TÉCNICO LISBOA

Towards Automated Checking of Input Data Usage with Facebook Infer

Rui Filipe da Silva Ferreira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João Fernando Peixoto Ferreira
Prof. Rui Filipe Lima Maranhão de Abreu

Examination Committee

Chairperson: Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Prof. José Faustino Fragoso Femenin dos Santos

January 2021

Acknowledgments

I would like to thanks my parents and my sister for the endless wisdom, guidance and motivation through my academic life. In every moment of struggle they were there with words of encouragement and ready to listen to my complaints, fears and doubts. This project could not have been done without their support.

I would also like to thank my dissertation supervisors Prof. João Ferreira and Prof. Rui Maranhão for the motivation to keep going when I had doubts and for the different perspective they gave to this project.

Last but not least, to all my friends and previous colleagues that helped me grow in the face of adversity and for their patience and wisdom in the times I needed most.

To each and every one of you – Thank you.

Abstract

In modern, data-intensive applications the use and modification of input data is very frequent. During the various transformations that the data suffers, parts can remain unused due to programming errors. These errors can be hard to detect and locate due to the high amount of data transformations, and can have real-life consequences. In this project we propose the implementation of a recent analysis proposed by Urban and Müller as an analysis for Facebook Infer, a popular static program analyser for Java, C, C++ and Objective C. We show that our prototype can effectively identify input data usage errors in the same benchmark used by related work.

Keywords

Data Usage; Static Analysis; Abstract Interpretation; Software Reliability.

Resumo

Em aplicações modernas e com uso intensivo de dados, o uso e modificação de dados de entrada é muito frequente. Durante as várias transformações que os dados sofrem, partes podem permanecer inutilizadas devido a erros de programação. Estes erros podem ser difíceis de detetar e localizar dado o elevado número de transformações de dados, podendo ter consequências na vida real. Neste projeto propomos a implementação de uma análise recente proposta por Urban e Müller na forma de uma análise no Facebook Infer, um popular analisador de programas estático para Java, C, C++ e Objective C. Demonstramos que o nosso protótipo deteta com eficácia erros de utilização de dados de entrada no mesmo benchmark usado no trabalho relacionado.

Palavras Chave

Uso de Dados; Análise Estática; Interpretação Abstrata; Fiabilidade de Software.

Contents

List of Figures

List of Tables

1

Introduction

Contents

1.1 Motivation

Data science applications normally deal with considerable amounts of input data that go through long pipelines of processes such as data acquisition, data cleansing, and data preparation. As data is processed, programming errors can cause parts of the input data to remain unused, leading to incorrect or unexpected outputs. The presence of such errors is difficult to detect because the errors are usually subtle and do not raise any compilation errors or warnings. Moreover, the results produced by these applications are usually plausible. The large amount of data transformations that can occur during program execution also hinders detection of these errors.

As pointed out by Urban and Müller [1], a real-world example that shows how input data usage errors can have nefarious societal consequences is that of the paper "Growth in a Time of Debt", published by economists Reinhart and Rogoff in 2010. This paper analyzed the correlation between economic growth, inflation and external debt, using data from forty four countries across two hundred years. However, it was later found by economists Herndon, Ash and Pollin that data from five countries was unintentionally excluded from the analysis due to a programming error [2]. It is worth noting that critics of this paper believe that it has led to unnecessary adoption of austerity policies in several countries [3].

To address this, Urban and Müller recently proposed the first static analysis, based on abstract interpretation, capable of detecting input data usage errors [1]. They implement their analysis in a research prototype, Lyra, that supports a subset of Python. We propose to adapt and implement their analysis for Facebook Infer's Intermediate language *SIL*. Infer is an industrial-strength static program analysis tool for Java, C, C++, and Objective C [4].

1.2 Hypothesis

Our main hypothesis is that we can develop an input data usage analysis, similar to Urban and Müller's analysis [1], that works for Infer's intermediate language *SIL*. Implicit in the main hypothesis are the sub-hypotheses that we can create an analysis that is useful in finding input data usage errors in real-world programs.

1.3 Research Questions

The main research questions addressed by our work are:

RQ1. Is it possible to implement an input data usage analysis similar to Urban and Müller's analysis [1] that works with the static analysis tool Infer?

RQ2. Can such an analysis work for several programming languages?

RQ3. Can such an analysis be used to analyze real code and find potential errors?

1.4 Contributions

The contributions of this paper are the following:

- Definition of an input data usage analysis similar to Urban and Müller's analysis [1] for Infer's intermediate language *SIL*.
- Implementation of the analysis as an Infer checker. At the time of writing, our implementation only supports the analysis of Java programs. Java is extensively used in data analysis programs [5–8] and has the potential to be used in data science programs since it is one of the most used programming languages [1].
- Evaluation of the analysis with a benchmark specifically created to test data input usage analyses and with real-world *Java* programs.

1.5 Organization of the Document

This thesis is organized as follows: Chapter 2 reviews the research done revelant to our work, and the projects aiming to solve the problem that we address.

Chapter 3 presents the solution that we developed, starting with an overview of our approach in Section 3.1, proceeding into the more technical details in Sections 3.2, 3.3, and 3.4. The chapter ends with the details of the implementation and its limitations.

Chapter 4 shows the results of our implementation for a benchmark specifically created to test input usage analyses. We also present results of executing our analysis in real-world Java programs.

Chapter 5 gathers our concluding thoughts about the implementation of this analysis, its usefulness, and the prevalence and effects of input data errors in real-world programs.

[1] Tiobe Programming Community Index: https://www.tiobe.com/tiobe-index/

2

Related Work

Contents

The work already done in this field is short since, to the best of our knowledge, only one program was developed that focuses specifically in input data usage errors in programs analyzing data. Another two tools were developed but are focused towards spreadsheet applications and on errors in the data, rather than the code that analyzes the data. We are going to start with by reviewing the two tools developed for spreadsheets, and conclude with the tool which our approach is based on.

2.1 Data Debugging for Spreadsheets

Barowy et al. [9] developed a data debugging tool, $CHECKCELL$, that "combines program analysis and statistical analysis to automatically find potential data errors". Barowy focused this tool on spreadsheets programs because of their intensive data use, and their extensive use of formulas that propagate an input error into possibly multiple other variables. The tool works as a Microsoft Excel add-in and works by identifying "cells that have an unusually high impact on the spreadsheet's computations".

The basis for this tool is that if some input data has a "disproportionate impact on the computation", then it is either wrong or very important data. Barowy explains that "a computation is not likely to be correct if the input data is not correct", and that even though there are a plethora of tools to find programming errors, there are few that look for data errors. The tools traditionally used by programmers to validate input are precise specifications, which are hard to define and fail to detect subtle errors, *data cleaning* which "copes with errors via cross-validation with ground truth data, which may not be present", and *statistical outlier detection* which typically "reports data as outliers based on their relationship to a given distribution (e.g., Gaussian)", considering that a valid input distribution is hard to find. Additionally a input data error does not need to have a different distribution for it to cause program errors.

Input data errors have 3 main types [10]:

- Data Entry Errors: Typographical and transcriptions errors.
- Measurement Errors: Data source error or malfunction.
- Data Integration Errors: Inconsistencies due to mixing of different data, for example unit of measurement mismatch.

This tool ranks inputs by the degree to which they influence the output. The author calls this approach "*Data Debugging*". One disadvantage of this approach is that the tool does not focus on finding small errors that do not cause a major effect on the output, instead focusing only on the big impact errors.

More specifically this tool starts by building a data dependence graph for the computations, and then analyzes data impact by "randomly re-sampling data items with data chosen from the same group (e.g., a range in a spreadsheet formula) and observing the resulting changes in computations that depend on

that data". In other words, $CHECKCELL$ tests if the replacement of a value by other in the same input vector causes big changes to the output.

2.2 Static Analysis of Spreadsheet Applications

Rival and Cheng et al. [11] created a static analysis in order to detect type-unsafe operations, like comparing integer values with string values in spreadsheets at run-time. The author concluded that this happens because spreadsheets use a weak type system, which mostly does not consider a type mismatch an error and because "data and formulas can be dynamically set, read or modified", allowing for this errors to happen while not generating an error for the user.

The analysis uses abstract interpretation to reduce the spreadsheet information (for example content types) into only the necessary properties and ties them into zones in spreadsheet tables. This "infers invariants by conservative abstract interpretation of spreadsheet applications". It also lets users choose which behaviors are deemed unsafe and should be reported, making it a more flexible tool.

Abstract Interpretation is a:

> *"theory of semantics approximation that is used for the construction of semantic-based program analysis algorithms" [12].*

In practice it abstracts a concrete set of properties (concrete domain) from a program into an abstract set of properties (abstract domain), reducing the amount of information used in an analysis. The state of the program at any point is called abstract state, which changes via a set of transfer functions, according to the program instructions.

2.3 Input Data Usage in Python

Urban and Müller et al. [1] developed a tool, Lyra[1] that uses static analysis based on abstract interpretation [13] to detect unused input in Python programs, a language extensively used in Data Science applications. The tool has two ways of analysing programs. One is based on *Syntactic Dependencies Between Variables*, and the other is based on *Strongly Live Variable Analysis*. Both analyses works backwards, starting the analysis in the last line of code of a program.

Strongly Live Variable Analysis is reliable for data usage in both terminating and non-terminating programs. Its downfall is that it is imprecise in terms of implicit dependencies between program variables. This analysis determines that a variable is strongly live if used in a statement other than an assignment, or if used to define other strongly live variables in an assignment.

[1] Lyra's webpage:https://caterinaurban.github.io/project/lyra

Syntactic Dependencies between Variables is more precise than the previous method. This method is based on monitoring if a variable influences another variable. For this the program examines the possible interactions between variables and gives them a classification, so that in the end of the analysis, the classification of the input variables determines if those have been used. Additionally, to capture implicit dependencies between program variables "...from variables appearing in boolean conditions of conditional and while statements, we track when the value of a variable is used or modified in a statement based on the level of nesting of the statement in other statements." [1]. In other words, the idea behind using the syntactic dependency between variables is to examine the possible interactions between variables and give them a classification, so that in the end of the analysis, the classification of the input variables determines if those have been used.

The analysis abstract state consists of a classification for each variable in the program. The possible classifications in Urban and Müller's analysis are: "Used" (U), "Not Used" (N), "Overwritten" (W) and "Below" (B). "Overwritten" signifies a variable that was previously used but was re-written, and "Below" a variable that was used at a lower nesting level.

Taking into account the classifications of all variables inside an instruction, the interaction between variables and the type of instruction being executed, the analysis will update each variable classification according to the transfer function Θ_Q that transforms an abstract state q as follows:

$$\begin{aligned}
\Theta_Q[\![skip]\!](q) &\stackrel{\text{def}}{=} q \\
\Theta_Q[\![x = e]\!](q) &\stackrel{\text{def}}{=} \mathsf{ASSIGN}[\![x = e]\!](q) \\
\Theta_Q[\![\text{if } b : s_1 \text{ else: } s_2]\!](q) &\stackrel{\text{def}}{=} \mathsf{POP} \circ \mathsf{FILTER}[\![b]\!] \circ \Theta_Q[\![s_1]\!] \circ \mathsf{PUSH}(q) \\
&\quad \sqcup_Q \mathsf{POP} \circ \mathsf{FILTER}[\![b]\!] \circ \Theta_Q[\![s_2]\!] \circ \mathsf{PUSH}(q) \\
\Theta_Q[\![\text{while } b : s]\!](q) &\stackrel{\text{def}}{=} \mathsf{lfp}_q^{\sqsubseteq_Q} \Theta_Q[\![\text{if } b : s \text{ else : skip}]\!](q) \\
\Theta_Q[\![s_1\ s_2]\!](q) &\stackrel{\text{def}}{=} \Theta_Q[\![s_1]\!] \circ \Theta_Q[\![s_2]\!](q)
\end{aligned}$$

The rule for assignments stipulates that a variable is considered "U" if it is utilized inside an assignment to another variable already considered "U" or "B". In that case, the second variable classification changes to "W" unless it is also present in the first variable, in which case it remains with the same classification. More formally, it is defined as:

$$ASSIGN[\![x = e]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} W & y = x \wedge y \notin \mathsf{VARS}(e) \wedge m(x) \in \{U, B\} \\ U & y \in \mathsf{VARS}(e) \wedge m(x) \in \{U, B\} \\ m(y) & \text{otherwise} \end{cases}$$

Note that if the variable being assigned does not have one of these classifications, then both variables remain unchanged.

Another way a variable can be classified as "U" is if it appears in the Boolean condition of a statement that uses or modifies another variable already classified as "U". This is captured in the definition of the $FILTER$ function, where *X* is the set of all variables:

$$FILTER[\![e]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} U & y \in \text{VARS}(e) \wedge \exists y \in X : m(y) \in \{U, W\} \\ m(y) & \text{otherwise} \end{cases}$$

Informally, the $FILTER$ function searches the domain for variables that are classified as "U" or "W". If it finds any variables in those conditions, then the variables present on the conditional statement get classified as "U". Otherwise, the current classifications remain unchanged.

Because of the nesting level changes and in order to detect classification changes inside them, the analysis maintains a stack of maps that "... grows or shrinks based on the level of nesting of the currently analyzed statement" [1]. In words, the $PUSH$ function changes the classification of all variables "U" or "W" so that the $FILTER$ function can capture any new variables being classified as "U" or "W" inside a conditional set of instructions.

$$PUSH(\langle m_0, m_1, ..., m_k \rangle) \stackrel{\text{def}}{=} \langle INC(m_0), m_0, m_1, ..., m_k \rangle$$

$$INC(m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} B & m(y) \in \{U\} \\ N & m(y) \in \{W\} \\ m(y) & \text{otherwise} \end{cases}$$

The POP function reverses the changes that $PUSH$ does, but only to the variables that have not been changed since. It is defined as:

$$POP(\langle m_0, m_1, ..., m_k \rangle) \stackrel{\text{def}}{=} \langle DEC(m_0, m_1), ..., m_k \rangle$$

$$POP(m, k) \stackrel{\text{def}}{=} \lambda y. \begin{cases} k(y) & m(y) \in \{B, N\} \\ m(y) & \text{otherwise} \end{cases}$$

Note that the POP function is used to return the variable classifications to a regular state, after an analysis of a conditional set of instructions finishes.

The results from the analysis are presented in a *Control Flow Graph* that presents the abstract domain (composed of every variable present in the program and its classification) in each node with the classification of each variable after each instruction analyzed.

3

Solution

Contents

In this Chapter, we describe the approach that we followed to implement an analysis that detects input data usage errors. We start with an overview and then: we describe *SIL*, the intermediate language of Infer on which we defined the analysis; we describe the abstract domain used and the transfer functions defined; and we present relevant details of our implementation.

3.1 Overview

Our work is based on the analysis originally developed by Urban and Müller [1]. Their analysis is implemented in a tool called Lyra[1]. The analysis is based on syntactic dependencies between variables and abstract interpretation, and it works backwards, starting the analysis in the last line of code of a program. The idea behind using the syntactic dependency between variables is to examine the possible interactions between variables and give them a classification, so that in the end of the analysis, the classification of the input variables determines if those have been used.

Possible variable classifications. The analysis classifies program variables into different categories. The possible classifications in Urban and Müller's analysis are: "Used" (U), "Not Used" (N), "Overwritten" (W) and "Below" (B). "Overwritten" signifies a variable that was previously used but was re-written, and "Below" a variable that was used at a lower nesting level. Our work introduces two additional classifications: "Below (Used before Push)" (BU) and "Not Used (Overwritten before Push)" (NW). We describe in Section 3.4 the reason for the creation of these new classifications and what they represent.

Abstract domain. To simplify presentation, in this Chapter we use as abstract domain a map that associates the name of each variable in the program with its classification and its location (for error message purposes). Later, we discuss how we extended the abstract domain to simplify the implementation of the analysis. At the start of the program analysis, the abstract domain consists of every input variable being classified as "N", and the output variables "U". The additional variables used during the program are added to the domain when they are first encountered by the analysis with the classification of "N".

Programming language. Urban's approach, as presented in Section 2.3, tracks instructions for assignment, conditional method bodies and composition of instructions. Since Infer translates the program being analyzed into *SIL* intermediate language our analysis needs to focus on how each instruction for assignment, conditional method bodies and composition of instructions are translated into *SIL*.

Chapter organization. In the Section 3.2 we discuss how Infer converts the program being analyzed and what are the main structures of *SIL*. Some changes to the *Abstract Domain* used were done to

[1] Lyra's webpage: https://caterinaurban.github.io/project/lyra

overcome Infer's limitations and are explained in Section 3.3. The resulting transfer functions in Section 3.4 represent a way to accurately detect the same types of instructions that Urban's solution does, based on the structure of the converted *SIL* code. Finally, in Section 3.5 we will look into the specifics of the analysis implementation.

3.2 Infer Framework

We implemented the analysis as a checker for Facebook's Infer[2], which is a static analysis tool that currently supports Java, C, C++ and C-Objective [4]. The implementation is written in OCaml. Infer supports *javac* and *clang* compilers, as well as *ant, buck, cmake, gradle, make, maven, xcodebuild* and *xctool* build systems. Additionally Infer supports the *gcc* compiler, but it will use *clang* internally to compile the code.

Infer provides a framework, Infer.AI[3], that enables the creation of analyses based on abstract interpretation. This framework substantially simplifies the task of creating analysis for different programming languages. As Infer translates the code analyzed into its Intermediate Language *SIL*, which is used in the analysis instead of the original code, one only needs to create one checker for all the languages supported by Infer. Moreover, the checker will be readily available to any new language supported by Infer.

Infer's workflow has two main phases:

1. Capture Phase: In this phase Infer translates the files under analysis into Infer's internal intermediate language, *SIL*;

2. Analysis Phase: This is the phase where each function and/or method is analyzed.

During the analysis phase, Infer uses Pure Variables, which only appear in the *SIL* representation, and Program Variables, which are the variables that appear in the source program under analysis. This means our analysis needs to register the relations between them in order to understand, for example, which Program Variable is being used in an assignment that has Pure Variables. Because of this the execution of some instructions may need to be delayed, and the structures to support this are shown in Section 3.3 as well as some examples.

As mentioned previously, Infer converts the program being analyzed into an Intermediate Language called *SIL*, by capturing commands during compile time. Additionally, it generates a *Control Flow Graph* whose information is used in the abstract domain of our solution. The *SIL* code is then analyzed by Infer's checkers, creating the possibility that a single analysis can work for all languages currently and future supported by Infer. Depending on the specifics of the analysis and how Infer works, not every

[2]Facebook Infer: https://fbinfer.com.
[3]Building checkers with the Infer.AI framework: https://fbinfer.com/docs/absint-framework.

analysis can support all languages Infer supports without adapting the analysis for each language. Our analysis does not support all languages that Infer supports because it requires explicit support to detect each type of input and output for each language.

The intermediate language SIL. In order to understand what instructions *SIL* translates the programs to, and what these do, there is a need to understand the mains structures of *SIL*:

- Ident: Pure Variables used in instructions, carrying information from Program Variables.
- Pvar: Program Variables that correspond to variables used in the original code being analyzed. Also mentioned as Lvar in Expressions.
- Var: Single abstraction for all variable types in *SIL*.
- Exp: Expressions in *SIL* that contain a sequence of variables, either Program Variables or Pure Variables
- Typ: Types of variables in *SIL*.
- Location: Location of an instruction. Carries the line, columns and file from the original program being analyzed.

The programs analyzed and translated into *SIL* are converted into five types of instructions, each specified as follows:[4]

Load: `Load of {id: Ident.t; e: Exp.t; root_typ: Typ.t; typ: Typ.t; loc: Location.t}`

The definition states the name of the variable, followed by the type of its variable (e.g., `id: Ident`). Please note that even though it is still included in the definition, "root_typ" is deprecated and will be removed in future versions of Infer.

The *Load* instruction loads a value (carried inside the expression *"e"*) into an Identifier (*"id"*). The instruction also detects the type of the expression and the identifier (in *"typ"*), as well as the root type of the expression (in *"root_typ"*) and location of the instruction (in *"loc"*).

Store: Next is the definition of the Store instruction:

```
Store of {e1: Exp.t; root_typ: Typ.t; typ: Typ.t; e2: Exp.t; loc: Location.t}
```

The *Store* instruction stores the value on an expression *"e2"* into another expression *"e1"*.

Like the *Load* instruction it also detects the type of both instructions and its location.

[4] Infer SIL Github: https://github.com/facebook/infer/blob/master/infer/src/IR/Sil.mli

Prune: `Prune of {exp: Exp.t; loc: Location.t; bool: Boolean; if_kind: if_kind}`

The *Prune* instruction prunes the state of the expression *"exp"* based on if the boolean *"bool"* is *True* or *False*. It also detects the type of prune instruction it is *"if_kind"* (for example *"for"*, *"if"*, *"while"*).

Call: `Call of {(ret_id: Ident.t; ret_typ: Typ.t); e_fun: Exp.t; arg_ts: (Exp.t; Typ.t) list; loc: Location.t; call_flags: CallFlags.t}`

The *Call* instruction represents a `[ret_id = e_fun(arg_ts)]` instruction, that calls a function in *"e_fun"*, whose arguments are *"arg_ts"*, and stores its value in *"ret_id"*.

Metadata: `Metadata of {instr_metadata}`

The *Metadata* instruction carries additional information, not strictly necessary for understanding the program semantics, like information about the original syntactic structure.

In order to understand how Infer translates programs into *SIL*, consider the following *Java* example from our set of benchmarks, created to test the analysis functioning with array accesses:

```
1  class Dict_single_case {
2          int[] test1 ( ){
3                  int v1 = 0;
4                  int[] d = new int[5];
5                  d[1] = v1;
6                  return d;
7          }
8  }
```

Infer translates the instruction in line 6 into the following *SIL* instructions, which are shown backwards, which is the order the analysis evaluates them:

```
INSTR=  *&return:int[_*_](*)=n$3 [line 6];
INSTR=  n$3=*&d:int[_*_](*) [line 6];
```

The first *SIL* instruction is a *Store* instruction between the *Program Variable* "*return*" and the *Pure Variable* "*n$3*". The second instruction is a *Load* instruction of the *Program Variable* "*d*" into the *Pure Variable* "*n$3*.

Infer translates a single instruction being analyzed into several instructions. It creates *Pure Variables* as value holders until the *Program Variables* are loaded. Our analysis has to keep this in consideration, and keep track of the association between *Pure Variables* and *Program Variables* in order to analyze an instruction correctly.

The transfer functions defined for each of these instructions are presented in Section 3.4, with the exception of the *Metadata* instruction which does not carry information necessary to the creation of this analysis.

3.3 Abstract Domain

The abstract domain used in our implementation is different from the one of Lyra, being composed of three main structures:

- Map (*m*) is the main structure, which holds the information about all the variables in the program regarding its current classification and location. The classifications change during the analysis but the location remains the latest appearance of the variable in the code.
- MapArray (*a*) is used when there is an assignment that accesses an array (e.g., `a=b[c]`). This is needed because the variable being assigned (`a`) is only declared after the assignment, and its classification is needed to determine the classification of the other variables being used (`b,c`). This occurs because the analysis is backwards.
- MapCall (*c*) is used when there is an assignment that contains a $Call$ instruction. This is needed because, given the backwards nature of the analysis, the variable being assigned appears before the assignment and before the $Call$ instruction. This map registers the variable being assigned, and the Pure Variables being assigned to that variable through the program. This ensures that when the assignment arises, the analysis knows what variable is being used.

Additionally, a structure called cfg_node (*node*) is also present in the abstract domain, holding information about the node currently being analyzed, that is used to detect nesting level changes in the following instruction.

3.3.1 MapArray

Considering the example code given in the previous Section 3.2, we are going to analyze the line 5 in order to understand the creation and function of MapArray (a):

```
d[1]= v1;
```

Infer translates the instruction into the following *SIL* instructions:

```
INSTR 1=  *n$1[1]:int=n$2 [line 5];
INSTR 2=  n$2=*&v1:int [line 5];
INSTR 3=  n$1=*&d:int[_*_](*) [line 5];
```

In "*INSTR 1*", SIL represents a $Store$ instruction between two pure variables ("*n$1*" and "*n$2*"), while "*INSTR 2*" and "*INSTR 3*" represent $Load$ instructions from the *program variables* (in this case "*v1*" and "*d*") to the *pure variables* used in the previous instruction.

In order for the analysis to execute the assignment of this array access, the relations between the *Program Variables* "*n$1*" and "*n$2*" are stored in MapArray (a), being "*n$1*" the key and "*n$2*" the corresponding value. When the analysis evaluates the second instruction, it replaces the value "*n$2*" for "*v1*". Finally when the third instruction is evaluated, the analysis now associates the *Pure Variable* "*n$1*" with the *Program Variable* "*d*", and executes the necessary instruction for the assignment between "*d*" and "*v1*".

If the instruction was to use another variable to determine which part of the array to access (e.g., `d[a]=v1`) then the array associates the same "*n$1*" as key to both *Pure Variables* representing "*a*" and "*v1*".

Another use for MapArray (a) is presented in the next subsection 3.3.2, as part of the solution to interpret function calls correctly.

3.3.2 MapCall

In order to understand how MapCall (c) functions, consider now the following example from the benchmark "*Dict_example_container*":

```
1  import java.util.HashMap;
2  class Dict_example_container {
3          int test1 (int value){
4                  HashMap <String, Integer> example = new HashMap <String, Integer>();
5                  example.put ("a", 0);
6                  example.put ("b", 1);
7                  example.put ("c", 2);
8                  String key = "b";
9                  example.put (key, value);
10                 int i = example.get("a");
11                 return i;
12         }
13 }
```

The instruction in line 9 of the code is translated into *SIL* instructions that follows, displayed in the reverse order, as the analysis evaluates them:

```
INSTR=  *&$irvar7:java.lang.Object*=n$23 [line 9];
INSTR=  n$23=_fun_Object HashMap.put(Object,Object)(n$19:java.util.HashMap*,
    n$21:java.lang.Object*,n$22:java.lang.Integer*) virtual [line 9];
INSTR=  n$22=*&$irvar6:java.lang.Integer* [line 9];
```

```
INSTR=  n$21=*&key:java.lang.Object* [line 9];
INSTR=  _=*n$19:java.util.HashMap*(root java.util.HashMap) [line 9];
INSTR=  n$19=*&example:java.util.HashMap* [line 9];
INSTR=  *&$irvar6:java.lang.Integer*=n$18 [line 9];
INSTR=  n$18=_fun_Integer Integer.valueOf(int)(n$17:int) [line 9];
INSTR=  n$17=*&value:int [line 9];
```

As this example demonstrates, a single *Call* instruction can generate several instructions in *SIL*. The analysis needs to go through each instruction and maintain an updated list of associations between *Pure Variables* and *Program Variables*. In this particular case, the *SIL Call* instruction is the second one, but there is a *Store* instruction before that, with the *Program Variable* "*irvar7*" that *SIL* creates. In this first instruction the analysis will add to MapCall (c) "*irvar7*" as key and "*n$23*" as value.

After that, in the second instruction, the analysis checks MapCall (c) to see if "*n$23*" is a value and, if so, the key associated with it is classified as "*U*" in Map (m). Finally, it uses MapArray (a) in order to keep the information of the relation of "*n$19*", "*n$21*" and "*n$22*" until the *Program Variable* associated with "*n$19*" is loaded so the assignment can be executed properly.

This approach helps differentiate between an instruction like in line 9 and an instruction like in line 10, because in that case the first instruction is not called "*irvar_*" but rather "*i*" which is the *Program Variable* that is going to store the result from the execution of the *Call* instruction. It is important to distinguish between these two types of calls because in the case of *line 9* the analysis treats it like an assignment between "*example*" and the variables used inside the function call ("*key*" and "*value*"), all of which are present inside "*arg_ts*", but in the case of line 10 the assignment is between "*i*" (outside of "*arg_ts*") and "*example*" and its variables used inside.

Additionally, MapCall (c) is used in instructions like `v1=d[a]` where there is an assignment and an array access on the right side. This is needed because the instruction that associates "*v1*" with a *Pure Variable* appears before the assignment instruction (that used the *Pure Variable*). In this case MapCall (c) stores the relation between "*v1*" and a *Pure Variable*, and checked for matches when an assignment with array access is detected.

3.4 Transfer Functions

Taking into account the classifications of all variables inside an instruction, the interaction between variables and the type of instruction being executed, the analysis will update each variable classification according to the transfer function Θ_Q that we define in this Section. The function is defined by cases on *SIL* instructions and it transforms the abstract state q. Each abstract state is tuple $(m, c, a, node)$, where m is Map that contains the classification of each variable, c is $MapCall$ that contains the relation

between variables in a $Call$ instruction, a is $MapArray$ that contains the relation between variables in an assignment with access to an array, and $node$ is cfg_node that contains the information needed to detect changes in nesting levels.

3.4.1 Load

For the $Load$ instruction, the transfer function is defined as:

$$\Theta_Q[\![\text{Load } id : e]\!](q) \stackrel{\text{def}}{=}$$
$$\begin{cases} \text{ASSIGN}[\![\text{KOV}[\![id]\!](c) = e]\!] \circ \text{CK}[\![id]\!] \circ \text{CK}[\![e]\!](m) & e \in \text{LINDEX} \\ \text{LOAD\_PVAR}[\![id, e]\!](q) \circ \text{CK}[\![id]\!] \circ \text{CK}[\![e]\!](m) & e \in \text{PVAR} \wedge id \in a \\ \text{LOAD\_GENERAL}[\![id, e]\!](q) \circ \text{CK}[\![id]\!] \circ \text{CK}[\![e]\!](m) & \text{otherwise} \end{cases}$$

The $Load$ instruction has different approaches depending on the type of the variable "id" and the variables inside expression "e". In every $Load$ instruction the analysis starts by executing the function CK (CHECK), defined in Equation 3.1, that verifies if both variables are present in Map (m), and if they are not it adds them to Map (m) with the classification "N". This also happens with $Store$ and $Call$ instructions.

$$CK[\![e]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} N & x \notin m \wedge y \in \text{VARS}(e) \\ m(y) & \text{otherwise} \end{cases} \tag{3.1}$$

Case 1. If the variable "e" is an *Array Index Offset* (LINDEX) the analysis performs an assignment between the key matching the "id" value in MapCall (c), and the variable "e"; this assignment is achieved using the function $ASSIGN$, defined in Equation 3.2. These instructions capture an instruction `a=b[c]` that accesses an array during a $Load$ instruction.

The rule for assignments stipulates that a variable is considered "U" if it is utilized inside an assignment to another variable already considered "U", "B" or "BU". In that case, the second variable classification changes to "W" unless it is also present in the first variable, in which case it remains with the same classification. More formally, it is defined as:

$$ASSIGN[\![x = e]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} W & y = x \wedge y \notin \text{VARS}(e) \wedge m(x) \in \{U, B, BU\} \\ U & y \in \text{VARS}(e) \wedge m(x) \in \{U, B, BU\} \\ m(y) & \text{otherwise} \end{cases} \tag{3.2}$$

Note that if the variable being assigned does not have one of these classifications, then both variables remain unchanged.

$$KOV[\![x]\!](c) \stackrel{\text{def}}{=} \lambda y. \begin{cases} z & y = x \wedge c(z) = y \\ y & \text{otherwise} \end{cases} \tag{3.3}$$

The function KOV (Equation 3.3) (*key of value*) accesses MapCall (c) and returns the key variable assigned to variable "x". If there is no key associated to the variable "x", then it returns that variable.

Case 2. If the variable "e" is a *program variable* and "id" is in MapArray (a), then the function $LOAD\_PVAR$ is executed which, depending if the variable "id" is one of the keys of MapArray (a), either executes the $ASSIGN$ function or the $UPDT\_KEY$ function.

$$\text{LOAD\_PVAR}[\![id, e]\!](q) \stackrel{\text{def}}{=} \begin{cases} \text{ASSIGN}[\![e = a(id)]\!](m) & id \in \text{a.keys} \\ \text{UPDT\_KEY}[\![id, e]\!](a) & \text{otherwise} \end{cases}$$

The presence of "id" in the keys of MapArray (a) determines that the analysis located the $Load$ instruction for the key present in MapArray (a), which is the last instruction in an assignment with an array access, like in the example presented before in the beginning of this Chapter. This triggers an assignment between the variable "e" and the values in MapArray (a) whose key is "id".

If "id" is not in the keys of MapArray (a) then it updates the value from "id" into "e" via $UPDT\_KEY$ (UPDATE KEY). This happens to update from *Pure Variables* that the assignment uses to *Program Variables*, which are the real variables used in the assignment, but that are loaded after the assignment instruction is called.

$$UPDT\_KEY\ [\![x, e]\!](a) \stackrel{\text{def}}{=} \lambda y. \begin{cases} e & a(z) = x \\ a(y) & otherwise \end{cases} \tag{3.4}$$

Case 3. If none of the previous conditions are verified then, if the length of expression "e" is not 1, the program updates the classification of "e" with the same classification of "id" (transferring the classification that the pure variable "id" carried into the program variable that it represents "e") using the function $UPDT\_C$ (UPDATE CLASS), defined in Equation 3.5.

$$UPDT\_C[\![x, e]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} m(x) & y \in \text{VARS}(e) \\ m(y) & \text{otherwise} \end{cases} \tag{3.5}$$

This corresponds to the second case of the following definition:

$$\text{LOAD\_GENERAL}[\![id, e]\!](q) \stackrel{\text{def}}{=} \begin{cases} \text{UPDT\_C}[\![id, e]\!](m) \circ \text{UPDT\_VAL}[\![id, e]\!](c) & \text{length}(e) = 1 \\ \text{UPDT\_C}[\![id, e]\!](m) & \text{otherwise} \end{cases}$$

On the other hand, if the length of expression "e" is 1, then it also updates the classification as described above, but before doing it, it updates the value that matched "id" to "e" using the function $UPDTE\_VAL$ (UPDATE VAL), defined in Equation 3.6. This happens to follow the associations between *Pure Variables* and *Program Variables* that happen after a $Call$ instruction, like showed in the beginning of this Chapter.

$$UPDT\_VAL[\![x, e]\!](c) \stackrel{\text{def}}{=} \lambda y. \begin{cases} e & c(z) = x \\ c(y) & \text{otherwise} \end{cases} \tag{3.6}$$

3.4.2 Store

The $Store$ instruction stores an expression "e2" into another "e1". The transfer function for the $Store$ instruction is defined as follows:

$\Theta_Q[\![\text{Store } e1 : e2]\!](q) \stackrel{\text{def}}{=}$

$$\begin{cases} \text{ASSIGN\_HASH}[\![e1, e2]\!](a) \circ \text{CK}[\![e1]\!] \circ \text{CK}[\![e2]\!](m) & e1 \text{ is LINDEX} \wedge e2 \text{ is VAR} \wedge e1 \notin m \wedge e2 \notin m \\ \text{STORE\_RETURN}[\![e1, e2]\!](q) \circ \text{CK}[\![e1]\!] \circ \text{CK}[\![e2]\!](m) & e1 \text{ is LVAR} \wedge e1 \text{ is RETURN} \\ \text{STORE\_LVAR}[\![e1, e2]\!](q) \circ \text{CK}[\![e1]\!] \circ \text{CK}[\![e2]\!](m) & e1 \text{ is LVAR} \wedge e2 \text{ is VAR} \wedge e1 \text{ not RETURN} \\ \text{STORE\_GENERAL}[\![e1, e2]\!](q) \circ \text{CK}[\![e1]\!] \circ \text{CK}[\![e2]\!](m) & \text{otherwise} \end{cases}$$

Just like $Load$, the analysis starts by executing a CK (CHECK) that verifies if both variables are present in Map (m), and if they are not it adds them to Map (m) with the classification "N" (see Section 3.1). The $Store$ instruction has four cases that are described below.

Case 1. If expression "e1" is an *Array Index Offset* (LINDEX), then this is the first instruction of an assign with an array on the left side and a Pure Variable "e2" on the right. When this is detected the analysis uses the function $ASSIGN\_HASH$, defined in Equation 3.7, to add to MapArray (a) the first variable in expression "e1" as key and the remaining variables of "e1" and "e2" as values.

$$ASSIGN\_HASH[\![x, e]\!](a) \stackrel{\text{def}}{=} \lambda y. \begin{cases} a(x) = y & y \in \text{VARS}(e) \wedge x \notin a \wedge e \notin a \wedge x \notin y \\ a(y) & otherwise \end{cases} \tag{3.7}$$

Case 2. If during the $Store$ instruction the expression "e1" is a *return* expression ("*return*" is considered a *Program Variable* by Infer) then the analysis executes the function $STORE\_RETURN$ defined below.

$$\text{STORE\_RETURN}[\![e1, e2]\!](q) \stackrel{\text{def}}{=} \begin{cases} \text{PUSH}(m) \circ \text{PUT\_USED}[\![e2]\!](m) & \text{PREDS}(node) = 2 \\ \text{PUT\_USED}[\![e2]\!](m) & \text{otherwise} \end{cases}$$

The first step is to execute the function PUT_USED, defined in Equation 3.8, that changes the classification of all variables contained in the expression "e2" to "U". If the node (that represents a single Java instruction) has two predecessors ("*PREDS*") then it means that the instruction before (that is going to be analyzed after this because the analysis is backwards) is the beginning of a conditional method body.

If this is the case, then the analysis will also apply the function $PUSH$, defined in Equation 3.9, that changes the classification of all variables "U" or "W" so that the function $FILTER$, defined in Equation 3.11, can capture any new variables being classified as "U" or "W" inside a conditional set of instructions.

$$PUT\_USED[\![x]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} U & y \in \text{VARS}(x) \\ m(y) & otherwise \end{cases} \tag{3.8}$$

$$PUSH(m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} BU & m(y) \in \{U\} \\ NW & m(y) \in \{W\} \\ m(y) & \text{otherwise} \end{cases} \tag{3.9}$$

A remark on using a single map. In order to facilitate our implementation, we deviate sligthly from Urban and Müller and, instead of keeping a stack of maps, we use a single map that is updated throughout the analysis. This changes the possible classifications of a variable, introducing the two new classifications "BU" and "NW", as discussed above. The classification "BU" is used when a variable that was classified as "U" suffers a $PUSH$. The classification "NW" is used when a variable that was classified as "W" suffers a $PUSH$.

We do this because the array introduced by Urban and Müller is only used for the $PUSH$ and POP functions, so the analysis ends up only using the last two classifications of a variable. This change saves the program from having to go through and modify an array for every variable.

Case 3. When a value is stored from a *Pure Variable* into a *Program Variable* we apply the function $STORE\_LVAR$ defined below.

$$\text{STORE\_LVAR}[\![e1, e2]\!](q) \stackrel{\text{def}}{=}$$
$$\begin{cases} \text{PUSH}(m) \circ \text{UPDT\_C}[\![e1, e2]\!](m) \circ \text{CK\_VARS}[\![e1, e2]\!](c) \circ \text{UPDT\_VAL}[\![e1, e2]\!](c) & \text{PREDS}(node) = 2 \\ \text{UPDT\_C}[\![e1, e2]\!](m) \circ \text{CK\_VARS}[\![e1, e2]\!](c) \circ \text{UPDT\_VAL}[\![e1, e2]\!](c) & \text{otherwise} \end{cases}$$

The first step is to execute the function $UPDT\_VAL$, defined in Equation 3.6, that changes the value in MapCall (c) associated with "e1" and replaces it with "e2". This happens in order to keep track of the association between *Pure Variables* and *Program Variables* in a $Call$ instruction that may have been done previously.

Next, the analysis executes the function CK_VARS (CHECK VARS), defined in Equation 3.10, which checks if "e2" is associated with a key in MapCall (c) and if it is not, then it adds "e1" as a key and "e2" as a value.

$$CK\_VARS[\![x, e]\!](c) \stackrel{\text{def}}{=} \lambda y. \begin{cases} e & \exists z \in X : c(z) = e \wedge y \in \text{VARS}(x) \\ c(y) & \text{otherwise} \end{cases} \tag{3.10}$$

Finally the program executes the function $UPDT\_C$ (UPDATE CLASSIFICATION), defined in Equation 3.5, that changes the classification of "e2" into the same classification of "e1". This happens to keep track of the association between *Pure Variables* and *Program Variables* in terms of classification. Just like $STORE\_RETURN$, in case the previous instruction has two predecessors ("*PREDS*"), meaning that the instruction before is the beginning of a conditional method body, the analysis executes a $PUSH$ instruction.

Case 4. If none of the previous conditions are true, then the function $STORE\_GENERAL$ is applied:

$$\text{STORE\_GENERAL}[\![e1, e2]\!](q) \stackrel{\text{def}}{=}$$
$$\begin{cases} \text{PUSH}(m) \circ \text{ASSIGN}[\![e1 = e2]\!](m) \circ \text{UPDT\_VAL}[\![e1, e2]\!](c) & \text{PREDS}(node) = 2 \\ \text{ASSIGN}[\![e1 = e2]\!](m) \circ \text{UPDAT\_VAL}[\![e1, e2]\!](c) & \text{otherwise} \end{cases}$$

In this case, the analysis updates the value in MapCall (c) associated with "e1" and replaces it with "e2" with the instruction $UPDT\_VAL$ (defined in Equation 3.6). After that, the analysis executes an assignment between "e1" and "e2". Once again the analysis performs a $PUSH$ instruction if the previous instruction has two predecessors ("*PREDS*").

3.4.3 Call

The $Call$ instruction represents the call to a function e_fun with the arguments arg_ts. The result from this call is stored in ret_id. The transfer function for this instruction is defined as follows:

$$\Theta_Q[\![\text{Call } ret\_id, e\_fun, arg\_ts]\!](q) \stackrel{\text{def}}{=}$$
$$\begin{cases} \text{CALL\_OBJECT}[\![ret\_id, arg\_ts]\!](q) \circ \text{CK}[\![id]\!] \circ \text{CK}[\![e]\!](m) & e\_fun \text{ is Object} \\ \text{CALL\_ARRAY}[\![ret\_id, arg\_ts]\!](q) \circ \text{CK}[\![id]\!] \circ \text{CK}[\![e]\!](m) & e\_fun \text{ is ArrayList} \\ \text{CALL\_PRINT}[\![ret\_id, arg\_ts]\!](q) \circ \text{CK}[\![id]\!] \circ \text{CK}[\![e]\!](m) & e\_fun \text{ is PrintStream} \wedge ret\_id \text{ is IRVAR} \\ \text{CALL\_GENERAL}[\![ret\_id, arg\_ts]\!](q) \circ \text{CK}[\![id]\!] \circ \text{CK}[\![e]\!](m) & \text{otherwise} \end{cases}$$

As before, the transfer function checks a series of conditions, starting with the instruction CK (CHECK) like in $Store$ and $Load$. There are four cases, which we describe below.

Case 1. If e_fun is an object then the analysis executes the function $CALL\_OBJECT$:

$$\text{CALL\_OBJECT}[\![ret\_id, arg\_ts]\!](q) \stackrel{\text{def}}{=}$$
$$\begin{cases} \text{PUT\_USED}[\![arg\_ts]\!](m) & \text{KOV}[\![ret\_id]\!](c) \in c \wedge m(\text{KOV}[\![ret\_id]\!](c)) \in U \\ \text{ASSIGN\_HASH}[\![arg\_ts[0], arg\_ts]\!](a) & \text{KOV}[\![ret\_id]\!](c) \in c \wedge m(\text{KOV}[\![ret\_id]\!](c)) \notin U \\ \text{ASSIGN}[\![\text{KOV}[\![ret\_id]\!](c), arg\_ts]\!](m) & \text{otherwise} \end{cases}$$

$CALL\_OBJECT$ verifies if a key associated with the value ret_id in MapCall (c) exists, and if that key is classified as "U". This would indicate that a function was called inside a prune condition, and would execute PUT_USED in order to change the classification of all variables inside the function called into "U", in order to follow the same procedure as $FILTER$ (defined in Equation 3.11).

If on the other hand the analysis detects that a key corresponding to the value ret_id in MapCall (c) but the classification of this key is not "U", then the analysis considers that this is the instruction of a call to a function (e.g.: `text.put(input1, input2)`), and then associates the first variable in arg_ts (as key) with the other variables in arg_ts (as values) in MapCall (c), with the instruction $ASSIGN\_HASH$.

If none of the above conditions verify, then the analysis executes an $ASSIGN$ between the key associated with ret_id and the variables of arg_ts, to account for code like "`int i = example.get("a");`".

Case 2. If the analysis verifies that e_fun is an *Arraylist* then it executes $CALL\_ARRAY$:

$$\text{CALL\_ARRAY}[\![ret\_id, arg\_ts]\!](q) \stackrel{\text{def}}{=}$$
$$\begin{cases} \text{PUT\_USED}[\![arg\_ts]\!](m) & \text{KOV}[\![ret\_id]\!](c) \in c \wedge m(\text{KOV}[\![ret\_id]\!](c)) \in U \\ \text{ASSIGN\_HASH}[\![arg\_ts[0], arg\_ts]\!] & \text{otherwise} \end{cases}$$

$CALL\_ARRAY$, like $CALL\_OBJECT$ also analyzes if a key associated with the value ret_id in MapCall (c) exists and if that key has a classification of "U", and if confirmed, also executes PUT_USED. However, if this condition is not met, then it executes an $ASSIGN\_HASH$ between the first variable of arg_ts and the other variables of the same variable.

Case 3. If the analysis verifies that e_fun is an *Printstream* then it executes $CALL\_PRINT$:

$$\text{CALL\_PRINT}[\![ret\_id, arg\_ts]\!](q) \stackrel{\text{def}}{=} \begin{cases} \text{PUSH}(M) \circ \text{PUT\_USED}[\![arg\_ts]\!](m) & \text{PREDS}(node) = 2 \\ \text{PUT\_USED}[\![arg\_ts]\!](m) & \text{otherwise} \end{cases}$$

$CALL\_PRINT$ turns the classification of the variables in arg_ts to "U" with PUT_USED, and if the analysis detects that the next instruction has 2 predecessors then the analysis performs a $PUSH$ in the end, to account for a new conditional method body. This function adds support for variables being printed to be counted as output, and so its classification becomes "U".

Case 4. If none of the previous clauses are verified for $CALL$ then the analysis applies the function $CALL\_GENERAL$:

$$\text{CALL\_GENERAL}[\![ret\_id, arg\_ts]\!](q) \stackrel{\text{def}}{=} \text{ASSIGN}[\![ret\_id, arg\_ts]\!](m) \circ \text{UPDT\_VAL}[\![ret\_id, arg\_ts[0]]\!](c)$$

$CALL\_GENERAL$ performs a $UPDT\_VAL$ that changes the value in MapCall (c) associated with "ret_id" and replaces it with "arg_ts[0]". This happens in order to keep track of the association between *Pure Variables* and *Program Variables* in a $Call$ instruction that may have been done previously. Additionally it performs an $ASSIGN$ between ret_id and arg_ts.

3.4.4 Prune

The $Prune$ instruction represents the beginning of a conditional method body with an expression exp as the condition and the boolean bol indicating which branch is being analyzed. The transfer function for this instruction is defined as follows:

$$\Theta_Q[\![\text{Prune } exp, bol]\!](q) \stackrel{\text{def}}{=} \begin{cases} \text{PRUNE\_TRUE}[\![exp]\!](q) & bol = \text{TRUE} \\ \text{PRUNE\_FALSE}[\![exp]\!](q) & \text{otherwise} \end{cases}$$

Case 1. If bol is *TRUE* then the analysis executes the function $PRUNE\_TRUE$.

$$\text{PRUNE\_TRUE}[\![exp]\!](q) \stackrel{\text{def}}{=} \begin{cases} \text{PUSH}(m) \circ \text{POP}(m) \circ \text{FILTER}[\![exp]\!](m) & \text{PREDS}(node) = 2 \\ \text{POP}(m) \circ \text{FILTER}[\![exp]\!](m) & \text{otherwise} \end{cases}$$

The first function to be applied is $FILTER$, defined in Equation 3.11, which searches the domain for variables that are classified as "U" or "W". If it finds any variables in those conditions, then the variables present on the conditional statement get classified as "U". Otherwise, the current classifications remain unchanged.

$$FILTER[\![e]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} U & y \in \text{VARS}(e) \wedge \exists x \in X : m(x) \in \{U, W\} \\ m(y) & \text{otherwise} \end{cases} \tag{3.11}$$

After that the analysis executes the POP instruction, defined in Equation 3.12, that returns the variables in Map (m) with the classification "BU" or "NW" to the previous "U" and "W" respectively. This returns to normal the alterations that $PUSH$ (defined in Equation 3.9) executed in the beginning of a conditional method body, because $FILTER$, defined in Equation 3.11, already detected if any new variables were transformed to "U" during the conditional method body. Once again, if the next instruction has 2 predecessors then the analysis performs a $PUSH$ in the end, to account for a new conditional method body.

$$POP(m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} U & m(y) \in \{BU\} \\ W & m(y) \in \{NW\} \\ m(y) & \text{otherwise} \end{cases} \tag{3.12}$$

Case 2. If bol is *FALSE* then the analysis applies the function $PRUNE\_FALSE$:

$$\text{PRUNE\_FALSE}[\![exp]\!](q) \stackrel{\text{def}}{=} \begin{cases} \text{PUSH}(m) \circ \text{POP}(m) & \text{PREDS}(node) = 2 \\ \text{POP}(m) & \text{otherwise} \end{cases}$$

The first instruction to be executed is POP, which returns the variables classifications in Map (m) into the previous classifications that $PUSH$ instruction changed.

Just like $PRUNE\_TRUE$, here the analysis also checks for the number of predecessors the instructions has, in order to verify if a new conditional method body starts in the next instruction being analyzed.

3.5 Implementation

The analysis presented in the previous sections is implemented as an Infer check, it works, and it is available online.[5]

The analysis was implemented in the language Ocaml, which is the language Infer was develop in, and registered as a checker in Infer configuration files. At the time of writing, the analysis only works for *Java* programs, because the analysis needs explicit support for each type of inputs and outputs specific to each language. The analysis was given an *id* (*data-usage-check*), which is used to call the analysis on the command line. A new error type was created and registered ("*VARIABLE_UNUSED*") in Infer, that represents a variable not used.

The analysis works backwards, starting to analyze the program from the last instruction and working its way up. In the beginning of the analysis the abstract domain only contains the variables that were passed as input into a function with the classification "N". In the following instructions the analysis checks if each variable is in the abstract domain, and if not introduces it with the classification of "N".

[5] Data Usage Check: https://github.com/Rui1995/DataUsageCheck-FBInfer

This happens because the analysis does not have the ability to know beforehand all of the variables used during the program.

The output of the program is detected if a variable is either returned or printed, in which case it is added to the abstract domain with the classification “U”. These are the first variables not classified as “N” and the starting point for all the other variables to change classification, by interacting with the outputted variables.

All variables, *Pure Variables* and *Program Variables*, are added to the abstract domain, but only the *Program Variables* are evaluated in the end of the analysis (because those are the ones corresponding to the actual variables in the program analyzed) where all variables with the classification “N” launch an error on the console, showing the line where the variable last appeared and the name of the variable. An example of this is presented in Figure 3.1, with the results of one of the analysis of the benchmark suite.

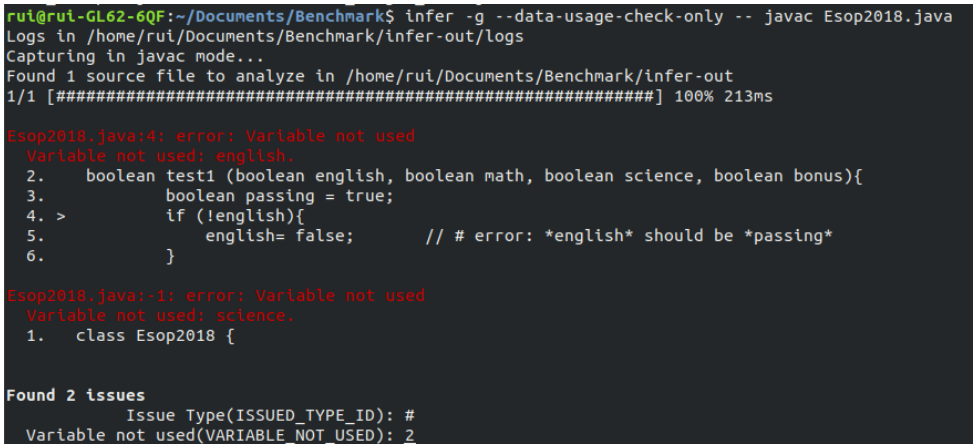

Figure 3.1: Input Usage Errors Found by Our Implementation

Additionally, Infer has a graphical interface that opens in a web browser, showing the code analyzed, errors detected, and *SIL* instructions for each Java instruction. An example of that follows in Figure 3.2.

This interface shows the code with proper indentation, followed by a link to each particular instruction, that shows the *SIL* instructions that it was converted to, and the evaluation for that line of each analysis running. An example of that is showed in Figure 3.3 for the instruction in line 9 in the code of Figure 3.2. In Figure 3.3 the abstract domain is showed before the analysis and after each *SIL* instruction. This is highlighted inside the blue boxes. Each *SIL* instruction is highlighted inside the green boxes.

The way this analysis works it not only reports input variables that were not used, but also other variables that were not used during the execution of the program. This is in line with Lyra’s approach which does the same.

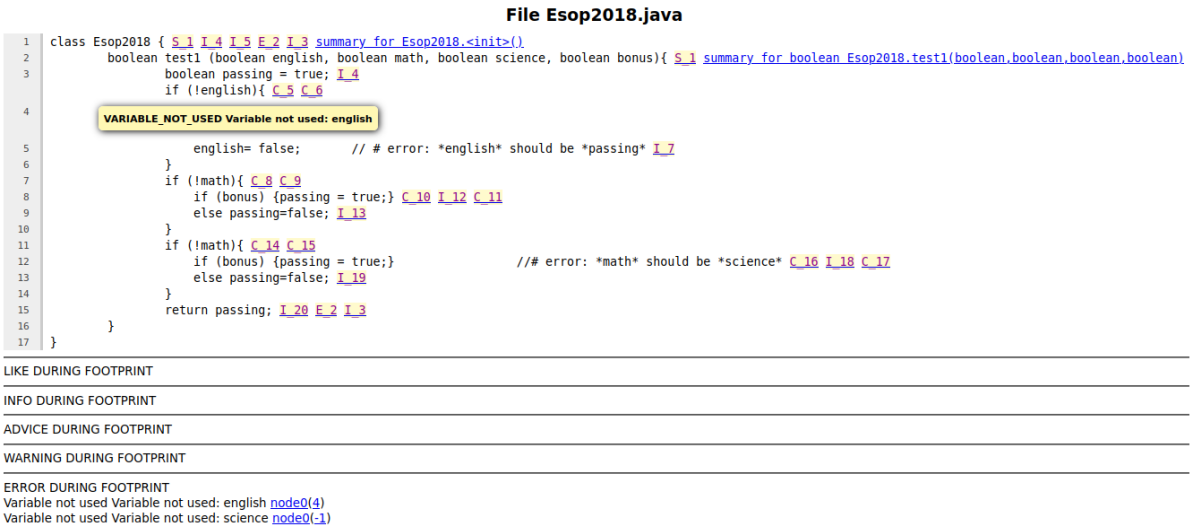

Figure 3.2: Graphical Interface

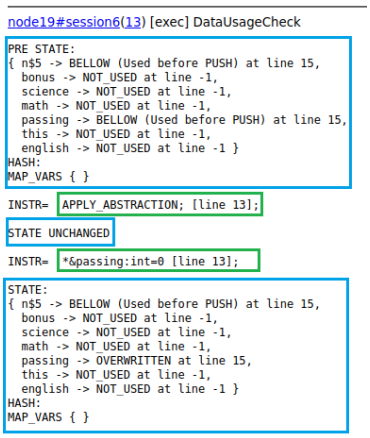

Figure 3.3: Single Instruction in Graphical Interface

3.5.1 Limitations

The analysis created has some limitations. It is necessary to implicitly add support for each type of inputs and outputs, making the adaptation to support other languages not trivial or automatic (for example the analysis needs to detect “scanf()” and classify its variables as inputs for a C program).

In case of functions that modify objects and do not return variables the analysis will consider all variables inside as not used. In case of packages in Java that either produce some input (e.g.: Java.Util.Scanner) or output, it is also needed to add explicit support for it in the analysis. This happens because Infer does not detect automatically all inputs and outputs of a program.

Some packages being imported and used in the code being analyzed also may need added explicit support (to classify the variables correctly) because it can be translated by Infer in a different way than

the previously supported functions (e.g.: some expressions may appear in different order and additional information may need to be stored).

In order to install Infer from source in order to analyze C and Objective-C programs, Infer compiles a custom version of *Clang* that may take a long time [6]. In our machine it took around two hours to compile in a Quad-Core 3.2Ghz computer. Programs compiled with *gcc* need to compile with *clang* too.

[6] Infer Installation Guide: https://github.com/facebook/infer/blob/master/INSTALL.md

4

Evaluation

Contents

In order to test the effectiveness of the analysis, we have done a series of tests that verify the effectiveness of the tool created. First, the set of benchmarks that Lyra uses were manually converted from Python into Java so that there is a basis for comparison between the two tools. Next we chose two data science and data analysis open-source projects in order to test the tool's capabilities in real-world applications.

4.1 Experimental Setup

The analyses were done in a Quad-Core 3.2Ghz system with 8GB of RAM, running *Ubuntu* 18.04.4 LTS stored on an SSD. The Infer version used to implement our analysis was v1.0.0-1e990455f. The Java version used was v11.0.9.1, except for the second Non-Controlled Benchmark in Section 4.3 that used Java version v1.8.0_275. The first program in Section 4.3 used *Gradle* version v6.6.1, while the second program used v5.4.1.

Infer was run with the flag "*–data-usage-check-only*" in order to disable the default analyses of Infer. Lyra's version used was v0.1, requiring Python v3.6.

4.2 Artificial Benchmark

In order to evaluate the effectiveness and accuracy of our implementation compared to Lyra, we assessed it with the same input data usage benchmark used to evaluate Lyra[1]. The benchmark contains 10 programs written in Python; since Infer does not support Python, we manually converted it to Java. The benchmark is specifically designed to test the detection of unused input data, ranging from simple examples with only variables containing Booleans as input and a single conditional set of instructions, to tests containing dictionaries as input.

It is important to notice that the files related to *CodeJam* in Lyra's repository were not evaluated as they did not focus on testing the input data usage analysis.

The benchmark suite, averaging 25 lines of code (min:12, max:39), included tests where inputs were not used and the analysis correctly caught those. There are also tests containing dictionaries (HashMaps in Java) that were dealt with correctly, and tests with nested conditional sets of instructions to check if the modification to the way the analysis dealt with them is valid, which it was. Additionally there are tests with no output, which made the analysis assume no variables were used, creating false-positives. These false-positives were expected since the analysis depends on the variables from the output to classify the rest of the variables.

[1] Lyra benchmark: https://github.com/caterinaurban/Lyra/tree/master/src/lyra/tests

Table 4.1: Detailed Artificial Benchmark Results

Artificial Benchmark					
File Name	#Lines	Errors	False-positives	True-positives	Time to analyze
BRCA_example	25	0	0	0	0,175s
Dict_descr_example	39	1	0	1	0,350s
Dict_example	26	6	0	6	0,091s
Dict_example_container	13	0	0	0	0,059s
Dict_if_example	18	0	0	0	0,062s
Dict_simple_example	12	0	0	0	0,058s
Esop2018	17	2	0	2	0,054s
Larger_expressions	36	6	0	6	0,367s
Running_example	32	1	0	1	0,335s
Three_dict_example	30	0	0	0	0,335s
TOTAL	248	16	0	16	1,886s

Our implementation behaved similarly to Lyra on the 10 programs: both our analysis as well as Lyra's presented no false-negatives or unexpected false-positives. In terms of performance, Lyra takes on average $0.0802s \pm 0.0820s$ and our implementation takes $0,1886s \pm 0,1336s$. The performance penalty was expected, since Infer has to build the intermediate representation before starting the analysis. In Table 4.1 we present the more detailed results from the benchmark.

4.3 Non-Controlled Benchmark

In order to test the real capabilities of the analysis we chose to run it in two data science and data analysis Java software packages. We focused our tests in open source packages created by experienced programmers.

First package. The first package analyzed was *Neo4j Graph Data Science Library*[2], a plugin for *Neo4j graph database* that consists of a library with graph algorithms. The analysis took on average $15,67s \pm 0,23s$ to package the library with *gradle* and convert the instructions into intermediate language *SIL*, plus an additional $21,29s \pm 0,13s$ on average to analyze the intermediate language.

The analysis evaluated 40 Java files with an average of 120 lines of code (min:44, max:260). The analysis identified 371 false-positives and 0 true-positives. The analysis raised a substantial amount of false-positives due to external packages being imported and utilized in the code. As mentioned before, one of the disadvantages of this analysis is that some additional packages in the code being analyzed need to get additional explicit support. Additionally some false-positives were also caused by methods that altered objects but did not return variables. One example of this is showed in the next example:

[2]Neo4j Graph Data Science Library: https://github.com/neo4j/graph-data-science

```
1  private CypherMapWrapper(Map<String, Object> config) {
2      this.config = config;
3  }
```

Second package. The second package analyzed was *Ananas Desktop*[3], an open source data integration and analysis tool that allows non technical users to edit data processing jobs and visualize data. Unfortunately, because the analysis launches an error that does not allow the program to compile, only three files were able to be analyzed individually.

The analysis took on average $1,94s \pm 1,38s$ to compile each Java file, convert the instructions into intermediate language *SIL* and to analyze the intermediate language.

The analysis evaluated three Java files with an average of 83 lines of code (min:69, max:103). The analysis identified 25 false-positives and 0 true-positives. Most of the false-positives were generated by methods that altered objects without returning any variables, with the remaining errors related to imports that were not handled properly by the analysis. As mentioned in “Limitations”, some imports may need added support for the analysis to work properly.

[3]Ananas Desktop: https://github.com/ananas-analytics/ananas-desktop

5

Conclusion

Contents

5.1 Conclusions

In conclusion we were able to adapt Lyra's approach to work in Infer by converting the transfer functions to Infer's Intermediate Language *SIL*. For this adaptation we created two additional support structures in order to properly evaluate the *SIL* instructions.

The analysis proved to have some limitations, more specifically the need for added support for each inputs and output for each program language supported, since Infer does not detect automatically all inputs and outputs. Additionally the analysis may need support for certain packages in order to work properly.

Given the results obtained, we answer the proposed research questions as follows:

RQ1: Is it possible to implement an input data usage analysis similar to Urban and Müller's analysis that works with the static analysis tool Infer? We were capable of adapting Lyra's analysis and implement it with Infer's framework, proving accurate in the same set of benchmarks that Lyra uses.

RQ2: Can such an analysis work for several programming languages? The analysis as of writing supports only Java programs, but support for the other languages supported by Infer can be extended in the future by adding support to the different inputs and outputs of each language.

RQ3: Can such an analysis be used to analyze real code and find potential errors? The analysis created can be used to analyze real code but it does not run in every real program and in the ones that it ran there were no potential problems detected.

5.2 Future Work

As part of our future work, we plan to:

1. Improve the analysis in order to analyze any Java program, more specifically improve how the analysis handles imported functions.

2. Add support to other languages supported by Infer by adding support for its input and output instructions.

3. Evaluate the analysis in wider benchmark samples, which was not possible due to the limitations of the current implementation.

Bibliography

[1] C. Urban and P. Müller, "An abstract interpretation framework for input data usage," in *European Symposium on Programming*. Springer, 2018, pp. 683–710.

[2] T. Herndon, M. Ash, and R. Pollin, "Does high public debt consistently stifle economic growth? a critique of reinhart and rogoff," *Cambridge journal of economics*, vol. 38, no. 2, pp. 257–279, 2014.

[3] J. Mencinger, A. Aristovnik, and M. Verbic, "The impact of growing public debt on economic growth in the european union," *Amfiteatru Economic Journal*, vol. 16, no. 35, pp. 403–414, 2014.

[4] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods Symposium*. Springer, 2015.

[5] P. Rahkila, "Grain—a java data analysis system for total data readout," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 595, no. 3, pp. 637–642, 2008.

[6] K. Mikolajczyk, M. Szabatin, P. Rudnicki, M. Grodzki, and C. Burger, "A java environment for medical image data analysis: initial application for brain pet quantitation," *Medical Informatics*, vol. 23, no. 3, pp. 207–214, 1998.

[7] B. Dysvik and I. Jonassen, "J-express: exploring gene expression data using java," *Bioinformatics*, vol. 17, no. 4, pp. 369–370, 2001.

[8] A. J. Saldanha, "Java treeview—extensible visualization of microarray data," *Bioinformatics*, vol. 20, no. 17, pp. 3246–3248, 2004.

[9] D. W. Barowy, D. Gochev, and E. D. Berger, "Checkcell: Data debugging for spreadsheets," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 507–523, 2014.

[10] J. M. Hellerstein, "Quantitative data cleaning for large databases," *United Nations Economic Commission for Europe (UNECE)*, vol. 25, 2008.

[11] T. Cheng and X. Rival, “Static analysis of spreadsheet applications for type-unsafe operations detection,” in *European Symposium on Programming Languages and Systems*. Springer, 2015, pp. 26–52.

[12] P. Cousot and R. Cousot, “Abstract interpretation and application to logic programs,” *The Journal of Logic Programming*, vol. 13, no. 2-3, pp. 103–179, 1992.

[13] ——, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.