Java Bytecode Verification for @NonNull Types

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

Computer Science Group,
Victoria University of Wellington, NZ
{malechri,djp,alex,dymnikkost}@mcs.vuw.ac.nz

Abstract

Java's annotation mechanism allows us to extend its type system with non-null types. However, 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

`NullPointerException`s are a common error arising in Java programs when references holding `null` are dereferenced. Java 1.5 allows us to annotate types and, hence, to extend the type system with `@NonNull` types. An important step in the enforcement of such types is the bytecode verifier which must efficiently determine whether or not non-null types are used soundly. The standard bytecode verifier uses a dataflow analysis which is insufficient for this task. To address this, we present a novel, lightweight dataflow analysis ideally suited to the problem of verifying non-null types.

Java Bytecodes have access to a fixed size local variable array and stack [37]. 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 illustrates 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 the `Integer` type is inferred for that stack location as a result.

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 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*. More specifically, this paper makes the following contributions:

- We formalise our non-null bytecode verifier for a subset of Java Bytecode.
- We detail an implementation of our system for Java Bytecode.
- We report on our experiences with using our system on real-world programs.

While there has already been considerable work on non-null types (e.g. [43, 20, 33, 9, 17]), 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.

2 Preliminaries

We extend Java types to allow references to be declared as non-null and for arrays to hold non-null elements (in §5.4 we extend this to Java Generics). For example:

```
Vector v1;
@NonNull Vector v2;
@NonNull Integer @NonNull [] a1;
```

Here, `v1` is a *nullable* reference (one which may be `null`), while `v2` is a non-null reference (one which may not be `null`); similarly, `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. We formalise a cut-down version of the non-null types supported by our system using the following grammar:

$$\begin{array}{lll} \alpha & ::= & \texttt{@NonNull} \mid \epsilon \\ T & ::= & T\ \alpha\ \texttt{[]} \mid \alpha\ C \mid \mathit{null} \mid \bot \end{array}$$

Here, the special *null* type is given to the `null` value, ϵ 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 deadcode, etc).

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

`@NonNull Integer @NonNull []` $\leq$ `Integer @NonNull []`

In fact, we require all array element types be identical between subtypes[1] A formal definition of the subtype relation for our simplified non-null type language is given in Figure 1.

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

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

$$\frac{\alpha_1 \leq \alpha_2 \quad C \texttt{ extends } B}{\alpha_1\ C\ \leq \alpha_2\ B} \qquad \text{(S-CLASS)}$$

$$\frac{\alpha_1 \leq \alpha_2}{T_1\ \alpha_1\ \texttt{[]} \leq T_1\ \alpha_2\ \texttt{[]}} \qquad \text{(S-ARRAYa)}$$

$$\frac{}{T_1\ \alpha\ \texttt{[]} \leq \alpha\ \texttt{java.lang.Object}} \qquad \text{(S-ARRAYb)}$$

$$\frac{}{\bot \leq T\ \alpha\ \texttt{[]}} \quad \frac{}{\bot \leq \alpha\ C} \quad \frac{}{\bot \leq \textit{null}} \qquad \text{(S-BOTa, S-BOTb, S-BOTc)}$$

$$\frac{}{\textit{null} \leq T_1\texttt{[]}} \quad \frac{}{\textit{null} \leq C} \qquad \text{(S-NULLa, S-NULLb)}$$

Figure 1: Subtyping rules for non-null Java types. We assume reflexivity and transitivity, that `java.lang.Object` is the class hierarchy root and, hence, is also $\top$.

An important property of our subtype relation is that it forms a *complete lattice* (i.e. that every pair of types T_1, T_2 has a unique least upper bound, $T_1 \sqcup T_2$, and a unique greatest lower bound, $T_1 \sqcap T_2$). This helps ensure termination of our non-null verification algorithm. A well-known problem, however, is that Java's subtype relation does not form a complete lattice [35]. This arises because two classes can share the same super-class and implement the same interfaces; thus, they may not have a unique least upper bound. To resolve this, we adopt the standard solution of ignoring interfaces entirely and, instead, treating interfaces as type `java.lang.Object`. This works because Java supports only single inheritance between classes. This is the approach taken in Sun's Java Bytecode verifier and, hence, our system is no less general than it.

3 Non-null Type Verification

Our non-null type verification algorithm infers the nullness of local variables at each point within a method. We assume method parameters, return types and fields are already annotated with `@NonNull`. 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, the method is type safe and cannot throw a `NullPointerException`. 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; it's edges represent the effects of the instructions.

We now formalise this construction process for methods. Constructors are ignored for simplicity and discussed informally in §5. Also, while the full Java Bytecode in-

struction set is supported, only a subset is considered here for brevity. Figure 2 details those bytecode instructions we are considering here.

3.1 Abstract Store

In the Java Virtual Machine (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). Our system models this using an abstract store, which we formalise as (Σ, Γ, κ), where Σ is the *abstract meta-heap*, Γ is the *abstract location array* and κ is the *stack pointer* which identifies the first free location on the stack. Here, Γ 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 $\kappa \leq n$). A type reference is a reference to a *type object* which, in turn, can be thought of as a non-null type with identity. Thus, we can have two distinct type objects representing the same non-null type. Crucially, this types-as-references approach allows two abstract locations to be *type aliases*; that is, refer to the same type object. For example, in the following abstract store, locations 0 and 2 are type aliases:

$$\Sigma = \{r_1 \mapsto \texttt{@NonNull Integer}, r_2 \mapsto \texttt{String}\}, \Gamma = \{0 \mapsto r_1, 1 \mapsto r_2, 2 \mapsto r_1\}, \kappa = 3$$

Here, the abstract meta-heap, Σ, maps type references to non-null types. It's called a *meta-heap* as Σ does not abstract the program heap; rather it is an internal structure used only to enable type aliasing.

Definition 1. An abstract store (Σ, Γ, κ) is well-formed iff $dom(\Gamma) = \{0, \ldots, n-1\}$ for some n, $ran(\Gamma) \subseteq dom(\Sigma)$ and $0 \leq \kappa \leq n$.

3.2 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 bytecode instructions considered in our formalism are given in Figure 3. Here, $\Gamma[r_1/r_2]$ generates an abstract store from Γ where all abstract locations holding r_1 now hold r_2. Several helper functions are used: **fieldT**$(\texttt{O}, \texttt{N})$, returns the type of field `N` in class `O`; **methodT**$(\texttt{O}, \texttt{M})$ returns the type of method `M` in class `O`; **thisMethT**$()$ gives the current method's type; finally, **validNewT**(T_1) holds if $T_1 \neq$ `@NonNull` T_2 α `[]` for any T_2. The latter prevents creation of arrays holding `@NonNull` elements, as Java always initialises array elements with `null` (see §5).

A useful illustration of our abstract semantics is the `arrayload` bytecode. This requires the array index on top of the stack, followed by the array reference itself; these are popped off the stack and the indexed element is loaded back on. Looking at the `arrayload` rule, we see κ decreases by one, indicating the net effect is one less element on the stack. The notation $\Gamma[\kappa-2 \mapsto r]$ indicates the abstract store is updated so that abstract location $\kappa-2$ now holds type reference r; thus, r has been pushed onto the stack and represents the loaded array element. The reference on top of the stack is ignored since this represents the actual index value, and is of no concern. The constraint $r \notin \Sigma$ ensures r references a *fresh* type object; such constraints are used to ensure an

Instruction	Effect on Stack / Description
`load` i	$[\ldots] \Rightarrow [\ldots, ref]$ Load reference from local variable i onto stack
`store` i	$[\ldots, ref] \Rightarrow [\ldots]$ Pop reference off stack and store in local variable i
`loadnull`	$[\ldots] \Rightarrow [\ldots, \texttt{null}]$ Load `null` constant onto stack.
`arrayload`	$[\ldots, arrayref, index] \Rightarrow [\ldots, value]$ Load value from array at index *index* and push onto stack
`arraystore`	$[\ldots, arrayref, index, value] \Rightarrow [\ldots]$ Pop value of stack and store into array at index *index*
`getfield O.N`	$[\ldots, ref] \Rightarrow [\ldots, value]$ Load value from field `N` of object referenced by *ref*.
`putfield O.N`	$[\ldots, ref, value] \Rightarrow [\ldots]$ Pop value off stack and write to field `N` of object referenced by *ref*
`invoke O.M`	$[\ldots, ref, p_0, \ldots, p_n] \Rightarrow [\ldots, value]$ Invoke method `M` on object referenced by *ref*.
`new` T	$[\ldots] \Rightarrow [\ldots, ref]$ Allocate and construct new object of type T and place on stack.
`ifceq` *dest*	$[\ldots, ref_1, ref_2] \Rightarrow [\ldots]$ Branch to *dest* if ref_1 and ref_2 reference the same object.
`return`	$[ref] \Rightarrow []$ Return from method using ref as return value.

Figure 2: Cut-down Java bytecode instruction set. Values are either object or array references and we assume all methods return a value. Our instruction set also includes a `goto` bytecode which transfers control, but otherwise has no effect on the stack. Note, `ifceq` is shorthand for `if_cmpeq`.

abstract location is not type aliased with any other. Another constraint ensures the array reference is non-null, thus protecting against a `NullPointerException`.

Considering the remaining rules from Figure 3, 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, $T_1 \sqcap T_2$ (recall §2). 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, $T_1 - T_2$, 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`; after a comparison `o!=null`, `null` is removed from this set. Thus, it is defined as follows:

Definition 2. $T_1 - T_2$ is `@NonNull` T, if $T_1 = \alpha\ T \wedge T_2 = null$, and 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.

Finally, the Java Bytecodes not considered in Figures 2 + 3 include all arithmetic operations (e.g. `iadd`, `imul`, etc), stack manipulators (e.g. `pop`, `dup`, etc), other branching primitives (e.g. `ifnonull`, `tableswitch`, etc), synchronisation primitives (e.g. `monitorenter`, etc) and other miscellaneous ones (e.g. `instanceof`, `checkcast`, `athrow` and `arraylength`). It is easy enough to see how our abstract semantics extends to these and our implementation (see §5) supports them all.

3.3 An Example

Recall our non-null verification system 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, (Σ, Γ, κ), 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 throw a `NullPointerException`.

Figure 4 illustrates the bytecode instructions for a simple method and its corresponding abstract representation. 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 4 has type `Test`; the remainder have nullable type `Integer`, with each referring to a unique type object (since we must conservatively assume parameters are not aliased on entry).

In Figure 4, 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 throw a `NullPointerException`.

We now consider what happens at join points in the CFG. The `return` instruction in Figure 4 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 4, the store coming out of the `invoke` instruction has a type aliasing relationship, whereas that coming out of the `loadnull` instruction does not; also, in the former, location 2 has type `@NonNull Integer`, whilst the latter gives

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

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

$$\frac{r \notin \Sigma \quad \Sigma' = \Sigma \cup \{r \mapsto \mathit{null}\}}{\texttt{loadnull} : \Sigma, \Gamma, \kappa \longrightarrow \Sigma', \Gamma[\kappa \mapsto r], \kappa+1}$$

$$\frac{\begin{array}{c}\text{validNewT}(T) \\ r \notin \Sigma \quad \Sigma' = \Sigma \cup \{r \mapsto \texttt{@NonNull}\ T\}\end{array}}{\texttt{new}\ T : \Sigma, \Gamma, \kappa \longrightarrow \Sigma', \Gamma[\kappa \mapsto r], \kappa+1}$$

$$\frac{\Sigma(\Gamma(\kappa-2)) = T\ \texttt{@NonNull []} \quad r \notin \Sigma \quad \Sigma' = \Sigma \cup \{r \mapsto T\}}{\texttt{arrayload} : \Sigma, \Gamma, \kappa \longrightarrow \Sigma', \Gamma[\kappa-2 \mapsto r], \kappa-1}$$

$$\frac{\Sigma(\Gamma(\kappa-1)) = T_1 \quad T_1 \leq T_2 \quad \Sigma(\Gamma(\kappa-3)) = T_2\ \texttt{@NonNull []}}{\texttt{arraystore} : \Sigma, \Gamma, \kappa \longrightarrow \Sigma, \Gamma, \kappa-3}$$

$$\frac{\begin{array}{c}\Sigma(\Gamma(\kappa-1)) = \texttt{@NonNull}\ C \quad r \notin \Sigma \quad \Sigma' = \Sigma \cup \{r \mapsto T\} \\ T = \text{fieldT}(\texttt{O}, \texttt{N})\end{array}}{\texttt{getfield O.N} : \Sigma, \Gamma, \kappa \longrightarrow \Sigma', \Gamma[\kappa-1 \mapsto r], \kappa}$$

$$\frac{\begin{array}{c}\Sigma(\Gamma(\kappa-1)) = T_1 \quad \Sigma(\Gamma(\kappa-2)) = \texttt{@NonNull}\ C \\ T_2 = \text{fieldT}(\texttt{O}, \texttt{N}) \quad T_1 \leq T_2\end{array}}{\texttt{putfield O.N} : \Sigma, \Gamma, \kappa \longrightarrow \Sigma, \Gamma, \kappa-2}$$

$$\frac{\begin{array}{c}(P_1, \ldots, P_n) \rightarrow T_r = \text{methodT}(\texttt{O}, \texttt{M}) \\ \Sigma(\Gamma(\kappa-n)), \ldots, \Sigma(\Gamma(\kappa-1)) = T_1, \ldots, T_n \quad \Sigma(\Gamma(\kappa-(n+1))) = \texttt{@NonNull}\ C \\ r \notin \Sigma \quad \Sigma' = \Sigma \cup \{r \mapsto T_r\} \\ T_1 \leq P_1, \ldots, T_n \leq P_n\end{array}}{\texttt{invoke O.M} : \Sigma, \Gamma, \kappa \longrightarrow \Sigma', \Gamma[\kappa-(n+1) \mapsto r], \kappa-n}$$

$$\frac{\begin{array}{c}(P_1, \ldots, P_n) \rightarrow T_r = \text{thisMethT}() \\ \Sigma(\Gamma(\kappa-1)) = T \quad T \leq T_r\end{array}}{\texttt{return} : \Sigma, \Gamma, \kappa \longrightarrow \emptyset, \emptyset, 0}$$

$$\frac{\begin{array}{c}r_1 = \Gamma(\kappa-2) \quad r_2 = \Gamma(\kappa-1) \\ \Sigma(r_1) = T_1 \quad \Sigma(r_2) = T_2 \quad r_3 \notin \Sigma \quad \Sigma' = \Sigma \cup \{r_3 \mapsto T_1 \sqcap T_2\}\end{array}}{\texttt{ifceq} : \Sigma, \Gamma, \kappa \xrightarrow{\mathit{true}} \Sigma', \Gamma[r_1/r_3, r_2/r_3], \kappa-2}$$

$$\frac{\begin{array}{c}r_1 = \Gamma(\kappa-2) \quad r_2 = \Gamma(\kappa-1) \\ \Sigma(r_1) = T_1 \quad \Sigma(r_2) = T_2 \quad r_3, r_4 \notin \Sigma \quad \Sigma' = \Sigma \cup \{r_3 \mapsto T_1 - T_2, r_4 \mapsto T_2 - T_1\}\end{array}}{\texttt{ifceq} : \Sigma, \Gamma, \kappa \xrightarrow{\mathit{false}} \Sigma', \Gamma[r_1/r_3, r_2/r_4], \kappa-2}$$

Figure 3: Abstract semantics for Java Bytecodes considered. Note, `ifceq` stands for `if_cmpeq`.

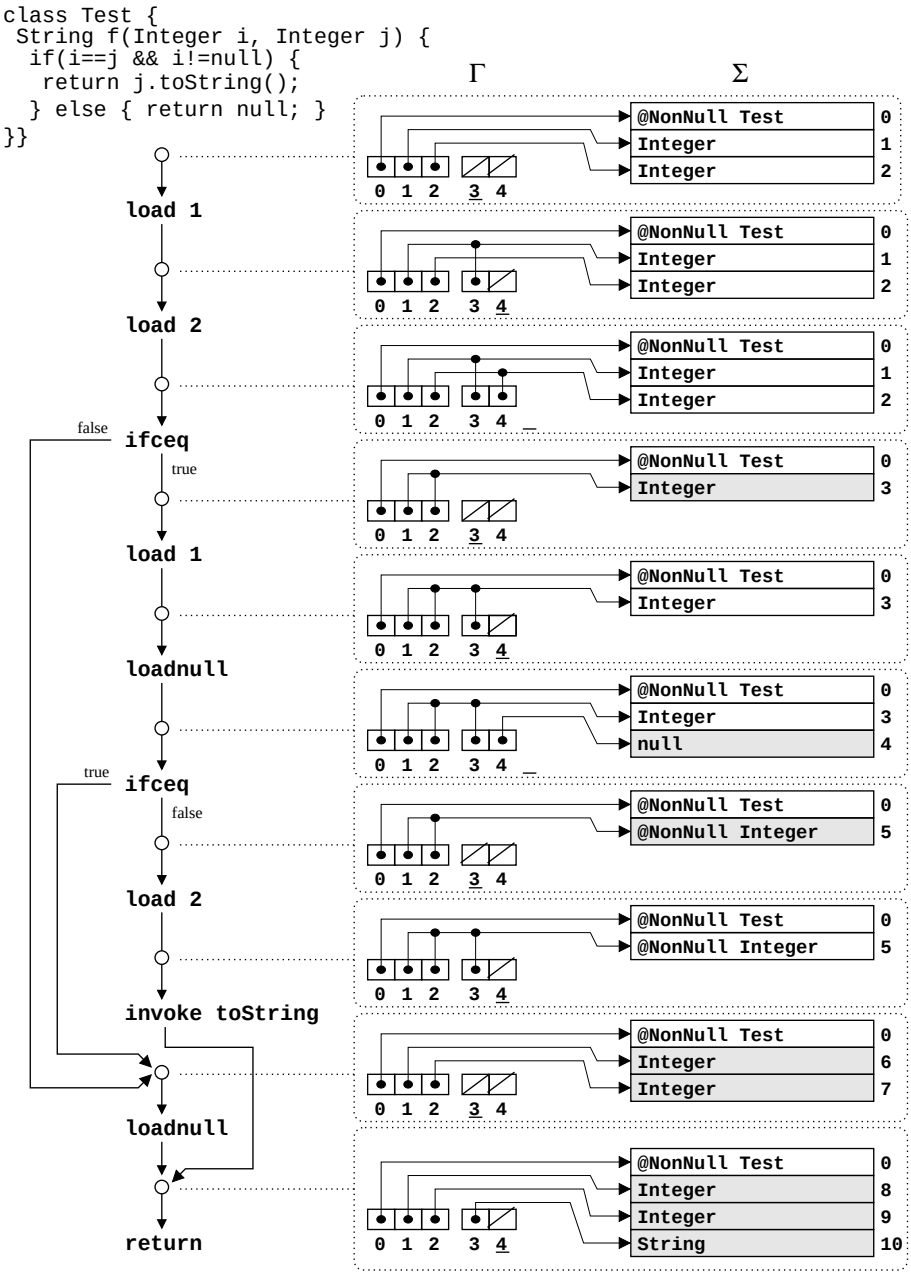

Figure 4: Bytecode representation of a simple Java Method (source given above) and the state of the abstract store, (Σ, Γ, κ), going into each instruction. The value of κ is indicated by the underlined abstract location; when the stack is full, this points past the last location. The type objects in Σ 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.

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:

Definition 3. Let $S_1 = (\Sigma_1, \Gamma_1, \kappa), S_2 = (\Sigma_2, \Gamma_2, \kappa)$ be well-formed abstract stores. Then $S_1 \leq S_2$ iff $\forall x, y \in \{0 \ldots \kappa\} \big[\Sigma_1(\Gamma_1(x)) \leq \Sigma_2(\Gamma_2(x)) \wedge (\Gamma_2(x) = \Gamma_2(y) \implies \Gamma_1(x) = \Gamma_1(y))\big]$.

Note, Definition 3 requires κ 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:

Definition 4. Let $G = (V, E)$ be the control-flow graph for a method M. Then, the dataflow equations for M are given by $S_M(y) = \bigsqcup_{x \xrightarrow{l} y \in E} f(I(x), S_M(x), l)$.

Here, the *transfer function*, f, is defined by the abstract semantics of Figure 3, $I(x)$ gives the bytecode at node x, and the edge label, l, distinguishes the true/false branches for `ifceq`. Thus, $S_M(y)$ gives the abstract store going into y. Finally, the dataflow equations can be solved as usual by iterating to a fixed point using a *worklist algorithm*. In doing this, our algorithm pays no special attention to the order in which nodes in the control-flow graph are visited. More complex iteration strategies (see e.g. [31, 6, 21]) and optimisations (e.g. [22, 19, 40]) could be used here.

4 Soundness

We now demonstrate that our algorithm *terminates* and is *correct*; that is, if a method passes our verification process, then it cannot throw a `NullPointerException`.

Several previous works have formalised Java Bytecode and shown the standard verification algorithm is correct (e.g. [28, 35]). Our system essentially operates in an identical fashion to the standard verifier, except that it additionally maintains type aliases and propagates `@NonNull` annotations. Indeed, our abstract semantics of Figure 3 would be identical to previous work (e.g. [35]) if we removed the requirement for `@NonNull` types at dereference sites and prohibited type aliasing relationships. Thus, we leverage upon these existing works to simplify our proof by restricting attention to those details particular to our system.

An important issue regarding our formalism is that it applies only to *methods*, not *constructors*. The reason for this is detailed in §5. Therefore, in the following, we assume all fields annotated with `@NonNull` are correctly initialised.

4.1 Termination

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

Strictly speaking, Definition 3 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 as type objects are Java Objects and, hence, $r_1 \neq r_2$ simply means different object addresses. Now, while S_3 and S_4 are distinct, they are also *equivalent*:

Definition 5. Let $S_1 = (\Sigma_1, \Gamma_1, \kappa), S_2 = (\Sigma_2, \Gamma_2, \kappa)$, 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 5 is that our subtype operator is not a partial order (since this requires anti-symmetry); rather, it is a *preorder*.

Lemma 1. *Let $S_1 = (\Sigma_1, \Gamma_1, \kappa), S_2 = (\Sigma_2, \Gamma_2, \kappa)$ with $dom(\Gamma_1) = dom(\Gamma_2)$. 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. Firstly, $U \neq \emptyset$ since $(\{r_1 \mapsto \texttt{Object}, \ldots, r_n \mapsto \texttt{Object}\}, \{1 \mapsto r_1, \ldots, n \mapsto r_n\}, \kappa)$ is an upper bound for any store where $dom(\Gamma) = \{1, \ldots, n\}$. Now, suppose for contradiction that we have two $u_1, u_2 \in U$, where $u_1 \not\equiv u_2$. Then, by Definition 5, 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 $u_1 = (\Sigma_{u1}, \Gamma_{u1}, \kappa)$ and $u_2 = (\Sigma_{u2}, \Gamma_{u2}, \kappa)$, then following Definition 3, there are three cases to consider:

- **i)** $\exists x, y$ where $\Sigma_{u1}(\Gamma_{u1}(x)) \not\leq \Sigma_{u2}(\Gamma_{u2}(x))$ and $\Sigma_{u2}(\Gamma_{u2}(y)) \not\leq \Sigma_{u1}(\Gamma_{u1}(y))$. However, we know $\{S_1, S_2\} \leq u_1$ and $\{S_1, S_2\} \leq u_2$, which implies some T exists where $\Big\{\Sigma_1(\Gamma_1(x)), \Sigma_2(\Gamma_2(x))\Big\} \leq T \leq \Big\{\Sigma_{u1}(\Gamma_{u1}(x)), \Sigma_{u2}(\Gamma_{u2}(x))\Big\}$ (otherwise, the subtype relation for non-null types is not a complete lattice which it is, recall §2). 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)** $\exists x_1, x_2, y_1, y_2$ where $\Gamma_{u1}(x_1) = \Gamma_{u1}(x_2)$, but $\Gamma_{u2}(x_1) \neq \Gamma_{u2}(x_2)$ and $\Gamma_{u1}(y_1) \neq \Gamma_{u1}(y_2)$, but $\Gamma_{u2}(y_1) = \Gamma_{u2}(y_2)$. Since $\{S_1, S_2\} \leq \{u_1, u_2\}$, it must hold in S_1 that $\Gamma_1(x_1) = \Gamma_1(x_2)$ and $\Gamma_1(y_1) = \Gamma_1(y_2)$ and, similarly, in S_2. This is a contradiction, as it implies in any minimal upper bound of S_1 and S_2 we have $\Gamma(x_1) = \Gamma(x_2)$ and $\Gamma(y_1) = \Gamma(y_2)$.
- **iii)** $\exists x, y_1, y_2$ where $\Sigma_{u2}(\Gamma_{u2}(x)) \leq \Sigma_{u1}(\Gamma_{u1}(x))$, but $\Gamma_{u1}(y_1) = \Gamma_{u1}(y_2)$ and $\Gamma_{u2}(y_1) \neq \Gamma_{u2}(y_2)$ (observe there is a symmetric case to this, with u_1 and u_2 reversed). In this case, we can construct a third upper bound u_3, where $\Sigma_{u3}(\Gamma_{u3}(x)) = \Sigma_{u2}(\Gamma_{u2}(x))$, and $\Gamma_{u3}(y_1) = \Gamma_{u3}(y_2)$. Thus, $\{S_1, S_2\} \leq u_3 \leq \{u_1, u_2\}$, which is a contradiction since it implies neither u_1 nor u_2 are minimal upper bounds.

□

Lemma 2. *The dataflow equations from Definition 4 are monotonic.*

Proof. Demonstrating f is monotonic requires showing each transition from our abstract semantics is monotonic. That is, if $i : S_1 \longrightarrow S_2$, then for all $i : S_1' \longrightarrow S_2'$ where $S_1 \leq S_1'$, it holds that $S_2 \leq S_2'$. Now, a given instruction i (ignoring `ifceq` for now) always manipulates

Γ and Σ in the same way, regardless of input store (e.g. `store 1` always overwrites location `1` with the top of the stack). Therefore, any type aliases introduced in S_2' by i must have also been introduced in S_2 and, likewise, any destroyed in S_2 by i must also have been destroyed in S_2'. Furthermore, 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 $\leq$ its counterpart in S_1', every location in S_2 is $\leq$ its counterpart in S_2'. For `ifceq`, we also require $T_1 \sqcap T_2 \leq T_1' \sqcap T_2'$ and $T_1 - T_2 \leq T_1' - T_2'$ if $T_1 \leq T_1'$ and $T_2 \leq T_2'$. The former holds as $\leq$ since forms a complete lattice (recall §2), whilst the latter follows immediately from its definition. □

4.2 Correctness

We now show the type aliasing information maintained is correct (Lemma 3), and that any location with `@NonNull` type cannot hold `null` (Lemma 4). This yields an overall correctness result for the subset of Java Bytecode we have formalised (Theorem 1).

Definition 6. A Java method is considered to be valid if it passes the standard JVM verification process [37].

The consequences of Definition 6 include: all conventional types (i.e. ignoring non-null types) are used safely; stack sizes are always the same at the meet points; method and field lookups always resolve; etc.

Lemma 3. *Let $S_M = (\Sigma, \Gamma, \kappa)$ be the abstract store for an instruction in a valid method M. If $\{l_1 \mapsto r, l_2 \mapsto r\} \subseteq \Gamma$, then the local array/stack locations represented by l_1, l_2 refer to the same object or array immediately before that instruction in any execution trace of M.*

Proof. Assume this is not the case. Then, there exists some instruction $i_1 \in M$ where $S_M(i_1) = (\Sigma_1, \Gamma_1, \kappa)$, $\{l_1 \mapsto r, l_2 \mapsto r\} \subseteq \Gamma_1$, but the local array/stack locations represented by l_1, l_2 are not aliased at that point during some execution trace of M. Now, let $i_0 \rightsquigarrow i_1$ be any path through the control-flow graph such that l_1, l_2 are not type aliases in $S_M(i_0)$, but are type aliases in $f(i_0, S_M(i_0))$ (recall f is the transfer function from Definition 4). Let us further assume (without loss of generality) that the local array/stack locations represented by l_1, l_2 are not aliased immediately after i_0 in any execution trace of M. Such a path $i_0 \rightsquigarrow i_1$ must exist as: firstly, the least upper bound operator, $\sqcup$, conservatively retains type aliases; secondly, the store on entry to the CFG contains no aliases by construction.

Thus, we have narrowed our search to one instruction, i_0, which introduced the incorrect type alias. We now demonstrate, by case analysis on the instruction types of Figure 3, that no instruction can introduce an incorrect type alias, leading to a proof by contradiction of Lemma 3. There are four main cases to consider:

- **i)** `arraystore`, `putfield` `O.N` and `return` cannot introduce type aliases since they do not update Γ.
- **ii)** `loadnull`, `new` T, `arrayload`, `getfield` `O.N` and `invoke` `O.M` also cannot introduce type aliases since they only assign fresh locations to locations in Γ
- **iii)** `load` i and `store` i both introduce type aliases between the local array and the stack. However, this correctly reflects their semantics.
- **iv)** `ifceq`. We must consider the true and false branch separately. On the true branch, a type alias is created between all locations l where $\Gamma(l) = r_1$ or $\Gamma(l) = r_2$. But, as this is the true branch we know the objects represented by r_1 and r_2 are equal according to Java's reference comparison. 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.

□

Lemma 4. *Let $S_M = (\Sigma, \Gamma, \kappa)$ be the abstract store for an instruction in a valid method M. Assume the parameters of M, the fields accessed by M and the return value of all methods invoked by M respect their declared non-null type. Then, if $\{l \mapsto r\} \subseteq \Gamma \wedge \{r \mapsto$ `@NonNull` $T\} \subseteq \Sigma$, the local array/stack location represented by l does not hold* `null` *immediately before that instruction in any execution trace of M.*

Proof. Assume this is not the case. Then, there exists some instruction $i_1 \in M$ and location $l \in dom(\Gamma_1)$ where $S_M(i_1) = (\Sigma_1, \Gamma_1, \kappa)$, $\Sigma_1(\Gamma_1(l)) =$ `@NonNull` T, but the location represented by l holds `null` at that point in some execution trace of M. Following a similar argument as for the proof of 4, there must exist an instruction i_0 where the incorrect type `@NonNull` T was first introduced. Again, we demonstrate, by case analysis on the instruction types of Figure 3, that no instruction can introduces an incorrect type `@NonNull` T, leading to a proof by contradiction of Lemma 4. There are three main cases to consider:

- **i)** `load` i, `store` i, `loadnull`, `arrayload`, `arraystore`, `getfield` `O.N`, `putfield` `O.N`, `invoke` `O.M` and `return` do not introduce any new `@NonNull` types other than for field and return types (which are correct by assumption for Lemma 4) and, thus, these bytecodes can be safely ignored.
- **ii)** `new` T. This bytecode introduce a type `@NonNull` T and places it on the stack. However, it is important to observe that our `new` instruction explicitly prohibits creation of arrays with `@NonNull` elements. Furthermore, since this instruction does actually create a new object and place a reference to it on the stack, the introduction `@NonNull` T is safe.
- **iii)** `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 2, 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 `loadnull` 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. Lemma 3 guarantees this is not the case, however.

□

Theorem 1. *If our abstract representation can be correctly constructed for all methods in a Java Bytecode program, then no method will throw a* `NullPointerException`*, assuming all fields are correctly initialised.*

Proof. Consider the call graph representing the execution of any valid Java Bytecode program starting at a special method `public static void main(String @NonNull [])` (we assume this takes a non-null array of possibly-null `String`s).

Now, there are two ways in which a `NullPointerException` could be thrown: either a location assigned a possibly-null type T by our system is dereferenced; or, a location assigned a type `@NonNull` T actually holds `null` in some execution trace of the program. The former is prohibited by the judicious use of subtyping constraints in the abstract semantics of Figure 3, while Lemma 4 ensures the latter could only happen in a method M if one of its assumptions is broken (i.e. a parameter, field, or the return value does not respect its non-null type).

Therefore, we now demonstrate by induction that the assumptions of Lemma 4 cannot be broken by a program consisting entirely of methods for which our abstract representation can be constructed (assuming all fields are correctly initialised). The induction is over the sequence of method calls arising in any execution trace of the program.

Inductive Hypothesis: If the parameters supplied to the k^{th} method call respected their non-null type, then so do the parameters to the $k^{th}+1$ method. Likewise, if all fields respected their non-null type immediately before the k^{th} method call, then they will also immediately before the $k^{th}+1$ method call.

k=1: The parameters supplied to the first method call respected their `@NonNull` type, as did all fields at this point. This call was to the `main` method, and it is reasonable to assume that the supplied parameters are correct (in fact, non-nullness of the array argument is implied by the JVM spec [37]).

k=n: Lemma 4 guarantees all local array/stack locations and fields respect their non-null type before the `invoke` bytecode for the $k^{th}+1$ method call. To understand why, suppose the current method makes no method calls (hence, the k^{th} call was into this method). Since the parameters and fields respected their non-null type on entry, we know the first assignment to a field will respect its non-null type and, hence, so will the second, and so on. Thus, by the end of the method, all fields still respect their non-null types and, furthermore, so must the stack location used by the `return` bytecode. Hence, the return value supplied to the method's caller respects its non-null type, meaning any subsequent field assignments or return values in that method do as well and so on, up to the $k^{th}+1$ method call. On the other hand, if the current method does make method calls, we can apply a similar line of reasoning to conclude that, since everything held on entry to the method, so it did for the first call it makes and, hence, the return from that call, and then for the next call, and so on. □

5 Implementation

We have implemented our system on top of Java Bytecode and we now discuss many aspects not covered by our discussion so far.

5.1 Constructors.

An important problem arises when dealing with constructors and, more specifically, default values for fields [20]. 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 5 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 [20], we use a secondary type annotation, `@Raw`, for references to indicate the object referred to may not be initialised. 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. As `@Raw` is strictly a supertype of a normal reference, 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.

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

Figure 5: 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!

Our use of `@Raw` here is a somewhat simplified version of that outlined in [20], where a more fine-grained type is used which can indicate exactly which fields are uninitialised. Finally, static field initialisers present an awkward problem, since an object's constructor can, in principle, access any static field, including those which are awaiting initialisation! While some proper solutions are possible for this, we simply allow static initialisers for types defined in the standard library; this works, since we know such types in cannot access static fields defined in client programs. However, it is not a general solution to the problem.

5.2 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*.

5.3 Field Retyping.

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

```
class Test {
 Integer field;
 void f() {
  if(field != null) {
   field.toString()
}}}
```

```
0.  load 0
2.  getfield Test.field
5.  ifnull 16
8.  load 0
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` or we are in a `synchronized` block involving the object in question. 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.

5.4 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 1 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`. However, 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` $\not\leq$ `@NonNull String`. To resolve this, 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.

5.5 Casting + Arrays.

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 an explicit *cast* to `@NonNull Integer[]` when the programmer knows the array has been fully initialised. Casts from nullable to non-null types are implemented as runtime checks which fail by throwing `ClassCastException`s. Their use weakens Theorem 1, since we are essentially trading `NullPointerException`s for `ClassCastException`s. While this is undesirable, it is analogous to the issue of downcasts in Object-Oriented Languages.

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

Figure 6: 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.

5.6 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, rending the cast redundant (note, an `instanceof` test never passes on `null`).

5.6.1 Finding Least Upper Bounds.

Finding least upper bounds for the subtype relation of Figure 1 is easy enough using a depth-first search of the class hierarchy. To Find the least upper bound of two abstract stores efficiently, we (essentially) maintain the abstract store as a bidirectional graph from abstract locations to type objects. This allows us to efficiently determine, for a given type object, the abstract locations aliasing it. To generate the least upper bound, we begin with an empty abstract location array, and then consider each of the locations $\{0 \ldots \kappa\}$ in turn; for each, we construct a fresh type object representing the least upper bound of the types at that location in each input store; then, we compute the intersection of the alias relationships this location is involved in across all input stores, assigning the new type object to each location in this; finally, we move on to the next location to be considered, whilst ignoring those which have been previously assigned during this process.

5.7 Type Annotations.

The Java Classfile format doesn't allow annotations on generic parameters or in the array type reference position. We expect future versions of Java will support such types directly and, indeed, work is already underway in this regard [18].

Therefore, we currently employ 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 6 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 type inference system infers the type '`@NonNull Integer @NonNull []`' for `x` after the custom cast.

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 annotations added, and the modifications needed for the program to type check. The most frequent modification, "Field Load Fix", was for the field retyping issue identified in §5.3. 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 dummy null checks. Examples include:

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

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 sub-packages.

	Annotated Types	Parameter Annotations	Return Annotations	Field Annotations	
`java/lang`	931 / 1599	363 / 748	327 / 513	241 / 338	
`java/io`	515 / 1056	322 / 672	96 / 200	97 / 184	
`jakarta-oro`	413 / 539	273 / 320	85 / 108	55 / 111	
`javacc`	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 annotations added and related metrics. "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 §5.3. "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 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());
  ...
```

Figure 7: 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.

- 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 7. 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 §5.4, `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, but passes this on to another requiring it be non-null. This works if the outer method catches `NullPointerException`s, as shown in Figure 8.

The "Other Fixes" category in Table 2 covers other miscellaneous modifications needed for the code to check. Figure 9 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 §5.6 that our tool automatically retypes local variables after `instanceof` tests, making numerous casts redundant.)

We were also interested in whether or not our system could help documentation. In fact, it turns out that of the 1101 public methods in `java/lang`, 83 were misdocumented. 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 pretty 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`.

Finally, a comment regarding performance seems prudent, since we have elided performance results for brevity. In fact, the performance of our system is very competitive with the standard bytecode verifier. This is not surprising, since our system uses a very similar algorithm to the standard bytecode verifier, albeit extended with type aliasing.

7 Related Work

Several works have considered the problem of checking non-null types. Fähndrich and Leino investigated the constructor problem (see §5.1) and outlined a solution using raw types [20]. 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` [33, 32]. Their approach does not employ type aliasing and provides no guarantee that all potential errors will be reported. While

```
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;
}
```

Figure 8: 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);
  ...
```

Figure 9: 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.

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 [23]. 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 JustAdd compiler [17]. 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 [43]. They present empirical data suggesting many internal null checks can be eliminated, and that this leads to a useful improvement in program performance.

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 [9]. To do this, they manually annotated existing code bases, and checked for correctness by testing and with ESC/Java.

Recent work has considered types which support arbitrary *type qualifiers* [24, 26, 11, 4, 12]. These so-called "pluggable type systems" [7] 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 [24, 26]. Building on this is the work of Chin *et al.* which also supports numerous qualifiers, including `nonzero`, `unique` and `nonnull` [11, 12]. 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 [13, 8]. 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 good 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. [39, 38, 34, 3, 42, 16, 47]). These, almost exclusively, assume the original program is completely untyped and employ set constraints (see [2, 30]) 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 *inter-procedural 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 [39] Such systems share a strong relationship with other constraint-based program analyses (e.g. [30, 19, 1, 46]), such as *alias* or *points-to* analysis (e.g. [25, 44, 5, 40, 36, 41]).

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 [14, 15], 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 endeavour would be those extended SSA forms which provide some support for conditional statements [27, 29].

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 [45]; 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 [10]; 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 [48].

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://www.mcs.vuw.ac.nz/~djp/JACK/`.

Acknowledgements.

Thanks to Lindsay Groves, James Noble, Paul H.J. Kelly, Stephen Nelson, and Neil Leslie for many excellent comments on earlier drafts. This work is supported by the University Research Fund of Victoria University of Wellington.

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 *Proc. 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 *Proc. PLDI*, pages 196–207. ACM Press, 2003.

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

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

[8] 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.

[9] P. Chalin and P. R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proc. ECOOP*, pages 227–247. Springer, 2007.

[10] 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.

[11] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *Proc. PLDI*, pages 85–95. ACM Press, 2005.

[12] B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *Proc. ESOP*, 2006.

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

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. K. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proc. POPL*, pages 25–35. ACM Press, 1989.

[15] 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.

[16] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proc. OOPSLA*, pages 169–184. ACM Press, 1995.

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

[18] M. Ernst. Annotations on Java types, Java Specification Request (JSR) 308, 2007.

[19] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proc. PLDI*, pages 85–96. ACM Press, 1998.

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

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

[22] C. Fecht and H. Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *Proc. ESOP*, pages 90–104. Springer, 1998.

[23] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245. ACM Press, 2002.

[24] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proc. PLDI*, pages 192–203. ACM Press, 1999.

[25] 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.

[26] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proc. PLDI*, pages 1–12. ACM Press, 2002.

[27] 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.

[28] A. Goldberg. A Specification of Java Loading and Bytecode Verification. In *Conference on Computer & Communications Security*, pages 49–58. ACM Press, 1998.

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

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

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

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

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

[34] 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.

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

[36] O. Lhoták and L. J. Hendren. Context-sensitive points-to analysis: Is it worth it? In *Proc. Compiler Construction*, pages 47–64. Springer, 2006.

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

[38] N. Oxhøj, J. Palsberg, and M. I. Schwartzbach. Making type inference practical. In *Proc. ECOOP*, pages 329–349. Springer, 1992.

[39] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proc. OOPSLA*, pages 146–161. ACM Press, 1991.

[40] 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.

[41] 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.

[42] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proc. OOPSLA*, pages 324–340. ACM Press, 1994.

[43] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proc. Compiler Construction*, pages 334–554, 2001.

[44] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Proc. OOPSLA*, pages 43–55. ACM Press, 2001.

[45] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proc. ESOP*, pages 366–381. Springer-Verlag, 2000.

[46] B. D. Sutter, F. Tip, and J. Dolby. Customization of Java library classes using type constraints and profile information. In *Proc. ECOOP*, pages 585–610. Springer, 2004.

[47] T. Wang and S. F. Smith. Precise constraint-based type inference for Java. In *Proc. ECOOP*, pages 99–117. Springer, 2001.

[48] Y. Zhang and F. Nielson. A scalable inclusion constraint solver using unification. In *Proc. LOPSTR*, page (to appear), 2007.