Using an universal 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

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 expressions by natives elements in *SLang*, and the notion of unreliable basic block for dataflow analysis. Our experiments compare the results of an implementation of a *null* pointer dereference checker over *SLang* and an implementation on the original language. The results show that we are able to find real issues with very low false positive rate, and also reach the same precision as the implementation on the original language.

Contents

Listings

List of Figures

List of Tables

1 Introduction

SonarSource is a company that develops static analysis tools 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 targeting 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. The next part of the work is to implement 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. Since each language produce a different tree, every check have to be implemented individually for every language, 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, that typically takes most of the production time. 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 [1], that apply to any programming language. 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 was the main motivation: avoid redoing the same work again and again, by implementing the checks on top of a common representation, reducing the complexity to a single implementation of each checks. 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 came 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 checks 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 current 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. It is therefore a balance between complexity (number of different feature that we support) and accuracy to be able to still report interesting issues.

The current grammar [2] and interface [3] of *SLang* 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 approximate 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 [4]. 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 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. The transformation is therefore incomplete, we are going to make abstraction of some concepts, but it is not a problem as long as the result of the checks are not affected. One interesting note is that there is 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 are the **native nodes**. During the translation, we are going to map all original nodes to their equivalent in *SLang*, if one has no equivalent, it is going to be mapped to a native node.

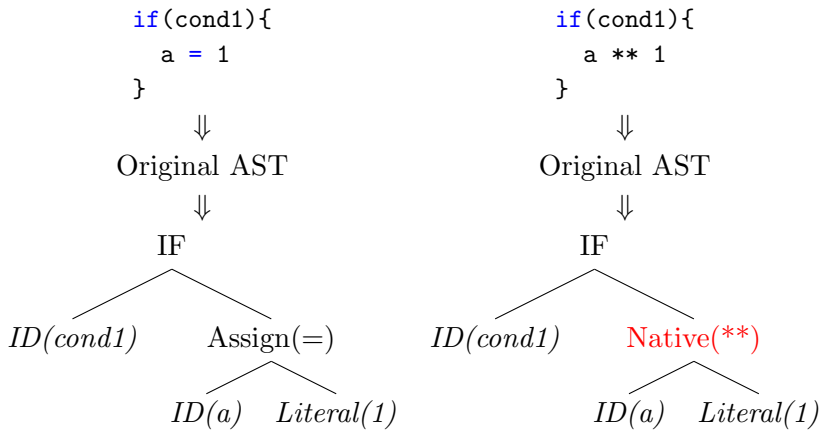


Figure 1: Example of native node in SLang

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, that we will still keep, but as a native node. Native nodes therefore represent nodes that are unknown, but we will still be able to compare them because we will keep the original type, list of their children and tokens inside the native node. 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 and where to expect a tree. For example, if we want to detect that two

functions body are similar, we can compare the child that correspond to the body, making abstraction of all its content. This is the key to ensure that the rule will work on any new language, for the function body comparison for example, we are going to add all elements that can differentiate two implementations inside the body, even if the node was not directly inside in the original language. The native nodes also enables to process to the implementation 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?* is now clear, 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 control-flow base 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 finish to compare our checker with other tools (Section 7), to see how we could improve our current version, and try to anticipate potential problem that can arise in the future of *SLang*.

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 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 specific needs, 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 are 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 checks

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 from the original language to *SLang*. If the prerequisite for the front-end are respected, the initial effort of 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 adapt it. 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 where the unexpected problems arise, that are due to an unknown language feature, a wrong approximation, and so on... Hopefully, there is multiple way to improve the results. The first one is to setup a parameter for the rule, that will adapt the behavior of the rule depending on the original language, and even depending on the user input. For example, all the rules of naming convention have to be setup with the convention of the language, and can be changed by the user by a custom setup. 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 obviously 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 less than 5% of false positive for our work. This is an arbitrary choice, other tools like FindBugs [5] initially targeted a rate of less than 50% of false positives for example. 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*. The first step is to find a good front end. Scalameta [6] 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 abstract syntax tree from a Scala file. At this point, we already have enough information to activate a first rule: file should parse. If Scalameta is not

able to parse the file, we report an issue. The first step from this tree is to extract comments, and translate the Scalameta token into *SLang* token. With this simple step, we are already able to enable new checks related to comments, like the tracking of comments with *TODO* and reporting code that is commented. The second step is to start the translation. As in any compiler phase that perform translation, the skeleton of the code 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. 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 sure that the meaning of a node in the original language is the one intended in *SLang* and that the metadata is correctly handled. Package and import declaration, literal and block are example of node that have a direct equivalent. 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 the implementation always follow the same process: we recursively build the children and eventually make sure 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 check, we have to make sure that everything make sense. In the case of *Scala*, some of the feature of the language greatly reduce the quality of the checks. One quick but naive solution when facing false positive is to map the problematic node to a native node, to remove the problematic case.

Listing 1: Pattern matching that can cause false positives

```
1 something match {
2   case "a" if(variable) => println("a")
```

```
3     case "a" => println("a")
4     case "b" if(variable) => println("b")
5   }
```

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 branch body in a conditional structure. If we map the match case to native, this solve the problem, but introduce false negative for 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 type 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 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%)
TermParam (3 & 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 more than 10% is mapped conditionally. This information can lead our research and lead to identify 4 potentials problematic situation.

1. ***Function with many parameter clauses***

Listing 2: Example of Scala function with many parameter clauses

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

Listing 2 shows an example of a Scala function with multiple parameters, that is common in Scala, but not necessarily in other languages. Mapping the whole function to a native node is a big downside, we are not going to be able to run all the checks that we could run inside functions. Adding the support for multiple list to *SLang* is a first solution, but we have to make sure that the benefit for this addition is worth the added complexity. A second solution is to merge all the parameters to a single list. This works fine, we recover the possibility to run all checks that apply to functions. However, the check that limits the number of parameters raise some unexpected issues. If we limit the number to one single argument, the code from listing 2 will raise an issue. The problem is that the implicit keyword is exactly here to not have to give this argument when calling this function, one can argue that implicit parameters should not be accounted. Despite this concern, we choose this solution, it does not change *SLang* itself, and a use can always configure the limit if he thinks that the checks raise unexpected issues.

This case exactly describe the challenges when implementing static analyzer, their is often multiples solutions, not really complex in themselves, but it requires a good understanding of the whole ecosystem, form the original language keyword *implicit* to the final implementation on *SLang*, to be able to produce good quality results.

2. ***Match statement with at least one conditional case***
   Listing 1, seen previously, also appear in the list. The current situation is that the whole match statement is converted to a native tree if only one conditional case is present. The granularity of this solution is not fine enough, we still want to be able to run the different checks related to match tree, even if one branch have an unknown structure! The solution chosen is to wrap the case tree inside a native in the case where a guard is present. This fine granularity is far better, we are know able to compare the body of the different cases and report duplicate ones, and also compare the pattern since we are still able to compare two natives nodes.

3. ***Function parameters with default value***

Listing 3: Example of Scala function with default value

```
def f1(i: String = "Default") = ...
```

In listing 3, we can see function parameters with default value. Once again, the first question is to know if it is worth it to add the support for such construct. For default value, *Ruby*, *Scala* and *Kotlin* have default value, it makes sense to add the support to *SLang*. This new structure adds new issues on bad naming convention, hard coded ip inside default value, unused parameters, and few others, comforting our choice.

4. ***Function parameters with modifier***

Listing 4: Example of Scala function with implicit modifier

```
def f(implicit param: Int) = {
  g
}

def g(implicit param: Int) = {
  print(param)
}
```

Listing 4 shows a problematic situation with the rule that check for unused parameters in Scala. In this case, the parameter *param* seems to be unused in *f* while it is implicitly passed to *g*. This is the reason why we had mapped parameter to native. At first glance, we might think that *Scala* is the only language that have an *implicit* modifier, but if we generalize the idea, we can expect other modifier and even annotation at this place. Annotation is more popular and could be useful in the future. The solution that we choose is to add the support for modifier in *SLang*, map both modifier and annotation to this node in *SLang*. Note that we are currently supporting only a fraction of the modifier possible, the majority fo them are going to be native nodes. We are now able to adapt the check for unused variable, but not reporting issues on variable with modifier. We might miss some real issues since the modifier is a native node that may be unrelated with the problem, but the approximation already gives good results.

3 Improving SLang: Null pointer consistency

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

3.1 What is null pointer consistency

Null pointer consistency is the verification that a pointer who is dereferenced is valid and not equal to *null*. Dereferencing a *null* pointer will result 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* [7], implemented on SonarJava [8], the tool developed at SonarSource to perform static analysis on Java code. The current implementation uses a complex symbolic execution engine to report potential *null* pointer exception. Symbolic execution try to estimate all possible execution paths, 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 paths 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 the ability to find more bugs. [9]. Our initial goal is not to find all the issues that the implementation on SonarJava reports, but to see if it is possible to still find interesting issues with an implementation that is less complex, and based on a common intermediate representation.

The idea of this first checker is to use facts implied by the code, that we will call **belief** [10]. 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 equal or not equal 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 demonstrates 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 individual instructions, and the edges represent 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 reduces 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 the 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 the 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 built for the current function being analyzed (intraprocedural), and will not have any access to other functions or others files (interprocedural).

Formally:

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

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

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

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 follow:

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 check

Analysis type	**N° 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 2.1.3. Table 3 shows the difference 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 rate of false positives is significantly higher, finding interesting issues is too hard with this noise. In addition, another downside of *MAY* analysis is that identifying true positive can be tricky, involving only specific path executed that will raise an exception, while discovering false positive is straightforward. In practice, to help the

user to better understand the issue, we could report multiple locations, for example the line where the pointer is used, and the one where it is dereferenced.

Intuitively, it is not surprising that the *MAY* analysis performs poorly if we do not take into account the paths that are unfeasible.

Listing 6: Example of false positive of MAY anaylsis

```
1 if(p != null){
2   p.toString()
3 }
4 (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 at line #4 does not mean that there is a potential exception. We will discuss possible amelioration to this situation in subsection 7.1.4.

3.4.2 Used then check, check then used

Listing 7: Pointer that is used then checked

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

Listing 8: Pointer that is checked then used

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

Listing 7 and 8 shows the difference between the two versions. The work presented before implements the former, however, the latter makes as much sense, if all paths 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 from the forward analysis 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 real 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

```
1 if(p == null) {
2   customThrow();
3 }
4 p.size();
5
6 customThrow() {
7   throw new MyException();
8 }
```

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

Custom functions that change the control flow is a weak point for flow based checker that does not perform interprocedural 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 first version (used then checked).

4 Implementation on SonarJava

We are first going to implement this check on 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 starting from the last element of the basic block to the first, 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 the *gen* set only if the use of the pointer follow the assignment, as expected.

Listing 10: Problematic situation with naive basic block creation

```
1 p.toString();
2 b = (p == nul);
3 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!
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 equal (or not equal). 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 when a pointer checked, it 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 example shown in listing 10, the check for *null* breaks 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 and compute the new *out* set of the current head. Since we are performing a forward analysis, if the new out set have

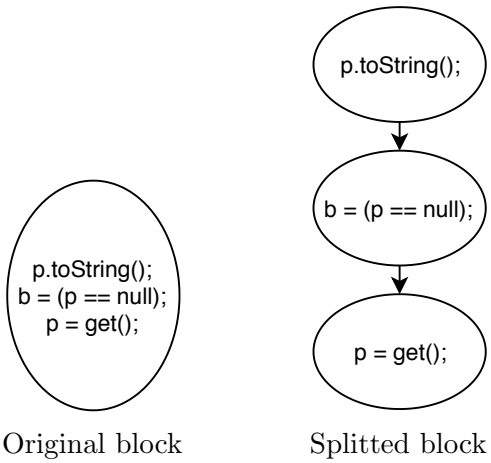


Figure 2: Example of the new way to split basic block

changed, this means that all the successors might potentially change as well. We therefore add the current basic block and all of it successor at the end of the work list, if the processed block sets have changed. We continue this process until the list is empty, meaning that we reached a fixed point.

At the end of this process, 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 basic blocks, if we see a check for *null* (that 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 documentation [11]:

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 tree, a control flow graph and the symbol resolution available. The first step to implement it on *SLang* is to identify what nodes and structures we need in order to implement everything that is required for this check.

5.1 Required 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 checks 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 do not need to know anything about the context in which the pointer is used, just that it is dereferenced at some point. When a function is called without a member selection, the tree will be an identifier (name of the function) and a list of argument, but in the case of a pointer use, the tree corresponding to the identifier will be a member selection, that we will use in the checker. By doing this, we are able to detect not only functions invocations, but also fields selections 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 mostly 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:

1. ***Variable declaration***

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

```
1  p.toString();
2  while (cond) {
3    Object p = getP();
4    if(p == null){ } // Compliant
5  }
```

In section 5.3.1, we are going to describe how we perform a naive semantic, that is assumed inside the check. In this semantic, we are not going to be able to differentiate if two pointers 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. In listing 11 the pointer is used at line #1 and check for *null* at line #4, but the two identifier p do not refer to the same symbol. In this case, removing p from the set when it is declared at line #3 will enable us to remove the false positive.

2. ***Function invocation***

Listing 12: Example of pointer used as a parameter of a function call

```
1  f(p.size())
2  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.

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.

5.1.1 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 represented as a member select, and can therefore be supported. Accessing or modifying the fields of *null* as if it were an array is however not supported. This will lead to false negatives in *SLang* that we will not have in the implementation of SonarJava, but the situation seems to be uncommon.

5.2 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 one of SonarPHP [12], 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 4 fields:
   (a) ***Predecessors***
   List of nodes that may be executed **before** the current block.
   (b) ***Successors***
   List of 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***
List of node following the current block if no jump is applied. This is not directly needed for our check, but it may be required for some check later 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 have a true and false successor block reference in addition to the simple block.

This is the only two interface that we need, we can now start to build the graph.

5.2.1 Building the control flow graph

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 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 contain 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.

1. ***Block and other nodes***
The simplest tree that we will face are the blocks, they represent a list of statements, we can therefore directly recursively build the graph for all children, that will add the content of the block inside the current basic block. 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 keep useful 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.

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 false and true successors, respectively. We can now recursively build the condition of the *If* tree from the branching block created before.

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 address this situation, we can introduce a **forwarding block**, a 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 temporary objects are here to store the link to the condition for emphcontinue, and the end of the loop for *break*. Before starting to build the body of the loop, we will push a breakable object to the top of the stack, and pop it once we are done. We use a stack to support nested loop, a break and continue will refers to the first enclosing loop.

   The current implementation of *For* loops is the same as *While* loops, as we do not need the exact behavior of loop for our check, we can accept this approximation. For *Do/While* loops, we are going to use the same idea but we are going to start start to build the condition before the body.

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* statement. 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. Non-fallthrough 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 an object (similar to 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* statements refer only to the closest enclosing match tree.

The next step is to create a forwarding block for the default case of the match tree, and build the different cases in the reverse order, 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 can build the sequence of the body of cases, re-starting each time we see a break.

Listing 13: Example of fallthrough pattern matching

```
1  x = 0;
2  match(cond) {
3   case pat1:
4     a = 1;
5   case pat2:
6     b = 2;
7     break;
8   case pat3:
9     c = 3;
10 }
11 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, for example $a = 1$ is executed both when *pat1* and *pat2* is true.

5. ***Jump Tree: break and continue***

Jump trees 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 directly add it to the current block. We will discuss in section 5.4 a solution to better support this situation. In the usual case, they will appear where we expect them and create an edge from the current

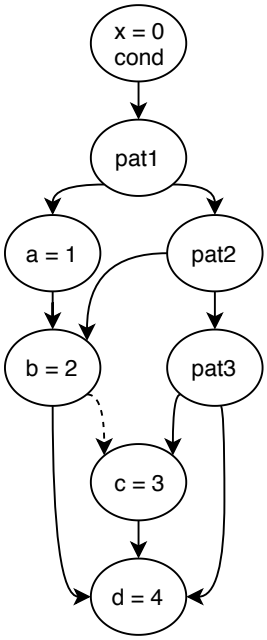


Figure 3: Corresponding CFG of the code of listing 13

block to the head of the stack that is filled as described in 3 and 4 sections.

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.

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 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 and where. 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.

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.

5.2.2 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, the task 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

We now 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 nodes that have an unreliable execution order 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 symbol of identifier, data 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 is a naive version that is not made to work well for all language, but used to show that with a proper semantic computation (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, but this would add some false positive.

Listing 14: Field can change value during a function call

```
1  String s = "";
2
3  void foo() {
4    s.toString();
5    changeS(); // An other method can change the value of s!
6    if(s == null) { } // Compliant, s changed
7  }
8
9  void changeS() {
10   s = null;
11 }
```

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 come 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, these new issues does not exactly respect the original description, we are not going to report them.

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 native nodes. For our concern, we will see the native nodes as a nodes that we do not know anything about, with a list of children.

Listing 15: Pseudo code with a ternary expression

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

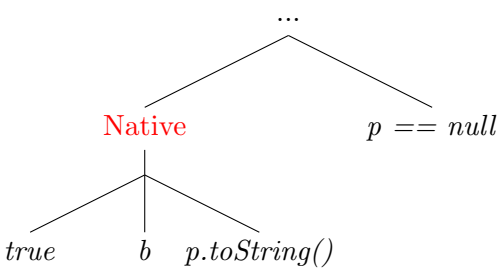


Figure 4: *SLang* AST from the code of listing 15

Figure 4 shows the result 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 represent the problem, but the construction can be any native nodes coming from any original language.
The problem here is that we have to represent 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 a typical example: in this case, we do not need to know what the native node (orange node) is exactly, but that the node assign p. We do not care what exactly happens in this native node, we just need to know that, at one point, p is assigned, even if it is possible that the

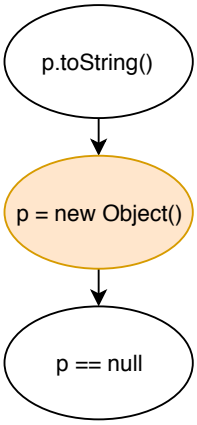


Figure 5: CFG with an assignment in a native node

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, making the assumption that the evaluation order is not important. This is in fact correct for a native node with only one child, where the evaluation order can obviously not change. If there is multiple children, we have to add 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 provide 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

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

Listing 16 some Java pseudo code with the corresponding control flow graph with 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 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 it is not in the real execution.

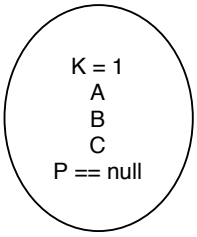
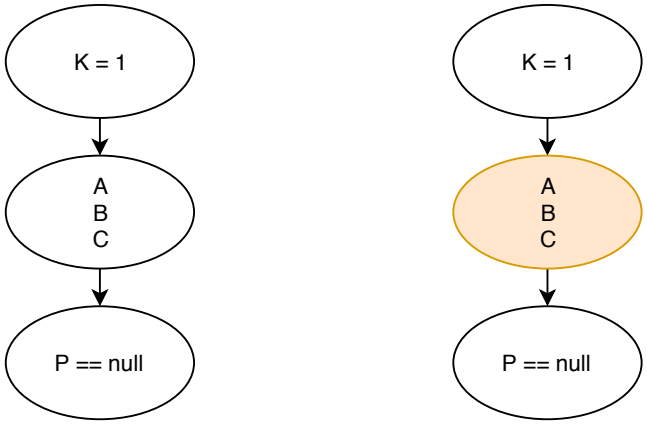


Figure 6: Basic block content of the code in listing above



Splitted CFG Splitted block with unreliable node

Figure 7: CFG with elements coming from natives nodes

We can therefore not ignore these nodes, and not naively add them to the blocks. The idea to solve the problem showed before is to put all elements that come from a native node in a separate basic block (figure 7, left), and mark the block as **unreliable** (figure 7, right), shown in orange. All control flow statement nested inside the native nodes will also lead to unreliable basic block.
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.

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 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 needs to really trust the control flow graph, it can make sense to directly stop the checker if the graph can not be trusted.

Language	**%**	**% of completely native**	**N° 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 as native. If more than 40% of nodes are not translated, this does not mean that our language lack a lot of nodes, but that we do not mapped some kind of nodes on purpose. This table shows us that native 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 native 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 consists 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.

$$newGen(n) = \begin{array}{l}\text{use of pointer in the node n,} \\ \text{except if the node is marked as unreliable}\end{array} \quad (1)$$

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 an issue, that is 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.

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 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 situations due to Boolean short circuit

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

With this addition, we are now able to avoid the false negative that were reported in the two examples of listing 21.

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 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 variations that are possible, since the number of known nodes is fixed (and only a faction of it in practice). This kind of problem are 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.

3. ***Lost Jump Statements***
   We have seen in section 5 that we directly a jump statement to the basic block in the case where we do not expect it, without doing anything special. In fact, we face similar uncertainties that happen with native 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 by this statement 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* (section 5) with the implementation on SonarJava (section 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 production code.

6.1 Experimental Setup

To test the checker, we are going to create a SonarQube instance [14], 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 project [15] used for this experiments.

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 two implementations, with the sources and setup described in section 6.1. Despite all our effort to prevent problematic situations done in the previous sections, the implementation have than 20% of false negatives 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*.

6.2.1 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*.

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

   Listing 22: Typical code structure with ternary expression

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

   The situation is not as obvious as the one presented before, listing 22 shows a possible situation were no issue 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.

2. ***Loop Header***
   Currently, no check uses the details of the for loop header, the three distinct part are therefore mapped to a single native tree.

   Listing 23: Pointer used inside loop header

   ```
   1 for (int i = 0; i < p.size(); i ++) {
   2 // ...
   3 }
   4 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 blocks of the loop header, as it can

depend on the original language for example. This is in fact the kind of behavior that we want to achieve, the language specific features do not produce false positives, but we accept false negatives. One way to solve this problem is to adapt the loop node in *SLang* that will better support this situation. If it makes sense for loop header as it is probably a structure that can be necessary in different situation, we have to keep in mind that adding a new node to *SLang* is not a solution that we should use in all situation, where the feature is really specific to a language for example.

6.3 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*, on the same source and setup (described in section 6.1), we manage to report exactly the same issues that the implementation of SonarJava was reporting!

Project	**N° 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 positives reported, for both forward and backward analysis.

6.3.1 Other languages

The check is implemented on top of *SLang* we can therefore run the checker on other language for free, if we make sure that all nodes described in subsection 5.1 are present. Currently, the mapping have been completed for Scala and Kotlin. The current limitation to find issues is the number of project that we can run our checker on. Recently SonarSource have prepared a setup to run an analyzer on more than 170'000 projects, coming from open-source project on Github with more than 50 stars [16]. This is a huge database, with billions of line of code, containing many languages, and of overall good quality. This is a nice occasion to test our tool.

Language	**N° of issues**	**N° of projects**	**True positive rate [%]**
Java	6572	88'871	>99 ?
Scala	99	2'561	89.9
Kotlin	10	1'134	0

Table 9: Number of issues found on more than 170K projects

Table 9 shows the number of issues per language that we have found during the analysis of the 170K projects, with the number of projects containing the language, and the true positives rate. The first observation is that there are issues for Java, Scala, and Kotlin!

1. ***Java***
   The number of issues reported on Java code is existing, there is a lot of interesting bug, and if we look at the number per project, it is hardly ever above 30 issues, meaning that it is not generating a lot of noise in overall. The true positives rate is hard to estimate with this number of issues, but by looking randomly in the list, we did not manage to find any false positive, and the majority of them are still on the master branch of their Github repository. If we can not guarantee that the true positive rate is at 100%, our goal to have less than 5% of false positive set in subsection 2.1.3 seems to be reached for Java!

2. ***Scala***
   Finding issues on Scala is a good news to consolidate our confidence on the strength of the checker, it is confirmed to be working on at least two languages with two different paradigm! The number is lower than

Java, it can be due to the fact that there is way less Scala projects than Java projects. A second reason is that Scala language provides the statement *Option*, that can be easily used to avoid null pointer. The experiment also show that this statement is not contently used by the community though.

Listing 24: Example of false positive in Scala

```
1 p.map(p => (p == null))
```

The true positives rate is slightly lower for Scala, this can be explained by situation similar to the one in listing 24. In this case, our naive semantic described in 5.3.1 consider that both pointer p refers to the same pointer, but it is not the case as the second one is the element of the list p, and not the list itself. The pointer will appear as used and then checked for *null* in the control flow, we will therefore have a false positive. This is an expected problem, the same can happen for variable that are shadowed in a pattern matching, but it is not a limitation in itself.

3. ***Kotlin*** The checker only reports 10 issues on Kotlin, and all of them are false positives.

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 that reflect the reason of the false positives. At line #3, we can see that the function *isBooleanOrInt* from the pointer a is called without a safe call with *.?*, it is exactly what we want to detect and might look like a true positive at first glance. However this code will not throw a *null* pointer exception since the function *isBooleanOrInt* is called, without dereferencing the pointer a! The is called an extension functions [17]

in Kotlin, it will extends a class with a new functionality, and when used, it will not dereference the pointer. Our checker is only looking at the content of one function, from his point of view this code can raise an exception. In fact, in Kotlin, we do not expect to detect any situations where an exception is possible due to the fact that the type system is built to prevents this kind of issues. As this checks does not make a lot of sense for Kotlin, we might want to remove it from the list of checks that we are going to run on the language.

6.4 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 seem to be relevant from a high level view and without any specific knowledge of the project, you can not easily justify the correctness of any of the issue reported. To estimate more reliably this interest, we can also look at the fix rate of the issues.

6.4.1 Fix-rate

Fix-rate is the rate of issues that are reported by a tool, and really fixed by the user. As discussed in section 2.1.3, static analysis tools have to deal with the fact that if we report too many 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 issues, and took some time to fix them.
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 give 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 come from version 9, that we will compare with the version 11.

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

Table 10: OpenJDK 9 issues fixed in version 11

Table 10 shows that 25% of the issues found on OpenJDK 9 have been fixed in the version 11. This may seem 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 [18] 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 issue. 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 priority, 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 in new code, the same way you would not start to mop the floor during a flooding without having fixed the origin of the leak.

A nice story related to the fix-rate is that, during the run on thousand of project described in section 6.3.1, the checker reported an issue on an old fork of the code of the Scala compiler! The issue has already been fixed, and the commit that fix the issue state:

Move null check case higher to avoid NPE

It is a nice result, this is exactly the kind of issue that we want to report, meaning that the issues that we are reporting really matters for the programmer and he is willing to fix it!

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 issue 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, reducing his interest in fixing the code. These kind of issues should however not directly be classified as false positive, as it can also report dead code.

Listing 26: Example of contradicting code that lead to dead code

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

Potential Null Pointer Exception or Dead code ?

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 26, we can see that this code will never raise an exception. It comes from the fact that we imply 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 for *null* is 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 Comparison with other tools

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

7.1 General features

Tools
SonarJava [8]
FindBugs [5] and SpotBugs [19]
Fbinfer [20]
ErrorProne [21]
IntelliJ IDEA [22]

Table 11: Tools that detect *null* pointer dereference

Table 11 shows the list of tools that detect *null* pointer dereference, that we can use to compare and understand the different technologies that are currently used. A first observation of these tools shows that they are globally using similar technology, with different level of efficiency. Our tool is however implementing only a fraction of these technology, mainly due to the fact that we did not target them in the first place since we tried to not become too complex. The next parts will discuss the main features that are used by others tools, for what this technology is good for, and if we can implement it on *SLang*.

7.1.1 Interprocedural

Our current checker is only supporting intraprocedural analysis, going further is obviously a way to find more issue, since it would enable us to learn belief from arguments not only inside one function, but also outside it. 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 a real challenge. 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 assigned 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 7.4. There is multiple other ways to perform interprocedural analysis, that represents more precisely the execution flow, however the ideas are complex and will not be described in this work.

7.1.2 Requires the build

Requiring the build can 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 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 requiring to have access to the binaries of the original language does not make a lot of sense, since it will contradict everything that is already in place. 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.

7.1.3 Guided by annotation

We have seen in section 7.1.1 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 problems. 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

```
1 @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 with-

out further checking.. For the parameters, it enforces that f can give *null* as a first parameter, but not for the second one.
They are multiple flavor on how to do it, depending on what 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 seem to be the most popular way to detect *null* pointer exception currently, especially for interactive tools, it enables to detect most of the exceptions 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 an old 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 check 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 checks 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 interprocedural analysis.

7.1.4 Path sensitivity

Currently, our tool is using flow sensitive analysis, meaning that we are only interested in the order in which the statements are executed. In addition, path-sensitivity computes and keeps additional information, based on 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. For our purpose, we could also learn the value of a given pointer with assignment for example.

Listing 28: Simple example of null pointer exception

```
1 Object p = null;
2 p.toString();
```

Listing 28 shows a simple code that obviously raise an exception. Our checker is currently not able to detect it, since it does not try to know the current value of a pointer. Path sensitive tools would be able to find this kind of issues. In addition, we could also use the path sensitivity to improve the *MAY* analysis introduce in subsection 3.4.1, to 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 seem to be possible.

7.2 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*.

7.2.1 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 the custom function that perform the *null* check.
Having configurable setting for a check 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 checks.

7.2.2 Error prone: Null away

Null away is a tool built on top of error prone. To perform *null* pointer consistency, 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 increasing complexity, skipping costly part, like the creation of the control flow graph, if it is possible. 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 prevent us to make bad decisions.

7.3 In-depth comparison: SpotBugs

SpotBugs [19] is the successor of Findbugs [23], an open-source static analysis tool, it implements multiple checks related to *null* dereference, it is therefore a good candidate to have a more complete comparison with.

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

Table 12: SLang and Spotbugs comparison on open-source projects

Table 12 shows the number of issues reported by the two tools on more than hundred open-source projects, with the setup described in section 6.1.

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.

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 13: Examples of rules reported by SpotBugs

Table 12 shows a subset of more than 30 checks related to *null* pointer dereference that are reported by SpotBugs. For our purpose, we are only interested by the checks that are labeled as correctness, as they represent 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 experiments because SpotBugs was crashing during the analysis of 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, and during the experiments presented in section 6.3.1, our plugin did not experience a single crash on a huge amount of file! This is particularly good: if we want to introduce our tool 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 corresponding to 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 are not missing any obvious issues.

While we would want to compute the intersection automatically, this number has to be computed by hand, since fully automatically computing the intersection is not a trivial task [24]. First, due to the fact that SpotBugs works on byte-code, we can not rely on the positions (even the line) of the reported issues reported by the tool. This problem is even worth since the tool seem to report the issues in an inconsistent way, sometimes in a check

for *null*, or at the line where 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 all of them into the comparison, we greatly increase the chance to have unrelated issues reported in the file.
One surprising observation is the number of issues, SpotBugs is reporting 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*, 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 by our tool.

7.4 SpotBugs specific features

The first reason of this huge difference is the difference described previously in subsection 7.1. In addition, we will look more in-depth into one additional feature that is implemented in SpotBugs and producing a big difference, and see if it may be implemented on *SLang*.

1. *Summary-based interprocedural analysis*
   This is an smart and easy first step to perform interprocedural 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 a parameter could be *non-null* to then report if *null* is passed as his argument. We can have this information by looking at annotation. If the annotation are not present, we can still perform intraprocedural analysis to infer 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 nodes and data required to build the summary. The feature currently missing in *SLang* is the method reference. 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 reference that we faced during in the previous parts, naive solution exist, but proper semantics have to be completed to obtain interesting results.

```
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

Listing 29: Example of true positive and false negative of SpotBugs

```
1 B b = new B();
2 b.foo(null); // True positive
3
4 A a = new B();
5 a.foo(null); // False negative
```

In listing 29, we have an example of a true positive and a false negative from SpotBugs. At line #2, the issues is correctly reported, the tool manages to identify statically that the pointer b is called with type B. At line #4 however, the tool is not reporting any issue. This is due to the fact that the type of a is B, the tool is not able to identify the potential run-time type of the variable.
Obtaining equivalents results does not require complex features, there is no strong push-back to implement it on *SLang*, and could already greatly increase the number of issues reported by *SLang*!

8 Related work

8.1 Micro grammar

How to Build Static Checking Systems Using Orders of Magnitude Less Code [9] 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 implement a checker that is order of magnitude smaller than typical systems. They results is encouraging, they manage to find hundreds of issues with an acceptable false positive rate, even 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.

9 Future work

9.1 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 [25].
Rules inference tries to solve the problem that programmers use a wide range of functions for the same goal, identifying which function does what is typically a cumbersome process that needs 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 make 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 during an execution [24], or to use an idea related to the belief style [10] 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 this example, the way a programmer typically deal with them is dependent of the language, and even dependent of the project!
This technique would enable us to find issues without knowing what is correct, everything in a language agnostic way!

9.2 Benchmarks

In section 7, 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 necessary exhaustive. To complement this work, it makes sense to test it against benchmarks, that aims to test as much situations as possible, to test with benchmark that uses languages specific features, like callback, high-order

and all the features that we have discussed in section **??**.

9.3 Improving the checker

The work presented under section 7 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 7.3 showed us that we can already greatly increase the number of true positive, without any complex feature and expected problem.

10 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.

References

[1] *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).

[2] *Grammar of Slang.* Mar. 2019. URL: https://github.com/SonarSource/slang/blob/master/slang-antlr/src/main/antlr4/org/sonarsource/slang/parser/SLang.g4 (cit. on p. 8).

[3] *Langauge API of Slang.* Mar. 2019. URL: https://github.com/SonarSource/slang/tree/master/slang-api/src/main/java/org/sonarsource/slang (cit. on p. 8).

[4] 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).

[5] 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 pp. 13, 50).

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

[7] *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. 18).

[8] *SonarJava, static code analyser for Java language.* Mar. 2019. URL: https://github.com/SonarSource/sonar-java (cit. on pp. 18, 50).

[9] 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. 18, 58).

[10] 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. 19, 59).

[11] *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. 25).

[12] *SonarPHP, static code analyser for PHP language.* Mar. 2019. URL: https://github.com/SonarSource/sonar-php (cit. on p. 29).

[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. 39).

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

[15] *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).

[16] *Public Git Archive: sourced repositories from GitHub with more than 50 stars.* Mar. 2019. URL: https://pga.sourced.tech/ (cit. on p. 45).

[17] *Kotlin documentation: Extension Functions.* Mar. 2019. URL: https://kotlinlang.org/docs/reference/extensions.html#extension-functions (cit. on p. 46).

[18] 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. 48).

[19] *SpotBugs: tool to detect bugs in java code. Successor of FindBugs.* Mar. 2019. URL: https://spotbugs.github.io/ (cit. on pp. 50, 54).

[20] *Static analysis to detect bugs in Java and C/C++/Objective-C.* Mar. 2019. URL: https://fbinfer.com/ (cit. on p. 50).

[21] *Static analysis tool for Java.* Mar. 2019. URL: https://errorprone.info/ (cit. on p. 50).

[22] *Static analysis tool for Java.* Mar. 2019. URL: https://www.jetbrains.com/idea/ (cit. on p. 50).

[23] *FindBugs: tool to detect bugs in Java code. University of Maryland.* Mar. 2019. URL: http://findbugs.sourceforge.net/ (cit. on p. 54).

[24] 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. 55, 59).

[25] 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. 59).