Using an incomplete intermediate representation to perform static analysis

Quentin Jaquier

School of Computer and Communication Sciences

A thesis submitted for the degree of Master of Computer Science at
École polytechnique fédérale de Lausanne

March 2019

Supervisor
Prof. Viktor Kuncak
EPFL / LARA

Company Supervisor
Dinesh Bolkensteyn
SonarSource

Abstract

TOOD: get from gdoc Bug finding using static analysis is a complex process that usually targets a single programming language. The task of generalizing such static analyzer tool to other programming languages is nontrivial and expensive in industrial settings. Therefore, static analysis companies typically minimize this effort by targeting multiple programming languages in one single intermediate representation. However, the main technical problem is to support and approximate different language features and paradigms in the same representation.

In this thesis, we study *SLang*, an intermediate representation defined by SonarSource, used as the input of a static analyzer for Kotlin, Scala, Ruby and Apex. The key novel ideas are the representation of unknown expression by natives elements in *SLang*, and the notion of unreliable basic block for dataflow analysis. In particular, we compare the results of an implementation of a *null* pointer dereference checker over *SLang* and an implementation on the original language. Our experiments show that we are not only able to find interesting issues, but also reach the same precision as the implementation on the original language. Our experiments show that the two implementation reach the same level of precision.

Contents

Listings

List of Figures

List of Tables

1 Introduction

SonarSource is a company that develops static analysis tool for more than 10 years, with the time, the team has developed a good expertise for static analysis. Static code analysis is the action of automatically analyzing the behavior of a program without actually executing it. This kind of analysis is particularly useful to identify potential issues as early as possible, reducing the effort needed to fix them. A year ago, SonarSource was supporting more than 20 languages, but they realized that they were not supporting some that were the most used by the community, and that a frequent request from user is to know when their beloved language will be supported for analysis. In 2018, SonarSource decided to respond to this demand and add the support for 5 new languages that they were not supporting: Go, Kotlin, Ruby, Scala and Apex.

1.1 Supporting 5 new languages

Supporting 5 new languages was a challenging objective since adding a new language to the list used to take month of work, the team had to question their whole process to tackle this challenge. Historically, the typical process to develop a new static analysis tool at SonarSource was to build a front-end, specifically a lexer and a parser. Then comes the core of the work: implementing the different checks, the metrics, copy-paste detection and syntax highlighting. The main content of the work is done, but it still needs to be regularly maintained to stay up to date. The complexity of the current situation is therefore a multiplication between the number of language, and the number of rule. As the objective is to increase the number of language, the current situation does not scale. The first observation that they made is that implementing the front-end for a language is a hard task. This is more or less implementing the front end of a compiler, doing it right and following the evolution of the language is not a trivial task. Hopefully, there exist open-source project that already provide complete and maintained parsing that we can use. This is the first important choice: SonarSource is not going to develop their own front end anymore, but re-use existing one. A second observation is that many rules are implemented the same way, and that some of them are common, they make sense for every programming language.

Unused local variables should be removed
Class names should comply with a naming convention
Credentials should not be hard-coded
Functions should not have too many parameters

Table 1: Example of the common rules list

Table 1 shows a sample of typical checks that can be considered as common rules [3] , that apply to any programming language. This was the main motivation: avoid redoing the same work again and again, by finding a way to implements these rules only once, changing the multiplication by an addition. At this point, the high level idea is to use an existing front-end to perform the parsing, to translate it to an universal intermediate representation and to implements the checks and metrics on it. This idea is promising, it would enable SonarSource to support new languages faster, avoiding duplication, and to reduce the maintainability cost, allowing them to reach their objective. After a few trial and error, the team comes up with SonarLanguage, or *SLang*, an incomplete universal intermediate representation.

1.2 Incomplete Universal Intermediate Representation: *SLang*

In order to be able to implement the rules only once, SonarSource has introduced an incomplete universal intermediate representation, a domain specific language for static analysis. The goal is to have a unified representation of common programming language, for easy, scalable and maintainable code smell and bug detection. The language is designed to implement the common rules introduced before, it is therefore not designed for mainstream programming, and in fact, the goal is not even to be able to compile it. It contains all the metadata and abstract syntax tree nodes that we need to support these rules, and only the one needed, and is therefore a balance between complexity (number of different feature that we support) and accuracy to be able to still report interesting issues.

Appendix A shows the grammar of the current version of the language. This grammar is not fixed, it is made to change and adapt to suits the needs that arise. We can see that it contains all the typical nodes of any programming language. The different nodes approximates the different programming concepts, to be able to support multiple input languages, but we do not need the translation to be faithful, as a transformation of source code requires for example [8]. For example, the loops are all mapped to one single node, with one child that represents the condition, and another for the body. Even if we still keep the original type of the loop, this procedure can still mutilate the input, reducing the three part of a for loop header into one condition, losing information but as long as the result of the checks are not affected, it is not a problem. One interesting note is that there is some important nodes that are not present, for example, there is no function invocation. The reason is that none of the rules use them, we eventually need to know the list of the arguments to report unused variable, but we do not need the concept of function invocation in itself. The specificity of this language is the native node. During the translation, we are going to keep all the original nodes, if one has no equivalent in *SLang*, it is going to be mapped to a native node.

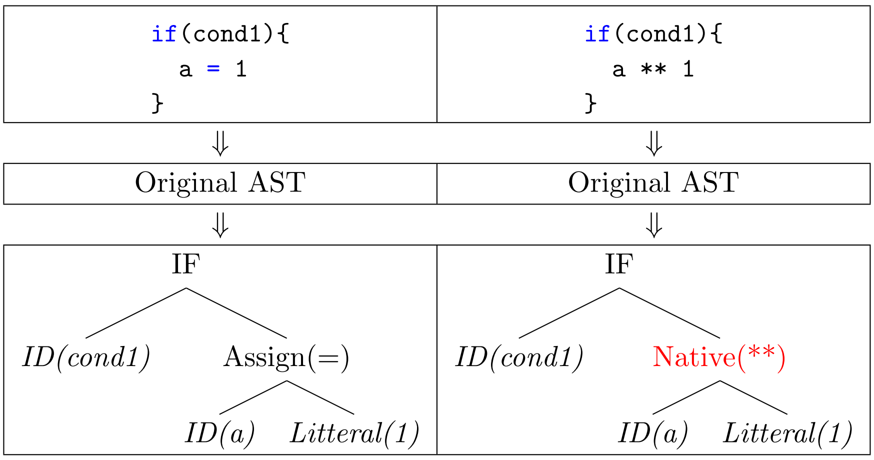


Figure 1: Example of native node in SLang

Figure 1 shows an example of native node present in a *SLang* tree. In the left tree, we understand that the equals is part of an assignment, but in the second case, we have an unknown expression, and keep it, but as a native node. Native nodes therefore represent nodes that are unknown, but we will still be able to compare them and have the list of their children and tokens. Since we can compare two native nodes, we are still able to find that two branches of a switch is the same for example, without knowing exactly what is inside. The other interesting point is that we now control the shape of the tree, we know what to expect, reducing the potential problematic situations. The native nodes also enables to process incrementally: we can implement the translation only for a few nodes, letting the others as native, and already be able to run some of the checks and see the first results.

If the *Why?* can be easily understood, we will discuss the *How?* in this work. To better understand the challenge of implementing a new language, we will start by describing the process of adding a new language to the ecosystem (Section 2), with the challenges and choices that need to be done (Subsection 2.2.2). We will then verify whether the results of a CFG-based check on *SLang* are comparable in quality (based on number of true positives, false positives, etc.) with the same checker over the original tree (Section 3, 4, 5 and 6). We will then investigate if we can push *SLang* further, to be able to support more complex checks, by trying to anticipate the problems that can arise in the future (Section 7) and compare our checker with other tools (Section 8).

2 Adding a new language to SLang

In this part, we are going to discuss the challenge to add a new language to *SLang*, with a general procedure and a concrete example with Scala.

2.1 General Procedure

The addition of a new language follows a general procedure that we can be described in a high level way. The first step is to choose a front end that we are going to use to perform the parsing of the language.

2.1.1 Front end

To choose a front-end, we have to take into consideration multiples points:

1. *License*
   The tool developed will be open source, we have to use an existing front end that that have a compatible license.

2. *Features*
   Static analysis requires specifics need, that is not necessarily provided by any front-end. For example, we need precise location of tokens to be able to report the issues to the user as precisely as possible. Another example is the comments, that is required for some of the common rule 1, but is typically removed early in a compiler front-end since they are not directly used.

3. *Maintenance*
   The last criteria is the completeness and maintenance. We want a tool that is regularly maintained and that support the eventual new feature of the language that will arise.

2.1.2 Incrementally add new mapping and enable rules

With the front-end, we now have access to the intermediate representation of the original language. We can start to work on enabling more and more rules. The work usually start by looking at which rule we want to implement.

Depending on the rule, we have to add the translation of more node. If the prerequisite for the front-end are respected, the initial effort adding the mapping for a new node is an easy task, we have to identify which node in the original language correspond to the node in *SLang*, and understand its structure to create a new node. Once we have added the nodes needed by the rule, and enabled it, we can then look at the results. This is the critical step, we have to make sure that the rule make sense on this new language, and that the issues reported are relevant.

In many cases, this is were the interesting part is, there is a lot of unexpected problems that can arise at this point. Hopefully, there is multiple way to improve the results. The first one is to setup a parameters of the rule, that will change the behavior. For example, all the rules of naming convention have to be setup with the convention of the language. Sometimes, the rules simply does not apply for the language. For example, if a language does not have a *switch* statement, all the rules related to *switch* will not apply to this language. The most challenging situation arise when the situation is not clear, where the approximation of the translation lead to situations where the rules could apply, but the language specific feature change the behavior of the check. These problematic situations have to be taken care case by case.

The problem of reporting relevant issues without too much noise is common in static analysis, and is often referred as the precision and recall trade-off.

2.1.3 Precision and Recall trade-off

In static analysis, a common challenge is to deal with the precision and recall trade-off. When reporting an issue, we can be in two situations:

1. *False Positive*
   The tool is reporting a non-existing bug.
2. *True Positive*
   The tool is reporting a real bug.

Similarly, we can have false and true negatives, for real issue not reported and non-existing bug not reported, respectively.

Precision is the number of true positives, over the total number of issues reported by the tool (*true positives* + *false positives*). *Recall* is the number of true positive over the number of issues present in the code. Finding good balance is a challenging task, in the first case, the programmer do not want to be surrounded with issues that he does not consider as relevant, it will hide the real issues and discredit the tool. In the other extreme, a checker that never report any issue will never report false positive, but obviously not be useful, and will contain a lot of false negative. There is no clear solution to this trade-off, we are going to target a rate of $< 10\%$ of false positive for our work, but this is an arbitrary choice, for example FindBugs [7] initially targeted a rate of less than 50% of false positive. It mainly depends on the context in which the tool is used, an analysis of the software of an aircraft might want to have a high recall while a user working on a small project would like precise tool. Targeting as little false positives as possible, accepting therefore more false negatives, but still report real issues is an important choice since it will greatly influence our design and implementation choices.

An important note is that we are not in the context of proving the absence of bugs, that our checker is sound, but we want to reduce at best their occurrences by reporting real problems, to be as complete as possible.

2.2 A concrete example: Scala

Scala is particularly interesting as it is the first functional language that is going to be added to *SLang*. Scalameta [12] provide all the features that we need, is widely used by the community and is intended to be used by static analysis tool. It seems to suit perfectly to the requirements for a good front end.

2.2.1 Incrementally mapping Scala to SLang

Now that we have chosen a front end, we can use it to obtain a Scala AST from a Scala file. At this point, we already have enough information to activate a first rule: file should parse. The first step from this AST is to extract comments, and translate the Scalameta token into *SLang* token. With this simple step, we are already able to enable the 4 new rules related to comments(S4663, S1134, S1135, S1451). The second step is to start the translation. As in any compiler phase that perform translation, the base will be a pattern matching on the current node. We will traverse the tree using a top-down approach. The initial step is to map all nodes to native tree, that will represents nodes that we do not know anything about. We still have access to the token of the native nodes, we can therefore already activate the copy paste detection and the different metrics. In addition, all the rules related to the structure of a file can already be enabled: length of line, tabulations, length of file. (S103, S105, S104). With only little effort, we manage to enable 8 rules, and provide a copy/paste detection and metrics. We will continue the effort by adding more and more nodes translations, and activating more and more rules.

Most of the nodes from the Scala AST have a direct equivalent in *SLang*, the translation effort is just to make *SLang* that the meaning of this node in the original language is the one intended in *SLang* and that the metadata is correctly handled. This is the case for package/import declaration, literal, block for example. Surprisingly, more complex nodes such as loop, if tree, pattern match also fall into this category. In fact,all nodes in *SLang* have an equivalent in Scalameta and they all follow the same process: we recursively build the children and eventually make *SLang* they exists, but no additional effort than building the children and grouping the metadata is required to build the node in *SLang*.

2.2.2 Reducing the false positives

SLang is driven by the rules, when we add a new node and enable a new rule, we have to make *SLang* that the current rules make sense. In the case of Scala, some of the features greatly reduce the quality of rules. One quick but naive solution when facing false positive is to map the problematic node to native, to remove the problematic case.

Listing 1: Pattern matching that can cause false positives

```
something match {
  case "a" if(variable) => println("a")
  case "a" => println("a")
  case "b" if(variable) => println("b")
}
```

For example, listing 1 shows a correct pattern matching, but with the current mapping, we only add the pattern *"a"* to the condition of the *match* case, and not the guard (*if(variable)*). This will incorrectly trigger the rule that report identical pattern in a conditional structure. If we map the match case to native, this solve the problem, but introduce false negative for the rules that report identical branch implementation, or other rules related to match tree.

Identifying which node can lead to false positive can be done during the mapping, but sometimes it is hard to have a feeling of where the problems can happen. To identify the potential problematic cases, we can store in all nodes, the original node from which it was created. After the translation, we can compute a mapping, from every original node to the node(s) in *SLang*. This gives us a huge list with all nodes present in Scalameta, that is not yet useful to do identify potential problems. The first observation is that the majority of the nodes are mapped 100% to native nodes. This is not a problem, we know that we do not need all the nodes from the original language to perform our checks. The more interesting cases are the original nodes that are mapped to a *SLang* node and a native node. All the rules that use the nodes that are conditionally translated are subject to false negatives.

DefnDef (1)	FunctionDeclaration(90%); Native(10%)
TermMatch (2)	Matchl(70%); BlockTree (21%); Native(9%)
Defn (3)	VariableDeclaration(64%); Native(36%)
Defn (3)	Native(65%); VariableDeclaration(35%)
TermParam (4)	Native(61%); Parameter(39%);

Table 2: Mapping from a node in Scalameta to the translated node in *SLang*, with the percentage

Table 2 shows the resulting table for Scala if we filter further to only keep the nodes where the partition is relevant. From this table, we can directly identify 4 problematic situations that lead to false positives.

1. *Function with many parameter clauses*

Listing 2: Example of Scala function with many parameter clauses

```scala
def add(i1: Int)(i2: Int): Int = {
  i1 + i2
}
```

Listing 2 shows an example of a Scala function with multiple parameters, that is common in Scala, but not in other languages. Treating all the parameters as belonging to a single list is not entirely correct, however simply mapping the whole function as native is a bit too strong, as all the rules using function declaration will not report any issue. [TODO...]

2. *Match statement with at least one conditional case*
   Listing 1, seen previously, also appear in the list. The solution chosen is [TODO...].

3. *Variable and constant definition if there is some templating*

Listing 3: Example of Scala templating

```
val  (x,y) = (1, 2) // x = 1, y = 2
var z :: zs = List(1, 2, 3) // z = 1, zs = List(2,3)
```

In listing 3 we are going to have false negative for both naming convention of variable, and unused local variable. [TODO...].

4. *Function parameters with default value or modifier.*

Listing 4: Example of Scala function with default value or with a modifier

```
def f1(i: String = "Default") = ...
def f2(implicit j: Int) = ...
```

In listing 4 [TODO...].

3 Improving SLang: Null pointer consistency

SLang has already demonstrated his power to add 4 new language, some of them in less than a month, and to implements more than 40 common rules. However, the language is still young and the current rules involve mainly syntactic element. In this section, we are going to attend to push *SLang* further, to implements more complex checks. To estimates the quality of the results of a checker implemented on *SLang*, we will use a variation *null* pointer consistency rule. We choose this rules because this is a well-known bug and well-studied in static analysis, a lot of different implementations exists with different complexity.

3.1 What is null pointer consistency

Null pointer consistency is the verification that a pointer that is dereferenced is valid and not equals to *null*. Dereferencing a *null* pointer will results at best to an abrupt program termination, and at worst could be used by an attacker, by revealing debugging information or bypassing security logic for example.

3.2 Belief style Null Pointer Checker

The goal is to build a checker that implements a variation of the current check *null pointers should not be dereferenced* [11], implemented on Sonar-Java [14], the tool developed at SonarSource to perform static analysis on Java code. The current implementation of the checker of SonarSource, SonarJava, use a complex symbolic execution engine to report potential *null* pointer exception. Symbolic execution try to estimates all possibles executions path, track the value of variables, and report when a pointer is dereferenced while it can be *null* on one path. One important limitation is that it uses a lot of assumptions to deal with the fact that the possible execution flow quickly explode. If it is possible to come up with good assumptions to report interesting bug, the complexity of the implementation also increase, preventing improvement and therefore to find more bugs [1]. This is one of our motivation, we want to see if it is possible to still find interesting issues with an implementation that is less complex, and based on an incomplete language. Our initial goal is not to find all the issues of this checker, but to show that it is still possible to find interesting issues, even if we implement the check on an incomplete intermediate representation such as *SLang*. The idea of this first checker is to use facts implied by the code, that we call belief [4]. It assumes that the programmer's goal is not to make his code crash. If two contradicting believes are detected, we report an issue. Concretely, we are going to try to detect the use of a pointer P, followed by a check for *null*. The check for *null* can be equals or not equals to *null*, both statements implying that the programmer believes that the pointer p can be *null*.

Listing 5: Typical example that the checker reports

```
1 p.toString();
2 //The programmer believes that p is not null, otherwise it will
    crash.
3 //... More code
4 if(p == null){//p is checked for null, we have a contradiction!
5 //...
6 }
```

Listing 5 demonstrate a typical example that the checker is able to report. From line #1, p is dereferenced without having been checked for *null*, we can imply that the programmer believes that, at this point, the pointer is not *null*, otherwise the program will crash. If later, at line #4, p is checked for *null*, it implies that the programmer thinks that p can in fact be *null*, contradicting the previous belief: we report an issue from this contradiction. To implement this check, we need to have a representation of the control flow of the program, that is typically represented by a control flow graph.

3.2.1 Control Flow Graph

A control flow graph is a directed graph that represents the execution flow of a program, the nodes of the graph are individuals instructions, and the edges represents the control flow. More precisely, there is an edge from a node *N1* to a node *N2*, if and only if the instruction of the node *N2* can be directly executed after the node *N1*.

Basic Block We initially described the nodes as individual instructions, however, we can easily see that many instruction are always executed unconditionally in the same sequence. We can regroup these instructions in the same node that we are going to call basic block, representing the maximum sequence of instruction that are executed unconditionally in sequence. This greatly reduce the number of nodes present in the graph, reducing therefore the complexity of future computation on top of the graph.

3.3 Formal definition of the checker

More formally, the idea is to check that a use of a pointer p post dominates the check of p for *null*, intuitively, we can say that all path arriving to the check of p goes through a use of p, without having been reassigned between the two. To do this, we are going to use a data-flow analysis using the control flow graph previously described.

3.3.1 Data-flow Analysis

The analysis tracks the pointer uses (set of pointer believed to be *non-null*) and flag when the same pointer is checked afterwards. The control flow graph will only be build for the current function being analyzed (intraprocedural), and will not have any access to other functions or others files (inter procedural).

Formally:

$$i_n = o_{p1} \cap o_{p2} \cap ... \cap o_{pk} \tag{1}$$

Where $p1, ..., pk$ are all the predecessors of n, i_n the input set of node n, and o_n the output set.

$$o_n = gen(n) \cup (i_n \setminus kill(n)) \tag{2}$$

Where

$$gen(n) = \text{pointer that is used in node n} \tag{3}$$

$$kill(n) = \text{assignment of pointer in node n} \tag{4}$$

Intuitively, we can see the analysis as follows:

1. The set of believed to be *non-null* pointer split at fork.

2. On join, we take the intersection of incoming path, this means that we will remove the ones kill on at least one path. Also called *MUST* analysis.

3.4 Variation of the rule

Analysis type	Number of issues	False Positive [%]
Forward - MUST	32	0
Forward - MAY	2500	> 90
Backward - MUST	65	80

Table 3: Number of issues per type of analysis, with the source, setup described in section 6.1

The version described before shows one way of doing the analysis, there is multiple small variation that we can do on the analysis that will greatly influence the results.

3.4.1 May vs Must analysis

With a MAY analysis, the computation of the input set from equation 1 becomes:

$$i_n = o_{p1} \cup o_{p2} \cup ... \cup o_{pk} \tag{1}$$

If a *MUST* analysis takes the intersection of all incoming path, the *MAY* analysis takes the union of the paths. It means that a pointer will be removed from the set only if all path re-assign this variable. The choice of *MUST* over *MAY* goes in the sense of the idea to have as little false positives as possible described in Precision and Recall trade off. Table 3 shows the differences between a May and a Must analysis of the checker ran on the same sets of sources. We can see that we have significantly more issues, but the rates of false positive is significantly higher, finding interesting issues is too hard with so much noise. In addition, another downside of *MAY* analysis is that identifying true positive can be tricky, involving specific path executed that will raise an exception, while discovering false positive can be within a few seconds. In practice, to help the user to better understand the issue, we could report multiple location, for example the line where the pointer is used, and the one where it is dereferenced.

Intuitively, this is not surprising, a *MAY* analysis means that it requires only one path that uses the pointer that is then checked to report an issues.

Listing 6: Example of false positive of MAY anaylsis

```
1 if(p != null){
2   p.toString()
3 }
4 
5 (p == null)
```

Listing 6 shows an example of a false positive that is reported by the *MAY* analysis. This is obviously an unfeasible path, the pointer p at line #2 is only used if it is not *null*, the check for *null* later does not mean that there is a potential exception. We will discuss possible amelioration to this situation in subsection 8.1.5.

3.4.2 Used then check, check then used

Listing 7: Pointer that is used then checked

```
p.toString();
if(p == null) {}
```

Listing 8: Pointer that is checked then used

```
if(p == null) {}
p.toString();
```

Listing 7 and 8 shows the differences between the two version of a checker. The work presented before implements the former, however, the latter makes as much sense, if all path that follow the check for *null* uses the pointer p, without re-assigning it, it probably means that an error is possible. In the implementation, this would be implemented using a backward analysis. As the name suggest, a backward analysis means that we take the intersection of all successor's input set to determine the output set of the current node.

For a backward analysis, equation 1 becomes:

$$o_n = i_{s1} \cap i_{s2} \cap ... \cap i_{sk} \tag{2}$$

Where $s1, ..., sk$ are all the successors of n.

And the computation 2 becomes:

$$i_n = gen(n) \cup (o_n \setminus kill(n)) \tag{3}$$

Surprisingly, the rate of FP is greatly increased, the number of false positive is greater than our goal of $< 5\%$, but the issues are more interesting than the *MAY* analysis, we can still find interesting issues, mainly due to the fact that identifying true positive is as easy as false positives.

Listing 9: User define function that changes the control flow

```
if(p == null) {
  customThrow();
}
p.size();

customThrow() {
  throw new MyException();
}
```

Listing 9 shows the typical example that generate the false positive, we can see that a function is called when p is *null*, that will throw an exception, therefore changing the execution flow order.

Custom function that changes the control flow is a weak point for flow based checker that does not perform inter procedural analysis, and we will probably face this problem both in an original language and in *SLang*. From now, we are only going to work with the used then checked version.

4 Implementation on SonarJava

We are first going to implements this check on the SonarJava ecosystem that already provide us all the tools that we need, specifically symbols resolutions, and a control flow graph. The implementation is a classical forward data-flow analysis: the first step is to generate for each basic block the gen and kill set as described before. We are going to store the symbols of the variable in the two set. We fill the set going top-down in the elements of the basic block, when a pointer is killed, we also remove it if it is present in the gen set. With this, a pointer that is used and assigned in the same basic block will be in gen set only if the use of the pointer follow the assignment, as expected.

Listing 10: Problematic situation with naive basic block creation

```
p.toString();
b = (p == nul);
p = get();
```

Listing 10 shows a potential problem of this method, all the different parts of the code will be added to the same basic block, we will therefore have a situation that we would want to report, but is not detected since the pointer is not in the gen set of this block. One naive solution would be to not aggregate statements in basic block, but we will have to compute the input and output set for every statements!
[TODO: ev explain performance diff] The alternative that we use in this work is done during the control flow graph creation: we break the basic block when we have a binary expression with an equals (or not equals). In order to support the backward and forward analysis, we should break before and after the check for *null*. We can now safely consider that a pointer that is checked will never be in the same basic block as where it is used or killed.

Figure 2 shows the old and new control flow graph for the code of listing 10. By doing this, we will be able to support the examples shown in listing 10, the check for *null* break the block in three, the use and check will therefore not be in the same basic block.

Once the *gen* 3 and *kill* 4 set has been generated for every basic block, we can start to run the analysis with a work list approach. The idea is to add all basic block in a queue, compute the new out set of the current

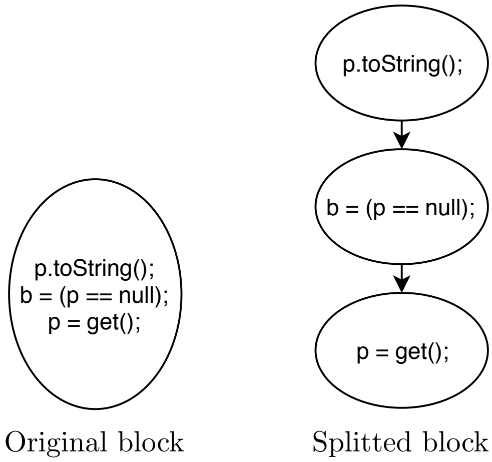


Figure 2: Example of the new way to split basic block

head. Since we are performing a forward analysis, if the new out set have changed, this means that all the successor might potentially change as well. We therefore add the current basic block and all of it successor at the end of the work list. We continue this process until the list is empty.

Finally, after having reached a fixed point, we have a set of pointer that are believed to not be *null* in each basic block. We can therefore do a second pass through the elements of the basics block, if we see a check for *null* (implies that this pointer could be *null*) in a block where the pointer is in the set of believed to not be *null*, we have a contradiction and report an issue.

4.1 Other way to add belief

Null pointer exception does not occurs only when a *null* pointer is dereferenced, but can also appear in the following cases for java, as defined in the Java documentation [10]:

1. Calling the instance method of a *null* object.

2. Accessing or modifying the field of a *null* object.

3. Taking the length of *null* as if it were an array.

4. Accessing or modifying the slots of *null* as if it were an array.

5. Throwing *null* as if it were a throwable value.

Currently, our checker is only using the first case, but we can use this information to improve our implementation: when we see one of these construct, we will add the pointer to the set of believed to be *non-null* (*gen* set) the same way as we would for a pointer that is used.

5 Implementation on SLang

In the implementation on SonarJava, we had access to the front end of the checker, with complete language, a control flow graph and the symbol resolution already present. The first step to implement it on *SLang* is to identify what nodes and structure we need in order to implement everything that is required for this check.

5.1 Nodes

The particularity of *SLang* is that it is incomplete. The IR does not need to includes all node types in order to work, only the one needed for the rules are mapped. We therefore have to make sure that we have all the nodes that we need in *SLang* for the implementation of the checker.

Checker specific	**Needed for CFG**	**Others**
Binary operation	If/else	Variable declaration
Identifier	Switch	Function invocation
Assignment	Exception handling, throw	Function declaration
Litteral : Null	Loops	Class declaration
Member select	Jump (break, continue, . . .)	

Table 4: All nodes needed for the null pointer dereference check

Table 4 shows the nodes that we need in order to implement the different parts of the checker. The first column lists the nodes needed to recognize the different structures used in the checker. We can see that the list is quite simple. The interesting node is the member select, in fact, to identify when a pointer is used, we will only use this node: we don't need to know anything about the context in which the pointer is used. For example, when a function is called without a member selection, the tree will simply be a identifier (name of the function) and a list of argument, but in the case of a pointer use, the tree will be a member selection, that we will use in the checker. This way, we are able to detect not only function invocation, but also field selection or anything that we consider as a member selection in the original language.

The nodes needed for the control flow graph are the one that we can expect for identifying the control flow of a program, that are common in all language and already implemented in *SLang*. The way we handle them will be described in subsection 5.2. The last column describes others nodes that are needed indirectly by the checker.

5.1.1 Variable declaration

Listing 11: Example of local scope inside a loop that can shadow field

```
 p.toString();
 while (cond) {
   Object p = getP();
   if(p == null){ } // Compliant
}
```

In section 5.3.1, we are going to describe how we perform a naive semantics, that is assumed inside the rule. In this semantics, we are not going ot be able to differentiate if two pointer that have the same name refer to the same declaration. The idea to better support this limitation is that we will kill the pointer in the analysis when we see a declaration, the same way we are killing it when we see an assignment. This will enable us to remove the kind of false positive of listing 11.

5.1.2 Function invocation

Listing 12: Pointer p is used as a parameter of a function call

```
f(p.size())
if(p == null) {} // Noncompliant
```

As discussed before, we do not need explicitly function invocation, only member selection. Due to a specific way we handle nodes that are not translated that we will explain in section 5.4, we will add function invocation to be able to report pointer use that happens inside a function call, as illustrated in listing 12.

5.1.3 Function and Class declaration

As our checker is only ran inside functions, we need function declaration to have our starting point. We also use this to improve our semantics, using the fact that the variable that is used inside a nested function is not checked.

Class declaration is used for the same idea as the function declaration, we use the assumption that variable declared in nested class are in an other scope.

5.1.4 Other nodes not supported

In 4.1, we saw multiple way to add the belief that a *null* pointer can be raised. Slang do not have arrays, so we could expect to not find all issues coming from them. The first is the length of the array. In Slang, this is simply represented as a field access/member select, and can therefore be supported. Accessing or modifying the slots of *null* as if it were an array is however not supported. This will lead to false negative in *SLang* that we will not have in the implementation of SonarJava.

5.2 Building a Control Flow Graph on SLang

SLang already have every control flow statements represented in the language, we can already start to build it the same way we would do it for any other language. In fact, the current implementation is greatly inspired by the implementation of the graph builder of SonarPHP [15], also developed at SonarSource. To build the control flow graph, we are going to use two main kind of basic blocks:

1. *CFG Block*
   This is the base of all basic block of the graph, it contains the following fields:
   (a) *Predecessors*
       Nodes that may be executed **before** the current block.
   (b) *Successors*
       Nodes that may be executed **after** the current block.
   (c) *Elements*
       List of instruction that are executed one after the other in this basic block.

(d) *Syntactic Successor*
Node following the current block if no jump is applied. This is not directly needed for any control flow graph, but it is sometimes required by some check that we may implement in the future.

2. *CFG Branching Block*
This interface represents blocks that include branching instruction, where the flow depend on the result of a Boolean expression. It inherits from CFG Block and simply have a true and false successor block reference in addition to the simple block.

Since we are going to build a graph for the content of a function, our starting point will be the list of the elements of the function. We are going to start to build the graph from the end of the execution, using a bottom-up approach. It enables us to always know the successor of the node that we are currently building, making easier to build the different instructions that contains control flow. We start by creating an END node, that contains no element and represents the end of the execution. We will then recursively build the graph by matching on the type of the tree.

5.2.1 Block and others nodes

The simplest tree that we will face are the blocks, they represent simply a list of statements, we can therefore directly recursively build the graph for all children. This behavior can also be applied to other known trees that does not change the flow of execution, as a default case. The only difference is that we are also going to add the current tree to the elements after having built the graph for the children, in order to not lose information. For example, having a list of identifier is not useful if we do not know that they are linked together by a binary expression, we will therefore add both the binary expression and the children to the elements of the current block.

5.2.2 If/Then/Else

This is the typical example that implements a branching block. We will first build the sub flow for the false and true branch, if present, and then construct a branching block with these two new blocks as successor. We can now recursively build the condition of the If tree from the branching block created before.

5.2.3 Loops: For, While, Do-While

The bottom-up approach makes the creation of the If tree straightforward, since we have already built the successor of the tree that we are currently building. However in the case of loops, the flow is not going to continue at the successor, but return at the condition of the loop, a predecessor's node that we have not built yet. To solve this issue, we can introduce a **forwarding block**, a simple basic block that is used to store a reference and will not contain any element. We can now start to build the loop flow by creating a forwarding block that will link to the condition, and build the body with this new block as the successor. Finally, we can build the condition of the loop as a branching block, with the true successor as the body of the loop, and the false as the block that follow the loop. There is one details that we have not address yet: break and continue. To support these two statements, we are going to use a stack, that will contain breakable objects. These objects are simply here to store the link to the condition for continue, and the end of the loop for break. Before starting to build the body of the loop, we will push a breakable to the top of the stack, and pop it once we are done. The stack is used to support nested loop, a break/continue will refers to the first enclosing loop.

The current implementation of For loops is the same as While loops, this may introduce problems that will be discussed later. We can use the same idea for Do/While loop, and simply start to build the condition before the body.

5.2.4 Match Tree

The particularity of a match tree is that it can behave differently depending on the original language. For example in Scala, only one match case can be executed, while in java, all cases are executed after a matching pattern, until a break. The second example is typically known as fallthrough. In Slang, both of them are mapped to the same node, however, identifying which one is the right behavior can be done by storing a flag in the node. Nonfallthrough match tree is an easy case, we can build all the cases separately, and create a block that will have multiples successors.

Fallthrough switch is more tricky, we first have to use the same idea as we used for loops: add a breakable to the stack, to store the reference to the block that is executed after the switch. We make the same assumption as we did with loop, that a break refers only to the closest enclosing switch.

We are going to start by creating a forwarding block for the default case, and build the different cases, in the reverse order and one after the other. We create one branching block per match cases, with the body of the case as true successor and the next pattern as the false successor. At the same time, we build the sequence of the body of cases, starting from a break.

Listing 13: Example of pattern matching in Scala

```
x = 0;
match(cond) {
 case pat1:
   a = 1;
 case pat2:
   b = 2;
   break;
 case pat3:
   c = 3;
}
d = 4;
```

Figure 3 shows the resulting control flow graph of the code of listing 13. We can see that the fallthrough behavior is represented as expected, *a = 1* is executed both when *pat1* and *pat2* is true.

5.2.5 Jump Tree: break and continue

Jump tree are not supposed to appear when we do not expect them, if we have an unexpected jump tree, we can not do anything and we will simply add it to the current block. We will discuss in section 5.4 a solution to better support this situation. In the usual case, we will expect them and create an edge from the current block to the head of the stack that is filled as described in 5.2.3 and 5.2.4 sections.

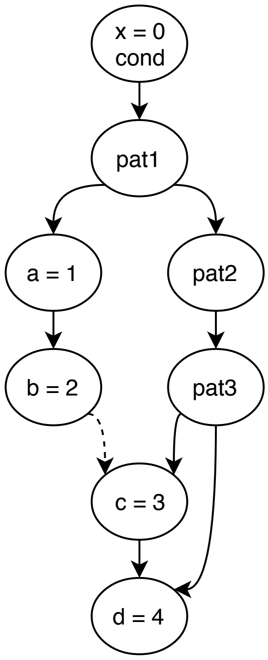


Figure 3: Corresponding CFG of the code of listing 13

5.2.6 Return

Once again, starting from the end greatly simplify the support of return statements: we can store a reference to the *END* block that is created at the beginning, use it as successor to the block that will contain the return expression.

5.2.7 Exception Handling Tree and Throw Tree

In our control flow graph, we are only going to consider exception that are explicitly thrown with a throw statements, and not add an edge to every statements where an exception can occur in reality.
We are going to start at the end of the exception handling tree. When an exception handling tree is executed, it can results in two possible cases: the exception is catched and the flow can continue, or it is not and the flow goes to the end of the function. To support this behavior, we are going to create a block with two successor: the *END* node recovered the same way we did for return statements and the successor previously created.
We can then create the finally block, if present, and the different catch case

separately, and continue to build the body of the try block. At this point we have to know where to jump in the case where an exception is thrown. To do this, we will use the same idea as we did with the jump trees: use a stack to push the target of the throw before building the body, and popping it after. If there is no catch block, the target will be the finally block, if there is one or more catch block, we will use the first catch case as target. This is an approximation that is due to the fact that we have no symbol resolution, we can not know which of the exception is catched or not. From now, we can know where to jump in the case of a throw three.
The last detail to take care of is the case where we have a return inside an exception handling tree. In this case, the finally block is executed after the return. To support this, we will store the exit target on a stack, pushing the reference to the finally block on top of the *END* block previously added.

5.2.8 Natives Nodes

The main challenge comes from the only new nodes that we have compared to any language: the *natives nodes*. The way we deal with native nodes will be described in subsection 5.4.

5.2.9 Normalization

The core of the graph is done, but we still need to perform a few modification in order to have a proper control flow graph. First, we are going to remove empty blocks. They can be introduced in multiple situations, when we create a temporary forwarding block or when the header of a for loop is empty for example. During the creation of the graph, we only knew the successors of the nodes, we still have to compute the predecessor set. Since we have all successors, computing the predecessor of all the block is straightforward. Finally, we can create a *START* node, that implements the same behavior as the *END* node, to indicate the beginning of the flow.
The hard work is done, we now have a complete control flow graph. There is still part of it that can be imprecise due to the fact that the different statements of the original language can behave differently, but we are going to discuss it later in section 5.4.

5.3 Data flow analysis

Now that we have all nodes needed and a control flow graph, we can start the implementation of the checker, that is in fact really similar to the one described in 4. The main difference is the the way we deal with "unreliable" nodes and the identification of local variable. The former will be described in 5.4 and the latter in the next section.

5.3.1 Identifying local variable

In SonarJava, we have access to symbols of identifier, information that we do not have in *SLang*. The current computation of local variable is quite simple: all variable declaration inside the function and all arguments are considered as local variables. This idea can work well for Java, but it not well suited for all language. In particularity for *Ruby*: there is no variable declaration in the language, running the checker as it is will lead to a lot of potential false negative since we will only report issues for parameters. This is a naive version, used to show that with a proper semantics (name definitions and scoping rules) we could expect results that are as good as the current naive version.
When we have this set of local variable, we can now check if the variable is in this set before reporting the issues. In practice, we could still report the issues that are not coming from local variable, this would add some false positive.

Listing 14: Field can change value during a function call

```
String s = "";

void foo() {
  s.toString();
  changeS(); // An other method can change the value of s!
  if(s == null) { } // Compliant, s changed
}

void changeS() {
  s = null;
}
```

Listing 14 shows an example of a false positive due to a function with side-effect that change the value of a field. Adding the issues that comes from non-local variable double the number of issues found, but the majority of them are false positives. Since a variable can be reassigned between the use and the check of a pointer p, these new issues does not exactly respect the original description.

5.4 How to deal with native nodes in a CFG based checker?

It is finally time to explain how we are going to deal with the natives nodes. For our concern, we will see the natives nodes as a nodes that we do not know anything about, with a list of children.

Listing 15: Pseudo code with a ternary expression

```
true ? b : p.toString();
p == null; // Compliant
```

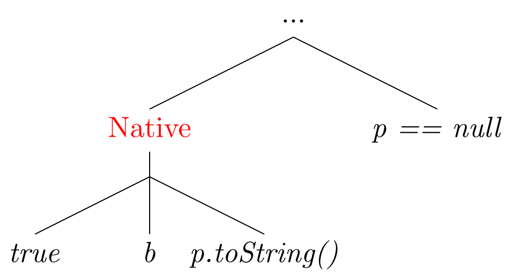


Figure 4: *SLang* AST from the code of listing 15

Figure 4 shows the results of the *SLang* tree created from the code from listing 15. In this example, we assume that we do not have ternary expression in *SLang*, and that they are not mapped to *if then else* statement. We will use ternary expression of Java to represents the problem, but the construction can be any native nodes coming from any original language.
The problem here is that we have to represents the control flow of a node that we do not know anything about. We can not trust the evaluation order of the children of the natives nodes, as it can be arbitrary. The first question that arise is why do we have to keep the content of a node that we do not know anything about? In fact, this is the root of the idea of the native nodes, we are not interested in the node itself, but only on the content.

Figure 5 shows an typical example: in this case, we do not need to know

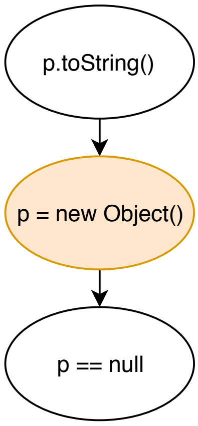


Figure 5: CFG with an assignment in a native node

what the native node (orange node) is exactly, but that the node assign p. We do not care what exactly happen in this native node, we just need to know that, at one point, p is assigned, even if in fact it is possible that the assignment is never executed. If it is the case, this will add false negative, but intuitively, we can assume that dead code is not common and this will not happen very often.

So, at this point, we know that we need to keep the content of the native nodes. The next step is to define what to do with them. A naive solution would be to add the content of the graph in the elements of the basic block, with the assumption that the evaluation order is not important. This is in fact correct for a native node with only one children, where the evaluation order can obviously not change. Here, we make the assumption that all the statements that change the flow of a program are represented in *SLang*. This is a reasonable assumption since programming language hardly ever provides exceptional statement that are break the control flow, and if it does, we can add it to *SLang* grammar.

Listing 16: Pseudo code with a ternary expression

```
K = 1;
A ? B : C;
P == null;
```

Listing 16 shows the pseudo code of a Java program with the control flow graph of the naive implementation in figure 6. Since the ternary expression will be mapped to a native node in *SLang*, if we take the children of the

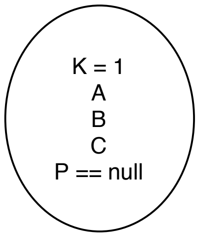
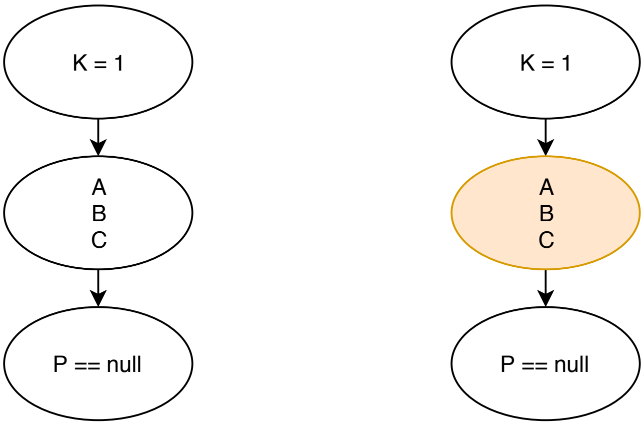


Figure 6: Basic block content of the code in listing above

native node in order, we will obtain the execution order of of the nodes in figure 6, that is obviously not correct, as the pointer will be seen as used then checked in this order, but not in the real execution order.



Splitted CFG

Splitted block with unreliable node

Figure 7: CFG with elements coming from natives nodes

The idea to solve the problem showed before is to put all elements that comes from a native node in a separate basic block (figure 7, left), and mark it as *unreliable* (figure 7, right), shown in orange. All control flow statement nested inside the native nodes will also lead to unreliable basic block. [Ev: show nested control flow nodes that are also unreliable]. Additionally, we will also mark the whole graph as unreliable.

This information can now be used by any checker that use a control flow graph, not only for the *null* pointer dereference checker.

5.4.1 How to use this information?

This information can be used in different ways to help to define the multiple level of granularity of the implementation of a new checker:

1. *Ignore this information*
   In some case, it may make sense to simply ignore this information, and to treat unreliable nodes as others. In our case, we have shown previously that this solution is not suitable, as it produces too many false positives.

2. *Don't run the checker on unreliable CFG*
   This is the opposite of the previous point: in the case where the checker need to really trust the control flow graph, it can make sense to simply stop the checker if the graph can not be trusted.

Language	**%**	**% of completely native**	**Number of files**
Scala	41	6.25	6126
Kotlin	47	6.5	26758
Ruby	39	5.2	7811

Table 5: Percentage of native and completely native nodes in the different languages

   Table 5 show the percentage of native nodes in *SLang* after translating open-source projects [13] to *SLang*. The percentage of completely native nodes refers to the nodes that have all their children that are native. If more than 40% of nodes are not translated, this does not mean that our language lack of information, because we do not mapped some kind of nodes on purpose. This table shows us that natives nodes are not rare, using the above approach will greatly reduce our chance to find any relevant issue as we will in the majority of the case, be in the presence of natives nodes in the body of a function.

   The two approach described before seems not well-suited for our checker, we may want something between the two extremes.

3. *Fine grain*
   We can use the fact that we know if an element comes from a native node or not to define a finer grain implementation of our checker. It

consist mainly in one modification of the data flow analysis described previously: we do not add a pointer that is used in an unreliable block to believed to be *non-null* set.

$$newGen(n) = \text{use of pointer in the node n except if the node is marked as unreliable} \tag{1}$$

Listing 17: First example of finer grain behaviour

```
true ? B : p.toString();
p == null; // Compliant
```

Listing 18: Second example of finer grain behaviour

```
b ? p.toString() : (p == null); // Compliant
```

With this idea, we are now avoiding to report any issue for the two correct pseudo code from listing 17 and 18. Since ternary expression are unreliable, we will not consider p as used, even though it may look like in the control flow graph.

Listing 19: Third example of finer grain behaviour

```
p.isEmpty() ? "" : (p == null); // Compliant
```

In listing 19 though, the real evaluation order use p and then check it for *null*. This code is not reported by our tool despite the fact that it should be. This is a false negative.

Listing 20: Fourth example of finer grain behaviour

```
p.toString;
(p == null) ? "" : "null"; // Noncompliant
```

In addition, the implementation will still report issues inside native nodes. For example, in listing 20, we can see that the pointer p is used, and then checked for *null* later in a un-trusted node. We do not know exactly what exactly happen in this node, but we can expect that a check for *null* still mean that p can be *null*. In this example, we report a true positive.

5.5 Other problematic situations

We have already presented the main problematic situations, coming form native nodes, but there is still a few cases that can raise false positive that we have to take care.

5.5.1 Boolean short-circuit

Our current implementation of the control flow graph does not encode the possible path due to Boolean short-circuit. This will in fact lead to a wrong evaluation order, even if the nodes are known. Since the evaluation order is not correct, it makes sense to treat them the same way we do with native nodes! We will hence keep all the content of these nodes, to be able to use them in the checkers, but mark them as unreliable.

Listing 21: Problematic situation due to Boolean short circuit

```
if(p == null || p.isEmpty())
p = a.get(1) == null || p.toString()
```

With this addition, we are now able to avoid the false negative that were reported in the well-known example of listing 21.

5.5.2 Order of evaluation of known nodes

In section 5.4, we have seen that the order of evaluation of the nodes are critical for our checker. When coming from different languages, even known nodes can have different evaluation order, as we have seen an example in section 5.2.4. The same situation can in fact arise for any statement, hopefully, the solution is often to add the support for the new behavior in *SLang*. This trick should be used with caution, the initial goal is to be language agnostic, we should ideally not have to modify *SLang* for every new language we add. The good news is that there is only a limited number of possible variations that are possible, since the number of known nodes is fixed (and only a faction of it in practice). This kind of problem is a difficulty that are hard to anticipate and will arise during the implementation of a new checker, but it is not in itself a strong limitation.

5.5.3 Lost Jump Statements

We have seen in section 5.2.5 that we add the jump statement to the basic block if we do not expect it, without doing anything special. In fact, we face a similar uncertainties that happen with natives nodes, we do not know exactly what is happening, but we can expect that something will go wrong. The same happen with jump tree with label, as the way the different language deals with label can be arbitrary, we can not assume anything. We just know that the statement will change the execution flow in an unreliable way, we will therefore mark the flow generated as *unreliable*.

6 Experimental evaluation: Running the checker on open source Java projects

In this section, we are going to compare the implementation of the checker on *SLang*(5) with the implementation on SonarJava (4), on a set of open source Java projects. It is a good source of data since it enables us to test the result on real situation.

6.1 Experimental Setup

To test the checker, we are going to create a SonarQube instance [16], with the version of the checker that we want to test. We are going to run the analysis with the plugin containing the implementation of our checker on more than 100 open source projects and publish the results on the SonarQube instance. Table 8 shows a sample of the projects list, and the complete list online [9].

6.2 Early results

The checker has been run on more than one hundred of project of various size, containing for instance OpenJDK, SonarJava and the *SLang* project itself. The idea is to run the implementation done on SonarJava and *SLang*, and compare the results.

SonarJava issues	Slang issues	%
37	29	78

Table 6: Early number of Issues reported by the two implementation, before improvement

Table 6 shows the number of issues reported by the implementation on SonarJava, and the issues reported by the implementation on *SLang*, with the source, setup described in section 6.1. Despite all our effort to prevent problematic situation done in the previous parts, the implementation is having more than 20% of false negative compared to the implementation on SonarJava. This is already a good start, but it is not enough for our objective set in section 2.1.3. We can wonder what are the reasons of this differences. We will discuss some of them in the following part, to see if this comes from a misbehavior of the implementation, or a real limitation of *SLang*.

6.3 Reducing the false negative from SonarJava

The difference between the two implementations is mainly due to the way ternary expression and loop header is currently handled in *SLang*.

6.3.1 Ternary expression

Ternary expression have been used as example previously, and they actually appear to be causing false positive in real project.

Listing 22: Typical code structure with ternary expression

```
int s = p.isEmpty()  ? "0" : p.length;
// More code ...
(p == null);
```

The situation is not as obvious as the one presented before, listing 22 show a possible situation were no issues will be reported. To solve this problem, one solution is to map ternary expression to if/else tree.This solution is already used for other checks and seems to solve our problem nicely.

6.3.2 Loop Header

Currently, no rules uses the details of the for loop header, it is therefore mapped to a native tree.

Listing 23: Pointer used inside loop header

```
for (int i = 0; i < p.size(); i ++) {
// ...
}
if(p == null) { //... }
```

Listing 23 shows the problematic situation. The pointer *p* is used, not re-assigned, and check for *null* later. It is exactly the kind of situation that we would like to report. However, the different parts of the header are in a native node, as described before, it will therefore not be added to the set of used pointer. This makes sense, from a language agnostic point of view, we can not know anything from the execution order of the different block of the loop header, as it can depends on the original language for example. This is in fact the kind of behavior that we want to achieve, the language specific features does not produce false positive, we only have false positive. One way to solve this problem is to simply add a node in *SLang* that will better support this situation. If it makes sense for loop header as it is probably a feature that is present in different situation, we have to keep in mind that this is not a solution that we should use in all situation, where the feature is really specific to a language.

6.4 Improved results

SonarJava issues	**Slang issues**	**%**
37	37	100

Table 7: Final issues found by the two implementations for Java

With the two modification done on the implementation on *SLang*, with the same source and setup described in section 6.1, we manage to report the same issues that the implementation of SonarJava was reporting!

Project	**Number of issues**
OpenJDK 9	12
ElasticSearch	7
Apache Abdera	5
Apache Tika	4
Ops4j Pax Logging	3
Apache Jackrabbit	2
RestComm Sip Servlets	1
Wildfly Application Server	1
Apache pluto	1
Fabric8 Maven Plugin	1
Total	37

Table 8: Final issues found by the two implementations for Java, with the source and setup described in section 6.1

Table 8 shows the projects that contains one or more issues and number of true positive reported, for both forward and backward analysis. All these issues have been reported by both the SonarJava implementation, and *SLang* with the modification done in Reducing the false negative from sonarJava.

6.4.1 Other languages

The mapping to *SLang* has been implemented for 5 languages: Java, Scala, Kotlin, Ruby, and Apex. Once we have an implementation of a checker that works for one language, the checker can be run out of the box on other languages if we make sure that all node described in subsection 5.1 are present.

[TODO: run on 1000s]

[[The checker seems to work on some sample example, but does not manage to find any real issues on the project that we tested. For Apex, this can be explained by the fact that we have only little open source code available out here. For Scala and Kotlin, the two languages provide a special construct to deal with *null*. Therefore, we can expect less issues on these two language. Comparing the performance of the checker for Scala and Kotlin with other tool is not really possible, since the only check that exists and make sense in these language is simply that you should never use *null*.]]

6.5 Are the issues found really relevant?

Table 8 shows the number of issues found per project. This includes all the true positives of the forward and backward analysis. A first observation is that the issues found are coming from various project and in various situations, it is not one anti-pattern that is repeated multiple times in the same project. Additionally, all the issues seems to be relevant from a high level view and without any specific knowledge of the project, you can not easily justify any of the issues reported. To estimate more reliably this interest, we can also look at the fix rate of the issues.

6.5.1 Fix-rate

Fix-rate is the rate of issues that are reported by a tool, and really fixed by the user. As discussed in Precision and Recall tradeoff, static analysis tools have to deal with the fact that if we report too much issues, we take the risk of reporting irrelevant ones and the user will not pay attention to them. This is where fix-rate may be useful, it shows that the user did really care about the issue, and took some time to fix it.
The problem is obviously that we can not define at a given instant this rate, we can only retroactively look at this number, depending therefore on the time we gives to the user to fix the issues. The goal is not to reach a precise number, but to find examples of issues that are fixed, to improve our confidence in the quality of the results.
The first way we will estimate the fix rate is by using some of our test project that are not updated for every version. In practice, there is only a few, the main one that we will use is the OpenJDK. The issues reported comes from version 9, that we will compare with the version 11.

OpenJDK V.9 issues	**Issues fixed in V.11**	**%**
12	3	25

Table 9: OpenJDK 9 issues fixed in version 11

Table above shows that 25% of the issues found on OpenJDK 9 have been fixed in the version 11. This may seems like a low number, but it seems to be the kind of results we can expect from this kind of estimation. For example, a research from Jetbrains [2] report that 32% of the issues reported by their tool were considered as useful (rated with high value) by the person that were confronted to the issues. We can explain this by the fact that developer have priorities, especially in such big open-source project, fixing a bug that is already here and is apparently not causing any trouble have low priorities, even if this is a legitimate issue. SonarSource often refers to this idea as the "Fix the leak" approach: it does not make sense to spend considerable effort to fix every bug already present in the code if you keep introducing new one on new code, the same way you would not start to mop the floor during a flooding without having fixed the origin.

One other way to estimate the fix-rate is to look into the issues reported by the tool, understand them, eventually write a unit test that raise an exception, and report this issues to let the owner of the project decide if this issue is worth the attention. One of the problem is that sometimes, it is indeed possible to write a unit test that target a specific function and throw a *null* pointer exception, but it will never happen in real execution due to the fact that the programmer have an implicit knowledge about his code, reducing his interest in fixing the code. For example, if a user only calls a function only when he finds a specific element in a list, he will assume that the list will never be *null*. These kind of issues should however not directly be classified as false positive, as it can also report dead code.

6.5.2 Potential Null Pointer Exception or Dead code ?

Listing 24: Example of contradicting code that lead to dead code

```
if (p != null) {
  p.toString();
  if (p == null) { //... }
}
```

Despite the fact that we try to find *null* pointer exception, some of the issues found can be considered as dead code, as they can never raise an exception in practice. For example, in listing 24, we can see that this code will never raise an exception. It comes from the fact that we implies beliefs from the code that a programmer writes, if he writes himself contradicting statement, we will still report an issue. In the situation of listing above, the checker does not take in consideration the check for *null* as a path-sensitive tool would do.
One similar situation is that sometimes, it is indeed possible to write a unit test that target a specific function and throw an exception, but it will never happen in real execution due to the fact that the programmer have an implicit knowledge about his code. For example, if a user only calls a function if he find a specific element in a list, he will assume that the list will never be *null* in this function, and therefore the check is simply dead code. This will however not degrade the quality of the results, this is still raising poor practice and poor code quality, since this will be dead code that can confuse the user.

7 Limitation of the approach: Anticipating other potential problems

In the previous parts, we have shown that we were able, with some assumptions, to find with *SLang* all the issues found by the implementation on SonarJava. This is encouraging, but it does not implies that nothing is lost during with this approach. In this part, we will try to briefly anticipate problems that were not central to this checker, but might come later, by first focusing on Kotlin example, then well known language features.

7.1 Kotlin

Listing 25: Kotlin code that raise a false positive

```
1  fun f() {
2      val a: Any? = null;
3      a.isBooleanOrInt();
4      if(a == null) { }
5  }
6
7  fun Any?.isBooleanOrInt(): Boolean = when(this) {
8      is Boolean, is Int-> true
9      else -> false
10 }
```

Listing 25 shows Kotlin code with an interesting situation. At line #3, we can see that the function *isBooleanOrInt* from the pointer *a* is called without a safe call with *.?*. Normally, the type system of Kotlin prevent this kind of issue if the variable is *@Nullable* (of type *Any?*). However this code will not raise a *null* pointer exception since in fact the function *isBooleanOrInt* is called, without dereferencing the pointer *a*! Our checker is only checking the content of one function, from his point of view this code can raise an exception. This is clearly a false positive that is reported in for Kotlin. In fact, in Kotlin, we do not expect to find any situations where an exception is possible, but only issues that are dead code, as described in section 6.5.2.

7.2 Other languages features

The goal of this section is to presents other potential problems that can arise due to different languages features. We are going to present a high level overview of the feature, and not precisely describe an exhaustive list the consequences and how to deal with these. This list is particularly useful when thinking about new rules that we can add to *SLang*, being aware of them can help to avoid bad surprises.

1. *Call by name or by value*

   Listing 26: Difference between call by name and by value

   ```
   f(p, p.toString)
   f(p1, p2) {
     if(p1 != null) { return p2 + "."} else { return "." }
   }
   ```

   Listing 26 shows pseudo code that demonstrate the potential problematic question. If the language uses call by value, the code will raise a *null* pointer exception, however, if we use a call by name, p2 will not be evaluated and therefore nothing wrong happen.

2. *Functional language*
   In subsection 2.2, we have described the addition of a new language on top of *SLang*. Adding a functional language does bring additional challenges, but supporting the current rules implemented on *SLang*has been done without blockers.
   One typical feature of functional language that we are going to face one day or an other are the high order functions.

   (a) *closure/high order function*

3. *Static vs dynamic scoping*

4. *Static vs dynamic typing*

5. *Declarative vs imperative language*

7.3 How can SLang cope with all these feature ?

All these feature are concretely related to the same challenge: choose what we want to add in *SLang*. We can always add more feature, but the language will quickly become very complex and unmaintainable. For example for the call by name/value, we can easily add the information in the node, and use this information in the checker. There is not clear answer, it is all about the tradeoff discussed in section 2.1.3, if one feature seems to be appearing a lot and causing problems, it would make sense to include it in the grammar.

8 Comparison with other tools

If the initial goal is not to find as many issues as an other tools, looking at the features they provide is a good way to look how to improve the current checker and to anticipate if it is possible to implement them on top of *SLang*.

8.1 General features features

Table 10: Technology used by different tools to detect *null* pointer dereference

Table 10 shows the different technology that the different tool uses. This is a high level description of the feature, they all implement them in their own way and with different level of efficiency. We can see that our tool is not using a lot of these technology, mainly due to the fact that we did not target them in the first place. The next parts will discuss what this technology good for, and if we can implement it on *SLang*.

8.1.1 Interprocedural

Our current checker is only supporting intraprocedural analysis, going further is obviously a way to find more issue, since it would enables us to learn belief from arguments not only inside the function, but also outside. The main difficulty is to define which function is called at run time, if it is possible to do it for one language, having a consistent way to do it in a language agnostic way is impossible. One of the way is to compute the summary of every function, and to use this information during the intraprocedural analysis. For example, we can store for every function if it can return *null*, then when the result of this function call is assign to a variable, we can consider it the same way as if it was *null*. This idea is used by SpotBugs and will be described in subsection 8.4. There is multiples others way to perform interprocedural analysis, that represents more precisely the execution flow, however the idea are complex and will not be described in this work.

8.1.2 Requires the build

Requiring the build can in fact be perceived as a disadvantage, since compiling code and calling it static analysis seems to be a contradiction. Using the build provides however so much information that the popular tools seems to all have opted for it. This can make sense when the checking for error is integrated to the build process, but this is a real handicap when we want to have interactive feedback in an IDE or a pull request analysis, and is simply not possible in a cloud computing scenario, when you do not have access to the binaries. The recent trend however is to avoid using the build due to the disadvantage stated before.

In the situation of *SLang*, we will obviously not have access to the binary of the original language. In addition, the goal of *SLang* is not to be a complete language, it is therefore far from being possible to compile this new code. This adds a new challenge that *SLang* will probably face in the future, but it brings enough benefits to make the effort worth it.

8.1.3 Guided by annotation

We have seen in section 8.1.1 that interprocedural analysis is difficult operation, to help to reduce the complexity of the analysis, we can use annotation to help the tool report possible problem. Annotation are typically used to declare that a function can return a *null* value, or that a function should never be called with *null* as argument.

Listing 27: Example of annotated code

```
@NonNull int f(@Nullable String s1, String s2);
```

In listing 27 for example, the function *f* is guaranteed to never return *null*, and the callee can directly dereference the result of this function without checking it for *null*. For the parameters, it enforce that *f* can gives *null* as a first parameter, but not for the second one.
They are multiple way to do it depending on if we want, for example, Error Prone is using a trusting analysis, meaning that method parameters, field, and method return are assumed @*Nullable* only if annotated so. If we see the problem the other way, we could alternatively ask to explicitly mark as *non-null* an argument that should never be *null*.

Annotations seems to be the most popular way to detect *null* pointer exception currently especially for interactive tools, it enables to detect most of the exception with a small effort on the programmer side. It is often worth to make this effort, it is useful not only for the checker, but also helps during the development of the program. The downside of this method is that it requires a consistent and coordinate use of annotation in a whole project, consistency that is hard to achieve, even more when we want to introduce it in a new project.

Annotation could be added on top of *SLang*, finding annotation that are relevant for any language is possible if they share the same concepts. For example, a *non-null* annotation makes sense in any language that has the concept of *null*. In other cases, this can be more tricky; for example, the annotation initializer, used in the context of *null* pointer exception, can not be used in an language agnostic way, since the initialization is not the same in any language.
One solution to this is to provide a way to configure the tool. Using configurable rule is always a danger for the user experience. For example, NullAway is providing more than 10 configuration flags, some of them being mandatory, all of them related to *null* dereference. In the context of *SLang*, having so much configuration to do for every rules and every language simply does not scale at all.
An interesting note is that annotation are principally used is to help to reduce the complexity of the analysis, however no annotation is required to detect all pointer if we have a perfect inter procedural analysis.

8.1.4 Language specific help

One tempting solution to solve all potential problem is to add language specific information to the different checker. We can always add metadata, to add information that reflect the different parts of the language. This is a step that can be useful if the concept is well known and shared between multiple language. This is an important element: the language is not designed to be perfect and to satisfy everyone, but to support at best the rules implemented on it. Therefore, if we see that 4 out of 5 languages define a feature (like ternary expression for example), and that we need it to provide precise issues, it makes sense to add it to *SLang*.

8.1.5 Path sensitivity

Currently, our tool is using flow sensitive analysis, namely we are only interested in the order in which the statements are executed. In addition, path-sensitivity computes and keeps additional information, based statements seen along the path and avoid infeasible path. For example, if a pointer is checked for *null*, the tool will know that the pointer is equals to *null* inside the true branch, it will therefore report if the pointer is used or given to a function that expect a *non-null* value. In addition, we could also use the path sensitivity to improve the MAY analysis introduce in subsection 3.4.1, to at least remove the obvious false positives. The main challenge of this kind of analysis is to deal with the fact that the number of path grow exponentially, making it hard to scale. In the current situation, we do not have path sensitivity in our checker, but we already have all the features required, implementing it with the same constraints as an implementation over a original language seems to be possible.

8.2 Other tools features

The idea here is not to describe in depth the exact functioning of a particular tool, but to describe some of the features implemented by other tools that can be interesting for our purpose of anticipating the potential problems of an eventual implementation on *SLang*.

8.2.1 IntellJ IDEA

IntellJ is an IDEA, this is a particularly interesting since it requires interactivity, a user wants to see the issues being raised while he writes code, without having to rebuild the whole project. This tool is also performing a *null* pointer analysis using annotation. It warns when the user use a pointer that is @*Nullable*, without checking it for *null*. To detect that a pointer is checked for *null*, it uses a pattern that is customizable. This will not work if the user is using custom methods to perform the null-check. In this case, the user can use a contract, that would for example say that this method fail if the argument is *null*, or more simply configure "not-null" check methods. Having configurable setting for a rule in *SLang* is far from being ideal, even if the effort to configure one check seems to be minimal, if we have a configuration to do for every rule, this can quickly becomes a nightmare for the user. All our work that try to reduce the overall complexity would be pointless if we have a configuration for every rules.

8.2.2 Error prone : Null away

Null away is a tool built on top of error prone. The process first checks if the value dereferenced is obviously *non-null* (annotated @Non-null). If it is not the case, it performs a data-flow analysis to try to show the that the value is *non-null*. The data-flow analysis is using existing *null* check into the code. Therefore, if a field is annotated as @*Nullable* and is dereferenced, an error will be reported only if the value is not checked for *null*. The key idea here is to perform the analysis in multiple steps of potentially increasing complexity. If the value is obviously *null*, we do not have to go through the creation of the control flow graph and the computation of the data flow analysis. Having multiple steps is a particularly good idea for *SLang*, not only for performance, but also for the quality of the results. If we can report an issue, or prevent a complex computation before facing the uncertainties due to *SLang*, it may prevents us to make bad decisions.

8.3 In-depth comparison: SpotBugs

SpotBugs [17] is the successor of Findbugs [5], an open-source static analysis tool, it implements multiples checks related to *null* dereference, it is therefore a good candidate to have a more complete comparison with.

SLang 21	**SLang ∩ SpotBugs** 21	**SpotBugs: annotations** 263
SpotBugs: others 161	**SpotBugs: correctness** 424	

Table 11: Slang and Spotbugs comparison on open-source projects

Table 11 shows the number of issues reported by the two tools on more than hundred open-source projects. For SpotBugs, we used the default configuration, namely confidence level and effort set to default, and took only the issues related to real potential bug.

Table 11 shows a subset of more than 30 rules related to *null* pointer dereference that are reported by SpotBugs. For our purpose, we are only

Rule	Category
Nullcheck of value previously dereferenced	Correctness
Possible null pointer dereference	Correctness
Load of known null value	Dodgy code
Method with Boolean return type should not return null	Bad practice

Table 12: Examples of rules reported by SpotBugs

interested by the rules that are labeled as “correctness”, as they represents the bugs that we try to identify and not the code smells that are not directly of interest for us.
Note that the number is different from the previous one because SpotBugs was crashing on some of the project (OpenJDK, elastic search). This leads to our first observation: our tool can be ran with no configuration directly on any of more than 100 of projects. This is particularly good: if we want to introduce these kind of tools on top of huge project like OpenJDK, it is extremely complex to debug if it does not work out of the box. The second observation is that every issues reported by *SLang*, is also reported by SpotBugs. This results may seem discouraging, we are not finding anything new, but it also shows that the issues reported by our tools does matter for others tools as well. These issues are reported by SpotBugs as “NullCheck of value previously dereferenced“, who is in fact exactly the issues that we report when we use the forward version of the analysis. In addition, *SLang* implementation is reporting all issues that are reported under this category, showing that we do not seem to be missing any obvious issues.

While we would want to compute the intersection automatically, this number has to be computed by hand. Fully automatically computing the intersection is not a trivial task [6]. First, due to the fact that Spotbugs works on byte-code, we can not rely on the positions (even the line) of the reported issue reported by the tool. This problem is even worth since the tools is reporting the issues in an inconsistent way, sometimes in a check for *null*, then when the pointer is used. One solution would be to look at the file level, and compare the number of issues. This would be possible if the issues were reported into the same category, but Spotbugs is reporting the issues related to *null* pointer in multiple categories, if we include multiple categories into the comparison, we greatly increase the chance to have other issues that are not related reported in the file.
The most important information is the number of issues, SpotBugs is report-

ing more than 20x more issues! We can split this number into two category: the first one is the issues related to annotation. It is interesting to do the differentiation to understand what we can gain from adding a given feature. The second is the other issues related to *null*. These issues are reported without the help of annotation. It can be interesting for us since it does not require any language specific knowledge, and can serve as a goal that can be reached with our tool.

8.4 SpotBugs specific features

The first reason of this huge difference is that we have such difference is the additional feature that we have described previously in subsection 8.1. In addition, we will look more in-depth into one additional feature that is implemented in SpotBugs and producing a big difference, and see that may be implemented on *SLang*.

1. *Summary-based inter procedural analysis*
   This is an smart and easy first step to perform inter procedural analysis. The idea is to compute a summary of every methods, and use this information during the intraprocedural analysis. In our case, we would like for example to store if a method can return *null*, or if an parameters could be *non-null* to then report if *null* is passed as his argument. We can have this information by simply looking at annotation. If the annotation are not presents, we can still perform intraprocedural analysis to find ourselves the annotation. For example, if a method ever return *null*, we can annotate the function as @*Nullable*, and if a function always dereference an argument without checking it for *null*, we can consider the argument as *non-null*. The good part is that we already have the tools required to build the summary. The feature currently missing in *SLang* is the method references. We obviously need to be able to identify which method is called to be able to retrieve the summary. This is related to the problem of the name references that we faced during in the previous parts, naive solution exist, but proper semantics have to be completed.

Listing 28: Example of true positive and false negative of SpotBugs

```
B b = new B();
b.foo(null); // True positive

```

```
abstract class A {
 foo(p);
}
```

```
class B extends A {
  foo(p) {
    p.toString();
  }
}
```

Figure 8: Pseudo code of a class that extends an abstract class

```
4 A a = new B();
5 a.foo(null); // False negative
```

In listing 28, we have an example of a true positive and a false negative from SpotBugs. At line #2, the issues is correctly reported, the tools manage to identify statically that the pointer *b* is called with type *B*. At line #4 however, the tools is not reporting any issues. This is due to the fact that the type of *a* is *B*, the tools do not identify the potential run-time type of the variable.

This method does not require complex features, there is not strong push-back to implement it on *SLang*, and can already greatly increase the number of issues reported by *SLang*!

9 Related work

9.1 Micro grammar

How to Build Static Checking Systems Using Orders of Magnitude Less Code [1] is raising a similar concern that we tried to solve. They observed that the current situation makes it hard to target new languages due to the complexity of the current systems. The main idea is similar to *SLang*, they implemented a checker that is based on an incomplete grammar, that is called micro-grammars. With this approach, they managed to implements a checker that is order of magnitude smaller that typical systems. They results is encouraging, they manage to find hundreds of issues with an acceptable false positive rate, with some of them that were not reported by they previous work. The idea is similar to island parsing, where the grammar only describe some part of a language, without requirement to have the whole syntax.

10 Future work

10.1 Rule inference

This work shows the potential of *SLang* to support the implementation of more and more checks, however, the list is limited, finding interesting rules that makes sense in a language agnostic way is difficult. One promising continuation is to work on rule inference to detect object usage anomalies [18].

Rules inference try to solve the problem that users uses a wide range of function for the same goal, identifying which function does what is typically a hard process that need to be done by hand and even impossible for manual approach if the function is user defined. If the basic idea is to generate rules specific to a project, it would makes sense to adapt it to generate rules specific to a language, everything in an agnostic way on top of *SLang*. The typical example is to look at temporal properties. There is multiples way to do it, looking at the sequence of method call online [6], or to use a related idea of the belief style [4] that we used in this project. The idea is to learn pattern of sequence of function call from the code, and report when this pattern is not respected. For example, if we see that the majority of the time, $< b >$ is used after $< a >$, it might imply the belief that $< b >$ should be called after $< a >$. If, in the minority of the cases, $< b >$ is not called after $< a >$, it contradict the belief and may therefore implies an error. Concretely, this idea should be able to detect that an unlock is called after a lock, or that a resources is closed. In the example of the lock, the way a programmer typically deal with them is dependent of the language, and even dependent of the project!

This technique would enables us to find issues without knowing what is correct, what are the name of the method, the typical process to use, and what is the way it is done in different language.

10.2 Benchmarks

In section 8, we have tested the tools on real-life project, it is a good step to understand the quality of the results on a set of real life situation, however, the list of potential issues present in the real life project is not at all exhaustive. To complement this work, it makes sense to test it against benchmarks, that aims to test as much situations as possible. In our situation, it would make sense to test with benchmark that uses languages specific features, like callback, high-order and all the features that we have discussed in section 7.2.

10.3 Improving the checker

The work presented under section 8 is a good starting point to see what can be done in the future for this check. For example, the comparison with SpotBugs in section 8.3 showed us that we can already greatly increase the number of true positive, without any complex features and expected problems.

11 Conclusion

We managed to run a null pointer dereference checker on top of an incomplete intermediate representation. The checker turns out to be as efficient as the implementation on more complete intermediate representation, after some effort to adapt the language. The control flow graph shows some weak spot due to the presence of the native nodes, but we manage to find a solution that report a good amount of issues with no obvious false positive. The algorithm contains no complex elements, and is still able to find more than 35 issues on top of existing open-source project.
[...]
Slang seems to have started in a very good way, but we have to keep in mind that the language is less than one years old, if it suits well for the situations that we have tested today, it still have many challenge to face.

Appendix A SLang Grammar

$$
\begin{aligned}
\langle \textit{StatementOrExpression} \rangle \text{ a } &::= \langle \textit{Block} \rangle \ [\langle \textit{StatementOrExpression} \rangle \text{ a}] \text{ a} \\
&\mid \quad \langle \textit{Break} \rangle \text{ a}
\end{aligned}
$$

References

[1] Fraser Brown, Andres Nötzli, and Dawson Engler. "How to Build Static Checking Systems Using Orders of Magnitude Less Code". In: *SIGPLAN Not.* 51.4 (Mar. 2016), pp. 143–157. ISSN: 0362-1340. DOI: 10.1145/2954679.2872364. URL: http://doi.acm.org/10.1145/2954679.2872364 (cit. on pp. 17, 59).

[2] Timofey Bryksin et al. "Detecting Anomalies in Kotlin Code". In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ISSTA '18. Amsterdam, Netherlands: ACM, 2018, pp. 10–12. ISBN: 978-1-4503-5939-9. DOI: 10.1145/3236454.3236457. URL: http://doi.acm.org/10.1145/3236454.3236457 (cit. on p. 46).

[3] *Common rules list: rules that are defined as common by SonarSource.* Mar. 2019. URL: https://jira.sonarsource.com/browse/RSPEC-5167?jql=project%20%3D%20RSPEC%20AND%20issuetype%20in%20(%22Bug%20Detection%22%2C%20%22Code%20Smell%20Detection%22%2C%20%22Finding%20Detection%22%2C%20%22Vulnerability%20Detection%22)%20AND%20%22Common%20Rule%22%20%3D%20Yes (cit. on p. 8).

[4] Dawson Engler et al. "Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code". In: *SIGOPS Oper. Syst. Rev.* 35.5 (Oct. 2001), pp. 57–72. ISSN: 0163-5980. DOI: 10.1145/502059.502041. URL: http://doi.acm.org/10.1145/502059.502041 (cit. on pp. 18, 60).

[5] *FindBugs: tool to detect bugs in Java code. University of Maryland.* Mar. 2019. URL: http://findbugs.sourceforge.net/ (cit. on p. 55).

[6] Mark Gabel and Zhendong Su. "Online Inference and Enforcement of Temporal Properties". In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: ACM, 2010, pp. 15–24. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806806. URL: http://doi.acm.org/10.1145/1806799.1806806 (cit. on pp. 56, 60).

[7] David Hovemeyer and William Pugh. "Finding Bugs is Easy". In: *SIGPLAN Not.* 39.12 (Dec. 2004), pp. 92–106. ISSN: 0362-1340. DOI: 10.1145/1052883.1052895. URL: http://doi.acm.org/10.1145/1052883.1052895 (cit. on p. 13).

[8] James Koppel, Varot Premtoon, and Armando Solar-Lezama. “One Tool, Many Languages: Language-parametric Transformation with Incremental Parametric Syntax”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018), 122:1–122:28. ISSN: 2475-1421. DOI: 10.1145/3276492. URL: http://doi.acm.org/10.1145/3276492 (cit. on p. 9).

[9] *List of open source projects used.* Mar. 2019. URL: https://github.com/quentin-jaquier-sonarsource/Master-Project-Report/blob/master/Data/open-source-projects.csv (cit. on p. 42).

[10] *Oracle documentation, Standard ed. 8, class NullPointerException.* Mar. 2019. URL: https://docs.oracle.com/javase/8/docs/api/?java/lang/NullPointerException.html (cit. on p. 24).

[11] *RSPEC-2259, Check implemented on SonarJava: null pointers should not be dereferenced.* Mar. 2019. URL: https://jira.sonarsource.com/browse/RSPEC-2697 (cit. on p. 17).

[12] *Scalameta, Library to read, analyze, transform and generate Scala programs.* Mar. 2019. URL: https://scalameta.org/ (cit. on p. 13).

[13] *Slang test sources, open-source projects used to test different features of SLang.* Mar. 2019. URL: https://github.com/SonarSource/slang-test-sources/tree/81dae65239b0665afafd9ea0f09a2f7942ddc052 (cit. on p. 38).

[14] *SonarJava, static code analyser for Java language.* Mar. 2019. URL: https://github.com/SonarSource/sonar-java (cit. on p. 17).

[15] *SonarPHP, static code analyser for PHP language.* Mar. 2019. URL: https://github.com/SonarSource/sonar-php (cit. on p. 28).

[16] *SonarQube: open-source platform for continuous code quality inspection, developed at SonarSource.* Mar. 2019. URL: https://www.sonarqube.org/ (cit. on p. 42).

[17] *SpotBugs: tool to detect bugs in java code. Successor of FindBugs.* Mar. 2019. URL: https://spotbugs.github.io/ (cit. on p. 55).

[18] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. “Detecting Object Usage Anomalies”. In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering.* ESEC-FSE ’07. Dubrovnik, Croatia: ACM, 2007, pp. 35–44. ISBN: 978-1-59593-811-4. DOI: 10.1145/1287624.1287632. URL: http://doi.acm.org/10.1145/1287624.1287632 (cit. on p. 60).