Sigma}_{u2}(\hat{\Gamma}_{u2}(y)) \not\leq \hat{\Sigma}_{u1}(\hat{\Gamma}_{u1}(y))]$. *However, we know that $\{S_1, S_2\} \leq u_1$ and $\{S_1, S_2\} \leq u_2$, which implies that some $\hat{T}$ exists where $\{\hat{\Sigma}_1(\hat{\Gamma}_1(x)), \hat{\Sigma}_2(\hat{\Gamma}_2(x))\} \leq \hat{T} \leq \{\hat{\Sigma}_{u1}(\hat{\Gamma}_{u1}(x)), \hat{\Sigma}_{u2}(\hat{\Gamma}_{u2}(x))\}$ (otherwise, the subtype relation is not a complete lattice which it is, recall §3.1). A symmetric*

*argument applies for $y$, leading to the conclusion neither $u_1$ nor $u_2$ are least upper bounds of $S_1$ and $S_2$!*

ii) $\exists x_1, y_1[\hat{\Gamma}_{u2}(x_1) = \hat{\Gamma}_{u2}(y_1) \wedge \hat{\Gamma}_{u1}(x_1) \neq \hat{\Gamma}_{u1}(y_1)]$ *and* $\exists x_2, y_2[\hat{\Gamma}_{u1}(x_2) = \hat{\Gamma}_{u1}(y_2) \wedge \hat{\Gamma}_{u2}(x_2) \neq \hat{\Gamma}_{u2}(y_2))]$. *Since $S_1 \leq u_1$ (respectively $S_1 \leq u_2$) we have that $\hat{\Gamma}_1(x_2) = \hat{\Gamma}_1(y_2)$ (respectively $\hat{\Gamma}_1(x_1) = \hat{\Gamma}_1(y_1)$). By a symmetric argument, $\hat{\Gamma}_2(x_2) = \hat{\Gamma}_2(y_2)$ and $\hat{\Gamma}_2(x_1) = \hat{\Gamma}_2(y_1)$ hold for $S_2$. Thus, in any minimal upper bound of $S_1$ and $S_2$ we have $\hat{\Gamma}(x_1) = \hat{\Gamma}(y_1)$ and $\hat{\Gamma}(x_2) = \hat{\Gamma}(y_2)$. This is a contradiction as it implies neither $u_1$ and $u_2$ are minimal upper bounds.*

iii) $\exists z[\hat{\Sigma}_{u1}(\hat{\Gamma}_{u1}(z)) \not\leq \hat{\Sigma}_{u2}(\hat{\Gamma}_{u2}(z))]$ *and* $\exists x, y[\hat{\Gamma}_{u1}(x) = \hat{\Gamma}_{u1}(y) \wedge \hat{\Gamma}_{u2}(x) \neq \hat{\Gamma}_{u2}(y))]$. *W.L.O.G. assume only one possible value for each of $x, y, z$. Since $S_1 \leq u_1$ and $S_2 \leq u_1$ it follows that $\hat{\Gamma}_1(x) = \hat{\Gamma}_1(y)$ and $\hat{\Gamma}_2(x) = \hat{\Gamma}_2(y)$ (respectively). This implies that $\hat{\Sigma}_{u2}(\hat{\Gamma}_{u2}(x)) = \hat{\Sigma}_{u2}(\hat{\Gamma}_{u2}(y))$ (otherwise, $u_2$ is not minimal). Likewise, since $S_1 \leq u_2$ and $S_2 \leq u_2$, we have $\hat{\Sigma}_{u2}(\hat{\Gamma}_{u2}(z)) = \hat{\Sigma}_1(\hat{\Gamma}_1(z)) \sqcup \hat{\Sigma}_2(\hat{\Gamma}_2(z))$ (again, otherwise $u_2$ is not minimal). This implies $\hat{\Sigma}_{u1}(\hat{\Gamma}_{u1}(z)) \geq \hat{\Sigma}_{u2}(\hat{\Gamma}_{u2}(z))$ (otherwise, the subtype relation is not a complete lattice which it is, recall §3.1). Finally, let us construct $u_3 = (\hat{\Sigma}_{u3}, \hat{\Gamma}_{u2}, \hat{\kappa})$ where $\forall i[i \neq z \implies \hat{\Sigma}_{u3}(\hat{\Gamma}_{u3}(i)) = \hat{\Sigma}_{u1}(\hat{\Gamma}_{u1}(i))], \hat{\Sigma}_{u3}(\hat{\Gamma}_{u3}(z)) = \hat{\Sigma}_{u2}(\hat{\Gamma}_{u2}(z))$ and $\forall i, j[\hat{\Gamma}_{u3}(i) = \hat{\Gamma}_{u3}(j) \iff \hat{\Gamma}_{u1}(i) = \hat{\Gamma}_{u1}(j)]$. Since $\hat{\Sigma}_{u1}(\hat{\Gamma}_{u1}(z)) \geq \hat{\Sigma}_{u2}(\hat{\Gamma}_{u2}(z))$, it follows immediately that $u_3 \leq u_1$. Likewise, since $\hat{\Sigma}_{u2}(\hat{\Gamma}_{u2}(x)) = \hat{\Sigma}_{u2}(\hat{\Gamma}_{u2}(y))$, it follows that $u_3$ is an upper bound of both $S_1$ and $S_2$. Thus we have a contradiction, since $u_1$ is not a minimal upper bound.*

□

We now demonstrate that our dataflow equations are *monotonic*, which is a normal requirement for termination.

**Lemma 2 (Monotonicity)** *The dataflow equations from Definition 5 are monotonic.*

**Proof 2** *Demonstrating $f$ is monotonic requires showing each transition from our abstract semantics is monotonic. That is, if $f(i, S_1, l) = S_2$ for some bytecode at position $i$, then for all $S_1'$, where $f(i, S_1', l) = S_2'$ and $S_1 \leq S_1'$, we have $S_2 \leq S_2'$.*

*Now, given a bytecode instruction $i$, $f(i, S_M(i), l)$ always manipulates $\hat{\Gamma}$ and $\hat{\Sigma}$ in the same way, regardless of input store (e.g. `store 1` always overwrites location `1` with the top of the stack). Observe that this correctly reflects the operational semantics of SJVM bytecodes (recall Figure 2), where each bytecode always manipulates the stack in the same manner.*

*Now, we first consider the issue of type aliasing. Let us assume, for contradiction, that $S_2 \not\leq S_2'$ because a type alias exists in $S_2'$ which does not exist in $S_2$. Now, we have two cases:*

i) *The offending type alias in $S_2'$ was not present in $S_1'$ and, hence, was introduced by $f(i, S_1', l)$. Now, type aliases are only introduced by the transfer function for*

*bytecodes* `load` *and* `ifceq` *(on the true branch). More importantly, they are always introduced in these cases, and all existing type aliases are always preserved. This gives a contradiction, since it implies the offending type alias was in* $S_2$.

ii) *The offending type alias in* $S_2'$ *was present in* $S_1'$. *Since* $S_1 \le S_1'$, *we know (by Definition 4) the offending type alias was also in* $S_1$. *Thus, the offending type alias was eliminated in* $f(i, S_1, l)$, *but not* $f(i, S_1', l)$. *Considering Figure 4, we see that every bytecode except* `load`, `loadconst` *and* `new T` *is capable of eliminating a type alias. This is because they pop references off the stack. However, they always manipulate the stack in the same manner. Thus, since the offending type alias was eliminated by* $f(i, S_1, l)$, *it must also have been eliminated by* $f(i, S_1', l)$ *which gives a contradiction.*

*We now consider the issue of subtyping. Let us assume, for contradiction, that* $S_2 \not\le S_2'$ *because there exists some location* $x$ *where* $\hat{\Sigma}_2(\hat{\Gamma}_2(x)) \not\le \hat{\Sigma}_2'(\hat{\Gamma}_2'(x))$, *where* $S_2 = (\hat{\Sigma}_2, \hat{\Gamma}_2, \hat{\kappa}_2)$ *and* $S_2'$ *similarly. Now, ignoring* `ifceq`, *we know from Figure 4 that the bytecode at position* $i$ *only assigns type objects already accessible from the location array and/or introduces type objects with the same type (e.g.* `new Integer` *always introduces an* `Integer` *type object). Therefore, since the type of every location in* $S_1$ *is* $\le$ *its counterpart in* $S_1'$, *every location in* $S_2$ *must be a subtype of its counterpart in* $S_2'$ — *giving a contradiction.*

*Finally, for* `ifceq`, *we also require that* $\hat{T}_1 \sqcap \hat{T}_2 \le \hat{T}_1' \sqcap \hat{T}_2'$ *and* $\hat{T}_1 - \hat{T}_2 \le \hat{T}_1' - \hat{T}_2'$ *if* $\hat{T}_1 \le \hat{T}_1'$ *and* $\hat{T}_2 \le \hat{T}_2'$. *The former holds as the subtype relation,* $\le$, *forms a complete lattice (recall §3.1), whilst the latter follows immediately from its definition.* □

### 4.2 Correctness

We now demonstrate that the information computed by our algorithm is a safe approximation of the SJVM during any execution trace of the program. To do this, we must first define more precisely what this means.

**Definition 8 (Effective Type.)** *Let* $(\Sigma, \Pi)$ *be a state during some execution trace of the* SJVM, *where* $\Pi = (\delta^M, pc, \Gamma, \kappa) :: S$. *Then, the* effective type *of a value* $\psi$ *is determined as follows, where* $\mathcal{I}_\mathcal{N} = \{$`null`$, \ldots, -2, -1, 0, 1, 2, \ldots\}$:

$$nntype(\psi, \Sigma) = \begin{cases} null & \text{if } \psi \in \mathcal{I}_\mathcal{N} \\ @\text{NonNull } C & \text{if } \Sigma(\psi)(\$) = C \end{cases}$$

The purpose of the above is to determine the appropriate abstract type for a particular variable at runtime. For example, a variable holding a reference to a `String` object in some state has an effective type of @NonNull `String`. However, its effective type may differ in other states where the variable holds different values.

We can now proceed to develop a notion of correctness for our algorithm. Informally, the algorithm is correct provided that, for every execution trace, every method parameter, return value and field respect their declared abstract types:

**Definition 9 (Safe Local Abstraction)** *Let $(\Sigma, \Pi)$ be an SJVM state during some execution trace at position $i$ in method $M$ (i.e. $\Pi = (\delta^{\text{M}}, i, \Gamma, \kappa) :: S$ and $\delta^{\text{M}}$ identifies $M$). Let $S_M(i) = (\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa})$ be the abstract store determined for position $i$ by our algorithm. Then, $S_M(i)$ is a safe abstraction of $(\Sigma, \Pi)$ iff for every location $x, y$ we have that $\hat{\Gamma}(x) = \hat{\Gamma}(y) \Longrightarrow \Gamma(x) = \Gamma(y)$, $nntype(\Gamma(x), \Sigma) \leq \hat{\Sigma}(\hat{\Gamma}(x))$ and $\hat{\kappa} = \kappa$.*

Definition 9 states that an abstract store $S_M(i)$ is a safe abstraction of a concrete state $(\Sigma, \Pi)$ if: firstly, every type alias in $S_M(i)$ is actually a type alias in $(\Sigma, \Pi)$; secondly, that the effective type of a reference in $(\Sigma, \Pi)$ is a subtype of that in $S_M(i)$; and, finally, that the stack pointers match. In a similar fashion, we define a safe abstraction of the program heap:

**Definition 10 (Safe Heap Abstraction)** *Let $(\Sigma, \Pi)$ be an SJVM state during some execution trace at position $i$ in method $M$ (i.e. $\Pi = (\delta^{\text{M}}, i, \Gamma, \kappa) :: S$ and $\delta^{\text{M}}$ identifies $M$). Let $\Delta_F$ be a map from fields to abstract types. Then, $\Delta_F$ is a safe abstraction of $(\Sigma, \Pi)$ iff for every $\delta^{\text{F}} \in dom(\Delta_F)$ and $\rho \in dom(\Sigma)$, where $\delta^{\text{F}} = (O, N, T)$ and $\Sigma(\rho)(\$) = O$, we have that $nntype(\Sigma(\rho)(N), \Sigma) \leq \Delta_F(\delta^{\text{F}})$.*

Finally, we bring these notions together to determine what it means for information computed by our algorithm to be a safe representation of the SJVM at any point.

**Definition 11 (Safe Abstraction)** *Let $(\Sigma, \Pi)$ be an SJVM state during some execution trace at position $i$ in method $M$ (i.e. $\Pi = (\delta^{\text{M}}, i, \Gamma, \kappa) :: S$ and $\delta^{\text{M}}$ identifies $M$). Let $S_M(i)$ be the abstract store determined for position $i$ by our algorithm, and $\Delta_F$ a map from fields to abstract types. Then, $(\Delta_F, S_M(i))$ is a safe abstraction of $(\Sigma, \Pi)$ iff $\Delta_F$ and $S_M(i)$ are both safe abstractions of $(\Sigma, \Pi)$.*

We now demonstrate that all aspects of a safe abstraction are ensured by our algorithm. Roughly speaking, this amounts to showing that: firstly, join operator always produces a safe abstraction, given safe abstractions as input (Lemma 3); secondly, that the transfer function, when given a safe abstraction as input, always produces a safe abstraction (Lemmas 4 and 5).

**Lemma 3 (Safe Join)** *Let $i \overset{l}{\to} j$ be an edge in the CFG of a valid method $M$, and $(\Sigma_i, \Pi_i) \rightsquigarrow (\Sigma_j, \Pi_j)$ be a transition in some execution trace of the SJVM, where $\Pi_i$ and $\Pi_j$ are at positions $i$ and $j$ (i.e. $\Pi_i = (\delta^{\text{M}}, i, \Gamma_i, \kappa_i) :: S$ and $\delta^{\text{M}}$ identifies $M$, etc). Assume $f(i, S_M(i), l)$ is a safe abstraction of $(\Sigma_j, \Pi_j)$. Then, $S_M(j)$ is a safe abstraction of $(\Sigma_j, \Pi_j)$.*

**Proof 3** *Suppose not. Let $\Pi_j = (\delta^{\text{M}}, j, \Gamma_j, \kappa_j) :: S$, $f(i, S_M(i), l) = (\hat{\Sigma}_i, \hat{\Gamma}_i, \hat{\kappa}_i)$ and $S_M(j) = (\hat{\Sigma}_j, \hat{\Gamma}_j, \hat{\kappa}_j)$. Then, following Definition 9, there are three cases to consider:*

(i) $\exists x, y[\hat{\Gamma}_j(x) = \hat{\Gamma}_j(y) \wedge \Gamma_j(x) \neq \Gamma_j(y)]$. *In other words, the join operator introduced an erroneous type alias, when joining $f(i, S_M(i), l)$ with $f(k, S_M(k), l)$, for some other predecessor $k$ of $j$. Recall from Definition 4 that the subtyping relation on abstract stores conservatively retains type aliases. In other words, when joining abstract stores together, an alias relationship is only present in the result if it was present in all input stores. This gives a contradiction since it implies the erroneous type aliasing relationships must have been present in $f(i, S_M(i))$.*

(ii) $\exists x[nntype(\Gamma_j(x), \Sigma_j) \not\leq \hat{\Sigma}_j(\hat{\Gamma}_j(x))]$. *In other words, the join operator introduced an erroneous abstract type, when joining $f(i, S_M(i), l)$ with $f(k, S_M(k), l)$, for some other predecessor $k$ of $j$. However, by Definitions 4 and 5, we know $\hat{\Sigma}_i(\hat{\Gamma}_i(x)) \leq \hat{\Sigma}_j(\hat{\Gamma}_j(x))$ because $S_M(j) = \bigsqcup_{k \to j} S_M(k)$ (and $i$ is a predecessor of $j$ by construction). Hence, we have a contradiction as $nntype(\Gamma_j(x), \Sigma_j) \leq \hat{\Sigma}_i(\hat{\Gamma}_i)$ by assumption.*

(iii) $\hat{\kappa}_j \neq \kappa_j$. *Recall that, since method $M$ is valid, we know that the $\kappa$ always has the same value at any given position in $M$. Now, since $f(i, S_M(i), l)$ is a safe abstraction of $(\Sigma_j, \Pi_j)$, we know that $\hat{\kappa}_i = \kappa_j$. Finally, by Definition 4, we know that two abstract stores can only be joined if their stack pointer has the same value and, hence, $\hat{\kappa}_j = \hat{\kappa}_i$.*

$\square$

**Lemma 4 (Safe Local Step)** *Let $i \xrightarrow{l} j$ be an edge in the CFG of a valid method $M$, and $(\Sigma_i, \Pi_i) \to (\Sigma_j, \Pi_j)$ be a single transition of some execution trace, where $\Pi_i$ and $\Pi_j$ are at positions $i$ and $j$ (i.e. $\Pi_i = (\delta^M, i, \Gamma_i, \kappa_i) :: S$ and $\delta^M$ identifies $M$, etc). If $S_M(i) = (\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa})$ and $\Delta_F$ are safe abstractions of $(\Sigma_i, \Pi_i)$, then $f(i, S_M(i), l)$ is a safe abstraction of $(\Sigma_j, \Pi_j)$.*

**Proof 4** *Suppose not. Firstly, since $(\Sigma_i, \Pi_i) \to (\Sigma_j, \Pi_j)$ is a single transition, we know the bytecode at position $i$ is not an* invoke *bytecode. Now, let $\Pi_j = (\delta^M, j, \Gamma_j, \kappa_j) :: S$, $S_M(i) = (\hat{\Sigma}_i, \hat{\Gamma}_i, \hat{\kappa}_i)$ and $f(i, S_M(i), l) = (\hat{\Sigma}_j, \hat{\Gamma}_j, \hat{\kappa}_j)$. Then, following Definition 9, there are three cases to consider:*

(i) $\exists x, y[\hat{\Gamma}_j(x) = \hat{\Gamma}_j(y) \wedge \Gamma_j(x) \neq \Gamma_j(y)]$. *In other words, applying the transfer function to $S_M(i)$ introduced an erroneous type alias. We now demonstrate, by case analysis on the instruction types of Figure 4, that the transfer function cannot introduce an incorrect type alias. There are four main cases to consider:*

  · putfield $\delta^F$ *and* return *cannot introduce type aliases since they do not update $\hat{\Gamma}$.*

  · loadconst, new $T$, getfield $\delta^F$ *also cannot introduce type aliases since they only assign fresh locations to locations in $\hat{\Gamma}$.*

  · load $i$ *and* store $i$ *both introduce type aliases between the local array and the stack. However, this correctly reflects their semantics (recall Figure 2).*

  · ifceq. *We consider the true and false branch separately. On the true branch,*

21

a type alias is created between all locations $l$ where $\hat{\Gamma}(l) = r_1$ or $\hat{\Gamma}(l) = r_2$. But, in this case, we know the references represented by $r_1$ and $r_2$ are equal according to the semantics of this bytecode (recall Figure 2). The false branch is simpler, as it (respectively) replaces $r_1, r_2$ with $r_3, r_4$, both of which are fresh and, hence, no type alias can be introduced.

(ii) $\exists x[nntype(\Gamma_j(x), \Sigma_j) \not\preceq \hat{\Sigma}_j(\hat{\Gamma}_j(x))]$. *In other words, applying the transfer function to $S_M(i)$ introduced an erroneous abstract type. Again, we demonstrate, by case analysis on the instruction types of Figure 4, that the transfer function cannot introduce an incorrect type. There are four main cases to consider:*

- `load` $i$, `store` $i$, `loadconst`, `putfield` $\delta^{\mathrm{F}}$, *and* `return` *do not introduce any new abstract types. Hence, since $S_M(i)$ safely abstracts $(\Sigma_i, \Gamma_i)$ by assumption, these bytecodes can be ignored.*

- `getfield` $\delta^{\mathrm{F}}$. *This bytecode places an abstract type $T = \Delta_F(\delta^{\mathrm{F}})$ onto the stack. However, by assumption, $\Delta_F$ is a safe abstraction of $\Sigma_i$ and, hence, the value loaded must be safe.*

- `new` $T$. *This bytecode introduces an abstract type @NonNull $T$ and places it on the stack. Furthermore, since this instruction does actually create a new object and place a reference to it on the stack, the introduction of @NonNull $T$ is safe.*

- `ifceq`. *Again, we treat true and false branches separately. The true branch introduces the greatest lower bound of the types of the two references that are equal. It produces a type @NonNull $T_1$ only when one operand has type @NonNull $T_2$. When the other operand has a possibly-null type, this is safe since the references are in fact equal according to Java's reference comparison.*

  *The false branch uses the type difference operator. According to Definition 3, if one of the two references compared has type null the other is given @NonNull status, otherwise no new type @NonNull $T$ is introduced. An important issue is that any location represented by an abstract location with type null can only hold null. This is trivially the case, since type null is only introduced by the `loadconst` bytecode, and null $\sqcup T \neq$ null unless $T =$ null.*

  *Finally, both branches replace $r_1, r_2$ by substitution, which could cause problems if any underlying type aliases were incorrect. Case (i) above guarantees this is not the case, however.*

(iii) $\hat{\kappa}_j \neq \kappa_j$. *From Figures 2 and 4 it is fairly evident that the transfer function manipulates the stack in an identical fashion to the SJVM.*

$\square$

**Lemma 5 (Safe Heap Step)** *Let $i \xrightarrow{l} j$ be an edge in the CFG of a valid method $M$, and $(\Sigma_i, \Pi_i) \to (\Sigma_j, \Pi_j)$ be a single transition of some execution trace, where $\Pi_i$ and $\Pi_j$ are at positions $i$ and $j$ (i.e. $\Pi_i = (\delta^{\mathrm{M}}, i, \Gamma_i, \kappa_i) :: S$ and $\delta^{\mathrm{M}}$ identifies $M$, etc). If $S_M(i) = (\hat{\Sigma}, \hat{\Gamma}, \hat{\kappa})$ and $\Delta_F$ are safe abstractions of $(\Sigma_i, \Pi_i)$, then $\Delta_F$ is a safe abstraction of $(\Sigma_j, \Pi_j)$.*

**Proof 5** *Suppose not. Again, since $(\Sigma_i, \Pi_i) \to (\Sigma_j, \Pi_j)$ is a single transition, the bytecode at position $i$ is not an* invoke $\delta^{\text{M}}$. *In fact, the bytecode at position $i$ must be a* putfield $\delta^{\text{F}}$, *since this is the only other bytecode which can modify the heap. Furthermore, this bytecode must be assigning* null *to a field matching $\delta^{\text{F}}$, where $\Delta_F(\delta^{\text{F}}) = $@NonNull $T$ (otherwise, the effective type of that field will be a subtype of $\Delta_F(\delta^{\text{F}})$, since $M$ is valid by assumption). This implies the abstract type on top of the stack in $S_M(i)$ had abstract type $T$, not @NonNull $T$ (since $S_M(i)$ is a safe abstraction by assumption). However, this gives a contradiction, since it implies $S_M(j)$ could not be constructed (because of the subtyping constraint on the* putfield *bytecode — recall Figure 4).* □

At this point, we have essentially shown that our verification algorithm is correct, provided we ignore invoke $\delta^{\text{M}}$ bytecodes. We now need to finish the proof by demonstrating that our treatment of these bytecodes is also safe.

**Theorem 1** *Let $i \xrightarrow{l} j$ be an edge in the CFG of a valid method $M$, and $(\Sigma_i, \Pi_i) \rightsquigarrow (\Sigma_j, \Pi_j)$ be a transition in some execution trace of the* Sjvm, *where $\Pi_i$ and $\Pi_j$ are at positions $i$ and $j$ (i.e. $\Pi_i = (\delta^{\text{M}}, i, \Gamma_i, \kappa_i) :: S$ and $\delta^{\text{M}}$ identifies $M$, etc). Assume $\Delta_F$ and $S_M(i)$ are safe abstractions of $(\Sigma_i, \Pi_i)$. Then, $\Delta_F$ and $f(i, S_M(i), l)$ is a safe abstraction of $(\Sigma_j, \Pi_j)$.*

**Proof 6** *Suppose not. Now, the bytecode at position $i$ must have the form* invoke $\delta^{\text{M}}$, *since all other bytecodes are already shown correct in Lemmas 4 and 5. Essentially then, one of two things must have happened during the execution of method $\delta^{\text{M}}$: either some field declared* @NonNull $T$ *was assigned* null; *or, some local variable contained* null *whose inferred abstract type was* @NonNull $T$.

*Let us first assume for contradiction that $\Delta_F$ was a safe abstraction of all* Sjvm *states during the execution trace $(\Sigma_i, \Pi_i) \rightsquigarrow (\Sigma_j, \Pi_j)$. Then, there must be some abstract store $S_{M1}(k)$ which did not safely abstract a corresponding* Sjvm *state $(\Sigma_k, \Gamma_k)$ in that trace. Assume WLOG, that $k$ is the first such position occurring in the trace. Let $(\Sigma_l, \Gamma_l)$ be the preceding state in the trace, and $S_{M2}(l)$ the corresponding abstract store (not necessarily in the same method). By assumption, $S_{M2}(l)$ is a safe abstraction of $(\Sigma_l, \Gamma_l)$. Now, if the bytecode at position $l$ is neither an* invoke *nor* return, *then an immediate contradiction follows from Lemma 4. So, suppose the bytecode at $l$ is either an* invoke *or* return. *The semantics of the former dictate that items are removed from the stack and then passed into the local array of the invoked method (recall Figure 2); for the latter, an item is removed from the stack of the method and placed onto the stack of the caller. Therefore, we still arrive at a contradiction since the items on the stack are a safe abstraction (by assumption), the abstract semantics of these bytecodes respect the declared (abstract) parameter and return types for the method in question and, hence, $S_{M1}(k)$ is a safe abstraction of $(\Sigma_k, \Gamma_k)$.*

*Thus, our conclusion is that $\Delta_F$ was not a safe abstraction of all* Sjvm *states during*

*the execution trace $(\Sigma_i, \Pi_i) \rightsquigarrow (\Sigma_j, \Pi_j)$. Let $(\Sigma_k, \Gamma_k)$ identify the first problematic state in the trace, and $S_M(k)$ be the corresponding abstract store. Now, it must hold that some field $\delta^{\mathrm{F}}$ holds* `null`*, but $\Delta_F(\delta^{\mathrm{F}}) = @$* `NonNull` *$T$. Let $(\Sigma_l, \Gamma_l)$ be the preceding state in the trace, and $S_M(l)$ the corresponding abstract store. In this case, the bytecode at position $l$ must be a* `putfield` *(since $(\Sigma_k, \Gamma_k)$ is the earliest erroneous state) and, thus, position $l$ is in the same method as $k$. Likewise, $S_M(l)$ must be a safe abstraction of $(\Sigma_l, \Gamma_l)$. However, this leads to a contradiction by Lemma 5, which implies $S_M(k)$ is a safe abstraction of $(\Sigma_k, \Gamma_k)$.* $\square$

## 5  Implementation

We have implemented our system on top of Java Bytecode and we now discuss many aspects not covered in our formalism for the SJVM.

### 5.1  Bytecodes.

Many JVM bytecodes were not considered in the SJVM formalism. There include all arithmetic operations (e.g. `iadd`, `imul`, etc), stack manipulators (e.g. `pop2`, `dup`, etc), other branching primitives (e.g. `ifnonnull`, `tableswitch`, etc), synchronisation primitives (e.g. `monitorenter`, etc) and other miscellaneous ones (e.g. `instanceof`, `checkcast`, `athrow` and `arraylength`). In the vast majority of cases, it is straightforward to see how our formalism can be extended to support these bytecodes.

Our implementation supports all except one bytecode — the `jsr` bytecode (Jump to SubRoutine). Subroutines are known to significantly complicate the bytecode verification process and, indeed, Sun's bytecode verifier makes several critical assumptions in dealing with them [46]. While various techniques for resolving them (see e.g [51,46,45,18]) could be applied here, we did not need them in practice. This is because the `javac` compiler almost always avoids using `jsr` by inlining subroutines and, as a result, we did not encounter them in practice.

### 5.2  Interfaces.

Our formalism requires that the subtype relation for non-null types formed a complete lattice (recall §3.1). The SJVM ensures this by ignoring Java `interfaces` altogether. In fact, it is a well-known problem that Java's subtype relation does not form a complete lattice [46]. This arises because two classes can share the same super-class and implement the same interfaces; thus, they may not have a

24

unique least upper bound. To resolve this, our implementation adopts the standard solution of ignoring interfaces entirely and, instead, treating interfaces as type `java.lang.Object`. This approach does mean our system will fail to verify some programs that we might expect it to pass. However, this is somewhat unlikely to occur in practice and, in fact, we have not encountered a real-world program where this was a problem.

*5.3  Constructors.*

An important problem arises when dealing with constructors and, more specifically, default values for fields [25]. Roughly speaking, the problem is that a field is given a default value until it is actually initialised by a constructor (if it ever is). In Java, the default value is a subtype of the field's declared type and, hence, this presents no problem. In our system, however, this is not necessarily the case; `null` is the default value assigned to fields of reference type, but this is clearly not a subtype of, for example, `@NonNull Integer`. Thus, a field of type `@NonNull Integer` will temporarily hold an invalid value inside a constructor. We must ensure such fields are properly initialised; furthermore, we must prevent accesses which assume such fields are already initialised (such as in a method called by the constructor).

Figure 6 highlights the problem. We must ensure such fields are properly initialised, and must restrict access prior to this occurring. Two mechanisms are used to do this:

(1)  A simple dataflow analysis is used to ensure that all non-null (instance) fields in a class declaration are initialised by that class's constructor.

(2)  Following Fähndrich and Leino [25], we use a secondary type annotation, `@Raw`, for references to indicate the object referred to may not be initialised. This implies any field of that object which has a reference type may currently hold `null`, regardless of whether it is annotated `@NonNull`. Reads from fields through these return nullable types. The `this` reference in a constructor is implicitly typed `@Raw` and `@Raw` is strictly a supertype of a normal reference. Thus, methods cannot be called on `this` whose receiver type is not declared `@Raw`. Likewise, we cannot pass `this` in a non-`@Raw` argument position, nor assign `this` to a non-`@Raw` field.

Our use of `@Raw` here is a somewhat simplified version of that outlined by Fähndrich and Leino [25], where a more fine-grained type is used which can indicate exactly which fields are uninitialised. However, we found this to be sufficient for the programs we annotated as part of our experimental study (see §6). It remains unclear whether the more detailed version of `@Raw` proposed by Fähndrich and Leino is actually necessary for checking programs in practice.

Finally, static field initialisers present an awkward problem, since an object's con-

```
public class Parent {
 public Parent() {
    doBadStuff(); // error #1, f1 not initialised
 }

 int doBadStuff() { return 0; }
}

public class Child extends Parent {
  @NonNull String f1;
  @NonNull String f2;

  public Child() {
   doBadStuff();       // error #2, f1 not initialised before call
   f1 = "Hello World";
                       // error #3, f2 not initialised yet
  }

  int doBadStuff() { return f1.length(); }
}
```

Fig. 6. Illustrating three distinct problems with constructors and default values. Error #3 arises as all @NonNull fields must be initialised! Error #2 arises as a method is called on this before all @NonNull fields are initialised. Error #1 arises as, when the Child's constructor is called, it calls the Parent's constructor. This, in turn, calls doBadStuff() which dynamically dispatches to the Child's implementation. However, field f1 has not yet been initialised!

```
public class Test {
 static Test x = new Test();
 static @NonNull String statf = "Hello World";

 @NonNull String f;

 public Test() {
  f = statf; // error, statf may not be initialised
 }
}
```

Fig. 7. Illustrating a problem caused by static initialisers. The issue is that the static field statf will not have been initialised when the Test constructor is called, despite its @NonNull annotation. Thus, the assignment will incorrectly assign null to field f.

structor can, in principle, access any static field, including those which are awaiting initialisation. Figure 7 illustrates the issue. Resolving the difficulty with static fields is not as easy as for normal fields, and cannot be achieved with something like the @Raw annotation. One approach is simply to prevent static fields from being annotated with @NonNull. However, this would be impractical as existing software typically assumes certain static fields (e.g. System.out) are non-null. Another approach might be to annotate every method with the static fields it accesses (including those read by all methods it invokes). This way, one could ensure that no static field was accessed before its initialiser was run. Again, this approach seems inherently impractical. A simpler approach would be to perform a runtime check whenever a static field annotated @NonNull is accessed. This would mean some problems were not caught at compile time, but were instead confined to a small number of places. While some better solutions may indeed be possible for this problem, for now we choose a simple workaround: we only allow static initialisers for types defined in the standard library; this works, since we know such types cannot access static fields defined in client programs. However, it is not a general solution to the problem.

*5.4   Inheritance.*

When a method overrides another via inheritance our tool checks that @NonNull types are properly preserved. As usual, types in the parameter position are *contravariant* with inheritance, whilst those in the return position are *covariant*. For example, consider the following:

```
public class Parent {
  String f(@NonNull Number x) { ... };
}
public class Child extends Parent {
  @NonNull String f(Number x) { ... };
}
```

The above code is allowed because: Parent.f() can return every value that Child.f() can return; and, likewise, Child.f() can accept every parameter value that Parent.f() can accept. In contrast, the following is not allowed:

```
public class Parent {
  String f(Number x) { ... };
}
public class Child extends Parent {
  String f(@NonNull Number x) { ... };
}
```

27

Here, the problem is that, given a variable v of static type `Parent`, it seems from the signature of `Parent.f()` that one should be able to call `v.f(null)`. However, if v actually refers to an instance of `Child`, then this assumption is broken.

## 5.5 Field Retyping.

Consider this method and its bytecode (recall local 0 holds `this`):

```
class Test {                    0.  load 0
 Integer field;                 2.  getfield Test.field
 void f() {                     5.  ifnull 16
  if(field != null) {           8.  load 0
   field.toString()            10. getfield Test.field
  }                            13. invoke Integer.toString
 }                             16. return
}
```

The above is not type safe in our system as the non-nullness of the field is lost when it is reloaded. This is strictly correct, since the field's value may have been changed between loads (e.g. by another thread). We require this is resolved manually by adjusting the source to first store the field in a local variable (which is strictly thread local).

An interesting observation here, is that there are some situations when we know an object's field cannot be modified — for example, if it's marked `final`. In these cases, our tool could correctly retype a field to be `@NonNull`. At the present time, however, we do not do this for simplicity, and leave it for future work.

## 5.6 Generics.

Our implementation supports Java Generics. For example, we denote a `Vector` containing non-null `Strings` with `Vector<@NonNull String>`. Extending the subtype relation of Figure 3 is straightforward and follows the conventions of Java Generics (i.e. prohibiting variance on generic parameters). Verifying methods which accept generic parameters is more challenging. To deal with this, we introduce a special type, $\top_i$, for each (distinct) generic type used in the method; here, $\top_i \leq$ `java.lang.Object` and $\top_i \not\leq \top_j$, for $i \neq j$. When checking a method `f(T x)`, the abstract location representing x is initialised to the type $\top_i$ used exclusively for representing the generic type `T`. The subtyping constraints ensure $\top_i$ can only flow into variables/return types declared with the same generic type `T`. So, for example, the following code gives a syntax error in our system as $null \not\leq$ `T`.

28

```
class Test<T> {
 T f() { return null; }
}
```

Whilst this ensures that user-defined classes are safe, an interesting problem arises with some existing library classes. For example:

```
class Hashtable<K,V> ... {
 ...

 V get(K key) {
  ...;
  return null;
}}
```

Clearly, this class assumes `null` is a subtype of every type; unfortunately, this is not true in our case, since e.g. `null` $\nleq$ `@NonNull String`. This example presents a particular problem as, unlike user-defined classes, the standard library code (particularly the interfaces it uses) cannot be changed. Thus, simply reporting an error and relying on the user to fix it, as our tool does for user-defined classes, will not eliminate the problem. Therefore, as a simple work-around, we prohibit instances of `Hashtable`/`HashMap` from having a non-null type in `V`'s position. Other classes, including `LinkedList`, `Stack` and `Queue` are likewise affected and resolved in the same fashion. Whilst this is not a general solution to the problem, it is sufficient to ensure the results reported in §6 are safe.

### 5.7   Casting + Arrays.

Our implementation supports arrays having both non-null references and elements. For example:

```
@NonNull Integer @NonNull [] a1;
```

Here, `a1` is a non-null reference to an array holding non-null elements. When annotating arrays, the leftmost annotation associates with the element type, whilst that just before the braces associates with the array reference type.

An important question is how our system deals with subtyping. For example, we do not allow the following:

```
@NonNull Integer @NonNull [] ≤ Integer @NonNull []
```

```
  void f(@Type("Vector<@NonNull␣Integer>") Vector<Integer> v,
         @Type("Integer␣@NonNull␣[]") Integer[] arr1,
         Integer[] arr2) {
  ...
  }
```

Fig. 8. Illustrating how the @Type() annotation is used to overcome Java's present limitations regarding where annotations can be placed. Notice that the @Type() annotation is only used when actually required. This code is then compiled down to Java bytecode in the normal fashion which, in turn, is read by our tool. Since annotations persist into Java Bytecode (if required), our tool is able to extract the required non-null type information from these annotations.

In fact, we require all array element types be identical between subtypes[1]. Finally, we must explicitly prevent the creation of arrays that hold non-null elements (e.g. new @NonNull Integer[10]), as Java always initialises array elements of reference type with null. Instead, we require the programmer provide an explicit *cast* to @NonNull Integer[] when he/she knows the array has been fully initialised. Casts from nullable to non-null types are implemented as runtime checks which fail by throwing ClassCastExceptions. Whilst casting from Integer to @NonNull Integer requires a single check, casting from Integer[] to @NonNull Integer[] requires checking every array element. The use of casts weakens Theorem 1, since we are essentially trading NullPointerExceptions for ClassCastExceptions. While this is undesirable, it is analogous to the issue of downcasts in Object-Oriented Languages.

*5.8 Instanceof.*

Our implementation builds upon the type aliasing technique to support retyping via instanceof. For example:

```
  if(x instanceof String) { String y = (String) x; .. }
```

Here, our system retypes x to type @NonNull String on the true branch, rendering the cast redundant (note, an instanceof test never passes on null).

---

[1] While this contrasts slightly with Java's treatment of arrays, we cannot do better without adding runtime non-null type information to arrays.

*5.9 Type Annotations.*

The Java Classfile format doesn't allow annotations on generic parameters or in the array type reference position. However, starting from Java 7, there will be direct support for such types based on JSR308 [23]. In the meantime, we employed a simple mechanism for encoding this information into a classfile. We employ a special annotation, `@Type()`, as a place holder for full type information where required. Figure 8 illustrates how this is used.

A second aspect of the type information problem is is caused by the erasure semantics of the JVM, where generic information is discarded. Full generic type and annotation information is available in Java Bytecode for class declarations, field types and method types (via the `Signature` and `RuntimeVisibleAttributes` attributes); however, this is not available in the bytecode instructions themselves. For the most part, this is not a problem and our bytecode instructions map directly to Java Bytecodes. Four bytecode instructions, however, are problematic: `new`, `anewarray`, `checkcast` and `instanceof`. Each of these encodes a type argument, but the type information is only partial (generic and annotation information is missing) where our system requires the full type. To get around this, we have implemented a custom cast operator using a set of special annotations: `@Cast1()`, ..., `@Cast10()`. These can be used to provide a full type in much the same way as for `@Type()`. To embed this type in a Java Bytecode instruction, we provide a set of hard-coded generic functions: `<T> T Cast1(T) ... <T> T Cast10(T)`. When a call to one of these is encountered, our system examines the corresponding `@CastX` annotation and creates a fresh type object representing this. The following illustrates this mechanism:

```
@Cast1("@NonNull Integer @NonNull []")
void aMethod() {
 Integer[] x = new Integer[];
 ...
 x = JACK.Cast1(x);
 ...
}
```

Thus, our inference system infers the type '`@NonNull Integer @NonNull []`' for `x` after the custom cast. Note, the reason for having different cast operators is simply to allow several casts for different types in a method. The choice of ten different operators is purely arbitrary, and is really just a work-around prior to Java 7.

31

# 6  Case Studies

We have manually annotated and checked several real-world programs using our non-null type verifier. The largest practical hurdle was annotating Java's standard libraries. This task is enormous and we are far from completion. Indeed, finishing it by hand does not seem feasible; instead, we plan to develop (semi-)automatic procedures to help.

We now consider four real-world code bases which we have successfully annotated: the `java/lang` and `java/io` packages, the `jakarta-oro` text processing library and `javacc`, a well-known parser generator. Table 1 details these. Table 2 gives a breakdown of the checking time required, the number of annotations added, and the modifications needed for the program to type check. The checking time records the total time to check the benchmark in question, excluding the time taken to compile it by `javac`. This was generated on a 2GHz Intel Core II Duo machine running Windows XP, and is provided only as a rough indication of performance.

The most frequent modification listed in Table 2, "Field Load Fix", was for the field retyping issue identified in §5.5. To resolve this, we manually added a local variable into which the field was loaded before the null check. Many of these fixes may represent real concurrency bugs, although a deeper analysis of each situation is needed to ascertain this. The next most common modification, "Context Fixes", were for situations where the programmer knew a reference could not hold `null`, but our system was unable to determine this. These were resolved by adding runtime casts. Examples include:

- `Thread.getThreadGroup()` returns `null` when the thread in question has stopped. But, `Thread.currentThread().getThreadGroup()` will return a non-null value, as the current thread cannot complete `getThreadGroup()` if it has stopped! This assumption was encountered in several places.

- Another difficult situation for our tool is when the nullness of a method's return value depends either on its parameters, or on the object's state. A typical example is illustrated in Figure 9. More complex scenarios were also encountered where, for example, an array was known to hold non-null values up to a given index.

- As outlined in §5.6, `Hashtable.get(K)` returns `null` if no item exists for the key. A programmer may know that, for specific keys, `get()` cannot return `null` and so can avoid unnecessary `null` check(s). The `javacc` benchmark used many `hashtables` and many context fixes were needed as a result. In Table 2, the number of "Context Fixes" for this particular problem are shown in brackets.

- An odd situation encountered is where a method accepts a nullable parameter,

```
public void actionPerformed(@NonNull ActionEvent ae) {
  ...
  JFileChooser jfc = new JFileChooser();
  ...
  int rval = jfc.showOpenDialog(null);
  if(rval == JFileChooser.APPROVE_OPTION) {
    File f = jfc.getSelectedFile();
    filePath.setText(f.getCanonicalPath());
  ...
```

Fig. 9. A common scenario where the nullness of a method's return type depends upon its context; in this case, if rval==APPROVE_OPTION, then getSelectedFile() won't return null. To resolve this, we must add a "dummy" check that f!=null before the method call.

but passes this on to another requiring it be non-null. This works if the outer method catches NullPointerExceptions, as shown in Figure 10.

The "Other Fixes" category in Table 2 covers other miscellaneous modifications needed for the code to check. Figure 11 illustrates one such example. Most relate to the initialisation of fields. In particular, helper methods called from constructors which initialise fields are a problem. This is because our system checks each constructor initialises its fields, but does not account for those initialised in helper methods. To resolve this, we either inlined helper methods or initialised fields with dummy values before they were called.

The "Required Null Checks" counts the number of explicit null checks (as present in the original program's source), against the total number of dereference sites. Since, in the normal case, the JVM must check every dereference site, this ratio indicates the potential for speedup resulting from non-null types. Likewise, "Required Casts" counts the number of casts actually required, versus the total number present (recall from §5.8 that our tool automatically retypes local variables after instanceof tests, making numerous casts redundant; furthermore, the count doesn't include any cast that was required solely because of the limitation in the current classfile format discussed in §5.9).

We were also interested in whether or not our system could help documentation. That is, whether or not using types (which are automatically checked) instead of hand-written documentation (which is not) would be more reliable. In fact, it turns out that of the 1101 public methods in java/lang, 83 were mis-documented. That is, the Javadoc failed to specify that a parameter must not be null when, according to our system, it needed to be. We believe this is actually fairly good, all things considered, and reflects the quality of documentation for java/lang. Interestingly, many of the problem cases were found in java/lang/String.

| benchmark | version | LOC | source |
|---|---|---|---|
| java/lang package | 1.5.0 | 14K | java.sun.com |
| java/io package | 1.5.0 | 10.6K | java.sun.com |
| jakarta-oro | 2.0.8 | 8K | jakarta.apache.org/oro |
| javacc | 3.2 | 28K | javacc.dev.java.net |

Table 1

Details of our four benchmarks. Note, java/lang does not include subpackages.

| | Checking Time (ms) | Annotated Types | Parameter Annotations | Return Annotations | Field Annotations |
|---|---|---|---|---|---|
| java/lang | 2578 | 931 / 1599 | 363 / 748 | 327 / 513 | 241 / 338 |
| java/io | 3344 | 515 / 1056 | 322 / 672 | 96 / 200 | 97 / 184 |
| jakarta-oro | 1219 | 413 / 539 | 273 / 320 | 85 / 108 | 55 / 111 |
| javacc | 2802 | 420 / 576 | 199 / 278 | 53 / 65 | 168 / 233 |

| | Field Load Fixes | Context Fixes | Other Fixes | Required Null Checks | Required Casts |
|---|---|---|---|---|---|
| java/lang | 65 | 61 | 36 | 281 / 2550 | 51 / 96 |
| java/io | 59 | 82 | 21 | 207 / 2254 | 54 / 110 |
| jakarta-oro | 53 | 327 | 29 | 73 / 2014 | 29 / 33 |
| javacc | 109 | 137 (28) | 74 | 287 / 5700 | 141 / 431 |

Table 2

Breakdown of checking time, annotations added and related metrics. "Checking Time" gives the total time to check the benchmark, excluding the time required by javac to compile it. "Annotated Types" gives the total number of annotated parameter, return and field types against the total number of reference / array types in those positions. A breakdown according to position (i.e. parameter, return type or field) is also given. "Field Load Fixes" counts occurrences of the field retyping problem outlined in §5.5. "Context Fixes" counts the number of dummy null checks which had to be added. "Required Null Checks" counts the number of required null checks, versus the total number of dereference sites. Finally, "Required Casts" counts the number of required casts, versus the total number of casts.

```
public static Integer getInteger(String nm,
                                 Integer val) {
  String v = null;
  try { v = System.getProperty(nm); }
  catch (IllegalArgumentException e) {}
  catch (NullPointerException e) {}
  if(v != null) { ... }
  return val;
}
```

Fig. 10. Illustrating a surprising use of exceptions which causes problems for our tool. `System.getProperty()` requires a non-null parameter and, without the extra `null` check, our tool issues an error since `nm` is not marked `@NonNull`. This code is taken from `java/lang/Integer`.

```
public ThreadGroup(String name) {
  this(Thread.currentThread().getThreadGroup(), name);
  ...
```

Fig. 11. An interesting example from `java.lang.ThreadGroup`. The constructor invoked via the `this` call requires a non-null argument (and this is part of its Javadoc specification). Although `getThreadGroup()` can return `null`, it cannot here (as discussed previously). Our tool reports an error for this which cannot be resolved by inserting a dummy `null` check, since the `this` call must be the first statement of the constructor. Therefore, we either inline the constructor being called, or construct a helper method which can accept a `null` parameter.

## 7 Related Work

Several works have considered the problem of checking non-null types. Fähndrich and Leino investigated the constructor problem (see §5.3) and outlined a solution using raw types [25]. However, no mechanism for actually checking non-null types was presented. The FindBugs tool checks `@NonNull` annotations using a dataflow analysis that accounts for comparisons against `null` [38,37]. Their approach does not employ type aliasing and provides no guarantee that all potential errors will be reported. While this is reasonable for a lightweight software quality tool, it is not suitable for bytecode verification. ESC/Java also checks non-null types and accounts for the effect of conditionals [28]. The tool supports type aliasing (to some extent), can check very subtle pieces of code and is strictly more precise than our system. However, it relies upon a theorem prover which employs numerous transformations and optimisations on the intermediate representation, as well as a complex back-tracking search procedure. This makes it rather unsuitable for bytecode verification, where efficiency is paramount.

Ekman *et al.* implemented a non-null checker within the JastAdd compiler [22].

This accounts for the effect of conditionals, but does not consider type aliasing as there is little need in their setting where a full AST is available. To apply their technique to Java Bytecode would require first reconstructing the AST to eliminate type aliasing between stack and local variable locations. This would add additional overhead to the bytecode verification process, compared to our more streamlined approach. Pominville *et al.* also discuss a non-null analysis that accounts for conditionals, but again does not consider type aliasing [57]. They present empirical data suggesting many internal null checks can be eliminated, and that this leads to a useful improvement in program performance.

Hubert *et al.* formalised an inference tool for non-null annotations [41] and implemented it on top of Java Bytecode [40]. This system employs interprocedural pointer analysis to produce annotations with high precision; however, no effort is made to ensure the annotations produced can be checked in a modular (i.e. intraprocedural) fashion. As a result, their tool cannot be used in conjunction with our system (since ours is inherently intraprocedural). Spoto developed a similar system, and argued it was faster and more precise in practice [61]. Again, however, the analysis is inherently interprocedural, and does not generate annotations that can be checked in a modular fashion.

Chalin *et al.* empirically studied the ratio of parameter, return and field declarations which are intended to be non-null, concluding that $2/3$ are [12]. To do this, they manually annotated existing code bases, and checked for correctness by testing and with ESC/Java.

Recent work has considered type systems which support arbitrary *type qualifiers* [29,31,14,4,15]. These so-called "pluggable type systems" [10] allow optional type systems to be layered on existing languages without affecting their semantics. The idea is that type systems can and should evolve independently from the underlying language to allow for domain-specific type systems. JavaCOP provides an expressive language for writing type system extensions, such as non-null types [4]. This system cannot account for the effects of conditionals; however, as a work around, the tool allows assignment from a nullable variable `x` to a non-null variable if this is the first statement after a `x!=null` conditional. CQual is a flow-sensitive qualifier inference algorithm which supports numerous type qualifiers, but does not account for conditionals at all [29,31]. Building on this is the work of Chin *et al.* which also supports numerous qualifiers, including `nonzero`, `unique` and `nonnull` [14,15]. Again, conditionals cannot be accounted for, which severely restricts the use of `nonnull`. The Java Modelling Language (JML) adds formal specifications to Java and supports non-null types [17,11]. However, JML is strictly a specification language, and requires separate tools (such as ESC/Java) for checking. Like us, this approach faces the formidable challenge of providing specifications for the Java libraries. While some progress has been made here, the majority of the libraries remain without specifications.

Related work also exists on type inference for Object-Oriented languages (e.g. [53,52,44,3,56,21,63]). These, almost exclusively, assume the original program is completely untyped and employ set constraints (see [2,34]) for inferring types. In such systems, constraints are generated from the program text, formulated as a directed graph and then solved using an algorithm similar to transitive closure. When the entire program is untyped, type inference must proceed across method calls (known as *interprocedural analysis*) and this necessitates knowledge of the program's call graph (in the case of languages with dynamic dispatch, this must be approximated). Typically, a constraint graph representing the entire program is held in memory at once, making these approaches somewhat unsuited to separate compilation [53] Such systems share a strong relationship with other constraint-based program analyses (e.g. [34,24,1,62]), such as *alias* or *points-to* analysis (e.g. [30,59,5,54,47,55]).

Using set constraints for type inference is a very different approach to that we have taken. Set constraints provide a powerful abstraction which is more amenable to efficient solving algorithms than traditional dataflow analyses. In light of this, it may seem peculiar that we did not employ set constraints in our system. However, it is important to realise that all of the constraint-based systems mentioned so far are *flow-insensitive* — meaning they assume a variable always has the same type. In contrast, our system must be flow-sensitive to support variables being retyped inside conditional statements. Furthermore, although one can obtain a flow-sensitive constraint system using a program transformation known as Static Single Assignment (SSA) form [19,20], this approach is not yet sufficiently developed to deal with the effects of conditionals or value aliasing. The traditional method of dataflow analysis which we adopt, on the other hand, is well suited to both of these. Nevertheless, we believe it would be interesting to try and develop a constraint-based type inference system equivalent to that presented here. A starting point in this endeavor would be those extended SSA forms which provide some support for conditional statements [32,33].

Several works also use techniques similar to type aliasing, albeit in different settings. Smith *et al.* capture aliasing constraints between locations in the program store to provide safe object deallocation and imperative updates [60]; for example, when an object is deallocated the supplied reference and any aliases are retyped to *junk*. Chang *et al.* maintain a graph, called the *e-graph*, of aliasing relationships between elements from different abstract domains [13]; their least upper bound operator maintains a very similar invariant to ours. Zhang *et al.* consider aliasing of constraint variables in the context of set-constraint solvers [64].

## 8  Conclusion

We have presented a novel approach to the bytecode verification of non-null types. A key feature is that our system infers two kinds of information from conditionals: nullness information and type aliases. We have formalised this system for a subset of Java Bytecode, and proved soundness. Finally, we have detailed an implementation of our system and reported our experiences gained from using it. The tool itself is freely available from `http://ecs.victoria.ac.nz/~djp/JACK/`.

## References

[1]  A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2–3):79–111, 1999.

[2]  A. Aiken and E. L. Wimmers. Solving systems of set constraints. In *Proc. symposium on Logic in Computer Science*, pages 329–340. IEEE Computer Society Press, 1992.

[3]  A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proc. conference on Functional Programming Languages and Computer Architecture*, pages 31–41. ACM Press, 1993.

[4]  C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 57–74. ACM Press, 2006.

[5]  M. Berndl, O. Lhoták, F. Qian, L. J. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI)*, pages 196–207. ACM Press, 2003.

[6]  P. Bertelsen. Dynamic semantics of Java bytecode. *Future Generation Computer Systems*, 16(7):841–850, 2000.

[7]  J. Bloch. *Effective Java: Programming Language Guide*. The Java Series. Addison-Wesley, 2001.

[8]  M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 405–422. ACM, 2007.

[9] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer, 1993.

[10] G. Bracha. Pluggable type systems. In *Proc. Workshop on Revival of Dynamic Languages*, 2004.

[11] P. Chalin. Towards support for non-null types and non-null-by default in Java. In *Proc. Workshop on Formal Techniques for Java-like Programs*, 2006.

[12] P. Chalin and P. R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 227–247. Springer, 2007.

[13] B.-Y. E. Chang and K. R. M. Leino. Abstract interpretation with alien expressions and heap structures. In *Proc. VMCAI*, pages 147–163. Springer-Verlag, 2005.

[14] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI)*, pages 85–95. ACM Press, 2005.

[15] B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *Proceedings of the European Symposium on Programming (ESOP)*, 2006.

[16] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. In *Proceedings of the ACM workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 21–31, 1999.

[17] M. Cielecki, J. Fulara, K. Jakubczyk, and L. Jancewicz. Propagation of JML non-null annotations in Java programs. In *Proc. Principles and Practice of Programming in Java (PPPJ)*, pages 135–140. ACM Press, 2006.

[18] A. Coglio. Simple verification technique for complex java bytecode subroutines. *Concurrency and Computation: Practice and Experience*, 16(7):647–670, June 2004.

[19] R. Cytron, J. Ferrante, B. K. Rosen, M. K. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, pages 25–35. ACM Press, 1989.

[20] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[21] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 169–184. ACM Press, 1995.

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

[23] M. Ernst. Annotations on Java types, Java Specification Request (JSR) 308, `http://pag.csail.mit.edu/jsr308/`, 2008.

[24] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI)*, pages 85–96. ACM Press, 1998.

[25] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 302–312. ACM Press, 2003.

[26] C. Fecht and H. Seidl. An even faster solver for general systems of equations. In *Proc. Static Analysis Symposium*, pages 189–204. Springer, 1996.

[27] C. Fecht and H. Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *Proceedings of the European Symposium on Programming (ESOP)*, pages 90–104. Springer, 1998.

[28] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI)*, pages 234–245. ACM Press, 2002.

[29] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI)*, pages 192–203. ACM Press, 1999.

[30] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proc. Static Analysis Symposium*, pages 175–198. Springer, 2000.

[31] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI)*, pages 1–12. ACM Press, 2002.

[32] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, 1995.

[33] P. Havlak. Construction of thinned gated single-assignment form. In *Proc. Workshop on Languages and Compilers for Parallel Computing*, pages 477–499. Springer, 1994.

[34] N. Heintze. Set-based analysis of ML programs. In *Proc. conference on Lisp and Functional Programming*, pages 306–317. ACM Press, 1994.

[35] S. Horwitz, A. J. Demers, and T. Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.

[36] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 132–136. ACM, 2004.

[37] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the ACM workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 9–14. ACM Press, 2007.

[38] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proceedings of the ACM workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 13–19. ACM Press, 2005.

[39] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. mei W. Hwu. Java bytecode to native code translation: the Caffeine prototype and preliminary results. In *Proceedings of the International Symposium on Microarchitecture*, pages 90–97. IEEE Computer Society Press, 1996.

[40] L. Hubert. A non-null annotation inferencer for java bytecode. In *Proceedings of the ACM workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 36–42. ACM, 2008.

[41] L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *Proceedings of the International conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 132–149. Springer-Verlag, 2008.

[42] K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, and T. Nakatani. Effectiveness of cross-platform optimizations for a java just-in-time compiler. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 187–204. ACM Press, 2003.

[43] J. Jorgensen. Improving the precision and correctness of exception analysis in SOOT. Technical report, McGill University, Canada, 2003.

[44] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proc. conference on LISP and Functional Programming*, pages 193–204. ACM Press, 1992.

[45] G. Klein and M. Wildmoser. Verified bytecode subroutines. 2003.

[46] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3/4):235–269, 2003.

[47] O. Lhoták and L. J. Hendren. Context-sensitive points-to analysis: Is it worth it? In *Proceedings of the Conference on Compiler Construction (CC)*, pages 47–64. Springer, 2006.

[48] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, 1999.

[49] C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for @NonNull types. In *Proceedings of the Conference of Compiler Construction (CC)*, volume 4959 of *LNCS*, pages 229–244, 2008.

[50] J. Miecznikowski and L. Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In *Proceedings of the Conference on Compiler Construction (CC)*, pages 153–184, 2002.

[51] R. O'Callahn. A simple, comprehensive type system for java bytecode subroutines. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, pages 70–78, 1999.

[52] N. Oxhøj, J. Palsberg, and M. I. Schwartzbach. Making type inference practical. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 329–349. Springer, 1992.

[53] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 146–161. ACM Press, 1991.

[54] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Journal*, 12(4):309–335, 2004.

[55] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for C. *ACM Transactions on Programming Languages and Systems*, 30(1), 2008.

[56] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 324–340. ACM Press, 1994.

[57] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proceedings of the Conference on Compiler Construction (CC)*, pages 334–554, 2001.

[58] C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/ HOL. *Lecture Notes in Computer Science*, 1579:89–103, 1999.

[59] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 43–55. ACM Press, 2001.

[60] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proceedings of the European Symposium on Programming (ESOP)*, pages 366–381. Springer-Verlag, 2000.

[61] F. Spoto. Nullness analysis in boolean form. In *In Proceedings of the IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 21–30. IEEE Computer Society, 2008.

[62] B. D. Sutter, F. Tip, and J. Dolby. Customization of Java library classes using type constraints and profile information. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 585–610. Springer, 2004.

[63] T. Wang and S. F. Smith. Precise constraint-based type inference for Java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 99–117. Springer, 2001.

[64] Y. Zhang and F. Nielson. A scalable inclusion constraint solver using unification. In *Proceedings of Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, pages (121 – 137), 2008.