A Non-Null Annotation Inferencer for Java Bytecode

Laurent Hubert

To cite this version:

Laurent Hubert. A Non-Null Annotation Inferencer for Java Bytecode. PASTE: Program analysis for software tools and engineering, Nov 2008, Atlanta, Georgia, United States. pp.10.1145/1512475.1512484, 10.1145/1512475.1512484. inria-00504006

HAL Id: inria-00504006
https://inria.hal.science/inria-00504006
Submitted on 19 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
ABSTRACT
We present a non-null annotations inferencer for the Java bytecode language. We previously proposed an analysis to infer non-null annotations and proved it soundness and completeness with respect to a state of the art type system. This paper proposes extensions to our former analysis in order to deal with the Java bytecode language. We have implemented both analyses and compared their behaviour on several benchmarks. The results show a substantial improvement in the precision and, despite being a whole-program analysis, production applications can be analyzed within minutes.

Categories and Subject Descriptors
F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—program analysis; D.3.3 [Programming Languages]: Language Constructs and Features—Data types and structures; D.1.5 [Programming Techniques]: Object-oriented Programming

General Terms
Languages, Theory, Experimentation

Keywords
Static analysis, NonNull, annotation, inference, Java

1. INTRODUCTION
A common source of exceptional program behaviour is the dereferencing of null references (also called null pointers), resulting in segmentation faults in C or null pointer exceptions in Java. Even if such exceptions are caught, exception handlers represents additional branches which can make verification more difficult (bigger verification conditions, implicit flow in information flow verification, etc.) and disable some optimizations. Furthermore, the Java virtual machine is obliged to perform run-time checks for non-nullness of references when executing a number of its bytecode instructions, thereby incurring a performance penalty.

As a solution, Fähndrich and Leino proposed a type system [10] to partition references between those which may contain the null constant and those which may not. The user had to annotate the program with non-null types which was not practical in the case of legacy code. We proposed a formal definition of this type system [16], proved its soundness and provided a whole-program dataflow analysis to infer those annotations.

The contribution we present is the tool NIT\(^1\), an implementation resulting from our provably sound analysis. This analysis had been designed on a relatively high level language, well suited for the definition of the analysis and the proofs but too far from the target language of the implementation: the Java bytecode. After recalling the sound analysis this paper is based on, we describe the improvements we have brought to the former analysis. We then present the implementations of both analyses and compare the results of the two implementations on practical benchmarks.

2. A SOUND INFERENCE ANALYSIS
2.1 Non-Null Annotations
The main difficulty in building a precise non-null type system for Java is that all objects have their reference fields set to null at the beginning of their lifetime. Explicit initialization of fields usually occurs during the execution of the constructor, potentially in a method called from the constructor, but it is not mandatory and a field may be read before it is explicitly initialized — in which case it holds the null constant. We consider a field that has not been explicitly initialized to be implicitly initialized to null at the end of the constructor.

Figures 1(a) shows a class C while Fig. 2 shows a model of the lifetime of an instance of class C. Assume no other method writes to C.f. The first part of the object’s lifetime is the execution of the constructor, which is mandatory and occurs to all objects. The field f is always explicitly initialized in the constructor and never written elsewhere, so any read of field f will yield a non-null reference. Despite that, if an annotation must be given for this field valid for the whole lifetime of the object, it will have to represent the

\(^{1}\text{The tool has been released under the GNU General Public License and is available at http://nit.gforge.inria.fr}\)
class C {
    @NonNull Object f;
    C(){
        this.f = new Object();
    }
    @NonNull Object m(@NonNull C x){
        return x.f;
    }
}

Figure 1: Motivating Examples

(a) Too Weak Annotations
(b) Motivation for @Raw Annotations
(c) @Raw Annotations

Figure 2: Lifetime of an Object

class:C
C.f:null
C.f:w

Figure 3: Grammar of the Idealized Language

E ::= x | E.f
I ::= x ← E | x.f ← E | x ← new C(E,...,E) | if (∗) l | x.m(E,...,E) | return

2.2 Sound Inference

One of the key ideas behind non-null annotation inference is to track field initialization in constructors and methods called from constructors. At the end of constructors, all fields defined in the current class which might not have been explicitly initialized are annotated with @NonNull while all other fields are annotated with the value they have been initialized with.

We defined concrete domains and a concrete semantics for the language.

We explained how to interpret State\(^d\) in terms of concrete values with the relation \(\sim \subseteq \text{State}^d \times \text{State}\).

We defined the property safe\((\{P\})\) which holds if all accessible states of the program \(P\) are safe (but \(\{P\}\) is not computable in general).

We defined the property safe\(^d\)\((S^d)\) which holds if \(S^d\) enforces safe\((\{P\})\) (assuming \(S^d\) admits the states of \(\{P\}\) as conservative interpretation, i.e. \(\forall S \in \{P\}. (S^d \sim S)\)).

Finally, we proved the soundness, i.e. we proved that if \(S^d\) is a solution of the constraint system for the program \(P\) (\(S^d \models P\)) and if safe\(^d\)\((S^d)\) holds, then safe\((\{P\})\) holds. (safe\(^d\)\((S^d)\) \& \(S^d \models P\) \Rightarrow safe([P]))
The analysis has been designed without return values but fields that are initialized at the end of the method (post signatures are inferred from the join of the calling context's args and an abstract value for each parameter (initialization state of the fields of the current object (as in 0-CF A [25]). A method signature includes the initialization state of all fields of instances of the current class (and not only fields of instances of the current class) is to keep the checker modular and annotations easy to understand by a human. To simplify the rules, we denote by $M^2$, $H^2$, $T^2$, $L^2$ a value of State². The main rule of the analysis is Rule (1) given in Fig. 6. The first two constraints of this rule state that annotations on method arguments are contravariant, i.e. the lower in the hierarchy the less precise is the annotation. It is standard in object oriented languages as a virtual call to a method in a top-level class may dynamically be resolved to a call in one of its sub-classes. The third constraint is conversely implied by the covariance of the method post-condition (post). The next two constraints link the method signatures with the flow-sensitive information used by the intra-procedural part of the analysis. Finally, the last line constrains the state depending on all instructions of the program using a judgment of the form

$$M^2, H^2, T^2, L^2 \models (m, i) : instr$$

for when the abstract state $M^2$, $H^2$, $T^2$, $L^2$ is coherent with instruction $instr$ at program point $(m, i)$. The two other rules are examples of such judgements.  

Rule (2) corresponds to the instruction that sets the field $f$ of the object stored in the local variable $x$ to the value of the expression $e$. The local variables are unchanged by such an instruction and the abstraction of the expression $e$ is computed ($[e]$) and added to the possible values of $f$. Then, depending on whether $x$ is this or not, the flow-sensitive information $T^2$ may be updated to reflect the fact that a field of the current object has been initialized.

The judgment for the return instruction is given in Rule (3). If the instruction is found in a constructor then the abstraction of the null constant (Nullable) is added to all fields that are not labeled as initialized at the end of the constructor.

## 3. TOWARDS A JVML ANALYSIS

In [16], we presented a first prototype of the analysis and some features that were mandatory to target the Java bytecode language (JVML). The prototype already had a simple must-alias analysis to track the references to this in the stack and some other conservative features. This section proposes three modifications of the analysis on the JVML.

The JVML is a stack language and includes some instructions to test variables for the null constant, such as ifnull n which pops the top of the stack and jumps n bytes of instructions if the popped reference is null. From such an instruction, the analysis infers that, when the test fails, the popped value is non-null but this information is useless to

---

**Figure 4: Concrete Domains**

In the following, we will first give some details about the concrete and abstract domains and the relation between them. Then, we will show some examples of constraint rules of the analysis.

### 2.2.1 Concrete and Abstract Domains

The concrete domains are the domains manipulated by the operational semantics of the language. Their definitions in Fig. 4 are standard except for two particular additions. In order to reason about object and field initialization, we instrumented the semantics and the domains so 1) each field of each object has a flag which indicates if the field has been (explicitly or implicitly) initialized, and 2) the set of classes for which a constructor has been executed is attached to each object. This addition is used to prove the correctness of the refinement of the lookup annotation.

Figure 5 presents the abstractions of the concrete domains. The Val² domain represents the annotation described in Sect. 2.1: references are either abstracted with NonNull, Nullable or Raw. Figure 5 also gives the correctness relations for Val², which express how to interpret Val² in terms of concrete values. The first rule defines Nullable as $\bot$, i.e. Nullable abstracts any value. The second rule defines NonNull as an abstraction of references to objects that have all their fields initialized. The third rule defines Raw as an abstraction of any non-null value. Finally, the forth rule defines Raw(A) as an abstraction of references to objects that have all their fields defined in A or super-classes of A initialized.

The Def² domain is used by TVal² to represent the initialization state of the fields of the current object (this in Java) that are defined in the current class. At the beginning of the object’s lifetime, the abstraction of the current object $T^2 \in TVal²$ associates to each field defined in the current class the abstract value UnDef. The abstraction then evolves as fields are initialized. The reason for limiting the information to fields defined in the current class (and not to all fields of instances of the current class) is to keep the checker modular and annotations easy to understand by a developer. The abstraction used for the heap does not differentiate objects and is flow-insensitive. This is quite standard and corresponds to the purpose of the analysis, i.e. giving one annotation for each field. As the annotations must be easy to read, they are context-insensitive, but, to achieve some precision, the analysis is inter-procedural and method signatures are inferred from the join of the calling contexts (as in 0-CFA [25]). A method signature includes the initialization state of the fields of the current object (this) and an abstract value for each parameter (args). A method signature also represents the result of the method, i.e. the fields that are initialized at the end of the method (post). The analysis has been designed without return values but

---

2.2.2 Constraint based data flow analysis

The analysis computes $S^2 \in \text{State}^2$ such that $S^2 \models P$, i.e. $S^2$ over-approximates all reachable states of the program $P$. We have specified this property as a constraint based data flow analysis. Expressing the analysis in terms of constraints over lattices has the immediate advantage that inference can be obtained from standard iterative constraint solving techniques for static analyses.

To simplify the rules, we denote by $M^2$, $H^2$, $T^2$, $L^2$ a value of State². The complete judgment set can be found in [16].
Val*I = \{Raw, Nullable, NullableInit, NonNull, Nullable<null>\}
∪ \{Raw(Y) | Y ∈ Class\}
Def* = \{Def, UnDef\}
TVaI* = \{\{Method\} \→ Def*\}
Heap* = \{\{x → Val*I\}\}
LocalVar* = \{\{M × N → Val*I\}\}
Method* = \{\{M → \{\{this ∈ TVaI*; args ∈ (Val*I)*; post ∈ TVaI*\}\}\}\}
State* = Method* × Heap* × (\{M × N → TVaI*\})
∩ (\{M × N → LocalVar*I\})

\[\begin{align*}
v ∈ Val & \quad \text{ Nullable } \sim_t v \\
v ∈ dom(h) & \quad \forall f ∈ dom(h(v)), IsDef(h(v)(f)) \\
NonNull & \sim_t v \\
v \not= null & \quad \text{ Raw } \sim_t h v \\
v ∈ dom(h) & \quad \forall f ∈ ∪_{C ≤ fields(C)}, IsDef(h(v)(f)) \\
Raw(A) & \sim_t v 
\end{align*}\]

Figure 5: Abstract Domains and Selected Correctness Relations

Determining the value of a variable may also be trusted when reading fields of variables annotated as NullableInit. A variable annotated as NullableInit may not point to an object that is not fully initialized, so field annotations can also be trusted when reading fields of variables annotated with NullableInit.

The JVM includes the instanceof C instruction which pushes 1 on the stack if the top of the stack is non-null and is an instance of the class C, otherwise it pushes 0. A conditional jump generally occurs few instructions after. Recovering information from such an instruction is not trivial: both the instanceof instruction and the jump are needed, they may be separated by some other instructions, and they interact in the concrete semantics through integer values. To be able to use this information, we have defined another analysis which computes a constraint of the stack such that, for each stack variable, it contains an under-approximation of the set of local variable that must be non-null if the corresponding stack variable is equal to 1.

Figure 6: Some Constraint Based Rules

Figure 7: Recovering Information from Tests

the analysis as this value cannot be reread. For the information to be exploitable, the analysis must know to which local variable the popped value was equal to. To infer equalities between stack and local variables we have implemented a must-alias analysis. For example, in Fig. 7, assuming x is annotated as Nullable at the beginning, this allows to infer that the second load loads a non-null Nullable value.

Assume we have two functions α ∈ 2Vlost → Val* and γ ∈ Val* → 2Vlost which computes the abstraction of a set of concrete values and the concretization of an abstract value, respectively. In Fig. 7, assume that x is either non-null and fully initialized or null at the beginning of the example. The analysis abstracts such a value by

\[\text{NonNull} \cup \alpha(\{\text{null}\}) = \text{Nullable}.\]

The test allows to recover some information but, as Nullable also abstract raw references, the most that can be recovered is

\[\alpha(\gamma(\text{Nullable}) \setminus \{\text{null}\}) = \text{Raw}^-.\]

Such configurations often occur in real programs as implicitly initialized fields are always annotated with Nullable, de-
Table 1: Annotation Results

<table>
<thead>
<tr>
<th>Project</th>
<th>Version</th>
<th>Fields</th>
<th>Parameters</th>
<th>Return</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># Non-null(%)</td>
<td># Non-null(%)</td>
<td># Non-null(%)</td>
<td># Non-null(%)</td>
</tr>
<tr>
<td>Java</td>
<td>BASIC</td>
<td>319</td>
<td>48.6</td>
<td>1663</td>
<td>29.2</td>
</tr>
<tr>
<td></td>
<td>OPT</td>
<td>319</td>
<td>55.8</td>
<td>1660</td>
<td>50.1</td>
</tr>
<tr>
<td>Soot</td>
<td>BASIC</td>
<td>3457</td>
<td>54.2</td>
<td>9793</td>
<td>44.5</td>
</tr>
<tr>
<td></td>
<td>OPT</td>
<td>3456</td>
<td>58.0</td>
<td>9783</td>
<td>54.5</td>
</tr>
<tr>
<td>ESC/Java</td>
<td>BASIC</td>
<td>3416</td>
<td>49.3</td>
<td>3075</td>
<td>28.0</td>
</tr>
<tr>
<td></td>
<td>OPT</td>
<td>744</td>
<td>52.6</td>
<td>3067</td>
<td>42.6</td>
</tr>
<tr>
<td>Julia</td>
<td>BASIC</td>
<td>396</td>
<td>39.4</td>
<td>1481</td>
<td>36.1</td>
</tr>
<tr>
<td></td>
<td>OPT</td>
<td>396</td>
<td>47.2</td>
<td>1481</td>
<td>48.3</td>
</tr>
<tr>
<td>JDTCore</td>
<td>BASIC</td>
<td>1018</td>
<td>45.0</td>
<td>3526</td>
<td>24.7</td>
</tr>
<tr>
<td></td>
<td>OPT</td>
<td>1018</td>
<td>47.0</td>
<td>3525</td>
<td>42.1</td>
</tr>
<tr>
<td>JavaCC</td>
<td>BASIC</td>
<td>116</td>
<td>50.0</td>
<td>292</td>
<td>33.2</td>
</tr>
<tr>
<td></td>
<td>OPT</td>
<td>116</td>
<td>50.0</td>
<td>292</td>
<td>43.5</td>
</tr>
<tr>
<td>Others</td>
<td>BASIC</td>
<td>783</td>
<td>50.2</td>
<td>2244</td>
<td>43.5</td>
</tr>
<tr>
<td></td>
<td>OPT</td>
<td>783</td>
<td>56.1</td>
<td>2244</td>
<td>56.0</td>
</tr>
<tr>
<td>Total</td>
<td>BASIC</td>
<td>8062</td>
<td>48.9</td>
<td>26662</td>
<td>36.3</td>
</tr>
<tr>
<td></td>
<td>OPT</td>
<td>8059</td>
<td>52.7</td>
<td>26650</td>
<td>48.7</td>
</tr>
</tbody>
</table>

as well as being efficient, have easily allowed us not to store bottom values (NonNull and Def). This is specially important as non-reference type are coded as bottom and most variables are non-null. We use sharing extensively and functionally programming has greatly helped us herein. E.g. the stack is implemented as a list and, between two instructions, the part of the stack that is unchanged by the instruction is shared in memory and, to some extent, the same applies to maps. Using sharing has also improved efficiency as it has then been possible to use physical equality tests instead of structural equality tests in some places. The result of the alias and instanceof analysis are compacted to remove the information for the instructions we know the results will not be used. E.g. for the result of the instanceof analysis, only the abstractions of stacks at conditional jumps (~5% of the instructions) are kept.

4. IMPLEMENTATION

The global inferencer is a whole program analysis composed of three analysis: the alias analysis, the analysis of instanceof instructions and the main non-null analysis described in this paper. The non-null analysis uses the results of both the alias and the instanceof analyses and the instanceof analysis uses the results of the alias analysis. Those communications between the analyses impose the simple scheduling of running first the alias analysis, then the instanceof analysis and in the end the non-null analysis. The three analyses have been implemented in a similar standard fashion. First we iterate over all instructions of the program to build transfer functions that take as argument a part of the abstract state and that return the parts of the abstract states that have been modified by the instructions. We store the functions in a hash-map with their dependencies as keys and we apply a work list algorithm to compute the fixpoint. The result of each analysis is the fixpoint computed.

The global result contains, for each program point, three abstractions, one of which, the non-null abstraction, containing already a lot of information. Such an analysis cannot be implemented without taking care of memory consumption. We have done the implementation in OCaml [21] so we have been able to use JavaLib [2] — a class file parser we maintain. We have put a lot of coding effort in reducing the memory consumption. We have implemented LocalVar\(^1\), TVal\(^2\) and Heap\(^3\) as balanced binary trees, which, as well as being efficient, have easily allowed us not to store bottom values (NonNull and Def). This is specially important as non-reference type are coded as bottom and most variables are non-null. We use sharing extensively and functionally programming has greatly helped us herein. E.g. the stack is implemented as a list and, between two instructions, the part of the stack that is unchanged by the instruction is shared in memory and, to some extent, the same applies to maps. Using sharing has also improved efficiency as it has then been possible to use physical equality tests instead of structural equality tests in some places. The result of the alias and instanceof analysis are compacted to remove the information for the instructions we know the results will not be used. E.g. for the result of the instanceof analysis, only the abstractions of stacks at conditional jumps (~5% of the instructions) are kept.

5. EMPIRICAL RESULTS

The benchmarks includes production applications such as Soot 2.2.4 [29], the JDT Core Compiler of Eclipse 3.3.3 [7], Julia 1.4 [19], ESC/Java 2.0b4 [6] and Jess 7.1p1 [18]. It also includes some other smaller applications such as JavaCC [17], Jasmin [23], TightVNC Viewer [28] and the 10 programs constituting the SPEC JVM98 benchmarks [26]. The implementation used for those benchmarks is NIT 0.4, coded in OCaml 3.10.2 [21], and uses the JavaLib 1.7 [2] library. We performed the whole-program analysis with the Java Runtime Environment of GNU gcj 3.4.6 [13] on a MacBook Pro with a 2.4 GHz Intel Core 2 Duo processor with 4 GB of RAM.

In our implementation, we have added switches to enable the modifications we have proposed in this article that may interact with the precision. The results obtained with all the modifications enabled are labeled as OPT and the version without modifications BASIC. Table 1 gives the number of annotations and the percentage of those which are non-null (Raw or NonNull) for field annotations, method parameters (except this) and method results. Notice the two inferencers do not give the same number of annotations: the more precise the analysis is the more dead code is removed. The improvement between the two analyses is substantial: while BASIC inferred 41.5% of non-null annotations, OPT inferred 50.9% of non-null annotations, i.e. the improvement brought by OPT over BASIC represents more than 9 points. Chalin and James experimentally confirmed [3] that at least two thirds of annotations in Java programs in general are supposed to be non-null. With regard to their experiment, 50.9% is already a significant fraction.

The purpose of annotations is to reduce the number of potential exceptions, so we instrumented our inferencer to count the number of dereferences that were proved safe with the inferred annotations. Table 2 shows the percentage of safe dereferences over the total number of dereferences for field reads, field writes, method calls and array operations.
(load, store and length). Note how BASIC implementation was able to prove $2/3$ of dereferences safe and how OPT reduced the number of unsafe dereferencing by a $1/3$, globally proving more than $80\%$ of dereferences in the studied benchmarks safe.

Finally, Tab. 3 gives the memory and time consumption for the most expensive benchmarks and the sum and the maximum memory and time consumption for the other benchmarks for the OPT inferencer. Assuming enough processors and memory, the analyses can be run in parallel so the resources needed correspond to the sum of the memory consumption but the maximum of time consumption, while if the analyses are run sequentially, the resources needed are the maximum memory consumption but the sum of time consumption. The worst case is the analysis of Jess (804,111 bytecode instructions including libraries and excluding dead code), which needs 887 MB and 144 s. Our results indicate that the implementation scales.

### 6. RELATED WORK

Fähndrich and Leino proposed the non-null type system \([10]\) on which this analysis is transitively based. Pap et al. propose a framework \([24]\) for Java source code annotations and, as an example, provide a checker for non-null annotations based on \([10]\). Ekman et al. propose an plug-in \([9]\) for JastAdd \([8]\) to infer and check the same non-null annotations. As their work predates the improvements we proposed in \([16]\) and as we further improved the analysis in this paper, our analysis is strictly more precise. Fähndrich and Xia \([11]\) propose another analysis to deal with object initialization which can deal with circular data structure but the generality of their framework prevents the analysis from being as precise as our on examples without circular data structures. Furthermore, their analysis can only infer a part of the annotations needed.

To infer type annotations, Houdini \([12]\) generates a set of possible annotations (non-null annotations among others) for a given program and uses ESC/Java \([20]\) to refute false assumptions. CANAPA \([5]\) lowers the burden of annotating a program by propagating some non-null annotations. It also relies on ESC/Java to infer where annotations are needed. These two annotation assistants have a simplified handling of objects under construction and rely on ESC/Java \([20]\), which is neither sound nor complete. Some other work has focused on local type inference, \(i.e.\) inferring nullness properties for small blocks of code like methods. One example hereof is the work of Male et al. \([22]\). Spoto very recently proposed another nullness inference analysis \([27]\) with a different domain that expresses logical relations between nullness of variables. He compared his implementation with an old version of our tool NIT which did not include improvements on precision and performance we made. We herein included in the benchmarks some of the programs used in \([27]\) and we can notice that, despite the context sensitivity of the analysis in \([27]\), both analyses have very close practical results.

FindBugs \([15, 14]\) and Jlint \([1]\) use static analyses to find null pointer bugs. To keep the false positive and false negative rates low they are neither sound nor complete.

### 7. CONCLUSION

We have proposed an improved version of our provably sound inference analysis along with an efficient implementation of both analyses. Despite being a whole-program analysis, it is possible to infer annotations for production programs within minutes. The precision of the analysis is not yet sufficient to certify existing code without handwork afterwards, but it is still of interest for code documentation, for reverse engineering and for improving the precision of control flow graphs, which is useful to native code compilers and other program verifications and static analyses.

While we still plan to further improve the inference analysis, we believe the need for a certified checker at the bytecode level is bigger than ever and hence this is in our priorities.
APPENDIX

A. REFERENCES


