Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Bachelor's Thesis

Provably Sound Nullness Analysis of Java Code

Author
Wouter Raateland

Supervisors
Sven Keidel
Dr. Sebastian Erdweg

Presented on July 3, 2018

Provably Sound Nullness Analysis of Java Code

Wouter Raateland, TU Delft

Abstract—**Null pointer dereferences in Java raise exceptions, occur often, are hard to debug and cost a lot of unnecessary effort and resources. Therefore, a lot of effort has been put in analyses spotting those null pointer dereferences. As developers rely on those analyses it is important that they are sound. However, proving null pointer analyses sound has been a complex problem. Hence, we developed a nullness analysis for Java that is provably sound. The analysis is built as an abstract interpreter upon the sturdy framework, that allows for compositional soundness proofs. This reduces the proof effort greatly. By creating a concrete interpreter, and by generating arbitrary programs, we were able to assess the soundness of the nullness analysis to great extend. Also, we have proven a part of the analysis to be sound in a formal way.**

Index Terms—**Static analysis, Abstract Interpretation, Nullness Analysis, Java, Arrows**

I. Introduction

JAVA allows one to store `null` values into both fields and variables. Dereferences of `null` pointers result in exceptions or segmentation faults.

Null pointer dereferences occur often and are notoriously hard to debug. According to Tony Hoare, the inventor of `null` pointers, the errors, vulnerabilities and system crashes caused by dereferencing `null` values have cost around a billion dollars. Hence, spotting `null` dereferences before they are executed is important. Java checks for nullness during run-time. By checking for nullness statically, Java programs have to make less checks during run-time and they would run more efficient. As developers rely on analyses spotting `null` dereferences, it is also important that the nullness analyses are sound. That is to say they don't miss any possible `null` dereferences.

Multiple static nullness analyses have already been developed. Some of these analyses [1] [2] [3] rely on abstract interpretation [4]. Of these analyses, only Loginov et al. [2] and Hubert et al. [1] have stated their analysis to be sound. Other existing analyses are based on other techniques: Some check nullness annotations and infer further nullness information from there [5]. Others infer nullness information from just code. Some analyses are not intended to be sound nor complete, they just attempt to find common `null` dereference errors and aim to be easy to use [6] [7]. Other analyses are complex and aim at high precision [8].

This work differs from this previous body of work by creating a static nullness analysis, based on abstract interpretation, that is provably sound and that requires significantly less effort to be proven sound than existing work. It describes a provably sound nullness analysis of Jimple [9], a preprocessed variant of Java, using the Sturdy framework[1]. Sturdy is a Haskell framework, providing utilities for creating abstract interpreters based upon arrows [10]. It enables compositional soundness proofs when different interpreters share parts of their semantics, reducing the effort required for these proofs.

This work is implemented using two interpreters and a shared interpreter interface: a concrete interpreter implementing concrete Jimple semantics, which we will call the concrete interpreter, an abstract interpreter implementing nullness semantics for Jimple, which we will call the abstract interpreter and a shared interpreter interface which describes the shared semantics of the concrete and abstract interpreter. First, the concrete interpreter is validated by running an extensive test suite. Then, the soundness of the abstract interpreter is assessed using the concrete interpreter together with QuickCheck[2] tests over each interface member. Finally, a partial soundness proof is made for the abstract interpreter. This illustrates the reduced complexity of the soundness proof over other proofs.

In this paper, we make three main contributions. We formalize the semantics of Jimple into a concrete interpreter and a shared interpreter interface that can be used to implement both an abstract nullness analysis and other analyses. We evaluate the correctness of the implemented concrete interpreter. We assess the soundness of our nullness analysis.

The rest of this paper is organized as follows. Section II gives information on the general structure and properties of the created interpreters. Section III describes important parts of the implementation of the interpreters. Section IV evaluates the correctness of the implementation and the soundness of the abstract interpreter. Section V describes other work done in this area. Section VI concludes the main findings. Section VII states potential directions for future work.

II. Background

Jimple programs are structured very similar to Java programs: Programs consist of multiple classes and interfaces, which may contain fields and methods. Also, class extension and interface implementation works just like in Java. The main difference between Jimple and Java programs is the way that method bodies are represented. This section describes the general structure and some important properties of our developed interpreters.

A. *Jimple Structure*

Our developed interpreters interpret desugared Jimple code. This code is generated in two steps: first, Java code is transformed into Jimple code by using the Jimple transformer from SOOT. Next, this Jimple code is manually desugared into an abstract syntax that is directly interpretable.

[1] https://github.com/svenkeidel/sturdy

[2] https://github.com/nick8325/quickcheck

We found that Jimple programs can be split in five different levels, which are all interpreted differently. These levels include: immediate values `Immediate`, boolean expressions (`BoolExpr`), normal expressions (`Expr`), statements (`[Statement]`) and complete programs (`[CompilationUnit]`).

Please note that expressions and boolean epxressions are two different constructs in Jimple. Boolean typed variables in Jimple are stored and manipulated as integers with values 1 and 0 for `True` respectively `False`. Boolean expressions only occur as conditions for `If` statements. Boolean expressions compare two immediate values with a comparison operator, such as the equality operator `Cmpq`, the non-equality operator `Cmpneq` and the ordering operators `Cmplt`, `Cmple`, `Cmpgt` and `Cmpge`.

B. Values

The concrete and abstract interpreter operate on the concrete value domain `Val` respectively the abstract value domain $\widehat{\texttt{Val}}$.

The concrete domain contains the following values:

```
data Val
  = IntVal Int
  | LongVal Int
  | FloatVal Float
  | DoubleVal Float
  | StringVal String
  | ClassVal String
  | NullVal
  | RefVal Addr
  | ArrayVal [Val]
  | ObjectVal String
      (Map FieldSignature Val)
```

Note here the value `RefVal Addr`. Java and Jimple implement objects and arrays as pointers to the actual record respectively array elements. This value mimmics this behaviour.

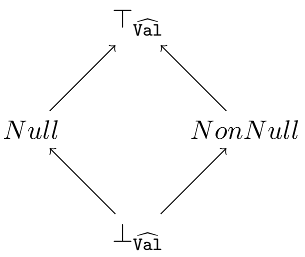
$\top_{\widehat{\texttt{Val}}}$
$Null$
$NonNull$
$\bot_{\widehat{\texttt{Val}}}$

Fig. 1: Complete lattice of nullness values $\widehat{\texttt{Val}}$, with top element $\top_{\widehat{\texttt{Val}}}$ containing all concrete values, $Null = \{\texttt{NullVal}\}$, $NonNull = \top_{\widehat{\texttt{Val}}} \setminus Null$ and $\bot_{\widehat{\texttt{Val}}} = \emptyset$.

The abstract interpreter uses the abstract domain for nullness defined by Cousot P. and Cousot R. [4], shown in figure 1.

The relation between the concrete domain and this abstract domain is easily described by the abstraction function $\alpha_{\texttt{Val}}$ of the Galois Connection for concrete values `Val` and abstract nullness values $\widehat{\texttt{Val}}$:

$$\alpha_{\texttt{Val}} : \mathcal{P}(\texttt{Val}) \leftrightarrows \widehat{\texttt{Val}} : \gamma$$

$$\alpha_{\texttt{Val}}(X) = \begin{cases} \top & \text{if } X \supset \{\texttt{NullVal}\} \\ Null & \text{if } X = \{\texttt{NullVal}\} \\ NonNull & \text{otherwise} \end{cases} \qquad (1)$$

C. Arrow Transformers

The concrete and abstract interpreter are implemented using the arrow transformer stack visible in figure 2a respectively figure 2b. The arrow transformers are defined in the sturdy framework and follow the definition from Hughes [10] where possible. To get a good understanding of the implementation of the interpreters, it is important to first understand the functionality of each arrow transformer in the stack. `Except` enables computations to fail with an exceptional value. In this implementation exceptional values can be either a `StaticException` or a `DynamicException`. `Reader` keeps track of the current method being interpreted. `Environment` enables scoped execution by adding an environment to a computation. The concrete interpreter keeps an environment mapping from variable names to addresses. The abstract interpreter keeps a mapping from variable names directly to values. `Store` simulates memory by keeping a mapping from addresses to values. `State` keeps track of latest memory address used. This makes sure that fresh addresses are allocated for each new variable declaration so that variables don't collide. `Const` keeps track of both the compilation units required to run the current program and the memory addresses containing the values of the static fields of these compilation units.

D. Context

It is visible from the arrow stack that the interpreter works in a context. For a local variable with name l, we denote the concrete context mapping l to concrete value v by $l \rightarrow_E v$ and the abstract context mapping l to abstract value $\hat{v}$ by $l \rightarrow_{\hat{E}} \hat{v}$.

III. Implementation

Now that the general principles of the interpreters are clear, it is time to move on to more specific implementation details. Hence, this section will illustrate the implementation of some crucial language constructs in our developed interpreters.

A. Declarations

Method bodies in Jimple consist of three separate parts: *declarations*, *statements* and *catch clauses*. The declarations contain all variables used in this method. When in interpreting a method body, first, its declarations are interpreted. A declaration consists of a type and a list of variable names. For each variable name in each declaration, first fresh memory is allocated using the `State` arrow transformer. Then the default value for the type of this declaration is assigned to this memory address using the `Store` arrow transformer. Finally the variable is made available for further use by inserting it into the environment. In the concrete interpreter, the default

```
newtype Interp x y = Interp
  (Except (Exception Val
    (Reader Context
      (Environment String Addr
        (StoreArrow Addr Val
          (State Addr
            (Const ([CompilationUnit],Fields)
              (->)))))) x y)
```

(a) Concrete stack

```
newtype Interp̂ x y = Interp̂
  (Except (Exception Val̂)
    (Reader Context
      (Environment String Val̂
        (StoreArrow FieldSignature Val̂
          (Const [CompilationUnit]
            (->))))) x y)
```

(b) Abstract stack

Fig. 2: Arrow transformer stacks used in concrete and abstract interpreters.

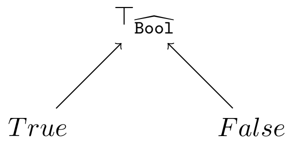

Fig. 3: Complete lattice of abstract booleans $\widehat{\texttt{Bool}}$ with top element $\top_{\widehat{\texttt{Bool}}} = \{\texttt{True}, \texttt{False}\}$, $True = \{\texttt{True}\}$ and $False = \{\texttt{False}\}$

.

value is `NullVal` for object and `null` typed variables. For primitive typed variables, the default value depends on the exact type, (e.g. `0` for `IntType`, `false` for `BoolType`). In the abstract interpreter, the default value is $Null$ for object and `null` typed variables and $NotNull$ for primitive typed variables.

B. *Label & Goto Statements*

Jimple contains `Label` and `Goto` statements. Both statements contain a label identifier:

```
Label "label1"
Goto "label1"
```

The effect of these statements is quite obvious. They could be compared to labels and break statements in Java. However, since a `Goto` statement can be used to move the code execution pointer to any `Label` defined in the current method, they are more powerful. As we will see further in this section, the `Label` statement is used for more language constructs.

C. *Assignments*

The only way to modify variables in Jimple is through the `Assign`. This statement takes a variable, an expression and a continuation. In interpreting an `Assign` statement, first the expression is evaluated to a value, then this value is assigned to a variable, which can either be a local variable or a reference variable. Reference variables include array elements, object fields and static fields. In the concrete interpreter a map from variable name to its address in memory is kept by the `Environment` arrow transformer and the `StoreArrow` arrow transformer maps these addresses to actual values. This structure enables many features, including nested objects and circular data structures. This structure also allows the concrete interpreter to assign a new value to a variable of each kind by only updating the `Store`. The abstract interpreter works on the more simple nullness domain (fig. 1). In this domain, all values are defined non-recursively. Therefor a mapping from variable names to addresses and from addresses to values is not required and all operations on variables can be executed using only the `Environment` arrow transformer. Assigning a new value to a variable is now simulated by running the continuation in a new environment with the updated value for the specified variable.

D. *If Statements*

`If` statements in Jimple have the same functionality as in `if` statements in Java. However their syntax is very different. As an example, consider the following Java code:

```
if (x < 2) {
  return x;
} else {
  return 3;
}
```

This is transformed into the following Jimple statements:

```
If (BinopExpr (Local "x") Cmplt
 ↪  (IntConstant 2)) "label1"
Return (Just (IntConstant 3))
Label "label1"
Return (Just (Local "x"))
```

The condition $x < 2$ is still there. However, the *if-branch* is replaced with the label ”label1” and the *else-branch* is put directly after the `if` statement. This is interpreted as follows: If the condition evaluates to a true value, then Java would continue in the *if-branch* and Jimple interpretation is continued at statement `Label "label1"`. Otherwise, Java code executes the *else-branch* and Jimple interpretation is continued at the next statement (`Return (Just (IntConstant 3))`).

The shared interpreter implements this behaviour using two continuations as follows:

```
If condition label -> do
  b <- evalBool -< condition
  if_
    (atLabel runStatements)
    runStatements -<
      ((b,condition),(label,stmts))
```

The implementation of `if_` in both the concrete and abstract interpreter is shown in figure 4. The `if_` operation takes two continuations `f` (=`atLabel runStatements`) and `g` (=`runStatements`) and requires both the condition and the value this condition is evaluated to as input. The concrete interpreter evaluates the condition to an ordinary boolean value. If b = `True`, then `f` is executed. If b = `False`, `g` is interpreted. The abstract interpreter evaluates the condition

```
if_ f g = proc ((b,_),(x,y)) ->
  case b of
    True -> f -< x
    False -> g -< y
```

(a) Concrete interpreter

```
if̂_ f̂ ĝ = proc ((b̂,BoolExpr i1 op i2),(x,y)) -> case b̂ of
  True -> f̂ -< x
  False -> ĝ -< y
  Top -> case (i1,op,i2) of
    (Local l,Cmpeq,NullConstant) -> specify -< ((l,Null,x),(l,NonNull,y))
    (NullConstant,Cmpeq,Local l) -> specify -< ((l,Null,x),(l,NonNull,y))
    (Local l,Cmpne,NullConstant) -> specify -< ((l,NonNull,x),(l,Null,y))
    (NullConstant,Cmpne,Local l) -> specify -< ((l,NonNull,x),(l,Null,y))
    _ -> (f̂ -< x) ⊔ (ĝ -< y)
  where specify = joined (extendEnv' f̂) (extendEnv' ĝ)
```

(b) Abstract interpreter

Fig. 4: Concrete and abstract implementation of the `if_` operation

to an abstract boolean value (fig. 3), which can be both `True` and `False` at the same time: $\top_{\widehat{\texttt{Bool}}}$. In most cases abstract interpretation of `if_` is the same as concrete interpretation: If $b = True$ or $b = False$, then the first, respectively the second continuation is interpreted. If $b = \top_{\widehat{\texttt{Bool}}}$, then `f` is interpreted assuming that condition c evaluated to $True$ and `g` is interpreted assuming that condition c evaluated to $False$. After interpreting both `f` and `g`, the results are combined using the least upper bound operation $\sqcup$.

By making the stated assumptions, the abstract interpreter can get more precise information on the program currently being interpreted in two specific cases. Namely, when a local variable with name l is compared to a `null` literal using either the equality operator `Cmpeq` ($\equiv$) or the non-equality operator `Cmpneq` ($\neq$). In these cases $b = \top_{\widehat{\texttt{Bool}}}$ and therefor $l \to_{\widehat{E}} \top_{\widehat{\texttt{Val}}}$. Now, the original condition `BoolExpr i1 op i2` is used. If `op` = `Cmpeq`, `f` and `g` are interpreted under the more precise assumptions that $l \to_{\widehat{E}} Null$ respectively $l \to_{\widehat{E}} NonNull$. If `op` = `Cmpeq`, `f` and `g` are interpreted under the more precise assumptions that $l \to_{\widehat{E}} Null$ respectively $l \to_{\widehat{E}} NonNull$.

E. Switch statements

Jimple contains two distinct switch statements, namely `LookupSwitch` and `TableSwitch`. Except for their name, both statements have identical syntax:

```
TableSwitch Immediate
  ↪  [(Case,LabelIdentifier)]
```

Here a `Case` can be an integer or a `default` case. The only difference between these statements is that they are normally interpreted using different algorithms for speed reasons. As speed was not a concern in this work, we implemented both equally. For interpreting a switch statement, the immediate value `Immediate` always has to be evaluated first. If the value resulting from this evaluation matches any integer valued case, interpretation is continued at the corresponding label. Else, interpretation is continued at the label corresponding to the `default` case. This is implemented in the shared interpreter as follows:

```
TableSwitch i cases -> do
  v <- evalImmediate -< i
  case_ (atLabel runStatements) -<
    ↪  (v,cases)
```

The shared interpreter evaluates `Immediate` and passes its result togeter with a continuation to the `case_` method, which is implemented by the specific interpreter. This continuation `atLabel runStatements` takes a label name as input, searches for the corresponding `Label` statement and continues interpretation from there. The concrete interpreter simply searches for the first matching case and runs the continuation using the matching label. The abstract interpreter does not distinguish between different integer values. Hence it is never certainly matches a case. Therefor it computes the continuation for all cases and combines the results using the least upper bound operation $\sqcup$.

F. Try Catch

Try Catch in Jimple is implemented by combining a method's catch clauses and `Label` statements. Take for example the following Java code where an exception is thrown and caught:

```
try {
  throw new Exception("foo");
} catch (Exception e) {
  return 1;
}
```

This is transformed into the following Jimple statements:

```
Label "label1"
Assign (LocalVar "$r2") (NewExpr (RefType
  ↪  "java.lang.ExceptionA"))
Invoke (SpecialInvoke "$r2" sig
  ↪  [StringConstant "foo"])
Throw (Local "$r2")
Label "label2"
IdentityNoType "$r3" CaughtExceptionRef
Return (Just (IntConstant 1))
```

and the following Jimple catch clause:

```
CatchClause {
  className = "java.lang.Exception",
  fromLabel = "label1",
  toLabel   = "label2",
  withLabel = "label2"
}
```

Our implementation supports two types of exceptions: `StaticException`'s and `DynamicException`'s. `StaticException`'s contain an error message and are thrown when the program contains an error, such as invoking a non-existing method. These exceptions stop the interpretation immediately. `DynamicException`'s contain a value, may be thrown either explicitly by a `Throw` statement, or implicitly by invalid operations such as a `null` dereference and can be caught using `CatchClause`'s. A `CatchClause` contains a *fromLabel*, *toLabel*, *withLabel* and a *className*. It catches a `DynamicException` when its value is an instance of *className* and when it is thrown in a statement between the `Label` statements corresponding with its *fromLabel* and *toLabel*. When it catches the exception, the exceptional value is assigned to a special variable named *@caughtexception* and interpretation is continued at the `Label` statement corresponding with the exception's *withLabel*.

The shared interpreter implements this behaviour using the `Reader` and `Except` arrow transformer as seen in figure 2 and the arrow operation `tryCatch` as follows:

```
Label labelName -> do
  (_,clauses) <- ask -< ()
  let clauses' = filter (\c ->
    fromLabel c == labelName &&
    Label (toLabel c) `elem` stmts)
     ↪ clauses
  tryCatch
    (pi1 >>> runStatements)
    catchException -< (rest,clauses')
```

This code uses the `Reader` arrow transformer to get the body of the current method, which may contain `CatchClause`'s. Then it filters the clauses and runs the `tryCatch` arrow operation. This operation tries to execute its first argument. If that throws an exception, then it catches that using its second argument, `catchException` which is implemented as follows:

```
catchException = proc ((_,clauses),e) ->
 ↪ case e of
  StaticException _ -> failA -< e
  DynamicException v -> catch
   ↪ handleException -< (v,clauses)

handleException = proc (v,clause) -> do
  addr <- alloc -< v
  extendEnv' runStatementsFromLabel -<
   ↪ ("@caughtexception", addr, withLabel
   ↪ clause)
```

This code passes clauses that may catch the exception `e` to `catch`, which is implemented in the concrete and abstract interpreter. `catch` also takes a continuation, `handleException`, which uses the `Store` and `Environment` arrows to allocate memory for and assign a value to the special variable *@caughtexception*. The concrete interpreter implements `catch` by finding the first clause such that the exceptional value `e` is an instance of the *className* of this clause. If no such clause can be found, then the exception is passed down the stack. The abstract interpreter does not discern between different object types. Hence it implements `catch` by both interpreting the given continuation for each given clause and by passing the exception down the stack and then joining the results using the least upper bound operation.

IV. Evaluation

In this research we evaluate whether it is feasible to implement a shared arrow-based interpreter for Java, we evaluate the correctness of the implemented concrete interpreter and we evaluate the feasibility of proving the implemented nullness analysis sound. The previous section already shows the feasibility of implementing a shared arrow-based interpreter for Java. Therefore, this section will evaluate the correctness of the concrete interpreter by describing test effort taken and this section will evaluate the feasibility of proving the implemented nullness analysis sound by assessing its soundness and by proving a part of its soundness.

The two interpreters that we created in this work are implemented on top of the non-leaky shared interpreter interface, which is shown in appendix A. It contains a total of 50 operations split over 5 classes: 38 value operations (class `UseVal`), 3 exception operations (class `UseException`), 7 operations using values and booleans (class `UseBool`), 1 environment operation (class `UseEnv`) and 1 constant operation (class `Useconst`). Together these operations describe the complete semantics of Jimple. By making use of the compositional soundness proofs that the Sturdy framework provides, proving the abstract interpreter sound, would require proving 50 soundness lemmas, one for each interface method. This was outside the scope of this research. However, the soundness of the abstract interpreter is assessed using QuickCheck tests. Also, to illustrate the reduced effort required to prove the interpreter sound, a soundness proof is given for `if_` which is used to interpret switch statements. In both assessing soundness of the abstract interpreter, and proving soundness of switch statements, a soundness proposition derived from the soundness proposition for the collecting semantics [11] of `evalImmediate`, `evalBool`, `eval` and of `runStatements` is used:

$$f \overset{\circ}{\sqsubseteq}_{\texttt{Interp}} \hat{f}$$
$$\iff \forall a \in \mathcal{P}(A), \forall \hat{a} \in \hat{A}, \alpha_A(a) \sqsubseteq \hat{a} \implies \alpha_{\hat{A}}(f(a)) \sqsubseteq \hat{f}(\hat{a})$$

Herein $f \overset{\circ}{\sqsubseteq}_{\texttt{Interp}} \hat{f}$ means that interpretation of $\hat{f}$ is sound with respect to f and $\sqsubseteq$ denotes an reflexive and transitive ordering relation.

A. Assessing correctness of the concrete interpreter

Without the concrete interpreter implementing Jimple correctly, assessing and proving soundness of an abstract interpreter that shares code with it, makes no sense. Hence, we will describe how we assessed the correctness of the concrete interpreter here.

To assess the correctness of the concrete interpreter, we made several efforts. First, for translating the syntax of Jimple into an abstract syntax tree that would be parseable in Haskell, the original Jimple documentation [9] was used in combination with the source code of Jimple[3]. As no official documentation was available, describing the semantics of Jimple code, the concrete semantics were derived from the source code. Then,

[3] https://github.com/Sable/soot/tree/develop/src/main/java/soot/jimple

we wrote small unit tests, testing the interpretation of immediate values, boolean expressions and expressions. Gradually larger tests were added, interpreting multiple statements or even complete programs consisting of multiple classes. In the end, we had a test suite containing 30 tests: 24 simple tests, testing interpretation up to statements and 6 tests which tested interpretation of complete Java programs. The test suite targeted most language constructs of Jimple, including program initialization, variable shadowing, operating on arrays and objects, throwing and catching exceptions, branching with if and case statements, handling arithmetic operations and using static fields. Because some rules for typecasting, like narrowing rules for primitives[4], have not been implemented, the test suite does not cover typecasting of variables.

B. Assessing Soundness of the Nullness Analysis

For each level of interpretation: immediate values, boolean expressions, general expressions, statements and complete programs, we designed QuickCheck tests. In this research, we only used these tests to assess the soundness of the abstract interpreter for nullness semantics. However, the tests were designed to test the soundness of any abstract interpreter for Jimple.

The QuickCheck tests made heavy use of the typeclass Arbitrary[5]. This allowed us to automatically generate random immediate values, identifiers, variable names and even random expressions and statements. Testing the interpreters in this way allowed for a high coverage without having to write many tests by hand.

To get a feeling of how such a QuickCheck test works, we show how the soundness of arbitrary unary operations is assessed. Please note that in order to be more readable in this paper format, the code shown here differs from the original code. Unary operations in Jimple are expressions with the format:

```
UnopExpr op immediate
```

To assess them, the method for assessing expression evaluation is called:

```
soundExpr :: (Arbitrary a,Show a,Arbitrary
 ↪ b,Show b,Galois (Pow vc) va,Galois
 ↪ (Pow rc) ra,Complete ra,Eq rc,Hashable
 ↪ rc,Show rc,Show ra) =>
  String ->
  (a -> [(String,vc)]) -> (b -> Expr) ->
  ([(String,vc)] -> Expr -> rc) ->
  ([(String,va)] -> Expr -> ra) ->
  Spec
soundExpr desc genMem genExpr runCon
 ↪ runAbs =
  it ("sound value approximation " ++
   ↪ desc) $ property $ \(a,b) -> do
    let mem = genMem a
    let mema = map $ second (alpha .
     ↪ singleton)
    let expr = genExpr b
    let rc = runCon mem expr
    let ra = runAbs mema expr
    rc `shouldBeApproximated` ra
    where c `shouldBeApproximated` a =
     ↪ unless
      (ca ⊑ alpha (singleton c))
      (expectationFailure "soundness
       ↪ failed")
```

This test method takes as input a memory generation function and an expression generation function. First it generates memory and an expression. Then, it runs the concrete and abstract interpreters on this generated memory and expression. Now, the value resulting from concrete interpretation is abstracted using the abstraction function (1). Finally, it checks whether the value resulting from abstract interpretation over approximates the abstracted concrete value.

The test is called in the following way:

```
soundExpr "Unary operations"
  (const []::() -> [(String,vc)])
  (uncurry UnopExpr)
  Con.eval Abs.eval
```

Testing in this way has allowed us to cover 92% of all top level definitions and 78% of all expressions in the concrete interpreter, abstract interpreter and shared interpreter interface combined. These results give us a strong indication that the abstract interpreter is actually sound.

C. Proving If Statements Sound

This section will provide a proof of the `if_` operation described in section III-D. We chose the `if_` operation as a first proof and as an example, because proving its soundness is neither trivial, nor to complex.

Lemma 1 (Soundness of `if_`.). *If* $\mathtt{f} \overset{\circ}{\sqsubseteq} \widehat{\mathtt{f}}$ *and* $\mathtt{g} \overset{\circ}{\sqsubseteq} \widehat{\mathtt{g}}$*, then* $\mathtt{if\_\ f\ g} \overset{\circ}{\sqsubseteq} \widehat{\mathtt{if\_}}\ \widehat{\mathtt{f}}\ \widehat{\mathtt{g}}$.

Proof. Let $\mathtt{f} \overset{\circ}{\sqsubseteq} \widehat{\mathtt{f}}$ and $\mathtt{g} \overset{\circ}{\sqsubseteq} \widehat{\mathtt{g}}$ and let $X \in \mathcal{P}(\mathtt{Bool})$ and $\hat{b} \in \widehat{\mathtt{Bool}}$ such that $\alpha_{\mathtt{Bool}}(X) \sqsubseteq \hat{b}$. Also let e be a boolean expression `BoolExpr` $i1$ *op* $i2$, where *op* is either an equality operation (`Cmpeq` or `Cmpneq`) or an ordering operation and $i1$ and $i2$ are two immediate values. Finally, let x and y be the two inputs to $\mathtt{f}$,$\widehat{\mathtt{f}}$ and $\mathtt{g}$,$\widehat{\mathtt{g}}$ respectively. Note that $\mathtt{f}$ and $\widehat{\mathtt{f}}$, and $\mathtt{g}$ and $\widehat{\mathtt{g}}$ take inputs of the same type. We distinct three separate cases:

"$\hat{b} = True$" Now $X \subseteq \{\mathtt{True}\}$, and thus

$$\begin{aligned} &\{\mathtt{if\_\ f\ g} \prec ((b, e), (x, y)) | b \in X\} \\ \subseteq\ &\{\mathtt{if\_\ f\ g} \prec ((\mathtt{True}, e), (x, y))\} = \{\mathtt{f} \prec x\} \end{aligned}$$

Hence,

$$\begin{aligned} &\alpha_{\mathtt{Bool}}(\{\mathtt{if\_\ f\ g} \prec ((b, e), (x, y)) | b \in X\}) \\ \sqsubseteq\ &\alpha_{\mathtt{Bool}}(\{\widehat{\mathtt{f}} \prec x\}) \sqsubseteq\ \widehat{\mathtt{f}} \prec x \\ =\ &\widehat{\mathtt{if\_}}\ \widehat{\mathtt{f}}\ \widehat{\mathtt{g}} \prec ((True, e), (x, y)) \end{aligned}$$

[4] https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html#jls-5.1.2

[5] http://hackage.haskell.org/package/QuickCheck-2.11.3/docs/Test-QuickCheck-Arbitrary.html

"$\hat{b} = False$" This case is similar to the first. Now $X \subseteq \{\texttt{False}\}$, and thus

$$\begin{aligned} &\{\texttt{if\_ f g} \mathrel{-\!\!<} ((b,e),(x,y)) | b \in X\} \\ \subseteq\ &\{\texttt{if\_ f g} \mathrel{-\!\!<} ((\texttt{False},e),(x,y))\} = \{\texttt{g} \mathrel{-\!\!<} y\} \end{aligned}$$

Hence,

$$\begin{aligned} &\alpha_{\texttt{Bool}}(\{\texttt{if\_ f g} \mathrel{-\!\!<} ((b,e),(x,y)) | b \in X\}) \\ \sqsubseteq\ &\alpha_{\texttt{Bool}}(\{\texttt{g} \mathrel{-\!\!<} y\}) \sqsubseteq\ \widehat{\texttt{g}} \mathrel{-\!\!<} y \\ =\ &\widehat{\texttt{if\_}}\ \widehat{\texttt{f}}\ \widehat{\texttt{g}} \mathrel{-\!\!<} ((False,e),(x,y)) \end{aligned}$$

"$\hat{b} = \top_{\widehat{\texttt{Bool}}}$" In this case $X \subseteq \{\texttt{True}, \texttt{False}\}$. To handle the more complex structure of the abstract interpreter we split this case into two subcases:

1). If then one of $i1$ and $i2$ is a local variable with name l, the other is a `null` constant and $op \in \{\texttt{Cmpeq}, \texttt{Cmpneq}\}$, $\widehat{\texttt{if\_}}$ will modify the interpretation context with `extendEnv'`. We can now reason as follows:

First consider $op = \texttt{Cmpeq}$. If now $b = \texttt{True}$, $l \rightarrow_E \texttt{NullVal}$. Otherwise, $l \rightarrow_E a$ for some $a \in Val, a \neq \texttt{NullVal}$ and

$$\begin{aligned} &\{\texttt{if\_ f g} \mathrel{-\!\!<} ((b,e),(x,y)) | b \in X\} \\ \subseteq\ &\{\texttt{if\_ f g} \mathrel{-\!\!<} ((b,e),(x,y)) | b \in \{\texttt{True}, \texttt{False}\}\} \\ =\ &\{\texttt{if\_ f g} \mathrel{-\!\!<} ((True,e),(x,y)), \\ &\ \texttt{if\_ f g} \mathrel{-\!\!<} ((False,e),(x,y))\} \\ =\ &\{\texttt{f} \mathrel{-\!\!<} x|_{l \rightarrow_E \texttt{NullVal}}, \texttt{g} \mathrel{-\!\!<} y|_{l \rightarrow_E a}\} \end{aligned}$$

Now

$$\begin{aligned} &\alpha_{\texttt{Val}}(\{\texttt{if\_ f g} \mathrel{-\!\!<} ((b,e),(x,y)) | b \in X\}) \\ \sqsubseteq\ &\alpha_{\texttt{Val}}(\{\texttt{f} \mathrel{-\!\!<} x|_{l \rightarrow_E \texttt{NullVal}}, \texttt{g} \mathrel{-\!\!<} y|_{l \rightarrow_E a}\}) \\ \sqsubseteq\ &\alpha_{\texttt{Val}}(\{\texttt{f} \mathrel{-\!\!<} x|_{l \rightarrow_E \texttt{NullVal}}\}) \sqcup \\ &\alpha_{\texttt{Val}}(\{\texttt{g} \mathrel{-\!\!<} y|_{l \rightarrow_E a}\}) \\ \sqsubseteq\ &\widehat{\texttt{f}} \mathrel{-\!\!<} x|_{l \rightarrow_{\widehat{E}} Null}\ \sqcup\ \widehat{\texttt{g}} \mathrel{-\!\!<} y|_{l \rightarrow_{\widehat{E}} NonNull} \\ =\ &(\texttt{extendEnv}'\widehat{\texttt{f}} \mathrel{-\!\!<} (x,l,Null)|_{l \rightarrow_{\widehat{E}} \top_{\widehat{\texttt{Bool}}}}) \sqcup \\ &(\texttt{extendEnv}'\widehat{\texttt{g}} \mathrel{-\!\!<} (y,l,NonNull)|_{l \rightarrow_{\widehat{E}} \top_{\widehat{\texttt{Bool}}}}) \\ =\ &\widehat{\texttt{if\_}}\ \widehat{\texttt{f}}\ \widehat{\texttt{g}} \mathrel{-\!\!<} ((\hat{b},e),(x,y))|_{l \rightarrow_{\widehat{E}} \top_{\widehat{\texttt{Bool}}}} \end{aligned}$$

We can use a similar argument to prove that soundness also holds when $op = \texttt{Cmpne}$.

2). Otherwise,

$$\begin{aligned} &\{\texttt{if\_ f g} \mathrel{-\!\!<} ((b,e),(x,y)) | b \in X\} \\ \subseteq\ &\{\texttt{if\_ f g} \mathrel{-\!\!<} ((b,e),(x,y)) | b \in \{\texttt{True}, \texttt{False}\}\} \\ =\ &\{\texttt{f} \mathrel{-\!\!<} x, \texttt{g} \mathrel{-\!\!<} y\} \end{aligned}$$

And thus,

$$\begin{aligned} &\alpha_{\texttt{Val}}(\{\texttt{if\_ f g} \mathrel{-\!\!<} ((b,e),(x,y)) | b \in X\}) \\ \sqsubseteq\ &\alpha_{\texttt{Val}}(\{\texttt{f} \mathrel{-\!\!<} x, \texttt{g} \mathrel{-\!\!<} y\}) \\ \sqsubseteq\ &\alpha_{\texttt{Val}}(\{\texttt{f} \mathrel{-\!\!<} x\}) \sqcup \alpha_{\texttt{Val}}(\{\texttt{g} \mathrel{-\!\!<} y\}) \\ \sqsubseteq\ &\widehat{\texttt{f}} \mathrel{-\!\!<} x \sqcup \widehat{\texttt{g}} \mathrel{-\!\!<} y \\ =\ &\widehat{\texttt{if\_}}\ \widehat{\texttt{f}}\ \widehat{\texttt{g}} \mathrel{-\!\!<} ((\hat{b},e),(x,y)) \end{aligned}$$

Now, for all cases,

$$\begin{aligned} &\alpha_{\texttt{Val}}(\{\texttt{if\_ f g} \mathrel{-\!\!<} ((b,e),(x,y)) | b \in X\}) \\ \sqsubseteq\ &\widehat{\texttt{if\_}}\ \widehat{\texttt{f}}\ \widehat{\texttt{g}} \mathrel{-\!\!<} ((\hat{b},e),(x,y)) \end{aligned}$$

Hence the lemma follows. □

V. Related Work

As far as we know, only one attempt at concrete interpretation of Jimple code has been done before. This project by University of Colorado Boulder[6] is incomplete and there has been no progress since October 2017.

The creators of SOOT have stated plans to write a concrete interpreter for Jimple[7], however no progress has been made there yet.

On static nullness analysis, a lot of work has been done. We will list the most relevant work done to date and compare it to our work.

Most works are based upon techniques other than abstract interpretation. These works include the following:

The Checker framework[8] contains a nullness checker that guarantees to find all null dereferences [9] and that is sound.

FindBugs[10] contains a static null pointer analysis, which they show to be very precise [8]. This analysis is not proven sound.

Both Eclipse IDE[11] and Intellij IDE[12] contain a simple nullness analysis[13][14]. These analyses are available for free and serve as a first check. They are neither sound nor complete.

The SOOT[15] analysis suite, contains a static nullness analysis [12]. This analysis is tested using the test suit from Soot. However, nothing could be found on the soundness of this analysis.

OpenJML[16] is a static analysis tool using Java annotations, which includes a null pointer analysis. It is the successor of ESC/Java and ESC/Java2. It is neither sound nor complete. CANAPA[17] and Houdini are analyses built on top op ESC/Java2 respectively ESC/Java. These analyses aim to reduce the effort of eliminating null pointer exceptions [6] [7].

JastAdd[18] is an extensible Java compiler, which includes modules for static non-null checking [5].

Our nullness analysis is based upon abstract interpretation of complete programs and is focused upon soundness rather than precision. Hence, our analysis differs from the above works.

Then, there is a nullness analyses based upon abstract interpretation that is not sound. Spoto et al. [3] created a

[6] https://github.com/cuplv/cuanto/tree/develop/src/main/scala/edu/colorado/plv/cuanto

[7] https://conf.researchr.org/track/issta-2018/panathon-2018

[8] https://checkerframework.org/

[9] https://checkerframework.org/releases/1.1.1/checkers-manual.html#nullness-checker

[10] http://findbugs.sourceforge.net/

[11] https://www.eclipse.org/ide/

[12] https://www.jetbrains.com/idea/

[13] https://wiki.eclipse.org/JDT_Core/Null_Analysis

[14] https://www.jetbrains.com/help/idea/nullable-and-notnull-annotations.html

[15] https://www.sable.mcgill.ca/soot/

[16] http://www.openjml.org/

[17] http://www.mimuw.edu.pl/ chrzaszcz/Canapa/

[18] http://jastadd.org/web/

null pointer analysis for Java bytecode. This work is mainly focused on precision.

Finally, there are two nullness analyses based on abstract interpretation that are sound.

Nit, the Nullability Inference Tool[19], is a static analysis tool that infers nullness annotations for Java bytecode [1]. The analysis underlying the tool is proven sound for some part [13]. They state that a soundness proof for the analysis for the complete Java bytecode would be necessarily large and that they would require machine-checking to verify this proof. The soundness proof for our developed analysis would also be large, but it would be compositional and would not require machine-checking.

Loginov et al. [2] presents a tool named SALSA, which uses a novel technique named expanding-scope analysis to soundly analyze null dereferences with reasonable precision. They state that their analysis is sound given any sound call-graph construction and alias analysis. In this work, no formal proof is given. Hence we cannot compare their proof effort to ours.

VI. Conclusion

We presented a novel approach to creating a static nullness analysis, based upon abstract interpretation using a shared interpreter interface. We implemented an interpreter for concrete Jimple semantics, an interpreter for abstract Jimple nullness semantics and a non-leaky shared interpreter interface containing the common functionality of the two interpreters using arrow transformers, making the abstract interpreter provably sound. We evaluated the soundness of the abstract interpreter in three steps. First, we assessed the correctness of the concrete interpreter using an extensive test suite. Then, we assessed the soundness of the abstract interpreter running both the concrete and abstract interpreter on randomly generated interpretable structures, giving a good indication that the abstract interpreter is actually sound. Finally, we have shown a proof of the soundness of the `if_` operation of the shared interpreter interface.

VII. Future work

Currently, the abstract interpreter will not terminate when interpreting programs containing loops and recursion. In future work, a fixpoint cache should be added to the interpreter. This would allow analysis of more complex programs including these language features. Also, in the current work, we focused on soundness rather than precision. In future work, the precision of the interpreter can be improved, for example by using a more precise abstract domain using Raw values [1] or by incorporating variable aliasing. Also, in this work, only a small part of the abstract interpreter is yet proven sound, it makes sense to expand this proof and to prove the complete interpreter sound. Most of the code of this work is written agnostic of the exact abstract interpreter. In future work, it would be interesting to create more complex analyses on top of the already created shared interpreter interface and to prove them sound as well.

Acknowledgment

In writing this article and during the research leading up to it, there was some cooperation with the authors of a similar article on abstract interpretation of JavaScript using *sturdy*. The authors would like to thank Sven Keidel and Sebastian Erdweg for writing the sturdy framework and for their effort and guidance during the research.

References

[1] L. Hubert, "A non-null annotation inferencer for java bytecode," in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '08. New York, NY, USA: ACM, 2008, pp. 36–42. [Online]. Available: http://doi.acm.org/10.1145/1512475.1512484

[2] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. Nanda, "Verifying dereference safety via expanding-scope analysis," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 213–224. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390657

[3] F. Spoto, "Precise null-pointer analysis," *Software & Systems Modeling*, vol. 10, no. 2, pp. 219–252, May 2011. [Online]. Available: https://doi.org/10.1007/s10270-009-0132-5

[4] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: ACM, 1977, pp. 238–252. [Online]. Available: http://doi.acm.org/10.1145/512950.512973

[5] T. Ekman and G. Hedin, "The jastadd extensible java compiler," in *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 1–18. [Online]. Available: http://doi.acm.org/10.1145/1297027.1297029

[6] M. Cielecki, J. Fulara, K. Jakubczyk, and L. Jancewicz, "Propagation of jml non-null annotations in java programs," in *Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java*, ser. PPPJ '06. New York, NY, USA: ACM, 2006, pp. 135–140. [Online]. Available: http://doi.acm.org/10.1145/1168054.1168073

[7] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, ser. FME '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 500–517. [Online]. Available: http://dl.acm.org/citation.cfm?id=647540.730008

[8] D. Hovemeyer and W. Pugh, "Finding more null pointer bugs, but not too many," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '07. New York, NY, USA: ACM, 2007, pp. 9–14. [Online]. Available: http://doi.acm.org/10.1145/1251535.1251537

[9] R. Valle-Rai and L. J. Hendren, *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. McGill University, Sable Research Group, 1998. [Online]. Available: https://pdfs.semanticscholar.org/1a8b/a074cf7a378392d549154d8b448915fe6e99.pdf

[10] J. Hughes, "Generalising monads to arrows," *Sci. Comput. Program.*, vol. 37, no. 1-3, pp. 67–111, May 2000. [Online]. Available: http://dx.doi.org/10.1016/S0167-6423(99)00023-4

[11] P. Cousot, "The calculational design of a generic abstract interpreter," in *Calculational System Design*, M. Broy and R. Steinbrüggen, Eds. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[12] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge, "A framework for optimizing java using attributes," in *CASCON First Decade High Impact Papers*, ser. CASCON '10. Riverton, NJ, USA: IBM Corp., 2010, pp. 225–241. [Online]. Available: http://dx.doi.org/10.1145/1925805.1925819

[13] L. Hubert, T. Jensen, and D. Pichardie, "Semantic foundations and inference of non-null annotations," in *Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, ser. FMOODS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 132–149. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68863-1_9

[19] http://nit.gforge.inria.fr/index.html

Appendix A
Shared Interpreter Interface

```haskell
class Arrow c => UseVal v c | c -> v where
  doubleConstant :: c Float v
  floatConstant :: c Float v
  intConstant :: c Int v
  longConstant :: c Int v
  nullConstant :: c () v
  stringConstant :: c String v
  classConstant :: c String v
  newSimple :: c Type v
  newArray :: c (Type,[v]) v
  and :: c (v,v) v
  or :: c (v,v) v
  xor :: c (v,v) v
  rem :: c (v,v) v
  cmp :: c (v,v) v
  cmpg :: c (v,v) v
  cmpl :: c (v,v) v
  shl :: c (v,v) v
  shr :: c (v,v) v
  ushr :: c (v,v) v
  plus :: c (v,v) v
  minus :: c (v,v) v
  mult :: c (v,v) v
  div :: c (v,v) v
  lengthOf :: c v v
  neg :: c v v
  instanceOf :: c (v,Type) v
  cast :: c ((v,Type),v) v
  defaultValue :: c Type v
  declare :: c x (Maybe v) ->
    c ((String,v),x) (Maybe v)
  readVar :: c String v
  updateVar :: c x (Maybe v) ->
    c ((String,v),x) (Maybe v)
  readIndex :: c (v,v) v
  updateIndex :: c x (Maybe v) ->
    c (((v,v),v),x) (Maybe v)
  readField :: c (v,FieldSignature) v
  updateField :: c x (Maybe v) ->
    c ((v,(FieldSignature,v)),x) (Maybe v)
  readStaticField :: c FieldSignature v
  updateStaticField :: c (FieldSignature,v) ()
  case_ :: c String (Maybe v) ->
    c (v,[CaseStatement]) (Maybe v)

class Arrow c => UseException e v c | c -> e v where
  failStatic :: c String a
  failDynamic :: c v a
  catch :: c (v,CatchClause) (Maybe v) ->
    c (e v,[CatchClause]) (Maybe v)

class Arrow c => UseBool b v c | c -> b v where
  eq :: c (v,v) b
  neq :: c (v,v) b
  gt :: c (v,v) b
  ge :: c (v,v) b
  lt :: c (v,v) b
  le :: c (v,v) b
  if_ :: c String (Maybe v) -> c x (Maybe v) ->
    c ((b,BoolExpr),(String,x)) (Maybe v)

class Arrow c => UseEnv env c | c -> env where
  emptyEnv :: c () env

class Arrow c => UseConst c where
  askCompilationUnits :: c () [CompilationUnit]
```