A Non-Null Annotation Inferencer for Java Bytecode *

Laurent Hubert
CNRS/IRISA
Campus Beaulieu
35042 Rennes, France
laurent.hubert@irisa.fr

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] (Nullability Inference Tool), 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 during the execution of the constructor to be implicitly initialized to null at the end of the constructors.

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

*This work was supported in part by the Région Bretagne

PASTE '08, Atlanta, Georgia USA

[1]The tool has been released under the GNU General Public License and is available at `http://nit.gforge.inria.fr`

```
class C {
 @Nullable Object f;
 C(){
  this.f = new Object();
 }
 @Nullable Object m(@NonNull C x){
  return x.f;
 }
}
```

(a) Too Weak Annotations

```
class C {
 @NonNull Object f;
 C(){
  m(this);
  this.f = new Object();
 }
 @NonNull Object m(@NonNull C x){
  return x.f;
 }
}
```

(b) Motivation for `@Raw` Annotations

```
class C {
 @NonNull Object f;
 C(){
  m(this);
  this.f = new Object();
 }
 @Nullable Object m(@Raw C x){
  return x.f;
 }
}
```

(c) `@Raw` Annotations

Figure 1: Motivating Examples

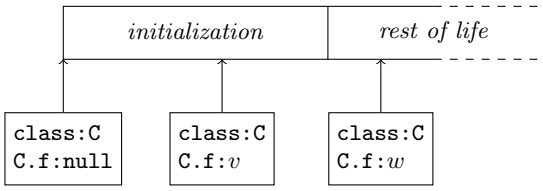

Figure 2: Lifetime of an Object

non-null reference put in the field by the constructor, but also the default null constant present at the beginning of the object's lifetime. Such an annotation is basically `@Nullable` where we would have clearly preferred a more precise information, such as `@NonNull`. The solution is to consider that annotations on fields are only valid after the end of the constructor, during the rest of the object's lifetime. The `@Nullable` annotations can now be replaced with `@NonNull` annotations.

Figure 1(b) shows the same class where `@Nullable` annotations have been replaced and a call to the method `m` has been added to the constructor. The method `m` simply reads and returns the value of `f`. Although this method is not a constructor, the object `x` may still be in its construction phase, *i.e.* the constructor of the object from which the field is read may not have been fully executed, and the value actually read may be null in contradiction with the field annotation. In fact, for each variable, we need to know whether the reference may point to an object that is still in its construction phase. We annotate with `@Raw` such variables. As the invariant described by the annotations is not yet established during the object initialization, reading a field of an object annotated as `@Raw` may return a null value (`@Nullable`) whatever is the annotation on the field. The example in Fig. 1(b) has been corrected in Fig. 1(c).

We refine those `@Raw` annotations by indicating the set of classes for which a constructor has been executed. Annotations concerning fields declared in those classes can be considered as already valid despite the fact the initialization of the object is not yet completely finished. The set of classes for which a constructor has been executed can be represented by a single class as the execution order of the constructors is constrained by the class hierarchy.

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 `@Nullable` while all other fields are annotated with the value they have been initialized with.

Variables	x	$\in$	$\mathbb{V}$
Fields	f	$\in$	$\mathbb{F}$
Class names	C	$\in$	$\mathbb{C}$
Labels	l	$\in$	$\mathbb{N}$
Method signatures	m	$\in$	$\mathbb{M}$

$$
\begin{array}{rcl}
E & ::= & x \mid E.f \\
I & ::= & x \leftarrow E \mid x.f \leftarrow E \mid x \leftarrow \textbf{new } C(E,\ldots,E) \\
 & & \mid \textbf{if } (\star)\ l \mid x.m(E,\ldots,E) \mid \textbf{return}
\end{array}
$$

Figure 3: Grammar of the Idealized Language

In [16], we chose to formalize and prove our analysis on a language close to the bytecode but without an operand stack. This trade-off came from the fact that we planed an implementation at the bytecode level but removing the stack avoided to deal with alias information, which simplified the presentation and the proofs. Figure 3 shows the grammar of the idealized language.

To build our provably sound inference analysis we defined an *abstract domain* $\text{State}^\sharp$ and a *constraint based specification* of the analysis that constrain $\text{S}^\sharp \in \text{State}^\sharp$ depending on a program P (denoted by $\text{S}^\sharp \models P$). To prove the soundness of the analysis we completed several tasks.

- We defined *concrete domains* and a *concrete semantics* for the language.
- We explained *how to interpret* $\text{State}^\sharp$ in terms of concrete values with the relation $\sim \subseteq \text{State}^\sharp \times \text{State}$.
- We defined the property $\text{safe}([\![P]\!])$ which holds if all accessible states of the program P are safe (but $[\![P]\!]$ is not computable in general).
- We defined the property $\text{safe}^\sharp(\text{S}^\sharp)$ which holds if $\text{S}^\sharp$ enforces $\text{safe}([\![P]\!])$ (assuming $\text{S}^\sharp$ admits the states of $[\![P]\!]$ as conservative interpretation, *i.e.* $\forall \text{S} \in [\![P]\!].(\text{S}^\sharp \sim \text{S})$).
- Finally, we proved the soundness, *i.e.* we proved that if $\text{S}^\sharp$ is a solution of the constraint system for the program P ($\text{S}^\sharp \models P$) and if $\text{safe}^\sharp(\text{S}^\sharp)$ holds, then $\text{safe}([\![P]\!])$ holds. ($\text{safe}^\sharp(\text{S}^\sharp) \wedge \text{S}^\sharp \models P \Rightarrow \text{safe}([\![P]\!])$)

$$
\begin{aligned}
\text{Val} &= \text{Loc} + \{\mathbf{null}\} \\
\text{Object} &= \mathbb{F} \rightharpoonup \text{Val} \times \{\mathbf{def}, \mathbf{undef}\} \\
\text{LocalVar} &= \mathbb{V} \to \text{Val} + \{\bot\} \\
\text{Heap} &= \text{Loc} \rightharpoonup \text{Object} \times \mathbb{C} \times \wp(\mathbb{C}) \\
\text{State} &= (\mathbb{N} \times \text{LocalVar} \times \text{Heap}) + \Omega
\end{aligned}
$$

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 `@Raw` annotation.

Figure 5 presents the abstractions of the concrete domains. The $\text{Val}^\sharp$ 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 $\text{Val}^\sharp$, which express how to interpret $\text{Val}^\sharp$ in terms of concrete values. The first rule defines Nullable as $\top$, *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 $\text{Raw}(A)$ as an abstraction of references to objects that have all their fields defined in A or super-classes of A initialized.

The $\text{Def}^\sharp$ domain is used by $\text{TVal}^\sharp$ 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 $T^\sharp \in \text{TVal}^\sharp$ associate 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 adding them is straightforward. Those domains are then combined to form the abstract state that correspond to all reachable state of the program. To be able to use strong updates [4], *i.e.* to precisely analyse assignments, the abstractions of the local variables and the current object are flow-sensitive while, as discussed earlier, the abstraction of the heap is flow-insensitive.

2.2.2 Constraint based data flow analysis

The analysis computes $\text{S}^\sharp \in \text{State}^\sharp$ such that $\text{S}^\sharp \models P$, *i.e.* $\text{S}^\sharp$ 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 $M^\sharp, H^\sharp, T^\sharp, L^\sharp$ a value of $\text{State}^\sharp$. 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

$$M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\text{m}, i) \ : \ \mathit{instr}$$

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

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 ($[\![e]\!]^\sharp$) and added to the possible values of f. Then, depending on whether x is `this` or not, the flow-sensitive information $T^\sharp$ 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

[2] The complete judgment set can be found in [16].

$$
\begin{aligned}
\mathrm{Val}^\sharp &= \{\mathrm{Raw}^-, \mathrm{NonNull}, \mathrm{Nullable}\} \\
&\quad \cup \{\mathrm{Raw}(Y) \mid Y \in \mathrm{Class}\} \\
\mathrm{Def}^\sharp &= \{\mathrm{Def}, \mathrm{UnDef}\} \\
\mathrm{TVal}^\sharp &= \mathbb{F} \rightharpoonup \mathrm{Def}^\sharp \\
\mathrm{Heap}^\sharp &= \mathbb{F} \to \mathrm{Val}^\sharp \\
\mathrm{LocalVar}^\sharp &= \mathbb{V} \to \mathrm{Val}^\sharp \\
\mathrm{Method}^\sharp &= \mathbb{M} \to \{\!|\mathsf{this} \in \mathrm{TVal}^\sharp; \mathsf{args} \in (\mathrm{Val}^\sharp)^*; \mathsf{post} \in \mathrm{TVal}^\sharp|\!\} \\
\mathrm{State}^\sharp &= \mathrm{Method}^\sharp \times \mathrm{Heap}^\sharp \times \left(\mathbb{M} \times \mathbb{N} \rightharpoonup \mathrm{TVal}^\sharp\right) \\
&\quad \times \left(\mathbb{M} \times \mathbb{N} \rightharpoonup \mathrm{LocalVar}^\sharp\right)
\end{aligned}
$$

$$
\frac{v \in \mathrm{Val}}{\mathrm{Nullable} \sim_h v}
\qquad
\frac{v \in \mathrm{dom}(h) \quad \forall f \in \mathrm{dom}(h(v)), \mathrm{IsDef}(h(v)(f))}{\mathrm{NonNull} \sim_h v}
$$

$$
\frac{v \neq \mathbf{null}}{\mathrm{Raw}^- \sim_h v}
\qquad
\frac{v \in \mathrm{dom}(h) \quad \forall f \in \bigcup_{A \preceq C} \mathrm{fields}(C), \mathrm{IsDef}(h(v)(f))}{\mathrm{Raw}(A) \sim_h v}
$$

Figure 5: Abstract Domains and Selected Correctness Relations

$$
\frac{\begin{array}{c}
\forall \mathrm{m}, \mathrm{m}', \mathrm{overrides}(\mathrm{m}', \mathrm{m}) \Rightarrow M^\sharp(\mathrm{m})[\mathsf{args}] \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{args}] \\
\forall \mathrm{m}, \mathrm{m}', \mathrm{overrides}(\mathrm{m}', \mathrm{m}) \Rightarrow \top_{\mathrm{class}(\mathrm{m}')} \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{this}] \\
\forall \mathrm{m}, \mathrm{m}', \mathrm{overrides}(\mathrm{m}', \mathrm{m}) \Rightarrow \top_{\mathrm{class}(\mathrm{m})} \sqsubseteq M^\sharp(\mathrm{m})[\mathsf{post}] \\
\forall \mathrm{m},\ M^\sharp(\mathrm{m})[\mathsf{this}] \sqsubseteq T^\sharp(\mathrm{m}, 0) \qquad \forall \mathrm{m},\ M^\sharp(\mathrm{m})[\mathsf{args}] \sqsubseteq L^\sharp(\mathrm{m}, 0) \\
\top_{\mathrm{class}(\mathbf{main})} \sqsubseteq T^\sharp(\mathbf{main}, 0) \qquad \mathrm{Raw}^- \sqsubseteq L^\sharp(\mathbf{main}, 0)(\mathbf{this}) \\
\forall \mathrm{m},\ \forall i,\ M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m}, i)\ :\ P_{\mathrm{m}}[i]
\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P} \quad (1)
$$

$$
\frac{\begin{array}{c}
\text{if } x = \mathbf{this} \wedge f \in \mathrm{fields}(\mathrm{class}(\mathrm{m})) \text{ then } T^\sharp(\mathrm{m}, i)[f \mapsto \mathrm{Def}] \text{ else } T^\sharp(\mathrm{m}, i) \sqsubseteq T^\sharp(\mathrm{m}, i+1) \\
L^\sharp(\mathrm{m}, i) \sqsubseteq L^\sharp(\mathrm{m}, i+1) \qquad [\![e]\!]^\sharp \sqsubseteq H^\sharp(f)
\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m}, i)\ :\ x.f \leftarrow e} \quad (2)
$$

$$
\frac{\begin{array}{c}
T^\sharp(\mathrm{m}, i) \sqsubseteq M^\sharp(\mathrm{m})[\mathsf{post}] \\
\mathrm{name}(\mathrm{m}) = \mathbf{init} \Rightarrow \forall f \in \mathrm{fields}(\mathrm{class}(\mathrm{m})).(T^\sharp(\mathrm{m}, i)(f) = \mathrm{UnDef} \Rightarrow \mathrm{Nullable} \sqsubseteq H^\sharp(f))
\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m}, i)\ :\ \mathbf{return}} \quad (3)
$$

Figure 6: Some Constraint Based Rules

```
load x      x ↦ Nullable
ifnull n    x ↦ NonNull
load x      x ↦ NonNull
            ...
```

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 (`@Raw`) value.

Assume we have two functions $\alpha \in 2^{\mathrm{Val}} \to \mathrm{Val}^\sharp$ and $\gamma \in \mathrm{Val}^\sharp \to 2^{\mathrm{Val}}$ 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

$$\mathrm{NonNull} \sqcup \alpha(\{\texttt{null}\}) = \mathrm{Nullable}.$$

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

$$\alpha(\gamma(\mathrm{Nullable}) \setminus \{\texttt{null}\}) = \mathrm{Raw}^-.$$

Such configurations often occur in real programs as implicitly initialized fields are always annotated with Nullable, despite they may never contain any raw value. To solve this, we add NullableInit, a new abstract value that abstract values that may not point to raw objects. We have

$$\mathrm{NonNull} \sqsubset \mathrm{NullableInit} \sqsubset \mathrm{Nullable}$$
$$\alpha(\gamma(\mathrm{NullableInit}) \setminus \{\texttt{null}\}) = \mathrm{NonNull}\ .$$

It allows to annotate more references as `@NonNull` and therefore to gain in precision as field annotations can then be trusted. It also allows a more direct gain in precision. 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 JVML 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 an abstraction 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.

Finally, when a possibly null value is dereferenced, if the control flow reaches the next instruction it means that, at this point, the reference is non-null. Therefore, it is pos-

Project	Version	Fields		Parameters		Return		Total	
		#	Non-null(%)	#	Non-null(%)	#	Non-null(%)	#	Non-null(%)
Jess	*BASIC*	319	48.6	1663	29.2	789	44.7	2771	35.8
	OPT	319	55.8	1660	50.1	788	51.1	2767	51.0
Soot	*BASIC*	3457	54.2	9793	44.5	4177	57.3	17427	49.5
	OPT	3456	58.0	9793	54.5	4177	63.4	17426	57.3
ESC/Java	*BASIC*	746	49.5	3075	28.0	1155	50.6	4976	36.5
	OPT	744	52.6	3067	42.6	1152	58.6	4963	47.8
Julia	*BASIC*	396	39.4	1481	36.1	842	52.1	2719	41.5
	OPT	396	47.2	1481	48.3	842	63.4	2719	52.8
JDTCore	*BASIC*	1018	45.0	3526	24.7	916	38.9	5460	30.9
	OPT	1018	47.0	3525	42.1	916	46.9	5459	43.8
JavaCC	*BASIC*	116	50.0	292	33.2	85	77.6	493	44.8
	OPT	116	50.9	292	43.5	85	82.4	493	51.9
Others	*BASIC*	783	50.2	2244	43.5	717	43.2	3744	44.8
	OPT	783	56.1	2244	56.0	717	50.6	3744	55.0
Total	*BASIC*	8062	48.9	26662	36.3	10592	48.8	45316	41.5
	OPT	8059	52.7	26650	48.7	10588	55.1	45297	50.9

Table 1: Annotation Results

sible to refine all instructions that dereference variables so those variables are inferred as non-null on outgoing non-exceptional edges of the control flow graph.

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$^\sharp$, TVal$^\sharp$ and Heap$^\sharp$ 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 functional 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 ($\sim 5\%$ 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

Project	Version	Field Read		Field Write		Method Call		Array Operations		Total	
		#	Safe(%)	#	Safe(%)	#	Safe(%)	#	Safe(%)	#	Safe(%)
Jess	*BASIC*	2725	85.4	919	97.7	10358	63.0	729	73.9	14731	69.8
	OPT	2721	97.4	917	98.0	10330	76.7	727	83.2	14695	82.2
Soot	*BASIC*	25323	76.5	7038	92.7	89161	73.4	3092	57.3	124614	74.7
	OPT	25319	80.7	7037	95.0	89138	82.2	3092	69.8	124586	82.3
JDTCore	*BASIC*	29200	75.2	7569	85.8	25205	50.2	10769	21.6	72743	59.7
	OPT	29196	88.3	7568	92.3	25197	72.6	10765	45.9	72726	77.0
JavaCC	*BASIC*	5733	81.7	1096	92.7	11772	66.0	2129	50.4	20730	70.2
	OPT	5733	92.6	1096	95.8	11772	71.8	2129	65.9	20730	78.2
ESC/Java	*BASIC*	10799	50.3	2570	90.9	19129	67.0	1441	54.3	33939	63.0
	OPT	10787	90.3	2568	98.4	19077	78.9	1441	79.9	33873	84.1
Julia	*BASIC*	4474	75.6	1065	90.0	15836	72.2	987	39.9	22362	72.3
	OPT	4474	82.9	1065	95.0	15835	85.2	987	55.0	22361	83.9
Others	*BASIC*	12341	72.8	3295	90.5	19189	66.4	3182	32.2	38007	67.7
	OPT	12341	79.3	3295	93.4	19182	76.4	3182	62.7	38000	77.7
Total	*BASIC*	104496	76.2	26882	91.1	230574	67.5	25864	34.9	387816	69.3
	OPT	104468	87.2	26874	95.0	230447	78.3	25858	56.5	387647	80.4

Table 2: Dereferencing Results

		Space (MB)	Time (s)
Jess		887	144
Soot		634	122
ESC/Java		421	51
Julia		363	44
JDTCore		331	36
JavaCC		310	34
Others	sum	1642	159
	max	253	30

Table 3: Time and Space Consumption

(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 dereferences 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. Papi *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. Those 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

[1] Artho, C., and Havelund, K. Applying Jlint to space exploration software. In *Proc. of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)* (2004), vol. 2937 of *LNCS*, Springer, pp. 297–308.

[2] Cannasse, N., Turpin, T., Hubert, L., Besson, F., and André, E. *JavaLib*. Inria, March 2007. `javalib.gforge.inria.fr`.

[3] Chalin, P., and James, P. R. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proc. of European Conference on Object-Oriented Programming (ECOOP'07)* (2007), vol. 4609 of *LNCS*, Springer, pp. 227–247.

[4] Chase, D. R., Wegman, M., and Zadeck, F. K. Analysis of pointers and structures. In *Proc. of the ACM conference on Programming Language Design and Implementation (PLDI'90)* (1990), ACM, pp. 296–310.

[5] Cielecki, M., Fulara, J., Jakubczyk, K., and Jancewicz, Ł. Propagation of JML non-null annotations in Java programs. In *Proc. of the 4th international symposium on Principles and practice of programming in Java (PPPJ'06)* (2006), ACM Press, pp. 135–140.

[6] Cok, D. R., and Kiniry, J. R. Esc/java2.

[7] Eclipse 3.3 (europa). `www.eclipse.org`.

[8] Ekman, T. *Extensible Compiler Construction*. PhD thesis, Lund University, June 2006.

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

[10] Fähndrich, M., and Leino, K. R. M. Declaring and checking non-null types in an object-oriented language. In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03)* (2003), vol. 2021 of *LNCS*, Springer-Verlag, pp. 302–312.

[11] Fähndrich, M., and Xia, S. Establishing object invariants with delayed types. In *Proc. of the Conference on Object Oriented Programming Systems and Applications (OOPSLA'07)* (2007), ACM, pp. 337–350.

[12] Flanagan, C., and Leino, K. R. M. Houdini, an annotation assistant for ESC/Java. In *Proc. of the International Symposium of Formal Methods Europe (FME'01)* (2001), vol. 2021 of *LNCS*, Springer-Verlag, pp. 500–517.

[13] The gnu compiler for the java programming language, 2007. `gcc.gnu.org/java`.

[14] Hovemeyer, D., and Pugh, W. Finding more null pointer bugs, but not too many. In *Proc. of the 7th workshop on Program Analysis for Software Tools and Engineering (PASTE'07)* (2007), ACM Press, pp. 9–14.

[15] Hovemeyer, D., Spacco, J., and Pugh, W. Evaluating and tuning a static analysis to find null pointer bugs. *SIGSOFT Software Engineering Notes 31*, 1 (2006), 13–19.

[16] Hubert, L., Jensen, T., and Pichardie, D. Semantic foundations and inference of non-null annotations. In *Formal Methods for Open Object-Based Distributed Systems* (June 2008), vol. 5051 of *LNCS*, Springer Berlin, pp. 132–149.

[17] Javacc. `javacc.dev.java.net`.

[18] Jess. `herzberg.ca.sandia.gov`.

[19] Julia. `profs.sci.univr.it/~spoto/julia/`.

[20] Leino, K. R. M., Saxe, J. B., and Stata, R. *ESC/Java user's manual*, technical note 2000-002 ed. Compaq Systems Research Center, October 2000.

[21] Leroy, X., Doligez, D., Garrigue, J., Rémy, D., and Vouillon, J. *The Objective Caml system*. Inria, May 2007. `caml.inria.fr/ocaml/`.

[22] Male, C., Pearce, D. J., Potanin, A., and Dymnikov, C. Java bytecode verification for @NonNull types. In *Proc. of the Conference on Compiler Construction (CC'08)* (2008), Springer-Verlag.

[23] Meyer, J., and Downing, T. *Java Virtual Machine*. O'Reilly Associates, 1997.

[24] Papi, M. M., Ali, M., Correa Jr., T. L., Perkins, J. H., and Ernst, M. D. Practical pluggable types for Java. In *Proc. of the 2008 International Symposium on Software Testing and Analysis (ISSTA 2008)* (July 22–24, 2008).

[25] Shivers, O. Control flow analysis in scheme. In *Proc. of the ACM conference on Programming Language Design and Implementation (PLDI'88)* (1988), ACM, pp. 164–174.

[26] SPEC JVM98 benchmarks. `www.spec.org/jvm98`.

[27] Spoto, F. Nullness analysis in boolean form. In *6th IEEE International Conference on Software Engineering and Formal Methods* (November 2008), IEEE Computer Society Press. To appear.

[28] Tightvnc. `www.tightvnc.com`.

[29] Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., and Co, P. Soot — a Java bytecode optimization framework. In *Proc. of the 1999 conference of the Centre for Advanced Studies on Collaborative research (CASCON'99)* (1999), IBM Press, p. 13.