NON-NULL TYPES FOR JAVALI

Peter Güttinger and Marcel Mohler

Advanced Compiler Design
ETH Zürich
Zürich, Switzerland

ABSTRACT

In most programming languages, an attempt to dereference `null` values or calling a method on a `null` reference results in a program fault or an exception.
Javali [1], an academic programming language used in various compiler courses at ETH Zürich, implements run time null checks and exits the program with an error code when trying to dereference a `null` reference.
We present an addition to the Javali language specification and an implementation we call *Javali!?* that allows the definition of `non-null` and `possibly-null` types.
This allows us to check for `null` references statically and completely eliminate the possibility of a `null` pointer dereference.

1. INTRODUCTION

Motivation. Null pointer exceptions are a common cause of troubles. In a conference in 2009, Tony Hoare calls the introduction of `null` references into `ALGOL W` (1965) his *billion-dollar-mistake* [2]. Issues caused by `null` dereferences were root of various, costly program errors and according to his estimate resulted in a billion dollar of pain and damage over the last 40 years. In a work by Patrice Chalin and Perry R. James they noted that 5% of bug fixes between release 3.2 and 3.3 of Eclipse JDT, a widely used Java development tool, were directly attributed to null pointer exceptions [3].
Listing 1 shows a simple code snippet, where accessing variable `a` might result in a null pointer exception.

```
A a;
if(*)
  a = new A();
a.foo() // possibly null
```

Listing 1. Example null pointer exception

By extending the type system of a programming language one can statically prove that a program is free of `possibly-null` dereferences and thereby provide these guarantees at compile time.
This comes at the cost of additional type modifiers to declare a type `non-null` or `possibly-null`. The programmer has to specify a reference type by `RefType!` if this is a `non-null` reference or by `RefType?` if the reference might also contain `null`.
We propose a *non-null type system* for Javali, a language that represents a subset of Java, which we call *Javali!?* and implemented it into the Javali compiler used in the Compiler Design and Advanced Compiler Design Courses at ETH Zürich[4]. The system has been designed to be as intuitive as possible for the programmer while trying to keep additional annotations to a minimum. To achieve proper initialization of objects and arrays, Javali got extended with the ability to define constructors and a new array creation method. Javali!? uses a data flow analysis to remain a powerful language and avoid restricting the programmer.
In Section 2.1 we provide a detailed look at the type system and its implementation. We continue with Section 2.2 on the issues of object construction and provide a description how we solved these issues. We present the data flow analysis and the impact on expressiveness in Section 2.3. An experimental analysis of the correctness, expressiveness and performance effects on Javali is described in Section 3. We conclude with some final remarks and possible further extensions in Sections 4 and 5.
Related work. In 2003 Manuel Fähndrich and K. Rustan M. Leino published the paper "*Declaring and checking non-null types in an object-oriented language*" [5], which describes the authors' experiences with a non-null type system for C#. It presents the basic concept as well as a description of the issues in regards of object construction.
There exist several programming languages that provide a non-null type system. Notable examples are *Spec#* [6][7], a research programming system by Microsoft, with focus on soundness and verification. *Rust*, a programming language by Mozilla [8] disallows the definition of a variable with `null`. Instead, it uses a construct with the `Some(T)` and `None(T)` option types and forces the programmer to handle these, while being able to provide safety-guarantees at compile-time. Also, recently released languages like *Swift*[9] and *Kotlin*[10] include non-null typesystems.

2. OUR SOLUTION

2.1. Type system

Syntax. In order to differentiate between `non-null` and `possibly-null` reference types the language syntax and in turn the parser have to be modified. The proposed non-null type system allows the programmer to annotate any reference type with `!` or `?`. `!` stands for `non-null`, so a reference of type `A!` must always contain a reference to an actual object of type `A`. `A?`, also called `possibly-null`, means that the reference may be `null`, i.e. does not have to point to an object. One can use these annotations on all reference types except in class definitions, casts and `new` expressions. If no annotation is provided, the type system will assume it has been declared as `non-null`, except for local variables, whose nullness will be inferred by the analysis later. From experience we conclude that `non-null` references are usually what the programmer wants. Null references are usually a special case that should be treated differently. Code examples can be seen in Listing 2.

```
A? a1; // a1 is possibly-null
A! a2; // a2 is non-null
A a3;  // a3 is non-null (default)
A? foo(A! arg1, A arg2) {} // parameters and return
     types. both arguments are non-null.
```

Listing 2. Example of new types

Arrays. Javali!? also support nullness types for arrays. It tracks the nullness of the array reference and the array elements separately and the syntax is shown in Listing 3.

```
A[!]! a1; // non-null elements and non-null reference
A[!]? a2; // non-null elements and possibly-null ref.
A[?]! a3; // possibly-null elements and non-null ref.
A[?]? a4; // possibly-null elem. and possibly-null ref.
A[] a5; // same as A[!]!
```

Listing 3. Example of array types

We claim that having one annotation inside the brackets and another one outside is intuitive and not a syntactical issue since we do not allow the use of the new keyword with nullness annotations. Therefore the bracket remains empty in all uses where nullness annotations are allowed.

Casts. We decided to disallow explicit casts between different null types. Instead, the programmer is always allowed to store a `non-null` reference in a `possibly-null` reference and he can wrap an assignment of `possibly-null` to `non-null` into a `non-null` check (e.g. `if(a != null)`).

```
A! a1; // a1 is non-null
A? a2; // a2 is possibly-null
a2 = a1; // allowed
a1 = a2; // disallowed
a2 = (A?)a1; // disallowed
a1 = (A!)a2; // disallowed
if (a2 != null) {
  a1 = a2; }// allowed
```

Listing 4. Example of possible casts

We will explain these semantics in detail in Section 2.3 about the data flow analysis. An overview of allowed and disallowed casts can be found in Listing 4.

Implementation. The null types are implemented as a modifier of an expression's type. When previously in Javali they had only a type, in Javali!? they now also have a nullness (or two for array types, as the component type also has a nullness). This allows to dynamically adapt the null types without interfering with the type system a lot, as would be required if each type were split into two types, a `non-null` one and a `possibly-null` one.

Type checking. Since the null annotations are not part of a type, they have to be type-checked in addition to the normal type check. Since those two checks are unrelated, the null type check can run after the analysis, which in turn can run after the normal type check. This is the best ordering for our purpose, as it allows the analyzer to be able to rely on the normal types, and can infer null types freely which will be checked afterwards.

2.2. Object construction

Constructors. All `non-null` fields of an object need to be initialized at the end of the constructor to be able to guarantee them being `non-null` for the rest of the program. Listing 5 shows possible issues that arise if object construction and initialization is not enforced.

```
class Main {
  void main() {
    A! a;
    a = new A();
    a = a.next; // allowed by type system
    a.foo(); // segfaults since a is unexpectedly null
  }
}
class A {
  A! next; // non-null field
  A() {} // empty constructor
  void foo() {}
}
```

Listing 5. Possible errors with object incomplete initialization/construction

The Javali language specification [1] and reference compiler do not include constructors so they had to be added. This involved adapting most parts of the compiler like parser, type checker, semantic analyzer and code generator. In Javali!?, the interpreter has also been modified to properly support the added functionality.

As seen in Listing 5, constructors are defined with the common Java syntax by defining a method with no return type and a method name that equals the class name.

In addition Javali!? also supports Java-like super calls which call the constructor of the super class but unlike Java allows them anywhere in the constructor as long as they are not nested inside an `if` or `while`, there must be at most one super call, and they must not occur after a possibly nested

`return` statement. If an explicit super call is omitted, and implicit one will be added at the beginning of the constructor. These rules make sure that the super constructor is always called exactly once.

Array constructors. Reasoning about the initialization of array elements is complex. Intuitively one can think of an array as an object with numbered fields. This means that, similarly to the object constructors, the semantic checker needs to be sure that all `non-null` elements are initialized after the array construction. However, the access to the elements can happen via arbitrary complex expressions that might not even be able to be evaluated at compile time (e.g. if they contain input variables). Therefore sound reasoning is a lot harder and might even be too restrictive.

Since arrays only allow either all elements to be `non-null` or all elements to be `possibly-null`, Javali!? allows a second parameter to array constructors which initialize all elements to a given default reference or default value. After the array constructor call the semantic checker can then safely assume the nullness type of the default reference for all array elements. The code snippet in Listing 6 shows their usage.

```
A! a; // non-null reference of a
A[!]! as1; // non-null array elements
A[?]! as2; // possibly-null array elements
a = new A(); // we can now assume a is properly
     initialized
as1 = new A[5, a]; // new array constructor which
     allows a default reference as a second parameter
// now all 5 elements of as1 point to a
as2 = new A[5, null]; // same as as2 = new A[5]
// now all 5 elements of as2 point to null
```

Listing 6. Examples of array initialization

2.3. Data flow analysis

General notes. Javali!? includes a powerful forward dataflow analysis to propagate information needed to reason about nullness properties.

It gets executed on the control flow graph as part of the semantic checker phase but before the conversion to SSA (single static assignment).

Javali!? extends the variety of existing semantic errors with specific errors targeted at different properties of the nonnull system to provide the programmer with reasonable error messages. The individual parts are described below.

Nullness analysis. In general, the analysis tracks nullness of certain references: local variables, parameters, fields of the current class and superclasses, and simple foreign fields and array locations (e.g. `x.y` or `a[b + 5]`, where `x` and `a` are in turn tracked references, and array indices must only consist of constants, unary and binary operations, and local variables). It tries to find the nullness of each tracked reference for every location in the code, as it may change over time, for example due to assignments and non-null checks. The analysis is local, i.e. no information is inferred from other classes and methods and exclusively uses declared information like non-null fields and return types. This makes the analysis lose all inferred information on fields and arrays whenever a method is called, as that method may change them. The analysis also loses all information on arrays on all assignments since it employs no points-to analysis.

Non-null checks. If a programmer inserts a manual nonnull check, the compiler is safe to assume that one of the branches satisfies the condition and propagates the nullness information accordingly. Javali!? supports binary checks for equality and inequality as well as conjunctions and disjunctions of these checks. One operand needs to be a `null` constant, while the other operand must be a tracked reference. Listing 7 illustrates how the compiler can use the information to allow a field access.

```
class Main {
  A? a; // assume A has an integer field x
  void foo() {
    if(a != null) {
      write(a.x); // allowed
    } else {
      write(a.x); // not allowed
    }
  }
}
```

Listing 7. A simple manual non-null check

Constructor analysis. In Javali!?, a constructor needs to make sure all `non-null` fields are initialized at the end of the constructor. The data flow analysis tracks definition of all fields (which also includes inherited fields, though the super call will usually set those) and reports an error in case some necessary field initializations are missing.

Points-to-this analysis. At some point in a constructor the `this` object can have already initialized and still uninitialized fields. So even though it is clearly not a reference to `null` it violates the contract that all `non-null` fields are initialized. In Listing 8 the unqualified method call on `this` leads to a null pointer exception in the method `len()` since the field `list` has not been initialized yet.

```
class A {
  List! list;
  A() {
    list = new List(len()); // unqualified call
  }
  int len() {
    return list.size(); // null pointer if called
  }                     // from constructor
}
```

Listing 8. An unqualified method call on this leads to a null pointer

To circumvent this and similar issues, the programmer is not allowed to call methods on `this` or use `this` as an argument to a method, constructor or super call.

However, local variables in the constructor could still point to `this`, and it may be required to store a reference to

`this` in a field. Therefore Javali!? keeps track of all variables and fields that might point to `this` to enforce the restrictions mentioned before while still allow assignments of `this`.
This analysis is also used to disallow assigning `this` to a field before calling the super constructor to prevent it from leaking `this`, and disallows casts on any reference that may be `this` to prevent access to fields of subclasses in super constructors, as those might not yet be assigned.

3. EVALUATION

Experimental setup. All measurements were run on an Intel i7-3667U processor with 2.00 GHz clock speed and 8 GB of DDR3 Memory. We used Arch Linux 64bit with a 4.5.4 Kernel and GCC 6.1.1. The test suite was also run on Windows to ensure platform independence.
Testing framework. Javali!? includes a JUnit [11] testing framework based on the code available from the Javali reference compiler. Figure 1 gives a schematic overview. We start by collecting all Javali files in a predefined folder. For each of these files, the parser gets invoked to check whether the source code is syntactically correct. If this is not the case, the test fails immediately.
If correct, the framework runs the semantic analyzer, which includes the non-null type checker and the control flow analysis. For each test, one can provide an *expected error file* which contains the expected semantic error. If no file is provided, the analyzer expects no error in the semantic phase.
In case the analyzer throws an error and matches the expected error of the error file, the test continues. Otherwise the testing framework reports a failure.
If all previous phases succeeded, Javali!? continues with the code generation and runs the compiled binary.
Additionally, the program gets executed with the interpreter. The interpreter features three additional counters. On each step of the interpreter, which corresponds to each call to the ExprVisitor a global counter is increased. Additionally a counter is increased on each step that includes a non null check in the reference Javali compiler. Namely, field accesses, array accesses and method invocations. Because these checks are removed in code produced by Javali!?, this is an easy way to count the number of saved null checks during execution without the need to run the reference compiler. The benefit over counting the null checks in the assembler file statically is that this method yields a precise counter of how many checks are actually avoided at run time.
Since the programmer might need to add explicit null checks, these are also counted and reported alongside the removed, implicit null checks.
To verify the correctness of the implementation of code generator and interpreter, both of their outputs are compared against each other and the test only succeeds if both contain the same result.

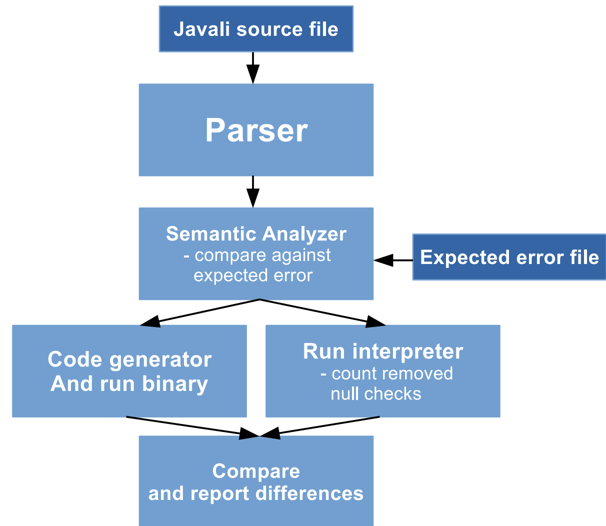

Fig. 1. Overview of the testing framework

Test suite. The test suite used in the evaluation consists of over 50 Javali test files which are meant to cover all different errors from the semantic phase.
Additionally we included 23 more advanced Javali programs from [4]. These include an implementation of QuickSort, BubbleSort, Fibonacci, Linked Lists and many more. We did exclude some of the programs that did not even run with the reference compiler since they require additional language features.
Correctness results. Correctness is the number one priority for a compiler. The test suite runs without errors where the compiler and the test suite do not behave as expected, though the suite may not be complete.
Completeness/Expressiveness results. In a very limited language it is easier to achieve correctness. And since our non-null type system increases the number of programs that are refused by the semantic checker we needed to make sure that we retain the expressiveness of the Javali language and do not restrict the programmer in the ability to write complex programs.
To ensure this, we modified the set of Javali programs from [4] to make them work under the additional constraints our type system imposes. Out of the 23 programs, 11 programs had to be modified to make use of constructors. Only 3 out of 23 needed additional `possibly-null` modifiers (`non-null` is default) and in one program a non-null binary check had to be added. 9 of 23 programs ran without any modification. This shows that a non-null type system can be helpful, with only very few changes to existing code that has been designed without such a system in mind.

Performance results. The Javali reference compiler inserts a null check for each method call, field access and array access, which compares the receiver, instance of the field's class or array reference respectively against null at run time. These checks are implemented as a call to a static method that is part of every Javali binary. The method is shown in Listing 9.

```
# Javali$CheckNull function___________________________
Javali$CheckNull:
enter $8, $0
and $-16, %esp
sub $16, %esp
cmpl $0, 8(%ebp)
jne label3
movl $4, 0(%esp)
call exit
label3:
leave
ret
```

Listing 9. Javali non-null check of the reference compiler

Apart from the additional safety, the non null type system also has the advantage that Javali!? can completely eliminate these automatically inserted checks. Only very few programmer inserted checks are needed as shown in the previous paragraph.

Since method calls, field access and array accesses are very common in Javali, we expect a significant increase in end-to-end performance of Javali programs.

We measured end-to-end run time with a subset of 10 out of the 23 more complex Javali programs. The subset was chosen out of the programs that worked without modifications to the source files on the reference compiler and had a significant run time. We compare them against the original version of these programs which were compiled using the Javali reference compiler (also called baseline). We also analyzed the output to make sure they produce the same results. Each measurement got repeated 5 times on each compiler. The results are drawn in Figure 2. In 6 out of 10 programs, the decrease in end-to-end execution time is larger than 25%. The remaining 4 speed up as well but less significant.

On average, Javali!? reduced the execution time by about 32% compared to the reference compiler.

When taking a look at the actual source code of BubbleSort one can see that most of the execution is spend in a long running while loop (see Listing 10), which consists of mainly array and field accesses. As mentioned before, they generate implicit non null checks and therefore a significant amount of the instructions of the loop are part of the non null check. We annotated the source code below with the actually inserted null checks by looking at the produced assembler file. As an example, `tmp = a[i].a` results in 3 non null checks. The left side of the expression needs none since tmp is a local variable. The right hand side first checks the nullness of `this` (since `a[]` is a field), then the nullness of the array `this.a` for the array index access, and finally the nullness of the object `a[i]` for the field access on `a`. With the non-null type system no non-null checks are needed.

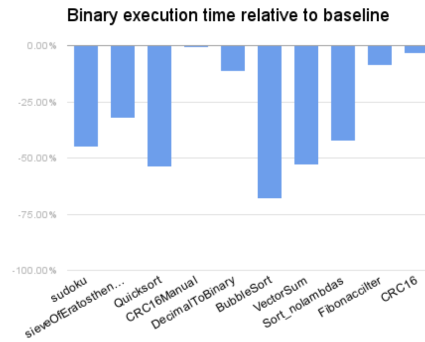

Fig. 2. Execution time comparison of 10 Javali programs

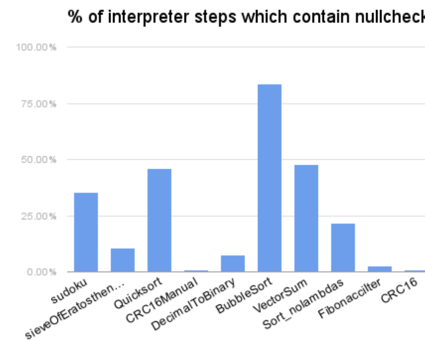

Fig. 3. Percentage of interpreter steps with non null check

```
while (i < SIZE) {
  if (this.a[i].a < this.a[i-1].a) {
    tmp = this.a[i].a; // 3 non-null checks
    this.a[i].a = this.a[i-1].a; // 6 non-null checks
    this.a[i-1].a = tmp; // 2 non-null checks
    changed = true;
  }
  i = i + 1;
}
```

Listing 10. Main loop of BubbleSort

Influence of non-null checks. To get a better impression on the influence of non-null checks we also ran a different experiment with the interpreter. As mentioned before, Javali!?'s interpreter counts the number of null checks that would be executed in case no non-null type system is used. For each step of the interpreter we counted whether or not it needs a non-null check. For the 10 analyzed programs the average percentage of steps including a null-check is 25.74%. The detailed results are shown in Figure 3. The graph shows a similar trend as the one obtained for the performance measurements. This confirms our hypothesis, that

the removal of the non-null checks positively affected the performance of the whole program. It shows that programs with a larger speedup have a higher percentage of interpreter steps that contain non-null checks and vice versa.

4. CONCLUSIONS

This paper presented an approach to add a non-null type system to the Javali language and described a working implementation on top of the existing Javali compiler. The type system adds `non-null` and `possibly-null` annoations for all reference types including arrays. To solve issues arising from object construction and initialization Javali got extended with constructors. The presented data-flow analysis allows the programmer to use the system intuitively and efficiently.
Furthermore a detailed evaluation showed that the non-null type system requires few annotations which greatly help the programmer to avoid issues from null pointer accesses.
Since Javali's automatically inserted run time non-null checks can be removed, Javali!? reduces the total execution time on a set of example programs by 31.94% on average.

5. FURTHER WORK

The evaluation showed that the system is already powerful and does not prevent us from expressing more complicated programs. However, there are certain limitations on which the system could be improved.
Constructors. Javali!? does not allow the `this` reference to be leaked out of the constructor. This forbids method calls with a `this` receiver, `this` as a method argument or defining a foreign field with `this`. One solution to overcome these limitations was proposed by Alexander Summers and Peter Müller [12] and consists of an additional, sound type system to capture the effect of object construction. However, this comes at the price of additional programmer annotations and a more complex language semantics and compiler.
Improved analysis. The compiler could employ a global analysis to get more accurate results for nullness of fields and arrays after method calls.
A points-to analysis would also improve the analysis when objects are aliased and reduces the amount of information lost on assignments.
Non-null checks. The propagation of information of non-null checks is rather simple. Currently it can only handle ==, `!=`, `&&` and `||`, and for example does not handle the unary not (`!`). The non-null checks also only work on local variables, fields and simple arrays. Complex array indices, e.g. with nested arrays and field accesses, would require a more complex analysis which was considered out of the scope for this work.

6. REFERENCES

[1] Laboratory for Software Technology, "Javali short summary," https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Spring2016/2810_Advanced_Compiler_Design/Homework/javali.pdf.

[2] Tony Hoare, "Null references: The billion dollar mistake," https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare.

[3] Patrice Chalin and Perry R James, "Non-null references by default in java: Alleviating the nullity annotation burden," in *ECOOP 2007–Object-Oriented Programming*, pp. 227–247. Springer, 2007.

[4] Laboratory for Software Technology, "Course: Advanced compiler design," http://www.lst.inf.ethz.ch/education/compiler-design.html.

[5] Manuel Fähndrich and K Rustan M Leino, "Declaring and checking non-null types in an object-oriented language," in *ACM SIGPLAN Notices*. ACM, 2003, vol. 38, pp. 302–312.

[6] Mike Barnett, K Rustan M Leino, and Wolfram Schulte, "The spec# programming system: An overview," in *Construction and analysis of safe, secure, and interoperable smart devices*, pp. 49–69. Springer, 2004.

[7] Mike Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs, K Rustan M Leino, Wolfram Schulte, and Herman Venter, "The spec# programming system: Challenges and directions," in *Verified Software: Theories, Tools, Experiments*, pp. 144–152. Springer, 2005.

[8] Mozilla, "Rust, frequently asked questions," https://www.rust-lang.org/faq.html#lifetimes.

[9] Apple, "Swift programming language," https://developer.apple.com/swift/.

[10] JetBrains, "Kotlin programming language," https://kotlinlang.org/.

[11] JUnit, "Testing framework," http://junit.org/.

[12] Alexander J Summers and Peter Mueller, "Freedom before commitment: a lightweight type system for object initialisation," in *ACM SIGPLAN Notices*. ACM, 2011, vol. 46, pp. 1013–1032.