Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Interprocedural Static Analysis by Abstract Interpretation

Master's Thesis

Flurin Rindisbacher

August 15, 2017

Advisors: Dr. Caterina Urban and Jérôme Dohrau

Prof. Dr. Peter Müller
Chair of Programming Methodology
Department of Computer Science, ETH Zürich

Abstract

Whether a developer follows the functional programming approach or they compose their programs using object oriented programming, real-world applications usually consist of multiple procedures that interact with each other.

Sample [2] is a generic static analyser developed at the Chair of Programming Methodology at ETH Zurich. It is generic with respect to the static analysis problem and supports intraprocedural analyses. That is, it analyses one procedure at a time. A static analyser with the ability to analyse beyond a procedure's boundaries provides a way to derive useful properties about the behaviour of the whole program.

In this thesis, we design and implement a generic interprocedural static analysis based on call-strings by Sharir and Pnueli [21]. We extend Sample with the interprocedural analysis and enable the analyser to be used for various static analyses beyond procedure-boundaries.

Contents

Contents **iii**

1 Introduction **1**
1.1 Motivation 1
1.2 Goals of the Thesis 3
1.2.1 Design and Implementation of a Generic Interprocedural Analysis 3
1.2.2 Instantiation of the Analysis 3
1.2.3 Evaluation 3
1.2.4 Specification Inference 3
1.3 Viper and Silver 4
1.4 Outline 4

2 Interprocedural Static Analysis **5**
2.1 Intra- vs. Interprocedural Static Analysis 5
2.2 Abstract Interpretation 6
2.2.1 Abstract Domains 6
2.3 Different Approaches to Interprocedural Analysis 7
2.4 Interprocedural Static Analysis using Call-Strings 8
2.4.1 Abstract Interpretation Tagged with Call-Strings . . . 10
2.4.2 Call-String Approximation 12
2.5 Generic Interprocedural Static Analysis 12
2.5.1 Overview 13
2.5.2 Targeted Programming Language 13
2.5.3 Framework Parameters 13
2.5.4 Creating the Supergraph and Callgraph 14
2.5.5 Determining where to Start the Analysis 15
2.5.6 Method Call Treatment 16
2.5.7 Entering and Leaving a Procedure. 17
2.5.8 Computing the Analysis Result 20

3 Sample **21**
3.1 Overview 21
3.1.1 Broad Overview 21
3.1.2 Programming Language 23
3.2 Sample for Intraprocedural Analyses 23
3.2.1 Program Representation 23
3.2.2 State Representation 24
3.2.3 Analysis Results 25
3.2.4 Intraprocedural Analysis Runner 25
3.2.5 Computing a Fixed Point Solution 25
3.3 Sample for Interprocedural Analyses 27
3.3.1 Overview 27
3.3.2 Source Code 28
3.3.3 ProgramResult 28
3.3.4 Call-String Representation 28
3.3.5 Interprocedural Analysis Runner 29
3.3.6 Top-Down / Bottom-Up Analysis 30
3.3.7 Interprocedural Interpreter 31
3.4 Implemented Analyses 38
3.4.1 Interprocedural Integer Interval Analysis 38
3.4.2 Interprocedural Integer Octagon Analysis 39
3.4.3 Interprocedural Strongly Live Variable Analysis 40

4 Specification Inference **43**
4.1 Numerical Abstract Domain (Octagons) 44
4.2 Top-Down / Bottom-Up Analysis 45
4.2.1 Bottom-Up Analysis Implementation 45
4.3 Extracting the Specification 49
4.3.1 Analysis Results 49
4.3.2 Extracting Preconditions 50
4.3.3 Extracting Postconditions 50
4.3.4 Extracting Loop Invariants 50
4.4 Implementation Details 51
4.4.1 Overview 51
4.5 Exporting the Inferred Specification to Viper IDE 52
4.5.1 Viper protocol 53
4.5.2 Implementation in Sample 53
4.5.3 Viper Protocol Messages 54

5 Evaluation **55**
5.1 Automated Tests 55
5.2 Testing Infrastructure 55
5.3 Numerical Forward Analyses 56
5.3.1 Fibonacci 56

5.3.2 McCarthy 91 Function 58
5.3.3 Multiple Callers 58
5.3.4 Review 60
5.4 Strongly Live Variable Analysis 61
5.4.1 Multiple Callers – No Recursion 62
5.4.2 Multiple Callers – Recursion 63
5.4.3 Review 65
5.5 Specification Inference 66
5.5.1 Program four() 66
5.5.2 Ackermann Function 66
5.5.3 Program getElementOrLast() 68
5.5.4 Strongly Connected Component in Callgraph 69
5.5.5 Review 70
5.6 Evaluation Review 71

6 Conclusions **73**
6.1 Future Work 74
6.2 Acknowledgements 75

A Appendix **77**
A.1 Bottom-Up Inferred Specification 77
A.2 Top-Down Inferred Specification 78
A.3 Running an Analysis 79
A.4 Top-Down inferred Specification for the Ackermann Function 80
A.5 Viper Protocol Message with Inferred Specification 84
A.6 Viper Protocol Error Message 85

Bibliography **87**

Chapter 1

Introduction

Static analysis provides mathematical guarantees about the behaviour of programs by analysing its source or object code. A wide range of useful properties can be derived by such an analysis. A compiler may for example leverage this information to perform program optimisations at compile-time. On the other hand, it may be possible to infer program specifications, such as preconditions, postconditions and loop invariants which will help reasoning about the correctness of a (possibly safety-critical) application. Many useful static analyses perform the analysis intraprocedurally, that is, one procedure is analysed at a time and procedure invocations are not supported. Interprocedural analyses, on the other hand, support procedure invocations and, by leveraging information about how a procedure is called, more precise analyses may be possible.

Sample ('Static Analyzer for Multiple Programming LanguagEs') [2] is a generic static analyser developed at the Chair of Programming Methodology[1] at ETH Zurich. It is based on abstract interpretation [5], a generic framework for the sound approximation of the behaviour of programs. Sample is generic in the sense that different (intraprocedural) analyses can be implemented within Sample's framework and it supports analysing multiple programming languages by first translating them into an internal format. A core goal of this thesis is to extend Sample with a generic and customisable interprocedural static analysis.

1.1 Motivation

Let us consider the small program written in a C-like programming language in Listing 1.1 and discuss possible analysis approaches. It is easy to see that the procedure *main()* will always access the array at position zero

[1] http://www.pm.inf.ethz.ch/

and therefore return the first element of the array. However, for an intraprocedural static analysis it is impossible to deduce that the variable *idx* stays within the bounds of the array. The problem of tracking possible numerical values of this programs variables needs to be tackled differently, that is, using an interprocedural approach.

```
1  int main() {
2    int array[5] = {0, 1, 2, 3, 4};
3    int idx = 2;
4    idx = decrement(idx);
5    idx = decrement(idx);
6    return array[idx];
7  }
8
9  int decrement(i: Int) {
10   return i - 1;
11 }
```

Listing 1.1: A small program that cannot be analysed using an intraprocedural static analysis.

One approach, called the *trivial treatment* in [18], is to simply assume that anything can happen when a procedure is called and therefore only conservative information about global variables, return values and the heap can be used. The variable *idx* would be set to an unknown value after the procedure call and no information about the array access would be available.

Using *inlining*, which is textual replacement of the procedure call by the called procedure's body, *main()* could be translated into a procedure without calls to *decrement()* and therefore analysed by an intraprocedural static analysis. This would work for the program in Listing 1.1 but not for recursive procedures because they would possibly need to be inlined infinitely many times.

A *context-insensitive* or, as [18] calls it, *naïve* approach treats a procedure invocation as a goto from the call location to the entry-point of the callee. A procedure's return is treated as a non-deterministic goto to the next statement after every call to this procedure. For the program in Listing 1.1, this would result in the gotos *from-line-number* $\rightarrow$ *to-line-number*: $4 \rightarrow 10$, $5 \rightarrow 10$, $10 \rightarrow 5$ and $10 \rightarrow 6$ which introduces the loop $5 \rightarrow 10 \rightarrow 5$ into the program. This loop is an infeasible path that cannot occur in the actual execution of the program. Due to this loop the numerical value of *idx* can still not be tracked precisely and again, the analysis results in unknown information for variable *idx*. This approach is called context-insensitive because it does not keep track where a call came from. A called procedure is analysed using all incoming information from all call-sites and the result of the procedure is returned to every caller. This analysis is cheap in terms of computing power but does not provide precise analysis results.

A more precise analysis is possible by using a *context-sensitive* approach, that is, an approach that prevents infeasible paths like the loop $5 \rightarrow 10 \rightarrow 5$ and

computes an answer for every call-site. We aim to design and implement such an analysis in Sample.

1.2 Goals of the Thesis

This thesis should extend Sample with a generic and customisable interprocedural analysis. The core goals consist of evaluating existing work, adapting it to our needs, implementing the analysis in Sample and instantiating as well as evaluating the generic implementation with actual analyses for different problems.

1.2.1 Design and Implementation of a Generic Interprocedural Analysis

The implemented generic analysis should meet the following criteria:

- The analysis can be instantiated for numerical and non-numerical analyses.
- The implementation should be extensible such that it can be used as a basis for other analyses in the future.
- The cost vs. precision ratio should be adjustable. For example a faster but less precise analysis like the context-insensitive approach or a more precise but slower context-sensitive approach should be usable depending on the use case.

1.2.2 Instantiation of the Analysis

An interprocedural analysis for standard numerical analyses should be instantiated. Furthermore, an analysis for non-numerical domains should be instantiated. We will implement a variant of live variable analysis, which works in backward direction, and instantiate an interprocedural analysis with it.

1.2.3 Evaluation

The instantiated analyses should be experimentally evaluated in terms of cost and precision. Strengths and weaknesses of the implementation should be shown.

1.2.4 Specification Inference

Sample is able to infer some specifications of a program. For instance, it supports the automatic inference of access permissions [9]. An extended

goal of the project is to use the new interprocedural analysis to implement a specification inference for numerical properties.

1.3 Viper and Silver

Our work within Sample will focus on the parts of the static analyser that interact with the Viper toolchain [16] and its verification language Silver. The Viper toolchain is a verification infrastructure for permission-based reasoning developed at ETH Zurich. It uses Sample to infer specifications for programs written in Viper's intermediate verification language Silver. "The Viper intermediate verification language (Silver) provides simple imperative constructs, as well as specifications and custom statements for managing permission-based reasoning. The assertion logic supported is a variant of implicit dynamic frames [22], which can be used to encode a variety of reasoning paradigms, including separation logics [20]. Silver provides support for mathematical types, user-defined predicates and pure functions, and a number of other unique features based on recent research." [3]
Going forward we will use Silver for our examples.

1.4 Outline

In Chapter 2 we first provide more information about abstract interpretation, interprocedural static analysis and existing approaches. Then, we introduce our generic interprocedural analysis. Chapter 3 provides more details about Sample and documents the implementation of the generic interprocedural analysis and the instantiations for numerical analysis and strongly live variable analysis [10]. Then in Chapter 4, we introduce specification inference and provide details about how the implemented interprocedural static analysis has been put to work to infer preconditions, postconditions and loop invariants for programs. Chapters 5 and 6 provide evaluation and conclusions of the thesis.

Chapter 2

Interprocedural Static Analysis

In this chapter we first provide some background regarding interprocedural static analysis. Then we present existing work and introduce how our generic interprocedural static analysis deals with method-invocations.

2.1 Intra- vs. Interprocedural Static Analysis

Whether a developer follows the functional programming approach or they compose their programs using object oriented programming, real-world applications usually consist of multiple procedures that interact with each other. An intraprocedural static analysis looks at each procedure in isolation. What if the analysed code contains a method-invocation? An intraprocedural analysis cannot deal with that. Interprocedural analyses on the other hand are able to analyse programs consisting of multiple procedures. If the information about how a procedure was called is taken into account, more precise analysis results can be achieved than by working on each procedure in isolation.

For an intraprocedural analysis a *control flow graph* (CFG) may be sufficient to represent the program under analysis. A CFG is a directed graph containing nodes for every statement in a procedure. In the interprocedural case additional information like the *callgraph* between procedures is needed. In other work the term *supergraph* [14, 19] — a combination of CFGs with edges connecting a call-statement in the caller with the CFG of its target (the callee) — sometimes is used. Depending on how this supergraph looks, an analysis may need to determine where to start. If the graph contains cycles and there is no *main* method as the entry-point to the program, it may not be obvious where to start an analysis. Another difficulty introduced in the interprocedural case is how to tackle (possibly infinite) recursion.

2.2 Abstract Interpretation

For a static analysis to be useful its result should always be correct.

Unfortunately the set of all possible executions of a program can be infinite. Therefore, most non-trivial questions about any programs behaviour are undecidable. This means that it is not possible for some program (the analysis) to answer a non-trivial question for any given program. To the cost of losing information a problem can be approximated and become decidable.

Abstract interpretation [5] by Cousot and Cousot is a generic framework for the sound approximation of the behaviour of a program. Concrete properties, for example the actual values of numerical variables at a certain point in the program, are approximated with abstract predicates. These abstract predicates generally form a lattice called the *abstract domain*. An analysis based on abstract interpretation soundly over-approximates the effect of a statement of the program in the abstract domain. Let $f\colon \mathbb{Z} \to \mathbb{Z}$ be a function in the concrete domain of integers that represents the effect of a statement in the program. Let $(D, \leq)$ be a lattice. For function f there would exist a *transformer* $g\colon D \to D$ that over-approximates the effect of f in D. These abstract transformers are iteratively applied starting with an initial element usually called $\bot$ until a fixed point is reached. To speed up finding a fixed point or to ensure termination for infinite lattices a *widening* operator exists that computes a post-fixed point, that is, a fixed point larger than the least fixed point. A thorough introduction to abstract interpretation can be found at [23] in Chapter 2.

2.2.1 Abstract Domains

Depending on the analysis, one must chose an abstract domain. Throughout the thesis we will use the *interval abstract domain* [4] and *octagon abstract domain* [15] for examples.

For integers we will use $[l, u]$, with $l, u \in \mathbb{Z}$, to denote a value in the interval domain. Other elements in the domain are $\top$ (any integer value) and $\bot$ (no information). The concrete value 0 would be written as $[0,\ 0]$. Furthermore, we will use $x \mapsto [l,\ u]$ to denote that a variable x has the abstract value $[l,\ u]$ in the abstract domain. A set of constraints could for example be $[x - y \leq 0,\ y - x \leq 0,\ x \leq 10,\ x + 10 \leq 0]$ which translates to the variables x and y having the same value somewhere between -10 and 10. For readability we will write constraints like $[x = y]$ instead of their octagonal representation $[x - y \leq 0,\ y - x \leq 0]$ in our examples.

2.3 Different Approaches to Interprocedural Analysis

In this section we briefly discuss some approaches to interprocedural analysis.

Summary-based Analysis. A well known approach to interprocedural analysis is the summary-based analysis proposed by Sharir and Pnueli [21]. Their analysis consists of two phases. In a first phase a summary is created that describes the effect of the procedure. In a second phase the program is analysed and, when a procedure call is encountered, the summary-function is applied on the current (abstract) state in the caller. This way a program can be analysed and the effect of a called procedure is applied when necessary. The name *functional approach* is also often use to refer to Sharir and Pnuelis summary-based analysis. Their framework excludes recursive procedures with local variables.

Dataflow Analysis via Graph Reachability. Reps et al. [19] showed that a large class of interprocedural problems can be precisely solved via reduction to graph reachability. They call this class of problems the *interprocedural, finite, distributive, subset* or *IFDS problems*. *IFDS* is a variant of the functional approach by Sharir and Pnueli with three modifications:

1. The abstract domain is restricted to be a subset domain 2^D for a finite D.
2. The transformers need to be distributive. Some transformer f is distributive if $f(x) \sqcup f(y) = f(x \sqcup y)$
3. Edges from a call node in the supergraph to the return-site can have an associated dataflow function.

With the third modification they allow the framework to analyse recursive procedures with local variables.

Compositional Separate Modular Analysis. IFDS and summary-based analysis follow the principle of global analysis where the whole program is represented as a system of equations to be solved. Cousot and Cousot [6] present methods for compositional separate modular static analysis of programs by abstract interpretation. A global analysis is decomposed into separated analyses that could be done in parallel. Several techniques about what separate analyses to run and how to compose them into a global analysis are discussed in [6]. Advantages of this approach are that the program is modularly analysed and it is therefore possible to save memory and time during analysis. Another advantage of this approach is that if a small part of the program changes then only the affected components need to be re-analysed whereas global analyses would need to analyse the whole program again.

2.4 Interprocedural Static Analysis using Call-Strings

As defined in our goals we aim for a generic (in terms of the abstract domain) and customisable interprocedural analysis to be added to the static analyser Sample. For a generic abstract domain we cannot assume that it is finite which is why IFDS cannot be used for our use case. A more general approach, that was also introduced by Sharir and Pnueli [21], is called *call-strings*. After evaluating existing approaches we chose call-strings as the basis of our analysis because it fits our requirements well and with modifications can be integrated into Sample's existing code base. For the rest of this section we explain the general idea of call-strings.

Call-Strings. The call-strings approach connects all the procedures into one big graph representing the program. As discussed for the context-insensitive approach in Section 1.1 this graph may contain infeasible paths. Call-strings provide a way to only take feasible paths into account during the analysis.

Feasible and infeasible paths in the graph. Listing 2.1 shows a small program written in Silver that increments the value of a local variable twice. Figure 2.1 shows two CFGs connected into one supergraph. The statements are annotated with the line numbers (l2, l3, l4, l8) in the actual program. The call-edges c_1 and c_2 represent the method calls. The return-edge r_1 to the statement at line 3 represents the transfer of the return-value to the target variable i. Some necessary special handling when entering the callee and returning from it is omitted for now. One feasible path between the two CFGs would start with the call-edge c_1 and end with the return-edge r_1. On the other hand a path containing call-edge c_2 and the return-edge r_1 exists in the graph but is infeasible in the actual execution of the program.

Simulating a call-stack. The analysis can separate feasible from infeasible paths by simulating the call-stack. A call-string γ is a sequence of locations in the program. Initially γ is empty. Every time a method is called, γ grows by adding the location of the last call to the call-string (similarly to a push() on the call-stack). When returning from a method, the location added last is removed from γ (similarly to a pop() on the call-stack). The call-string therefore always represents the stack of method calls that have not returned yet. For the program in Figure 2.1 three different call-strings exist: $\gamma_1 := Nil$, $\gamma_2 := l3 : Nil$, $\gamma_3 := l4 : Nil$ where Nil represents the empty list and : prepends an element to a list (similarly to the *cons* operator in functional programming languages). The empty call-string γ_1 is used when no calls are on the call-stack, which is the case for the entry-method. The other two call-strings γ_2 and γ_3 represent the two method calls at lines 3 and 4. Assuming that at the end of the *increment()*-CFG we have call-string

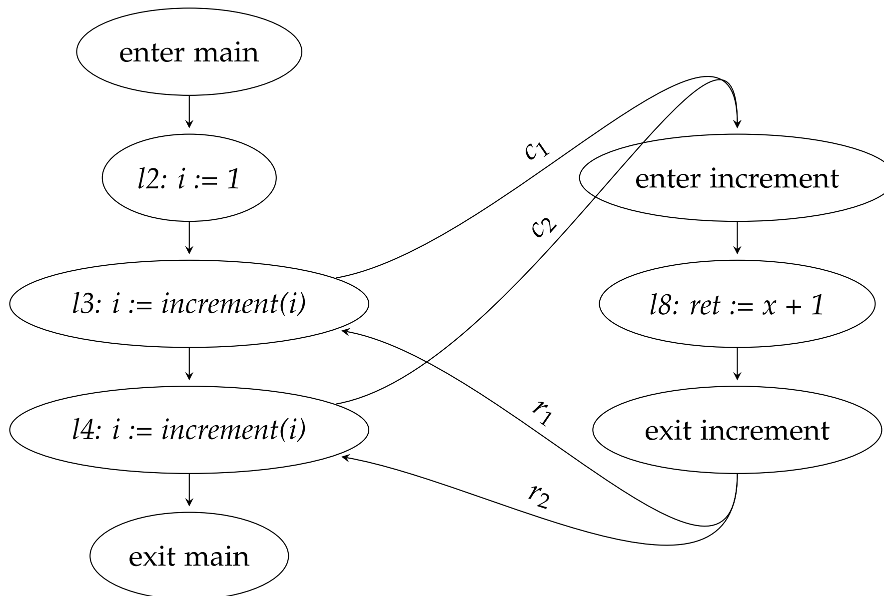

Figure 2.1: Supergraph consisting of two connected control flow graphs.

$\gamma_2 := l3 : Nil$ then by looking at the head of the call-string we know the only feasible return-edge can be r_1.

Recursive procedures. As explained, the call-string grows with each method call. For unbounded recursion this may lead to an infinitely growing call-string. For now we just note that there is a way to tackle this, and we will explain this in more detail in Section 2.4.2.

```
1 method main() {
2   var i: Int := 1
3   i := increment(i)
4   i := increment(i)
5 }
6
7 method increment(x: Int) returns (ret: Int){
8   ret := x + 1
9 }
```

Listing 2.1: A simple program written in Silver

2.4.1 Abstract Interpretation Tagged with Call-Strings

In this section we formally introduce call-strings adapting definitions and notations from [21] and [8] to our setting. Then we briefly discuss some drawbacks of the original solution. We will base our approach on these definitions.

A supergraph $G_{super} \stackrel{\text{def}}{=} (V,\ E \cup E_{interproc})$ is a directed graph containing all CFGs of all methods in the program. The set V consists of all nodes in the methods CFGs. Each node represents either a statement in the program or it may be a node that denotes that a method is entered or exited. E represents the edges in the control flow graphs and $E_{interproc} \stackrel{\text{def}}{=} E_{call} \cup E_{return}$ are additional method call- and return-edges.

Tagging a given Abstract Interpretation. The main idea behind call-strings is to translate a given abstract interpretation into a new one that tags its abstract values with call-strings. Let $\mathcal{A} \stackrel{\text{def}}{=} ((D, \leq), f_{mn}, d_0)$ be a given abstract interpretation with the lattice $(D, \leq)$ the transformers f_{mn} and an initial value d_0 with which an analysis is started.

Set of call-strings. Let Γ be the set of all call-strings representing interprocedurally valid paths in the supergraph. For a program without recursion this set is finite.

Growing and shrinking call-strings. The operator $\circ$ describes how a call-string $\gamma \in \Gamma$ is affected by an edge in the supergraph. We use : as the *cons* operator that prepends an element to a list whereas *.tail* returns the remainder of a list without its first element. Additionally we will use *.head* to denote the element that was added last to the call-string. This element represents the most recent method call in the call-stack.

$$\gamma \circ (m, n) \stackrel{\text{def}}{=} \begin{cases} \gamma & \text{if } (m, n) \in E \\ m : \gamma & \text{if } (m, n) \in E_{call} \\ \gamma.tail & \text{if } (m, n) \in E_{return} \text{ and } \gamma.head = n \end{cases} \tag{2.1}$$

The intuition behind the formula is that for (intraprocedural) edges in the CFG the call-string does not change. But for every call-edge the call-string grows whereas a return-edge $(m, n) \in E_{return}$ shrinks the call-string if the most recent method call matches the target of the return-edge. In other cases[1] the function is not defined which ensures that call-strings only represent feasible paths in the program.

[1]For example a return-edge that does not correspond to $\gamma.head$.

Constructing a new abstract interpretation. A new lattice $(D^*, \leq^*)$ whose elements are the point-wise lifting $D^* \stackrel{\text{def}}{=} \Gamma \rightarrow D$ is introduces. The order relation $\leq^*$ is the point-wise extension of $\leq$ in D.

Let $d^* \in D^*$. For a given call-string $\gamma \in \Gamma$, $d^*(\gamma)$ denotes the abstract value that was propagated by the analysis through the method calls in γ. A new initial value with respect to the initial value in the given analysis is defined as:

$$d_0^*(\gamma) \stackrel{\text{def}}{=} \begin{cases} d_0 & \text{if } \gamma = Nil \\ \bot & \text{otherwise} \end{cases} \tag{2.2}$$

Transformers for the lattice D^* are defined as:

$$f_{mn}^*(d^*)(\gamma) \stackrel{\text{def}}{=} \begin{cases} f_{mn}(d^*(\gamma_l)) & \text{if there exists a } \gamma_l \text{ s.t. } \gamma_l \circ (m,n) = \gamma \\ \bot & \text{otherwise} \end{cases} \tag{2.3}$$

The intuition behind this is that for valid call-strings the original transformers are applied. Otherwise the path is infeasible and $\bot$ is used.

Finally $\mathcal{A}^* \stackrel{\text{def}}{=} ((D^*, \leq^*), f_{mn}^*, d_0^*)$ is a new abstract interpretation that tags each abstract value with a call-string.

The analysis $\mathcal{A}^*$ yields an abstract interpretation adapted to only take feasible paths into account. If the transformers in the given $\mathcal{A}$ are distributive then the new transformers are as well [8]. The authors show that this analysis terminates for programs with finite Γ (without recursion). With a minor modification to the $\circ$ operator the analysis can be adapted to a backward analysis. In this case the $\circ$ operator would grow the call-string for return-edges and shrink it when the matching call-edge is encountered.

Drawbacks

The previously presented analysis $\mathcal{A}^*$ in its current form has two drawbacks that need to be solved to be useful for our use case.

1. It does not terminate for infinite Γ.
2. It does not support procedures with arguments and recursion with local variables.

Convergence mainly depends on Γ being finite but for recursive procedures the set of valid call-strings is infinite. For this problem the original paper [21] provides a solution to approximate call-strings. The authors only consider procedures without arguments and recursion without local variables. This needs to be adapted for our use case to make a static analysis of the programs like the one in Listing 2.1 possible.

2.4.2 Call-String Approximation

This section briefly describes how Sharir and Pnueli approach programs with recursion. We will use their *call-string suffix*[2] *approximation* [21] with modifications for procedures with parameters and local variables in recursive methods.

The program in Listing 2.2 recursively increments a counter x until its value is greater than or equal to ten. The set of valid call-strings would consist of $\Gamma_{rec} := \{Nil, l3 : Nil,\ l3 : l3 : Nil,\ ...\}$ which is infinite. To ensure termination of an analysis based on call-strings a finite set is needed. An upper bound on the length of the call-strings k is introduced. As long as the call-string smaller than k the call-string is modified according to the definition of the $\circ$ operator. Otherwise for method calls the location is still added to the list but then only the k locations added last are kept as the call-string. Older call-locations are discarded. Return-edges are treated as follows: Let $\gamma := l_i : l_j : l_k$ be a call-string containing the locations l_i, l_j, l_k and $e_r \in E_{return}$ a return-edge going back to the location l_i. Following the edge e_r will drop l_i from the call-string and then *add all* calls to the call-string that lead to the oldest call which is l_k. Returning from a method therefore can produce multiple call-strings and the abstract value is reused at multiple call-locations. Setting $k = 0$ results in propagating the abstract value to *every* caller which results in a context-insensitive [18] analysis.

```
1 method rec(x: Int) returns(r: Int){
2   if(x<10) {
3     r := rec(x + 1)
4   } else {
5     r := x
6   }
7 }
```

Listing 2.2: A recursive method yielding an infinite amount of call-strings

2.5 Generic Interprocedural Static Analysis

In this section we present our generic interprocedural static analysis. The analysis extends call-strings and adds support for procedures with parameters or recursion with local variables. The framework is generic in the sense that we assume an (intraprocedural) analysis in the form of an abstract interpretation to be given. We also discuss how to treat programs with multiple entry-points or disconnected supergraphs.

[2]Sharir and Pnueli append new elements to the right side of the call-string whereas we prepend new elements to the left. A *suffix* of length k contains up to k locations that were added last to the call-string.

2.5.1 Overview

Our approach uses a given intraprocedural analysis as a black box to compose an interprocedural whole-program analysis. It can also be adapted to a modular analysis for which we provide more details in Section 4.2. Both forward and backward direction of analysis are supported. The interprocedural analysis consists of the following steps:

1. Translate the program into a supergraph
2. Determine at what location in the program to start the analysis
3. Computing the analysis using the previously discussed call-string approach with special treatment for entering and leaving a procedure.

2.5.2 Targeted Programming Language

We target programming languages similar to Silver. A procedure supports multiple parameters and return values. The grammar rule for the syntax of a Silver call-statement with returns is shown in Listing 2.3.

```
1 call-statement ::= // call [with return target]
2   [ident^,* :=] ident "(" exp^,* ")"
```

Listing 2.3: Grammar rule of a call-statement in Silver.

A call-statement consists of assignments to the targets of the return values and the actual method call. If a procedure is declared to return values then every call to it must provide all target variables in its call-statement. The analysis is restricted to programs in which the target of a method-invocation can statically be determined. We therefore do not support dynamic dispatch in object oriented programming languages or higher-order functions from the functional programming world.

2.5.3 Framework Parameters

An instance of our framework requires the following parameters:

- A program consisting of a set of procedures in CFG form. We assume every CFG to contain an *entry* and *exit* node.
- An abstract interpretation $\mathcal{A} \stackrel{\text{def}}{=} ((D_s, \leq), f_{mn})$ consisting of a lattice $(D_s, \leq)$ of abstract states (defined below) and the abstract transformers f_{mn}. We use $\mathcal{A}$ as a black box providing an intraprocedural analysis and extend it to treat method call-statements.
- A $direction \in \{forward, backward\}$ of the analysis.
- An initialiser function $init : V \to D_s$ that provides an initial abstract state for a methods entry (forward) or exit (backward) node.

- An unification function $unify : (D_s,\ D_s) \rightarrow D_s$ to combine two abstract states.
- An optional argument $k \in \mathbb{N}$ that bounds the call-string length.

Abstract State

Elements in D_s represent a *state* and the analysis should collect a state providing information about the programs variables for every location in the program. Let *Ident* be the set of variable identifiers existing in the program. For every state $d_s \in D_s$ there exists a set $i \in \mathcal{P}(Ident)$ that represents the set of variable identifiers over which the state is defined. In non-relational domains the state d_s maps each identifier in i to an abstract value. Relational domains on the other hand consist of a set of constraints for the variables in i.

2.5.4 Creating the Supergraph and Callgraph

We use the same notation for the supergraph as in the previous section. The list of CFGs is translated into a supergraph $G_{super} \overset{\text{def}}{=} (V,\ E \cup E_{interproc})$ with $E \overset{\text{def}}{=} E_{call} \cup E_{return}$ in a straightforward way in linear time in terms of the number of nodes: First a new graph G_{super} is created and every CFG with all its nodes and edges are added to V and E correspondingly. This forms a disconnected graph with as many components as CFGs. Then all statements of every CFG are traversed *once* and when a call-statement is encountered two new directed edges are created. One edge that connects the call-statement to the entry-node of the callee is added to E_{call}. The other edge connecting the exit-node of the callee with the call-statement is added to E_{return}. Note that this results in a directed graph that may be disconnected. Listing 2.4 shows a program that results in a disconnected supergraph consisting of two connected components. The information in the supergraph is also used to build a callgraph $G_c \overset{\text{def}}{=} (V_c,\ E_c)$. The set V_c consists of all entry-nodes in all CFGs and the set of edges can be defined as $E_c \overset{\text{def}}{=} \{(m', n') \mid (m, n) \in E_{call} \wedge m'\ (n')\ \texttt{is the entry-node in the same CFGs as}\ m\ (n)\}$. Figure 2.2 shows a callgraph for the program in Figure 2.1.

```
1 method connected() {
2   callee()
3 }
4 method callee() {}
5 method disconnected() {}
```

Listing 2.4: A program with a disconnected supergraph

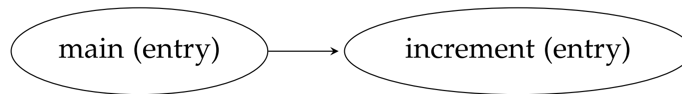

Figure 2.2: Callgraph created using the supergraph in Figure 2.1.

2.5.5 Determining where to Start the Analysis

The analysis is started at a specific location in the program with an initial abstract state supplied by the user of the framework. Where to start exactly depends on the direction of the analysis and on whether a top-down or bottom-up analysis should be run. Here we describe a top-down analysis where we always start with an entry-method of the program and then proceed into the callees. In Section 4.2 we present a bottom-up analysis as well. After the entry-method(s) have been determined the forward or backward analysis can be started, respectively, at the entry-node or exit-node of the method.

We consider any method without an in-edge in the callgraph as an entry-method and use it as a starting-point with the initial abstract state provided by the initialiser function *init()*. If every node in the callgraph contains at least one in-edge then a callgraph could look like the one marked with Greek letters in Figure 2.3. The nodes α and β form a strongly connected component and it is not clear whether a client of this program would call α or β and we therefore assume both nodes as possible entry-methods.

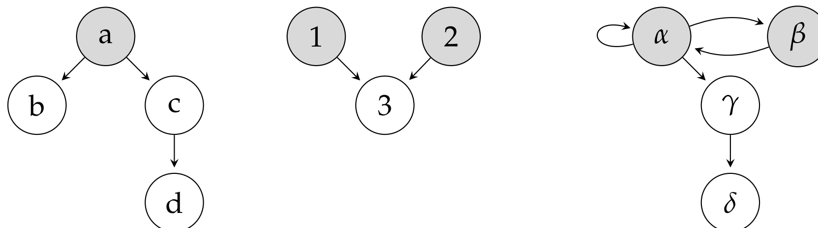

Figure 2.3: Three different callgraphs with their entry-points (highlighted).

For a callgraph that looks like the one on the right in Figure 2.3 the entry-nodes α and β can be derived the following way:

- Compute the strongly connected components (SCC) in the callgraph.
- Build a new graph, which we call *condensed callgraph*, and add a vertex for each SCC.
- Add a directed edge connecting two components if they are not connected yet and if there exists an edge (m, n) in the callgraph where m and n are in different SCCs.

- Every vertex without in-edges in the new graph now represents a set of procedures that should be treated as entry-methods.

This pre-processing must be run no matter what shape the given supergraph has. Only searching for nodes without in-edges is not enough because a supergraph might for example contain all methods marked with numbers and Greek letters in Figure 2.3. The described pre-processing would find 1, 2, α and β as entry-methods whereas only looking for nodes without in-edges would omit the methods α and β and never analyse them.

2.5.6 Method Call Treatment

The analysis $\mathcal{A}$ passed to the framework is an intraprocedural analysis and therefore cannot handle method call-statements. To be able to use $\mathcal{A}$ as is we rewrite method calls into a sequence of statements that the intraprocedural analysis can handle. Let $c \in V$ be a node in the supergraph representing a method call. Let $n_p \in \mathbb{N}_0$ be the number of parameters and $n_r \in \mathbb{N}_0$ the number of return values the called method declares. Let $d_{pre} \in D_s$ be the abstract state before the method call-statement in a forward analysis and $d_{post} \in D_s$ the abstract state directly after the method call in a backward analysis. Let $i_{pre}, i_{post} \in \mathcal{P}(Ident)$ be the set of variable identifiers over which d_{pre}, respectively d_{post} are defined. Depending on the direction of analysis (forward/backward) calling a method is treated differently.

Forward. We create and add n_p new identifiers to i_{pre}. These identifiers are *in-transfer-variables* and we use them to pass arguments into the callee. In our implementation in Section 3.3 we used *arg_#* prefixes for those identifiers simply because they are not valid identifiers in the programming language we target.

Backward. In a backward analysis n_r new identifiers are added to i_{post}. These identifiers are *out-transfer-variables* and they are used to return values from the callee into the caller. Similarly to the forward case we implemented this using a prefix *ret_#* that is not valid in the targeted programming language.

The actual method-invocation is then reduced to evaluating the arguments, branching into the callees CFG and after returning assigning the return values to the target variables defined in the call-statement. A simple way to let the black box (the intraprocedural analysis) $\mathcal{A}$ treat call-statements is by rewriting the node c in the CFG into a sequence of statements that assign the parameter-expressions to the in-transfer-variables and the out-transfer-variables to the target variables of the method call-statement. The unsupported call-statement is now replaced and the given intraprocedural analysis can interpret the rewritten nodes. Given a method call *r1, r2, r3 := foo(exp1,*

exp2) the node in the CFG would be rewritten to a sequence of statements as shown in Figure 2.4. It should be noted that our implementation does not actually rewrite the supergraph but performs these assignments on-the-fly when a call-statement is encountered. Before proceeding to the next statement after the method call the (temporary) in-/out-transfer-variables are removed from the current state.

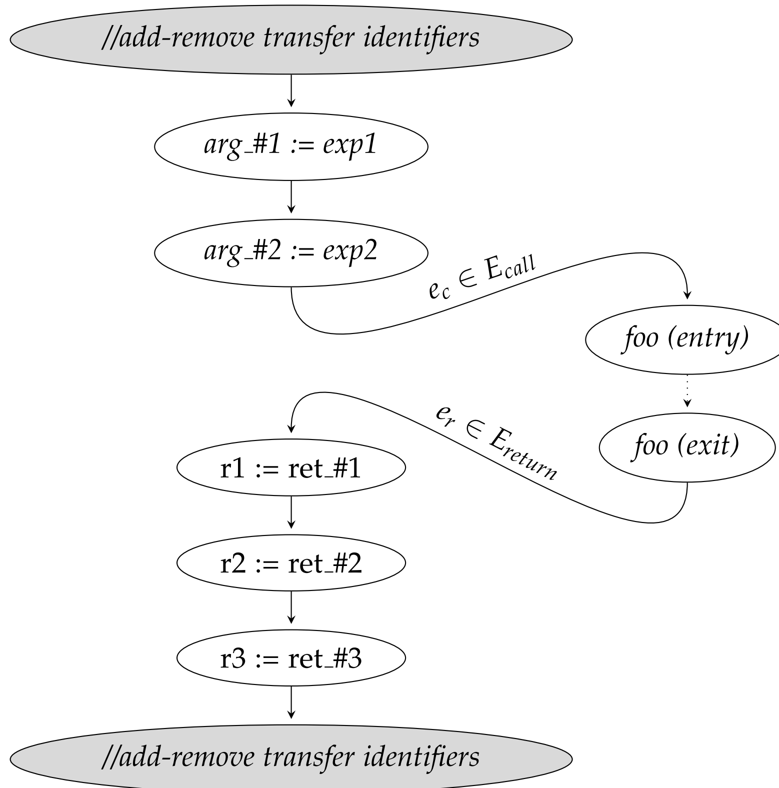

Figure 2.4: The call-statement *r1, r2, r3 := foo(exp1, exp2)* after rewriting.

2.5.7 Entering and Leaving a Procedure.

The problem. Recall that the original call-string based analysis simply tags an abstract value and propagates it along an edge $e \in E_{interproc}$ into a callee or back to a caller. Let us assume we want to track the numerical values of the variables in the program in Listing 2.5. Starting at the *main* method and assuming the framework has been instantiated with *integer interval analysis* as $\mathcal{A}$ and *forward* direction, at the call at line 3 we would propagate the abstract state $\{x \mapsto [0, 0], r \mapsto \top, arg_\#1 \mapsto [0, 0]\}$ tagged with the call-string *Nil* over the call-edge. For this simple program several problems arise.

- Simply propagating the abstract state into the callee does not set the value of the parameter *l*.
- The declaration of variable x at line 6 conflicts with the propagated value of x from the caller.
- At the exit-block of the callee the local variable with identifier x has the abstract value $[1, 1]$. Propagating this state back over the return-edge into the caller would overwrite the main-methods own local variable x.

```
1 method main() returns(r: Int){
2   var x: Int := 0
3   r: = callee(x)
4 }
5 method callee(l: Int) returns (r: Int) {
6   var x: Int := 1
7   r := x + l
8 }
```

Listing 2.5: Main and callee use overlapping variable names.

Using a Transfer-Value. Reps et al. [19] allow an edge $e \in E_{interproc}$ in the supergraph to have a transformer instead of just propagating the abstract state. Our approach creates and propagates a *transfer-value* through the edges in $E_{interproc}$. A transfer-value $d_t \in D_s$ is an abstract state that is propagated into or out of a called method. This abstract state should carry all necessary information for the analysis of the callee. For the program in Listing 2.5 and the framework instantiated with *forward* direction and *integer interval analysis*, d_t would carry the abstract value of the argument into the *callee* and the abstract value of the return variable out of the method. We require the transfer-value to not use any identifiers that exist in the caller or callee. Instead of propagating the whole state from the caller into the callee only information that is necessary[3] to analyse the callee is transferred. Instead of transferring *l* or *r* a temporary identifier *arg_#1* for the first argument to the callee and *ret_#1* for the first return variable could be used. That is, the set of identifiers over which the propagated state is defined only contains identifiers for the temporary in-/out-transfer-variables. The call in Listing 2.5 would propagate $\{arg_\#1 \mapsto [0, 0]\}$ into the callee and $\{ret_\#1 \mapsto [1, 1]\}$ back to the caller. Propagating a transfer-value solves the problems regarding procedure parameters and shared local variable names outlined before. It remains to be explained how a transfer-value can be merged with an existing abstract state.

[3] It depends on the actual analysis what information is necessary. For a numerical analysis it is enough to transfer the abstract values of arguments and global variables.

Abstract State unification. The framework needs a way to combine two abstract states into one. Since this heavily depends on the given analysis the generic framework requires a unification function $unify : (D_s,\ D_s) \rightarrow D_s$ that is able to combine two abstract states. Let $d_1, d_2 \in D_s$ be two abstract states with $i_1, i_2 \in \mathcal{P}(Ident)$ being their set of variable identifiers over which they are defined. Unifying the two states $d_u := unify(d_1, d_2)$ should result in a state d_u whose set of identifiers $i = i_1 \cup i_2$ contains all identifiers of both states. Non-relational domains should unify the states in such a way that the abstract value for each of the variables is preserved. Our framework uses $unify$ to merge an abstract state with a transfer-value. Because a transfer-value by definition uses new identifiers in general two states passed to $unify$ do not share common identifiers. If they do share common identifiers then it is up to the actual analysis to define how two states should be unified. In Section 3.4.2 we provide information about such a case for the octagon abstract domain for integers.

Passing call/return edges. Now that transfer-values and the $unify$ function are introduced it remains to be explained what the framework does when passing an edge $e \in E_{interproc}$. The following two paragraphs describe the forward analysis case. For a backward analysis minor modifications are necessary and we provide an implementation in Section 3.3.7.

Passing a call-edge. When the analysis passes an edge $(m, n) \in E_{call}$ the node n is a method-entry node and $d_t \in D_s$ is the corresponding transfer-value. To be able to analyse the called procedure an initial abstract state needs to be created. This initial state is $d_{0,n} := unify(init(n),\ d_t)$. This state also contains the abstract values for the in-transfer-variables. Similarly to the preparation of the method call a sequence of assignment statements need to be introduced to assign the in-transfer-variables to the actual method parameters. After these assignments the temporary transfer-variables are removed from the state.

Example 2.1 *Let us assume the framework is instantiated with an integer interval analysis, forward direction and an initialiser function that sets every variable to top. For the method call in Listing 2.5 a transfer-value* $d_t = \{arg_\#1 \mapsto [0,\ 0]\}$ *would be used to create an initial state* $d_{0,n} = \{l \mapsto \top,\ r \mapsto \top,\ arg_\#1 \mapsto [0,\ 0]\}$.

Passing a return-edge. When the analysis encounters a return-edge $(e, c) \in E_{return}$ the node e is the exit-node of the called method. Let $d_{ret} \in D_s$ be the previously described transfer-value containing the abstract values of the return variables. Let $d_{call} \in D_s$ be a state at node c after evaluating any arguments. This state contains the abstract values of all caller-local variables but also the *transfer-variables* created to call the callee. To merge the effect of the callee into the caller state at node c these two states are unified into

$d := unify(d_{call}, d_{ret})$. After unifying the states the assignment statements (see Figure 2.4) will assign the *transfer-variables* to the actual target variables of the method call.

Example 2.2 *For the call in Listing 2.5 and assuming the framework is instantiated with an integer interval analysis the transfer-value would be* $d_{ret} = \{ret_\#1 \mapsto [1, 1]\}$ *and the state after evaluating the arguments would be* $\{x \mapsto [0, 0], r \mapsto \top,$ $\text{arg_\#1} \mapsto [0, 0]\}$*. The unified state* $d = \{ret_\#1 \mapsto [1, 1], x \mapsto [0, 0], r \mapsto \top, arg_\#1 \mapsto [0, 0]\}$ *would contain all information necessary to assign ret_#1 to r and therefore merging the effect of callee() back into the method main(). After the assignment and removing all temporary variables the state in main() after the method call would be* $\{x \mapsto [0, 0], r \mapsto [1, 1]\}$.

2.5.8 Computing the Analysis Result

Finally the call-strings approach discussed in Section 2.4.1 combined with our special treatment for method-arguments and returns yields a call-string tagged abstract interpretation with support for local variables in recursion and procedure parameters. This analysis can be solved with standard fixed point solving techniques like a worklist algorithm in [17]. For non-recursive programs this is guaranteed to terminate [21]. In case of recursion we over-approximate the solution using Sharir and Pnuelis *call-string suffix approximation* as described in Section 2.4.2. Our special handling for call- and return-edges can directly be applied to the approximate case too.

Chapter 3

Sample

Sample ('Static Analyzer for Multiple Programming LanguagEs') [2] is a generic static analyser based on abstract interpretation [5] developed at the Chair of Programming Methodology at ETH Zurich. It is generic with respect to the underlying abstract domain, the heap analysis that approximates all runtime heap structures and the analysed language because Sample translates the language under analysis into an intermediate representation ([2]). Generic intraprocedural analyses are supported but an implementation for interprocedural analysis is missing. In this chapter we first give an overview of Sample's technical code base before this project and then present how Sample has been extended with the generic interprocedural analysis described in the previous chapter. Sample's source code including the work of this project is available online at: `https://bitbucket.org/viperproject/sample`.

3.1 Overview

The Sample code base consists of many different modules and is used in multiple research projects. Some of them being *TouchGuru* [1] or *Viper* [16] introduced below. Our work focused on the parts of Sample that interact with Viper and therefore its verification language Silver. Sample includes translations from Silver to Sample's internal representation and back. We did a small change to the existing implementation to also translate method call-statements.

3.1.1 Broad Overview

Figure 3.1 from the Sample project page [2] gives an overview of Sample's overall architecture. The top-left component is the source code to be analysed. Researchers can extend components, for example the compiler,

to integrate their own analyses or add more supported programming languages to Sample. The core of Sample consists of the internal representation of the analysed program (see Section 3.2.1), the fixpoint engine to over-approximate the analysis result and data structures to represent the result of the analysis for a given CFG. The components at the bottom of Figure 3.1 show the output to a developer like for example assertion warnings or inferred specifications (See [2]). Since Sample's focus so far has been intraprocedural analysis the core components work on one CFG at a time and return an analysis result for this CFG. The biggest part of our work focused on these core components to add the interprocedural analysis.

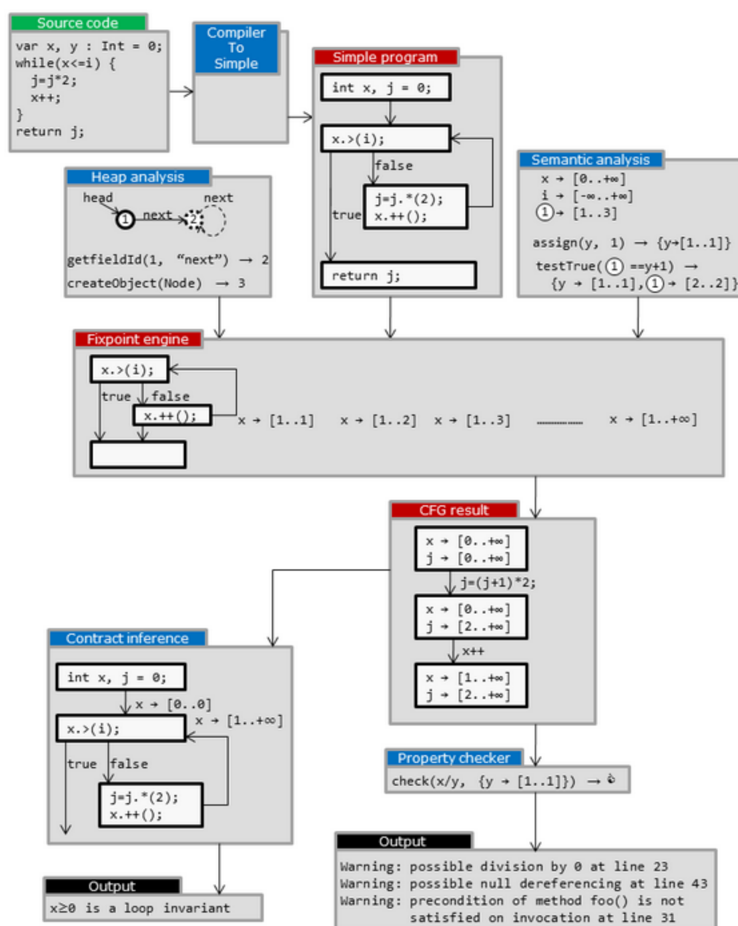

Figure 3.1: Sample overview taken from the project page [2].

3.1.2 Programming Language

Sample is written in the programming language Scala[1]. It provides object orientation but also the benefits of functional programming while being compatible to Java code. We will use some Scala code examples and assume the reader has a basic understanding of the language and its constructs. Readers experienced with other programming languages can get up to speed quickly with the *Scala School*[2]. All implementation work in this project has been done using Scala.

3.2 Sample for Intraprocedural Analyses

In this section we explain Sample's existing intraprocedural analysis infrastructure and highlight parts of the code base that are especially relevant for our implementation.

3.2.1 Program Representation

Control Flow Graph. A program is represented as a control flow graph (CFG). Unlike the CFG used in Chapter 2, where every node represented a statement in the program, the class *SampleCfg* represents a graph whose nodes are of type *Block*. Implementations of the Scala trait *Block* are *StatementBlock*, *PreconditionBlock*, *PostconditionBlock*, *LoopHeadBlock* and *ConstrainingBlock*. A *Block* can contain zero or many statements, depending on its type. Edges in the CFG can either be conditional or unconditional edges. A conditional edge is annotated with its condition and represents the case when the body of an *if* or *while* statement is entered. The method in Listing 3.1 is represented by the CFG in Figure 3.2. Every *SampleCfg* has an entry-block and may or may not have an exit-block. The first line of each block in Figure 3.2 defines its type and the number in parentheses uniquely identifies that block. Statements inside a method (CFG) can be identified by a *BlockPosition*. A *BlockPosition* is a data structure that contains a reference to a block and a zero-based *index* for the element it points to. The position *BlockPosition(18, 1)*[3] would for example point to the statement *t2 = MC(t1)* in the block identified by the number 18.

Statements. In Sample there is no distinction between expressions and statements. Everything is represented as a statement. Every subtype of *Statement* has an attribute *programpoint* of type *ProgramPoint* that maps the statement to the actual location in the source code. Figure 3.2 shows that arithmetic expressions like *n + 11* are represented as a method call *n.+(11)*.

[1] http://scala-lang.org/
[2] https://twitter.github.io/scala_school/
[3] The actual implementation keeps a reference to the block and not the block-number.

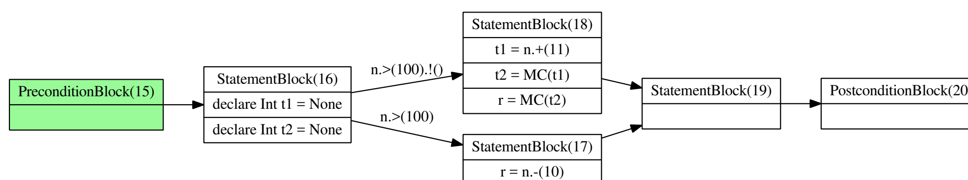

Figure 3.2: Control flow graph of the McCarthy 91 function.

```
1  // McCarthy91:
2  // if (n>=101) then r:= n-10 else r: = 91
3  //
4  method MC(n: Int) returns (r: Int) {
5    var t1 : Int
6    var t2 : Int
7    if (n>100){
8      r := n - 10
9    } else {
10     t1 := n + 11;
11     t2 := MC(t1);
12     r := MC(t2);
13   }
14 }
```

Listing 3.1: Implementation of the McCarthy 91 function in Silver.

3.2.2 State Representation

The state of an analysis is represented as a subtype of the trait *State*. The state forms a lattice, that is, it is a partially ordered set and for any two elements a least upper bound and greatest lower bound exists. As briefly explained in Section 2.2 abstract interpretation iteratively applies abstract transformers until a fixed point is reached in the lattice.

```
1  trait State[S <: State[S]] extends Lattice[S]
```

The type *SilverState* extends *State* and already provides some transformers related to the Silver programming language. Developers implementing a new static analysis need to extend this type and provide the transformers for their abstract domain by implementing the abstract methods defined by *State*. There exist various implementations such as *IntegerIntervalAnalysisState* for the integer interval abstract domain or *IntegerOctagonAnalysisState* for the integer octagon abstract domain.

3.2.3 Analysis Results

The intraprocedural analyses in Sample store the result of an analysis in a data structure of type *CfgResult*. A *CfgResult* stores for any location in the source code identified by a *ProgramPoint* the state of the analysis directly before and after that location. In an intraprocedural analysis there is one *CfgResult* per analysed method.

3.2.4 Intraprocedural Analysis Runner

A *SilverAnalysisRunner* is responsible for running the static analysis of a Silver program using Sample. The analysis runner compiles a Silver program into Sample's representation and then analyses each method and outputs the result. A *SilverEntryStateBuilder* is used to create the initial state with which the analysis of a method is started. Figure 3.3 shows a sequence diagram of an intraprocedural analysis implemented in Sample. The entry state builder can be understood as the implementation of the $init()$ function described in Chapter 2.

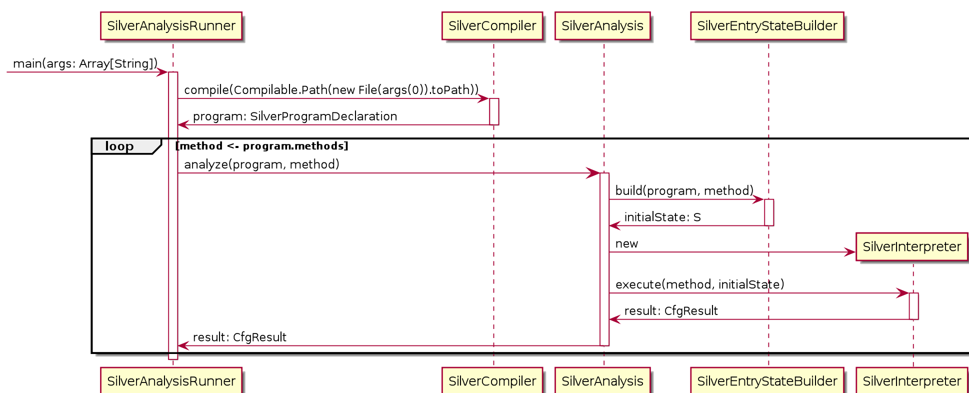

Figure 3.3: Sequence diagram of an intraprocedural analysis.

Except the *SilverCompiler* all types shown in the diagram require a generic argument S representing the abstract state (see Section 3.2.2). Sample's implementation makes heavy use of Scala generics and therefore allows a developer to compose different analyses by combining the necessary components. A developer implementing a new kind of analysis would need to provide an entry-state builder and implement a subtype of *SilverAnalysis* in which depending on the direction of analysis a *SilverInterpreter* is invoked.

3.2.5 Computing a Fixed Point Solution

Sample provides two implementations of a *SilverInterpreter* that interpret a program in either forward or backward direction (Figure 3.4). These inter-

preters start at the first (forward) or last (backward) position in a CFG and iteratively apply the transformers on an initial state until a fixed point is reached. The fixed point is computed using a worklist algorithm [17]. That is, blocks of the CFG are added to a worklist as long as the states propagated along the edges of the CFG change. In Listing 3.2 a simplified Scala implementation of the worklist algorithm for the forward analysis is shown. The method *execute()* takes a *SampleCfg* and an initial state of generic type S as argument and computes the analysis.

```
1  def execute(cfg: SampleCfg, initial: S): CfgResult = {
2    val cfgResult = initializeResult(cfg, initial.bottom())
3    val worklist = mutable.Queue[SampleBlock](cfg.entry)
4    val iterations = mutable.Map[SampleBlock, Int]()
5
6    while(worklist.nonEmpty) {
7      val current = worklist.dequeue()
8      val iteration = iterations.getOrElse(current, 0)
9      val oldEntryState = cfgResult.getStates(current).head
10
11     // join incoming states
12     var entryState = bottom
13     val edges = cfg.inEdges(current)
14     for(e <- edges) {
15       val predecessorState = cfgResult.getStates(edge.source).last
16       // ... handling of conditional edges omitted
17       entryState = entryState lub predecessorState
18     }
19
20     // widening
21     if(edges.size > 1 && iteration > SystemParameters.wideningLimit) {
22       state = oldEntryState widening entryState
23     }
24
25     // interpret the statements in the block
26     if(!(entryState lessEqual oldEntryState)){
27       val states = ListBuffer(entryState)
28       current.statements.foldLeft(entryState) {
29         (preState, statement) =>
30           val successor = executeStatement(statement, preState)
31           states.append(successor)
32           successor
33       }
34
35       // save states, iteration count and enqueue successors
36       cfgResult.setStates(current, states.toList)
37       worklist.enqueue(cfg.successors(current): _*)
38       iterations.put(current, iteration + 1)
39     }
40   }
41 }
```

Listing 3.2: Simplified worklist algorithm for a forward analysis.

Let $(D, \sqsubseteq, \sqcup, \sqcap)$ be the lattice representing states in the analysis. The worklist is a queue of blocks in the CFG and it is initialised with the entry-block of the currently processed method (line 3). For every element in the worklist an $entryState := \bigsqcup\{s \mid s \in D, \texttt{ s is a state propagated along an in-edge}\}$ defined as the join of all incoming states is computed (line numbers 12 to 18). As long as the entry-state is strictly greater than the entry-state of a previous iteration all statements of the block are executed by applying the transformers representing those statements (lines 26 to 33). The intuition is, that as long as the entry-state for a block is strictly greater than in a

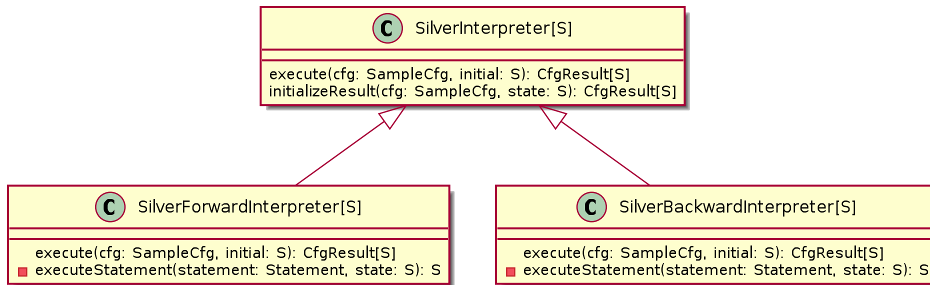

Figure 3.4: Intraprocedural Silver interpreters. The squares denote *private* methods.

previous iteration new information is available and a fixed point has not been reached yet. If one blocks entry-state did not change between two iterations there is no need to re-analyse its statements. After processing a block the successor blocks in the CFG are added to the worklist (line 37). The backward interpreter works very similarly with the differences that the algorithm starts at the exit-block of the method, statements of a block are executed in reverse order and instead of computing an entry-state an exit-state is computed by joining states propagated along all outgoing edges of the current block. Additionally the backward interpreter adds predecessors of the current block to the worklist. Depending on the system parameter *wideningLimit* widening is applied on the entry-state (forward) or exit-state (backward) to ensure termination of the worklist algorithm in the presence of loops (lines 21 to 23).

3.3 Sample for Interprocedural Analyses

A core goal of the project is to extend Sample with a generic and customisable interprocedural analysis. The existing code is extended in a way to make it reusable for an interprocedural analysis without breaking existing intraprocedural analyses. In this section we present implementation details and design decisions regarding the implemented interprocedural analysis. Basic understanding of Sharir and Pnueli's call-string approach [21] is required (see Chapter 2).

3.3.1 Overview

The previously existing implementation in Sample invoked the *SilverInterpreter* once for each method. But in the interprocedural case the interpreter needs all called methods to be available. Now, the interpreter not only is able to analyse a single control flow graph but also supports that blocks of different control flow graphs are added to the worklist in the same execution

of the worklist algorithm. Since multiple control flow graphs are analysed in the same execution, it is no longer possible to return the analysis result as one CfgResult. First, we introduce the new data structure *ProgramResult* which is the counterpart of *CfgResult* for programs. Then, we explain how a call-string is modelled in the system and we provide details on how the previously existing interpreters have been extended to the interprocedural case. The implemented analysis can be parametrised with a *CallStringLength* and, as before, it is possible to control the widening limit using a system parameter.

3.3.2 Source Code

The described implementation can be found in Sample's source code at bitbucket.org[4]. Most of the implementation was done in the module 'sample-silver'.

3.3.3 ProgramResult

To store the result of an analysis a new type *ProgramResult* has been introduced. A program result can be used to store multiple instances of *CfgResult*. The *ProgramResult* separates different *CfgResults* by a *SilverIdentifier* and an optional *CfgResultTag*. A *SilverIdentifier* is Sample's representation of an identifier in a Silver program. A method-name is for example a *SilverIdentifier*. The *ProgramResult* allows storing multiple *CfgResult* instances per identifier. To make them distinguishable, results can be tagged with any type that extends *CfgResultTag*. If no tag is given, results are marked with *CfgResultTag.Untagged*.

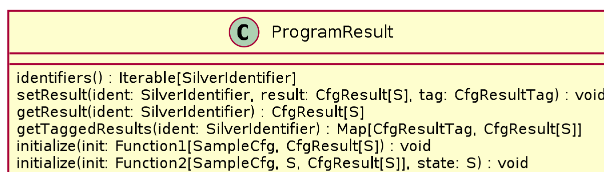

Figure 3.5: Class diagram of ProgramResult.

3.3.4 Call-String Representation

A call-string is represented as an immutable type *CallString*. It implements *CfgResultTag* and can therefore be used to tag CFGs in a *ProgramResult*. The implementation emulates the stack of method calls using a list of *ProgramPoint*. The interface to the *CallString* is similar to the interface of a stack.

[4] https://bitbucket.org/viperproject/sample

The method *push()* grows the call-string and *pop()* shrinks it by removing the last method call. In our implementation a call-string always grows and shrinks by two elements. One element represents the location of the method call-statement and the other element stores the location of the entry-point of the called method.

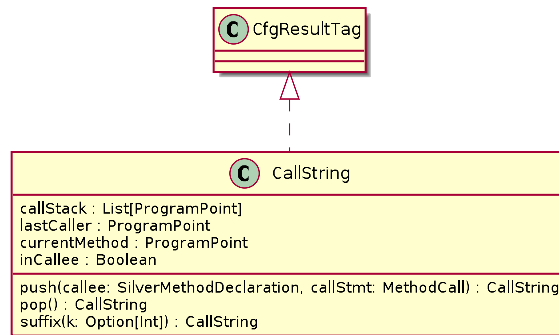

Figure 3.6: Class diagram of CallString.

Call-String Approximation

The method *suffix(k: Option[Int])* is used to compute a call-string's suffix for the approximation described in Section 2.4.2. If a suffix-length k is given then up to k of the last method calls are returned as a call-string. Calling *suffix()* with k=*None* returns the original unmodified call-string.

3.3.5 Interprocedural Analysis Runner

Interprocedural versions of *SilverAnalysisRunner*, *SilverAnalysis* and *SilverInterpreter* have been implemented. Figure 3.7 shows a high-level view of the steps involved in the analysis of a text-file containing Silver source code. Note that compared to the intraprocedural case here the interpreter is invoked once for the whole program and the result is of type *ProgramResult*. Again, the generic arguments have been left out of the diagram for better readability.

A developer can create an interprocedural analysis by providing an implementation of *SilverState*, an *EntryStateBuilder* and then creating a subtype of *InterproceduralSilverAnalysis* that invokes either the forward or backward interpreter. We implemented interprocedural analyses for integer intervals (forward), integer octagons (forward) and strongly live variable analysis (backward).

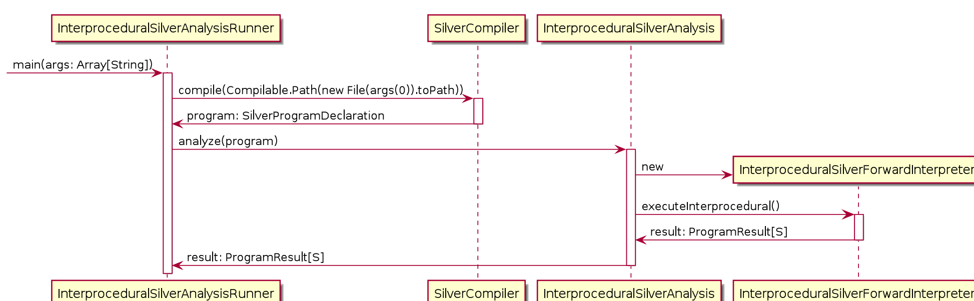

Figure 3.7: Sequence diagram of an interprocedural analysis.

3.3.6 Top-Down / Bottom-Up Analysis

We provide analysis runners and interpreters for both top-down and bottom-up analysis. Here we briefly introduce both analyses and in the rest of the chapter the top-down implementation is explained. Chapter 4 will provide implementation details for the bottom-up analysis.

Top-Down analysis. A top-down analysis is a whole-program or global analysis. That is, the analysis starts at one or many entry-methods of a program with an initial abstract state and proceeds into the called methods. An entry-method resides at the top of the callgraph and is analysed without any assumptions about its parameters. Called methods on the other hand are analysed context-sensitively and therefore some information about their parameters is given from the abstract state at the method-call. The data structure *ProgramResult* separates analysis results for different method calls by the call-string. A top-down analysis can therefore make precise statements about the effect of a method call for a specific caller but for callees the analysis only provides analysis results that depend on a given calling-context. We previously explained in Section 2.5.5 how the entry-methods of the program are determined.

Bottom-Up analysis. In bottom-up analysis we analyse methods modularly starting at the bottom of the callgraph and without any assumptions about their parameters. As soon as a callee has been analysed, all its callers can be analysed as well. When a method calls a callee, the previously computed analysis result is used instead of analysing the callee context-sensitively. For a relational abstract domain this analysis result will provide information about the relationship between parameters and return variables of the called method. Since no information about the actual parameters was used to analyse the callee, reusing this analysis result will over-approximate the effect of the method call in the caller.

3.3.7 Interprocedural Interpreter

For interprocedural top-down analysis two new interpreter traits are provided:

- *InterproceduralSilverForwardInterpreter*
- *InterproceduralSilverBackwardInterpreter*

These interpreters are a specialisation of their intraprocedural counterpart and reuse most of the worklist code previously shown in Listing 3.2. They require a *program* of type *SilverProgramDeclaration* and a *CallStringLength* of type *Option[Int]* as constructor arguments. Since the interpreters analyse the whole program they use *ProgramResult* to store the analysis results. We now discuss the implementation of the forward interpreter in detail and at the end highlight what is different for the interprocedural backward interpreter. The class diagram in Figure 3.8 visualises how the new forward interpreter specialises the existing intraprocedural interpreter (*SilverForwardInterpreter*). In the diagram we only list methods and attributes that are needed for understanding the explanations and most method parameters have been omitted for better readability. The trait *InterprocHelpers* provides functionality that is shared between the forward and backward interprocedural interpreters.

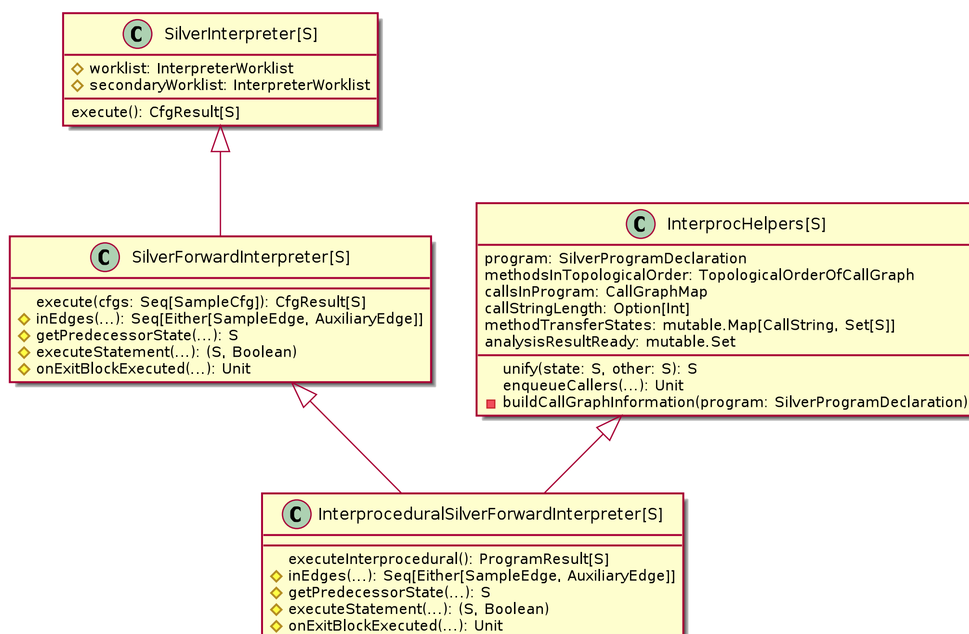

Figure 3.8: Class diagram for the interprocedural forward interpreter. Diamonds and squares denote, respectively, *protected* and *private* visibility.

Outlook

In the rest of this section we provide implementation details for the interprocedural interpreters. After discussing how the starting-points of the analysis are determined we show how the existing intraprocedural interpreters have been extended to make the worklist algorithm reusable for the interprocedural case. This includes refactoring of the existing code and a new worklist type. After that, we explain how a method call-statement is treated for calling a method and returning from the callee back to the caller. Then, we explain how the interpreter implicitly creates a supergraph, that is, how the CFGs are connected into one graph by creating method call-edges or simulating return-edges. Finally we discuss how these changes need to be adapted for the interprocedural backward interpreter.

Starting-Points of the Analysis

In the intraprocedural case the analysis is invoked for every method under analysis. In the interprocedural analysis the Silver program is passed to the interpreter as a whole and the interpreter determines the starting-point on its own. The interpreters analyse the callgraph of the program according to the description in Section 2.5.5 and then enqueue the entry-block (forward) or exit-block of every control flow graph belonging to a possible starting-point. Analysis of the callgraph does not depend on the direction of analysis and is implemented in *InterprocHelpers.buildCallGraphInformation()*. The interpreters keep two attributes *callsInProgram* and *methodsInTopologicalOrder* that are the result of this callgraph preprocessing.

Refactoring the Existing Interpreters

The worklist algorithm in *SilverForwardInterpreter* (Listing 3.2) has been refactored into multiple smaller methods that can be overriden by a subclass. These methods are marked with a diamond in the class diagram of Figure 3.8. Additionally the forward interpreter calls *onExitBlockExecuted()* every time the exit-block of a CFG has been interpreted. This call signals that a method has been interpreted and therefore possible callers may need to be added to the worklist. In the intraprocedural case these helper methods have the following functionality:

- *inEdges()*: Returns all incoming edges in a CFG for a given block.
  (Replaces line 13 in Listing 3.2)
- *getPredecessorState()*: For a given edge, return the state that is propagated along that edge. That is, for an edge $(m, n) \in E$ in forward analysis the last state of block m is returned.
  (Replaces lines 15 in Listing 3.2)

- *executeStatement()*: Execute/Interpret a program statement by applying the corresponding transformer on the current state.
  (This existed in the worklist algorithm but the method was *private* and now needs to be *protected*.).

- *onExitBlockExecuted()*: This method is only necessary for interprocedural interpreters. In the intraprocedural case the method is executed but implemented as a *no-op*.

Another change introduced by the interprocedural analysis is that the interpreter no longer keeps a reference to one *CfgResult* but looks it up depending on the current element under analysis. A different *CfgResult* needs to be used, depending on the current call-string and to which control flow graph the currently processed element belongs.

New Worklist Type

The existing implementation used a queue of CFG blocks as the worklist. The new implementation introduces a new type called *WorklistElement*. A *WorklistElement* consists of a *BlockPosition* and a boolean flag *forceReinterpretStatement*. Using a *BlockPosition* allows the interpreter to enqueue a specific position, and therefore a specific statement inside the block of a CFG, for (re-)analysis. Since the worklist algorithm only interprets the statement of a block if its entry-state changed (line 26 in Listing 3.2) it may happen that a specific position in the block was enqueued for re-analysis but the block itself will not be analysed because the entry-state is the same. The boolean flag *forceReinterpretStatement* can be used to override the check at line 26 of the worklist algorithm and then continue to interpret the elements starting at the element identified by the *BlockPosition*. This allows us to enqueue method call-statements to be re-analysed when a callee has been analysed. We therefore simulate a return-edge in the CFG by adding the method call-statement to the worklist after the callee was interpreted. Figure 3.9 shows the *WorklistElement* and its implementations *SimpleWorklistelement* and *TaggedWorklistElement*. A *SimpleWorklistelement* is used for the existing intraprocedural analyses. Using a *TaggedWorklistElement* it is possible to tag an element with a call-string. This is useful to enqueue blocks of the same method multiple times depending on the calling context. New worklist elements are always created by calling *createSuccessorForEnqeueue()* on the current worklist element. This allows a *TaggedWorklistElement* to preserve the same call-string for all worklist elements created for the same CFG.

Now that the intraprocedural interpreter and the worklist have been extended, let us discuss how method call-statements are treated for entering a callee and returning to the caller.

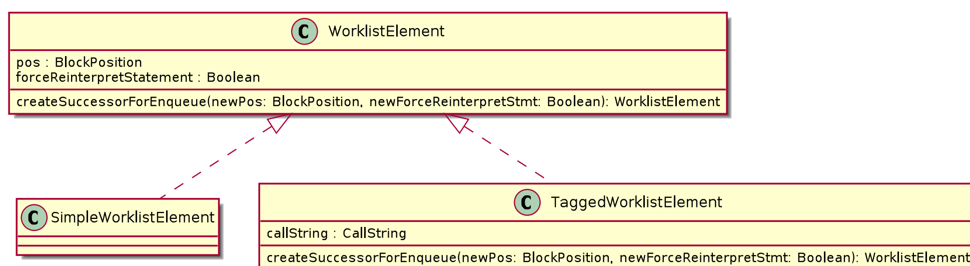

Figure 3.9: New types that replace *SampleBlock* as elements in the worklist.

Calling a Method

In Sample, method calls are represented as an instance of the case class *MethodCall:*

```
1  case class MethodCall(
2    pp: ProgramPoint,
3    method: Statement,
4    parametricTypes: List[Type],
5    parameters: List[Statement],
6    returnedType: Type,
7    targets: List[Statement] = Nil
8    ) extends Statement(pp){
9    //...
10 }
```

The interprocedural interpreters implement method call support by overriding *executeStatement()* and handling statements of type *MethodCall*. Every other statement (for example an assign statement) is handled by the super class as before. In the control flow graph in Figure 3.2 we saw that arithmetic expressions like *n+11* are also represented as a *MethodCall*. What we describe here is only applied to 'real' method calls and we let the parent class handle such expressions. Let k be the optional upper limit on the call-string length (the constructor argument *CallStringLength*). If a *MethodCall* is encountered then the callee is added to the worklist for analysis according to the following steps (as described Section 2.5.6):

- Evaluate the method calls arguments resulting in a new abstract state $d_{preCall}$
- Grow the current call-string into a call-string γ_{call}
- Create a transfer-value d_t that only contains abstract values for the temporary argument variables (e.g. *arg_#1*) by removing all other variable identifiers from $d_{preCall}$

- Save d_t tagged with γ_{call} in a global map called *methodTransferStates*
- Add the called methods entry-block tagged with the k-length suffix of γ_{call} for analysis to the worklist.

We tag the worklist entry with the k-length suffix to ensure that for unbounded recursion only up to k calls are analysed. In *methodTransferStates* the transfer-values are tagged with the full-length call-strings to separate different calls. When analysing the callee, these different transfer-values will then be joined into one state depending on their suffix.

As seen in the worklist algorithm in Listing 3.2 all statements of a block are executed in a loop. If a called method has just been enqueued to the worklist, then the statements following the method call should not be analysed immediately after. Therefore, *executeStatement()* returns a boolean flag telling the super class (the intraprocedural interpreter) whether the remaining statements of a block should be executed or not. This flag is set to true for every statement except a method call whose effect is not available yet (see next section). As a an example, let us consider the program in Listing 3.3. The main-method calls *one()* and then the assert checks whether the return value is 1. After analysing the callee this assert would succeed. But if the interpreter executes the assert statement at line 3 before *one()* has been analysed then Sample would trigger a warning that the assertion might not hold.

```
1 method main() returns(r: Int) {
2   r := one()
3   assert r == 1
4 }
5
6 method one() returns (r: Int){
7   r := 1
8 }
```

Listing 3.3: The assert at line 3 must not be interpreted before the analysis result of the method call is available.

Returning From a Method Call

There are two reasons for the interpreter to execute a *MethodCall* statement. On the one hand, the method should be enqueued for analysis as described in the previous section. On the other hand, the called method has been analysed and the result should be merged back into the caller. Using a global attribute *analysisResultReady* the interpreter keeps track of which method and call-string combinations have been analysed yet. If the analysis result of the called method and call-string is ready then the effect of the callee is

merged back into the caller state. This is done according to the description in Section 2.5.7 with the following steps:

- First the steps for preparing a method call (as discussed in the previous section) are taken. This results in a state $d_{preCall}$ and the call-string of the called method γ_{call}
- Using the k-length suffix of γ_{call} the *CfgResult* of the called method is retrieved from the *ProgramResult*. The last state in the *CfgResult* which we call d_{exit} represents the exit state of the called method.
- All local variables declared in the callee are removed from d_{exit}. Return-variables are renamed into temporary out-transfer-variable names (prefixed with *ret_#*) and parameters that exist in d_{exit} are renamed to temporary in-transfer-variable names (prefixed with *arg_#*)[5].
- The effect of the callee is merged into the caller state by calling $unify(d_{preCall}, d_{exit})$ (Section 2.5.7) and then assigning the temporary return variables (denoted by *ret_#* prefixes) to the targets of the *MethodCall*.
- At the end, all temporary in-/out-transfer-variables are removed from the abstract state.

The function $unify()$ must be provided by the abstract state and is implemented by handling a *UnifyCommand*. We provide implementations of $unify()$ for integer interval domain, integer octagon domain and an abstract domain for live variable analysis. The described variable renaming results in only propagating a transfer-value along a return-edge in the implicit supergraph. After the effect of the callee has been merged back into the caller then *executeStatement()* returns the boolean flag true to the super class to signal that the remaining statements in the block should be executed too. Please refer to Example 2.2 in Section 2.5.7 for an example.

Creating In-Edges For Called Methods

Our implementation does not keep a supergraph in memory but only CFGs and call-edges are created on demand during interpretation. The modified worklist algorithm in Listing 3.2 calls *inEdges()* for every currently processed worklist element. In the intraprocedural case this returns the in-edges of the control flow graph. In the interprocedural case the current worklist element is tagged with a call-string. Let γ_{curr} be the call-string that tags the currently analysed element. The global map *methodTransferStates* contains all transfer-values tagged with call-strings. Let k be the optional maximum call-string

[5]As explained in Chapter 2, it depends on the specific analysis what a transfer-value contains. For return-edges we also transfer the renamed parameters to preserve the relationship between parameters and return variables in the case of relational abstract domains.

length and let $calls \stackrel{\text{def}}{=} \{v \mid (\gamma, v) \in \texttt{methodTransferStates},\ \gamma.suffix(k) = \gamma_{curr}.suffix(k)\}$ be all calls to this method tagged with the same k length suffix. If the current element on the worklist represents an entry-block of a CFG then for every transfer-value in $calls$ the method *inEdges()* dynamically creates an edge that represents a method call with the given transfer-value. The worklist algorithm will then compute the join of all incoming states and analyse the method with that joined state. If the length of the call-string is not restricted, that is, the optional k has the value *None*, then $calls$ will always contain only one transfer-value. If there is a call-string length limit k then *methodTransferStates* can contain multiple call-strings with the same k-length suffix and multiple in-edges are created.

Creating Initial States for In-Edges

Using the method *getPredecessorState()* the worklist algorithm fetches a state that is propagated along an in-edge. For edges that represent a method call the interprocedural interpreter creates a new initial state with information from the method call. Let d_t be the transfer-value propagated along that edge. Let $(m, n) \in E_{call}$ be the current edge. Using $d_{init} \stackrel{\text{def}}{=} unify(initial(n), d_t)$ an initial state containing the transfer-value is created. Then the abstract values of the temporary in-transfer-variables are assigned to the parameters of the method and finally the temporary variables are removed from the state. The $init()$ function as described in Section 2.5.3 can be provided to the interprocedural analysis by implementing the abstract method *SilverEntryStateBuilder.buildForMethodCallEntry()* in a subtype of *SilverEntryStateBuilder*.

Enqueueing Callers For Analysis

Now that we discussed how edges for a method call are created, we will explain how the transfer-values are propagated back into the caller. The intraprocedural interpreter has been extended to call *onExitBlockExecuted()* every time the exit-block of a method has been interpreted. At that time the analysis result of a callee is available and this is remembered by adding the *(call-string, CFG)* tuple to *analysisResultReady*. Additionally the interprocedural interpreters enqueue all callers for re-analysing the *MethodCall* statement, and therefore merging the effect of the callee into the caller. Let γ_{callee} be the current call-string with methods and attributes according to the call-string class diagram in Figure 3.6. Let k be the optional call-string length limit. In case there is no upper bound on the length (k is *None*) the only possible caller is $\gamma_{callee}.lastCaller$ and the *BlockPosition* of that method call together with the call-string $\gamma_{callee}.pop()$ can be added to the worklist. If the call-string is approximated then all existing call-strings are checked and when their suffix matches the suffix of γ_{callee} all these locations are enqueued for analysis. If for example $k = 0$ then every call-statement that called the

current method would be added to the worklist. This corresponds to the context-insensitive approach. When adding call-statements to the worklist, the boolean flag *Worklistelement.forceReinterpretStatement* is set to true. As previously explained this forces the worklist algorithm to re-analyse a block starting with the enqueued statement even though the entry-state of the block did not change.

Differences Between Forward and Backward Interpreters

To wrap up the interpreter details we now briefly explain what differs to the implementation of the interprocedural backward interpreter. The same refactorings previously described have also been applied to the backward interpreter. Since the analysis runs backward the analysis result of a callee is marked as available when the entry-block has been interpreted. Therefore, backward interpreters provide a method *onEntryBlockExecuted()* to signal an available result. Furthermore, *executeStatement()* enqueues the last block of a callee's CFG for analysis and instead of passing information about the arguments into the callee, the transfer-value passes information about the callee's return variables into the called method. Growing and shrinking the call-string, as well as, enqueueing callers for analysis when the result of a callee is available work the same and are implemented in a shared Scala trait.

3.4 Implemented Analyses

In this section we briefly introduce the top-down analyses that have been implemented to evaluate the implementation. We implemented two numerical forward analyses and a variant of live variable analysis as a backward analysis.

3.4.1 Interprocedural Integer Interval Analysis

The Scala object *InterproceduralIntegerIntervalAnalysis* provides a numerical forward analysis based on interval abstract domain [4] for integers. The call-string length is bounded to *SystemParameters.callStringLength* which is 5 by default. A second Scala object *ContextInsensitiveInterproceduralIntegerIntervalAnalysis* provides an integer interval analysis with a call-string length limited to zero length and therefore resulting in a context-insensitive analysis. Integer interval is a non-relational abstract domain and the abstract state tracks the abstract value of each variable on its own. An implementation of integer interval domain was given before the project.

Implementation of init(). The *init*() function is implemented using *IntegerIntervalAnalysisEntryState.build()*. When entering a callee all parameters

are initialised with the abstract value $\top$.

Implementation of unify(). Unifying two integer interval states is implemented as *join* of two abstract states in *IntegerIntervalAnalysisState.command()*:

```
1 case UnifyCommand(other) => other match {
2   case o: IntegerIntervalAnalysisState => this lub o
3   case _ => super.command(cmd)
4 }
```

Assuming two states do not share any identifiers then the *join* (operator *lub* in the Scala code) will preserve the abstract values of all identifiers in their state. In the case of entering a callee the two states passed to $unify()$ never share identifiers. But when the result of a method call is merged back into the caller the state $d_{preCall}$ would contain identifiers for in-transfer-variables and the local variables in the caller (Section 3.3.7, 'Calling a Method'). After renaming, d_{exit} would contain identifiers for in-transfer-variables and out-transfer-variables (Section 3.3.7, 'Returning From a Method Call'). The two states $d_{preCall}$ and d_{exit} therefore have the in-transfer-variable identifiers in common. But since we will not use those temporary variables in the caller any more (they will be removed after unify) *join* works for that case too.

3.4.2 Interprocedural Integer Octagon Analysis

A relational forward analysis was implemented using the octagon abstract domain [15] for integers. Like integer intervals the implementation of the abstract domain itself existed in Sample before this project. The Scala object *InterproceduralIntegerOctagonAnalysis* provides the instantiation and it requires the path to a Silver source file as the first command line argument. By default the call-string length is bounded by *SystemParameters.callStringLength* but the limit can also be passed as an argument to the analysis.

Implementation of init(). Similarly to integer intervals $init()$ uses the entry state builder provided by the existing intraprocedural analysis. The initial state contains variables for all arguments but no constraints.

Implementation of unify(). Two states are unified by creating a state containing all identifiers from both given abstract states. Then the *meet* of the constraints sharing identifiers is computed. This is similar to the unify[6] function in Apron [12] a library of numerical abstract domains for static analysis.

[6] http://apron.cri.ensmp.fr/library/0.9.10/apron/apron_103.html

Example 3.1 *Let us assume a method is called and the caller passes the variable* $b = 15$ *as an argument to the callee. A state in the caller with constraints* $[b = 15, arg_\#1 = b]$ *and a state* $[arg_\#1 = ret_\#1 + 1]$ *returned from the caller should be unified into a state in which for example* $[b == ret_\#1 + 1]$ *would exist too.*

The fact that, when returning from a callee the two unified states share the identifiers of the in-transfer-variables (prefixed with *arg_#*) ensures that the octagon domain can infer a relationship between arguments passed to a callee and return variables.

3.4.3 Interprocedural Strongly Live Variable Analysis

Live variable analysis is a backward analysis that starts at the end of a procedure. A variable is considered to be live at position *p* if its value is used in the program before the variable is redefined. Strongly live variable analysis [10] is a variant of live variable analysis where variables are only considered to be strongly live if they are used in a definition of another strongly live variable. For our implementation we consider the return values of a programs entry-method (usually a main-method) to be strongly live. Therefore, all variables used to define the value of the returned variables are considered to be strongly live too. Listing 3.4 shows the set of live variables for each step when going backward in that procedure.

```
1 method main(x: Int) returns (r: Int) {
2   // parameter x is strongly live at method entry
3   var y: Int := 1
4   // strongly live: {x} (y has been redefined)
5   y := x
6   // strongly live: {y} (r has been redefined)
7   r := y
8   // strongly live: {r} (by definition)
9 }
```

Listing 3.4: Example of intraprocedural strongly live variable analysis.

In the interprocedural case we consider return variables only to be strongly live if they are either the return variables of an entry-method (main-method) or if their return value is used to define another strongly live variable in the caller. The parameter *a* in Listing 3.5 is considered to be strongly live because it is used to define the return value *r* via a method call to *callee()*. Variables used in conditions of an *if* or *while* statement are always considered to be strongly live.

The Scala object *LiveVariableAnalysis* provides an implementation of an interprocedural strongly live variable analysis. By default this also uses the

system wide *SystemParameters.callStringLength* but this can also be controlled by passing a custom length to the constructor.

```
1  method main(a: Int, b: Int) returns (r: Int) {
2    // strongly live: {a}
3    var x: Int := 0
4    // strongly live: {a} (defines i in callee())
5    r, x := callee(a, b)
6    // strongly live: {r} (by definition)
7  }
8
9  method callee(i: Int, j: Int) returns (c: Int, d: Int) {
10   // strongly live: {i}
11   c := i
12   // strongly live: {c}
13   d := j
14   // strongly live: {c} (defines r in main())
15 }
```

Listing 3.5: Example of interprocedural strongly live variable analysis.

Abstract Domain

The abstract domain for strongly live variable analysis was implemented in the trait *LiveVariableAnalysisState*. A variable in the abstract domain can either be live or dead. The Hasse diagram in Figure 3.10 shows the complete lattice for a program with three variables. The top element $\{x,y,z\}$ denotes all variables to be live whereas the bottom element $\{\}$ stands for all variables being dead. Note that unlike octagons or integer intervals this abstract domain is finite.

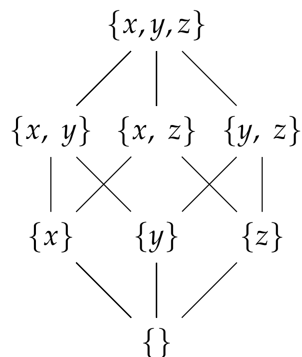

Figure 3.10: Complete lattice for strongly live variable analysis with three variables in the program.

Implementation of init(). Strongly live variable analysis needs two different *init*() functions depending on the context. If an initial state for an entry-/main-method should be created then *LiveVariableAnalysisEntryState.build()* is used. This initialisation function marks all return variables as being strongly live by definition. For a callee on the other hand *LiveVariableAnalysisEntryState.buildForMethodCallEntry()* is implemented to create an abstract state where no variable initially is live. Only those return variables are then set to live that would define a strongly live out-transfer-variable (prefixed with *ret_#*) in the transfer-value that is propagated over the method return-edge.

Implementation of unify(). Unify is implemented as the *join* in the lattice. If a variable was live in any of the two unified states then it will be live in the unified state.

Chapter 4

Specification Inference

Software verification is a powerful technique to check if a program follows its specification. A developer can provide such a specification by annotating methods with preconditions, loop invariants and postconditions. If a specification exists a verifier software can automatically check if the written program follows it. Preconditions must be satisfied before a method is entered whereas postconditions must be true after the method has been executed. Loop invariants are properties that hold before and after each iteration of a loop.

Example. Let us assume a developer wrote the program in Listing 4.1 with the intention that *getElementOrLast()* returns the element at position *pos* in the sequence *xs* or the last element if *pos* is greater than the maximum position. For empty sequences or if a negative position is given then -1 should be returned. For a verifier to be able to prove that the access to *xs[bounded]* at line 6 is within bounds the developer needs to provide a postcondition highlighted at line number 14.

```
1  method getElementOrLast(pos: Int, xs: Seq[Int])
2    returns (r: Int)
3  {
4    var bounded: Int
5    bounded := upperBound(pos, |xs| - 1)
6    if (bounded >= 0) {
7      r := xs[bounded]
8    } else {
9      // error, xs is empty or pos is negative
10     r := -1
11   }
12 }
13
```

```
14 method upperBound(n: Int, upper: Int)
15   returns (r: Int)
16   ensures r <= upper
17 {
18   if (n > upper) {
19     r := upper
20   } else {
21     r := n
22   }
23 }
```

Listing 4.1: The postcondition for *upperBound()* is necessary to prove that access to *xs* is within bounds.

Inferring a specification. A developer may not be used to writing a specification and it can be quite hard to do it correctly. Furthermore, sometimes a specification like the postcondition in Listing 4.1 may be obvious for the programmer but it needs to be written for the verifier anyway. A static analysis can assist developers by inferring some preconditions, loop invariants or postconditions.

Chapter structure. As an extension of this project we implemented an interprocedural specification inference for numerical properties using the octagon abstract domain [15]. A specification is inferred by first running a static numerical forward analysis. Then the specification is extracted from the analysis result and it can either be returned to the user as a list of preconditions, postconditions, and invariants, or a program annotated with the inferred specification can be generated. In this chapter we first provide information about how and what static analysis is run. Then we discuss how the specification is extracted from the analysis results. Finally, implementation details about how the inference can be integrated into Viper IDE [1], the visual environment for developing and verifying programs written in Silver, are given.

4.1 Numerical Abstract Domain (Octagons)

Let us consider the procedure in Listing 4.2 which increments an integer variable. In general, it is useful to have weak preconditions and strong postconditions. Since the implementation of *increment()* has no requirements on the parameter x we can use the weakest possible precondition *requires true* (which could also be omitted). A valid but not very useful postcondition

[1] https://bitbucket.org/viperproject/viper-ide/

could be *ensures true*. A more useful and valid postcondition would be *ensures ret* $>$ *x* whereas the most useful precondition for the method *increment()* would be *ensures ret == x + 1*.

```
1 method increment(x: Int) returns (ret: Int){
2   ret := x + 1
3 }
```

Listing 4.2: Increment method written in Silver.

For this example the most useful specification relates the parameter to the return variable. A simple non-relational numerical domain like integer intervals would not be able to infer the fact that the value of *ret* depends on the value of *x*. In a relational domain like the integer octagon abstract domain this is possible. We will use the interprocedural integer octagon analysis as a basis for the specification inference.

4.2 Top-Down / Bottom-Up Analysis

We infer a specification from the analysis result of a numerical forward analysis in either top-down or bottom-up manner. In this section we provide implementation details regarding the bottom-up analysis. Details about the top-down analysis have already been provided in Section 2.5.

4.2.1 Bottom-Up Analysis Implementation

Like the rest of Sample the bottom-up analysis is implemented in a generic and reusable way. Our implementation provides an abstract type *BottomUpForwardInterpreter[S]* with *S* being the generic argument for the abstract state (*IntegerOctagonAnalysisState* for specification inference). The actual implementation *FinalResultInterproceduralBottomUpForwardInterpreter[S<:State[S]]* of this abstract type requires the constructor arguments *program*, *builder* and *CallStringLength* to run an analysis of the given *program* using initial states provided by the entry-state builder *builder* and the optional call-string bound *CallStringLength*. This bottom-up interpreter specialises the *InterproceduralSilverForwardInterpreter[S]* discussed in Section 3.3.7 and makes use of its call-string analysis functionality. The top-down analysis is adapted to the bottom-up case with the following modifications:

- The analysis starts with callees instead of entry-methods.
- *MethodCall* statements are treated context-insensitively.
- Methods are enqueued to the worklist in different order.

Start of the Analysis.

Recall that the top-down interpreters build a *condensed callgraph* and a topological ordering of it to determine the starting point of the analysis (see Section 2.5.5). We use the same example callgraphs in Figure 4.1 as for the top-down analysis to explain how the starting point of the bottom-up analysis is determined. For a bottom-up analysis the idea is to analyse each procedure on its own, that is, we analyse callees without any assumptions about their parameters and then use this over-approximated analysis result in callers. Therefore, a callee should be analysed before any callers. To achieve this, methods are analysed in reversed order of the topologically ordered callgraph of the program. We analyse the program using the same code as for the other interpreters (*InterprocHelpers.buildCallGraphInformation()*) and topological ordering information is made available as the attributes *callsInProgram* and *methodsInTopologicalOrder* to the interpreter. The last node in the topological ordering is considered to be the starting point of the bottom-up analysis. Since nodes in the topological ordering consist of sets of methods (if they build a strongly connected component) it may be possible that more than one method is considered to be the starting point of the analysis.

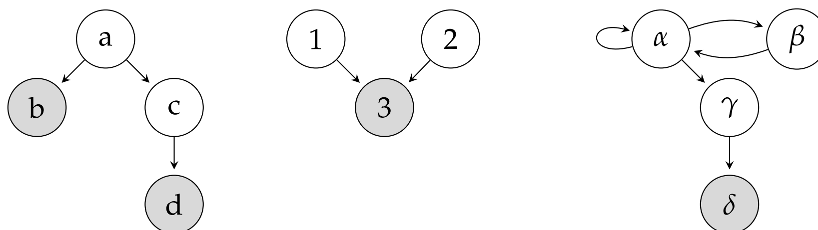

Figure 4.1: Example callgraphs with highlighted nodes marking candidates for the bottom-up analysis starting point.

It should be noted that the topological ordering of a graph is not unique. For the callgraph in the middle and the graph on the right in Figure 4.1 the highlighted nodes will always be at the end of the topological ordering. The starting point is therefore the same for every analysis run. For the callgraph on the left valid topological orderings may either put the nodes *b* or *d* at the end of the ordering. The analysis will therefore use any of the two nodes as a starting-point for the bottom-up analysis. These starting-points are added to the worklist and the entry-block of their CFG is analysed using the initial state provided by the *init*() function (*SilverEntryStateBuilder*).

Method Call Treatment

As in the top-down interpreter the intraprocedural interpreter handles the analysis of the control flow graph of a method. Only the *MethodCall* statement is treated specially by overriding the *executeStatement()* method in the bottom-up interpreter. There are three different cases that need to be considered when a method-call is encountered:

1. The callee is in a different component (node) of the topological ordering.
2. The callee is in the same component of the topological ordering.
3. The callee is the same method as the caller (recursion).

Different component in topological ordering. In the first case, caller and callee do not belong to the same node of the topological ordering. The callee has therefore already been analysed and a *MethodCall* consists of evaluating the arguments of the call, fetching the *CfgResult* tagged with the empty call-string from the *ProgramResult*, and merging the exit-state into the caller. Note that the callee is not added to the worklist.

Same component in topological ordering. In the second case, both caller and callee belong to the same node in the topological ordering. They therefore form a strongly connected component in the callgraph and have both not been analysed yet. Analysing this strongly connected component is delegated to the super class (the top-down interpreter) and both caller and callee are analysed using the call-string approach with an initial starting state. Once both methods have been analysed the *MethodCall* statement is treated by merging the exit-state of the *CfgResult* tagged with the empty call-string into the caller. The result tagged with the empty call-string corresponds to the analysis started with the initial state and therefore without any assumptions about the parameters.

Recursive calls. Recursive *MethodCall* statements are delegated to the top-down interpreter (the super class) and analysed using the call-string approach. Standard call-string approximation and widening are applied like for the top-down case.

In all three cases merging the exit-state into the caller stands for creating a transfer-value, unifying it with the state in the caller and assigning the temporary out-transfer-variables to the method calls target variables, as previously described in Section 3.3.7.

Enqueueing Methods to the Worklist

We already discussed that the analysis starts by adding to the worklist the method(s) at the bottom of the topologically ordered callgraph. Recall that the to top-down analysis uses *onExitBlockExecuted()* to signal when a callee has been analysed and then adds the *BlockPosition* of the method call to the worklist. For bottom-up analysis *onExitBlockExecuted()* is also used to signal an available analysis result. Bottom-up analysis uses a second worklist called *secondaryWorklist* to handle callgraphs in the form of Figure 4.2.

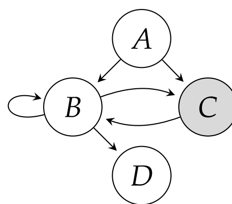

Figure 4.2: Callgraph with *B* and *C* forming a strongly connected component. If the analysis result of the highlighted method *C* is available then *A* will be added to the *secondaryWorklist*.

The methods *B* and *C* form a strongly connected component (SCC) in the callgraph. As discussed in the previous section, this SCC is analysed using the call-string approach. Let us assume the analysis result of method *C* is available but method *B* is still under analysis. If *C* were now to add its caller *A* to the worklist then, *A* would be analysed with the assumption that analysis results of all callees are available. This is not the case for method *B*. The bottom-up analysis therefore enqueues *A* to the *secondaryWorklist* and this worklist will not be worked on unless the primary worklist is empty and therefore also the analysis result of *B* is available. As soon as the primary worklist is empty, all elements from the secondary worklist are moved to it. If *onExitBlockExecuted()* is signalled then there needs to be a case-distinction for the following two cases:

1. The current call-string is empty.
2. The current call-string is not empty.

Empty call-string. If an empty call-string has been used, the analysis result of the current CFG without any assumptions about the parameters is available for all callers to be used. Let *curr* be the node that represents the current method in the topological ordering. Let *pred* be its predecessor. Instead of adding all callers to the worklist we only add methods in *pred* to the *secondaryWorklist*. We add the *BlockPosition* of their entry-block to the worklist and not the position of a method call as it was the case in top-down analysis. Since *B* and *C* form a SCC in Figure 4.2, analysing the exit-block of method

D would result in enqueueing both B and C even though C is not a caller of D. Analysing methods in backward order of their topologically ordered callgraph ensures that no method is analysed before all analysis results of their callees are available.

Call-string is not empty. In case there exists a non-empty call-string at the end of a methods exit-block then either a recursive method or a call-chain between methods of a strongly connected component is analysed (see case 'Same component in topological ordering.' in the previous section). Similarly to treating recursive method calls, the bottom-up interpreter delegates this case to the super class. Therefore, callers are enqueued to the worklist the usual way until an analysis result for the empty call-string is available as well.

4.3 Extracting the Specification

In this section we explain how a specification is extracted from the static analysis results provided as a *ProgramResult*. First, we discuss which *CfgResults* are used to extract the specification. Then, we explain how the information is extracted from the octagon domain.

4.3.1 Analysis Results

Let *result* be the *ProgramResult* for an integer octagon forward analysis ran in either top-down or bottom-up manner. For every method at least one *CfgResult* is available in *result*. Results tagged with *CfgResultTag.Untagged* correspond to method analyses without assumptions about the parameters. On the other hand a *CfgResult* tagged with a non-empty call-string represents an analysis result with assumptions about the methods parameters. Our implementation treats an empty call-string and the tag *CfgResultTag.Untagged* as the same. Depending on the inference (bottom-up vs. top-down) one or many *CfgResults* are used to infer the specification. Our implementation in the trait *InterproceduralSilverInferenceRunner* keeps an attribute *resultsToWorkWith* which represents a list of *CfgResult* relevant for the currently processed Silver method.

Bottom-up inference. For bottom-up inference only the untagged / empty call-string result is needed. This can be obtained by calling *result.getResult()* on the *ProgramResult*.

Top-down inference. For top-down inference every available *CfgResult* is added to *resultsToWorkWith*. Therefore, information about every analysed call-string will be used.

4.3.2 Extracting Preconditions

As we know, an abstract state in the octagon domain consists of constraints relating program variables to each other. To extract a precondition given the *CfgResult* of a method we extract the constraints from the state before the *ProgramPoint* that represents the entry-block of the method. If this state does not contain any constraints, which is the case for bottom-up analysis, then a precondition *true* is assumed. Otherwise the precondition is the conjunction of the constraints in that state. When *resultsToWorkWith* contains multiple results, then we use a disjunction of all existing conjunctions as the precondition.

4.3.3 Extracting Postconditions

Let *pre* be the conjunction of the constraints that represent the precondition for one *CfgResult*. By definition, a postcondition must be true at the end of a method execution. Therefore, we use the constraints from the post-state directly after the last *ProgramPoint* in the method as the postcondition. Some of these constraints may contain information about local variables and we therefore filter them and only use constraints related to method parameters or return variables. Let *post* be a conjunction of the filtered constraints. For one *CfgResult* we infer $pre \implies post$ as the postcondition. For multiple *CfgResults* an implication for each result would be used as postcondition. In a bottom-up inference *pre* is *true* and the postcondition would therefore be *post*.

4.3.4 Extracting Loop Invariants

By definition, a loop invariant needs to be true before and after each loop iteration. Therefore, the constraints representing the invariant need to be extracted from a location in the program that is reached before and after the loop body. In a control flow graph like the one in Figure 4.3, a loop can be represented using three nodes. The first node represents the *head* of the loop (*LoopHead(37)*). A second node that can be reached from the *head* node as long as the loop conditions holds represents the *body* of the loop (*StatementBlock(41)*). And a third node that is reached when the loop condition no longer holds represents the exit of the loop (*StatementBlock(42)*. Since the *head* and *body* nodes form a cycle the entry-state of the *head* represents the state before and after each loop iteration. Therefore, we use the constraints of the pre-state of the loop *head*. Let *inv* be the conjunction of the constraints at that position for one *CfgResult*. Similarly to the postcondition we infer the loop invariant as an implication $pre \implies inv$ for each available *CfgResult*.

The CFG in Figure 4.3 corresponds to a method *double(i: Int)* that iteratively increases the return value *r* until its value has doubled the parameter *i*. List-

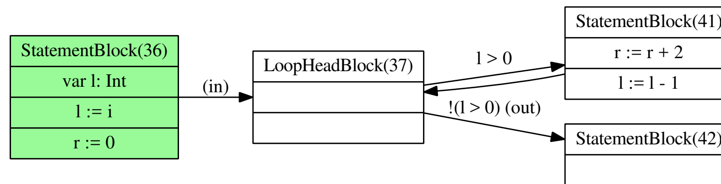

Figure 4.3: Control flow graph of a method containing a loop.

ings A.1 and A.2 in the Appendix show an implementation of the *double()* method together with a *main()* method. In the first listing the specification has been inferred using bottom-up analysis. The second listing shows the top-down inferred specification where loop invariants and postconditions are implications. Comparing the two results we notice that the top-down inference provides a stronger postcondition for the *main()* method than the one inferred bottom-up. But the postcondition of the called *double()* method does not provide very useful information. Bottom-up analysis on the other hand may provide weaker postconditions for entry-methods but it infers useful information for *all* available methods in the program.

4.4 Implementation Details

In this section we provide a very broad overview of Sample's specification inference implementation. The idea is to provide pointers to the code and mention what is implemented where. For more details we refer to the source code available online[2].

4.4.1 Overview

The top-down integer octagon inference is provided by the Scala object *InterproceduralIntegerOctagonInference*. The bottom-up inference is provided by the Scala object *InterproceduralIntegerOctagonBottomUpInference*. Similarly to the analysis runners discussed in Chapter 3, these objects provide a main method that expects the path to a text-file containing Silver source code as a command line argument. When run, a specification is inferred and a Silver program annotated with the inferred specification is printed to standard output. Figure 4.4 gives a broad overview of the generic implementation. Dashed lines represent that a trait is mixed-in to extend the functionality. The bottom-up analysis for example extends the numerical inference runner and mixes-in the *InterproceduralSilverBottomUpInferenceRunner* to run the static analysis bottom-up. The specification extraction described in the previous section takes place in *InterproceduralNumericalInferenceRunner* by over-

[2]https://bitbucket.org/viperproject/sample

riding the abstract methods declared in *SilverInferenceRunner*. The type *InterproceduralSilverInferenceRunner* is responsible for populating *resultsToWorkWith* with *CfgResults* for top-down analysis whereas *InterproceduralSilverBottomUpInferenceRunner* does this for the bottom-up case (see Section 4.3.1). The two traits *SilverExtender* and *SilverExporter* provide functionality to extend a Silver program into a program annotated with the inferred specification and to export the inferred specification in a different format. A Silver program is extended by traversing Sample's internal representation of the program and adding the inferred specification to loop and method declarations. Before this project an intraprocedural numerical inference existed. We contributed bottom-up analysis runners, the interprocedural inference and exporting functionality that will be discussed in the next section to Sample's specification inference implementation.

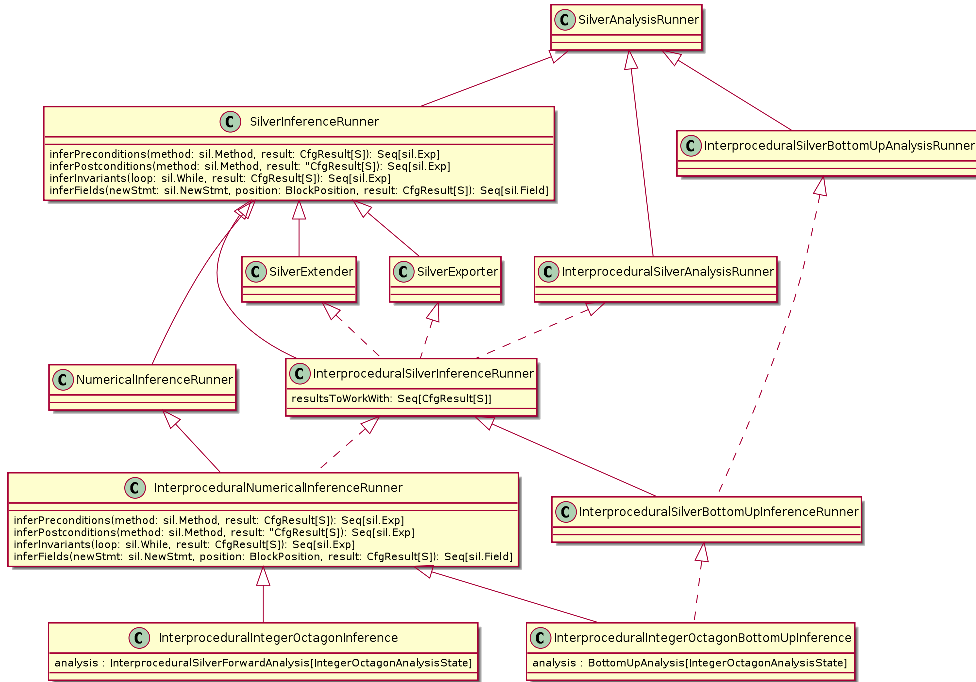

Figure 4.4: Hierarchy of the Silver inference implementation. Dashed lines represent a trait mix-in.

4.5 Exporting the Inferred Specification to Viper IDE

The Viper tools include a visual environment for developing and verifying Silver programs called *Viper IDE*. As discussed in the introduction of this chapter a static analysis can assist a developer by annotating the program

with a specification. Therefore, it is natural to aim for integration of the specification inference into Viper IDE. In this section we provide implementation details about Sample's interface to Viper. We briefly explain the protocol. Then, we give a short overview of the implemented changes in Sample. Finally, we show two example messages transferred from Sample to Viper IDE.

4.5.1 Viper protocol

The Viper IDE defines a JSON[3] based protocol that allows it to communicate with other components of Viper. For instance, Viper IDE can verify a Silver program by talking to a verification backend. The verification result is then displayed to the user. Viper protocol, the communication protocol used between IDE and backends, is defined on Page 51 under Section 5.2.3 of [13]. Every JSON message contains an attribute *type* and the remaining attributes change depending on the message type. Error messages use the type *Error* and then for each error a position in the source code, a tag identifying the error and a message-text is provided (Page 52 in [13]). For Sample we introduced *SpecificationInference* as a message type and *sample.error.inferenceOmitted* as a tag for error messages. Communication between Viper IDE and Sample is as follows: Viper IDE runs Sample's specification inference and provides the path to the Silver source code file as the first command line argument. Sample then prints either a JSON message of type *SpecificationInference* or of type *Error* to standard output.

4.5.2 Implementation in Sample

Sample provides two traits *SilverExporter* and its subtype *SilverJsonExporter* that can be mixed into a *SilverInference* object. These two traits are mixed into the Scala object *InterproceduralIntegerOctagonBottomUpInferenceWithJsonExport* to export a programs bottom-up inferred specification in JSON format. Listing 4.3 shows the steps involved in creating a message according to the Viper protocol.

```
1 val interproceduralResults = interprocedural.run(program)
2 interprocedural.exportProgram(program, interproceduralResults)
3 println(interprocedural.specificationsAsJson(file))
```

Listing 4.3: Steps from a Silver program to its specification in JSON format.

First, the program is analysed using the interprocedural octagon bottom-up analysis. This will provide *interproceduralResults* of type *ProgramResult*. Next, *exportProgram()* of the mixed-in trait *SilverExporter* is run to extract the

[3]JavaScript Object Notation

specification from the analysis results (see Section 4.3). Finally, *specificationsAsJson()* from the mixed-in trait *SilverJsonExporter* is called to convert the extracted specification into JSON format and then it is printed to standard output. The trait *SilverJsonExporter* implements the message format according to the Viper protocol. Sample will tell the IDE at what location in the source code the inferred preconditions, loop invariants and postconditions should be added. Since Viper IDE only adds a specification to the Silver program and never removes or replaces it, we do not support programs with an existing specification. If *SilverJsonExporter* notices that preconditions, loop invariants or postconditions existed before, then a message of type *Error* with tag *sample.error.inferenceOmitted* will be returned to the IDE to notify the user. Note that this check is done *after* the bottom-up analysis. To speed up feedback to the user either Viper IDE or Sample should introduce a check before invoking the static analysis in a future version of the inference.

4.5.3 Viper Protocol Messages

Listings A.5 and A.6 in the Appendix show two example messages that may be sent back to Viper IDE. Listing A.5 contains loop invariants and postconditions together with their location in the source code. Listing A.6 is an error message because the analysed program already contained a specification. The analysed program can be seen in Listing A.1.

Chapter 5

Evaluation

In this chapter we experimentally evaluate the implemented interprocedural analyses using hand-crafted Silver programs, programs used in the annual Competition on Software Verification (SV-COMP)[1] or programs used by other static analysers like the *Interproc analyzer* [11]. First we evaluate the implemented forward and backward top-down analyses. Later, the specification inference is evaluated. In each section we review the results and point out possible drawbacks or future work. Finally, at the end of the chapter we provide a review of the whole evaluation.

5.1 Automated Tests

Together with the implementation this project provides a set of automated tests based on *ScalaTest*. These tests are implemented in the Scala classes *TrivialInterproceduralAnalysisTest*, *ContextInsensitiveInterproceduralAnalysisTest* and *ContextSensitiveInterproceduralAnalysisTest* and allow Sample developers to quickly check that they did not break something fundamental to the interprocedural analysis when changing code in Sample.

5.2 Testing Infrastructure

All analyses were run using Ubuntu Linux 16.04 on a computer with a 2.6 GHz 4-Core CPU (Intel i7-2600U) with 12GB of RAM. Logging in Sample was set to the minimal level *info* and all analyses were run from within *sbt shell* to minimise the start-up time of *sbt*. Listing A.3 in the Appendix provides an example on how to use *sbt shell* to start an analysis.

[1] https://sv-comp.sosy-lab.org/2017/index.php

5.3 Numerical Forward Analyses

In this section we evaluate the implemented interprocedural numerical analyses using the integer interval and octagon abstract domain. Each program will be analysed by both analyses using different call-string lengths. To repeat an experiment the stated Silver programs can be put into a text-file and then either *InterproceduralIntegerOctagonAnalysis* or *InterproceduralIntegerIntervalAnalysis* can be started by providing the path to the source code file as the first command line argument. The upper bound on the call-string length can be controlled by modifying *callStringLength* in *SystemParameters.scala*.

5.3.1 Fibonacci

The program in Listing 5.1 computes the nth Fibonacci number. We analysed two variants of the program. One that invokes *fibonacci()* with $n = 7$ and another one with $n = \top$, that is, any possible integer value. The 7th Fibonacci number is 13 and therefore the most precise result would be $r = 13$. The source of this program is the SV-COMP *Fibonacci04* [2] C program which was translated to Silver. Table 5.1 shows the analysis duration and results for different call-string bounds k. For call-strings smaller than 5, the octagon analysis does not provide any information about the variable r. Integer intervals on the other hand provide a lower bound zero. This is surprising because the octagon abstract domain is more powerful than interval abstract domain and we analyse and discuss this in Section 5.3.4. With increasing k the results get more precise for both analyses. If k is set to *None*, and therefore no limit on the call-string length is provided, the value of r is computed precisely. As an additional test we ran an octagon analysis with full-length call-strings for the same program to compute the 15th Fibonacci number. The analysis terminated with the precise result $r = 610$ after several minutes.

	Integer Intervals		Integer Octagons	
k	Time (s)	*main()* exit-state	Time (s)	*main()* exit-state
0	< 1	$[r \to [0, \infty], n \to [7, 7]]$	< 2	$[n = 7]$
3	< 1	$[r \to [0, \infty], n \to [7, 7]]$	< 2	$[n = 7]$
5	< 2	$[r \to [12, \infty], n \to [7, 7]]$	< 3	$[12 \leq r, n = 7]$
6	< 2	$[r \to [13, 13], n \to [7, 7]]$	< 3	$[r = 13, n = 7]$
None	< 2	$[r \to [13, 13], n \to [7, 7]]$	< 3	$[r = 13, n = 7]$

Table 5.1: Analysis time and exit-states of the program that computes the 7th Fibonacci number.

[2] https://github.com/sosy-lab/sv-benchmarks/blob/master/c/recursive/Fibonacci04_false-unreach-call_true-no-overflow_true-termination.c

```
1  method fibonacci(n: Int) returns(r: Int) {
2    var x: Int := 0
3    if (n < 1) {
4      r := 0;
5    } elseif (n == 1) {
6      r := 1;
7    } else {
8      r := fibonacci(n-1)
9      x := fibonacci(n-2);
10     r := r + x
11   }
12 }
13
14 method main() {
15   var n: Int := 7
16   var r: Int := 0
17   r := fibonacci(n);
18 }
```

Listing 5.1: Silver program that computes the 7th Fibonacci number.

Table 5.2 shows the analysis results if *main()* invokes *fibonacci()* with the argument $n = \top$. Again, the octagon analysis does not provide information about r when analysed with *k=0*. But in the context-sensitive case both analyses provide the same analysis result. The results do not get more precise after *k=1* while the running time starts growing fast with call-strings greater than 4.

	Integer Intervals		**Integer Octagons**	
k	Time (s)	*main()* exit-state	Time (s)	*main()* exit-state
0	< 2	$[r \to [0, \infty], n \to \top]$	< 2	$[\,]$
1	< 2	$[r \to [0, \infty], n \to \top]$	< 2	$[0 \leq r]$
2	< 2	$[r \to [0, \infty], n \to \top]$	< 2	$[0 \leq r]$
3	< 2	$[r \to [0, \infty], n \to \top]$	< 2	$[0 \leq r]$
4	< 2	$[r \to [0, \infty], n \to \top]$	< 2	$[0 \leq r]$
5	11	$[r \to [0, \infty], n \to \top]$	< 10	$[0 \leq r]$
10	> 900	–	–	–

Table 5.2: Analysis time and exit-states of the program that computes the nth Fibonacci number. The analysis for *k=10* was aborted after 15 minutes.

We also implemented a version of the Fibonacci program in the programming language analysed by the Interproc analyzer [11] and ran an octagon

analysis with default settings. Interproc infers the same constraint $[0 \leq r]$ as Sample does for the Fibonacci function invoked with $n = \top$. Furthermore we tested what Interproc does when the *fibonacci()* procedure is invoked with a specific value. In that case Interproc ignores the argument and analyses the procedure without assumptions about its arguments.

5.3.2 McCarthy 91 Function

Listing 5.2 shows an implementation of the McCarthy 91 function invented by John McCarthy. The result of this recursive function is 91 for all arguments $n \leq 100$ and $n - 10$ otherwise. Our Silver implementation follows the McCarthy 91 example program used by the Interproc analyzer [11].

```
1  // exact semantics:
2  //    if (n>=101) then n-10 else 91
3  method MC(n: Int) returns (r: Int) {
4      var t1 : Int
5      var t2 : Int
6      if (n>100){
7          r := n - 10
8      } else {
9          t1 := n + 11;
10         t2 := MC(t1);
11         r := MC(t2);
12     }
13 }
14
15 method main(a: Int) returns (r: Int) {
16     r := MC(a)
17 }
```

Listing 5.2: Silver implementation of the McCarthy 91 function.

As Table 5.3 shows, a correct lower bound of the return value is computed for the restricted call-string lengths 0 and 5. If the call-string is not bounded then the analysis will not terminate. Because of the recursive calls the call-string just grows infinitely and widening will never be applied. The Octagon analysis also recognizes the constraint that tracks the relationship between the argument and the return value when the argument n is greater than 100. Our octagon analysis collects the same constraints as the Interproc analyzer [11] when run with the default settings and octagon domain.

5.3.3 Multiple Callers

The program in Listing 5.3 was designed to test the effect of having multiple callers to the same callee. The method *callee()* expects an argument of

	Integer Intervals		Integer Octagons	
k	Time (s)	*main()* exit-state	Time (s)	*main()* exit-state
0	< 1	$[r \to [91,\ \infty],\ a \to \top]$	< 2	$[91 \le r,\ a - r \le 10]$
5	< 2	$[r \to [91,\ \infty],\ a \to \top]$	< 2	$[91 \le r,\ a - r \le 10]$
None	–	–	–	–

Table 5.3: Analysis results for the McCarthy 91 function. Without a call-string bound the analysis does not terminate.

type integer and recursively increases or decreases the argument i until it is within the range of $-5 \le i \le 5$. The method *caller1()* passes a specific value to *callee()* whereas *caller2()* passes the argument i which can have any integer value.

```
1  method caller1() returns (r: Int)
2  {
3    r := callee(4)
4  }
5
6  method callee(i: Int) returns (k: Int)
7  {
8    if(i < -5) {
9      k := callee(i + 1)
10   } elseif(i > 5){
11     k := callee(i - 1)
12   } else {
13     k := i
14   }
15 }
16
17 method caller2(i: Int) returns (r: Int)
18 {
19   r := callee(i)
20 }
```

Listing 5.3: Silver Program with two entry-methods *caller1()* and *caller2()*

Tables 5.4 and 5.5 show the analysis results for integer intervals and octagons. Both caller methods are considered to be entry-points to the program and analysed with an initial abstract state. Since both methods are analysed in the same run, the time shown in the table is the total time Sample ran for the whole program. In the case where the call-string was limited to $k = 0$ the analysis result of the callee is reused in both callers (context-insensitive analysis). For the call-string length limit 5 the analysis results are

Integer Intervals			
k	Time (s)	*caller1()* exit-state	*caller2()* exit-state
0	< 2	$[r \to [-5,\ 5]]$	$[r \to [-5,\ 5], i \to \top]$
5	< 2	$[r \to [4,\ 4]]$	$[r \to [-5,\ 5], i \to \top]$
None	–	–	–

Table 5.4: Integer Intervals analysis results for the multiple caller experiment.

Integer Octagons			
k	Time (s)	*caller1()* exit-state	*caller2()* exit-state
0	< 2	$[-5 \leq r \leq 5]$	$[-5 \leq r \leq 5]$
5	< 3	$[r = 4]$	$[-5 \leq r \leq 5]$
None	–	–	–

Table 5.5: Integer Octagons analysis results for the multiple caller experiment.

separated and for both caller methods the most precise value of r is found.

Note that, as it was the case for the McCarthy 91 function, the analysis of the program in Listing 5.3 does not terminate if no upper bound for the call-string length is provided.

5.3.4 Review

For both the McCarthy 91 function and the Fibonacci function invoked with $n = \top$ Sample infers the same constraints in the octagon abstract domain as the Interproc analyzer does. Furthermore, the analysis of the Fibonacci program invoked with the arguments $n = 7$ or $n = 15$ shows that precise analysis results are achieved for intervals and octagons when the call-string is not bounded. This confirms that our approach of using transfer-values works to add support for method parameters and local variables in recursive programs to the call-strings approach. Because the octagon abstract domain is more powerful than intervals, we expected octagons to always infer at least the same or more properties about the program than intervals. This is the case for the *Multiple Callers* experiment and the McCarthy 91 function. For the latter also the relationship between parameter and return variable is inferred. For Fibonacci this is not the case as we will discuss in the next section.

Fibonacci Analysis Results – Octagons vs. Intervals

For the Fibonacci program when invoked with a specific n and with a call-string length smaller than 5, integer interval analysis yields a lower bound

for variable r whereas octagon analysis does not (see Table 5.1).

Analysis. Starting at $k = 5$ octagons provide a lower bound for r and for $k = 6$ the analysis provides the precise result for r because the limit k is high enough to contain the whole call-stack that is created to compute the 7th Fibonacci number. The reason why no information is inferred for k smaller than 5 is because, due to the infeasible paths introduced by the call-string approximation, octagons correctly provides an over-approximation for r, meaning no constraints for r are known. This happens because octagons preserve the relationship between in-transfer-variables, passed to the callee, and the local variables. When a transfer-value is passed back to multiple callers then the relationship between local variables, in-transfer-variables and out-transfer-variables may not be preserved and, as it is the case with $k = 0$ for the Fibonacci program, the constraints for the return variable are over-approximated. For integer intervals the abstract values of the return-variables are simply copied into the caller(s) identified by the call-string suffix. For $k = 0$ in the Fibonacci program the implementation therefore copies the result $[0, \infty]$ to all callers whereas octagons try to preserve the relationship between the variables in the calling-context and the returned transfer-value.

Conclusions. As we see in the analysis results for the McCarthy 91 function octagons provide useful constraints for $k = 0$ (see Table 5.3). When Fibonacci is invoked with a specific value and the call-string is bounded, then octagons may infer the correct but imprecise result of no constraints. Better results are achieved if the *fibonacci()* method is analysed without any assumption about its parameters. This is what we do in bottom-up analysis and it is also what the Interproc analyzer does in every case (a specific argument to *fibonacci()* will be ignored). We therefore draw the conclusion that our interprocedural integer octagon analysis is most useful when run top-down without a call-string bound or bottom-up where we get the same context-sensitive analysis results as the Interproc analyser.

5.4 Strongly Live Variable Analysis

In this section we evaluate the top-down backward analysis using the implementation of interprocedural strongly live variable analysis. The analysis is implemented in the Scala object *LiveVariableAnalysis* and it requires the path to a Silver source code file as the first command line argument when run.

5.4.1 Multiple Callers – No Recursion

First let us consider the program in Listing 5.4. The method *compute()* does a numerical computation and its three return values are computed using different parameters. A strongly live variable analysis would declare x, y and z to be live at the method entry-point because they are used to define the return values. The methods *noUse()*, *allUse()* and *partialUse()* are at the top of the callgraph and therefore entry-points to the program. As their name suggests they call *compute()* and either use no, all or parts of the return values. The *helper()* method is used to call *compute()* and sum (and therefore use) all return variables of *compute()*.

```
1  method compute(x: Int, y: Int, z: Int)
2      returns (r1: Int, r2: Int, r3: Int) {
3    r1 := x * x
4    r2 := y + z
5    r3 := y
6  }
7
8  method helper(a: Int, b: Int, c: Int)
9      returns (r: Int) {
10   var x: Int
11   var y: Int
12   var z: Int
13   x, y, z := compute(a, b, c)
14   r := x + y + z
15 }
16
17 // possible program entry-point
18 method noUse(a: Int, b: Int, c: Int)
19     returns (r: Int) {
20   r := helper(a, b, c)
21   r := 0
22 }
23
24 // possible program entry-point
25 method allUse(a: Int, b: Int, c: Int)
26     returns (r: Int) {
27   r := helper(a, b, c)
28 }
29
30 // possible program entry-point
31 method partialUse(a: Int, b: Int, c: Int)
32     returns (r: Int) {
33   var x: Int
```

```
34      var y: Int
35      var z: Int
36      x, y, z := compute(a, b, c)
37      r := y
38  }
```

Listing 5.4: *The* compute() *program.*

Table 5.6 shows the analysis result for this program. The table shows the method parameters that are considered to be strongly live at method-entry point because they are used to define the return values. The analysis was run with three different call-string bounds k. Since the program does not contain recursion and the stack of method calls never grows beyond 5 calls both k=5 and k=*None* produce the precise result. Again, for the context-insensitive case the analysis result is reused in all three callers.

Strongly Live Variables at entry-state				
k	Time (s)	*noUse()*	*allUse()*	*partialUse()*
0	< 2	$\{a, b, c\}$	$\{a, b, c\}$	$\{a, b, c\}$
5	< 2	$\emptyset$	$\{a, b, c\}$	$\{b, c\}$
None	< 2	$\emptyset$	$\{a, b, c\}$	$\{b, c\}$

Table 5.6: Strongly live variable analysis results for the multiple caller – no recursion experiment.

For the method *noUse()* the empty set is the correct precise result because even though *helper()* is called and its result is assigned to the return variable r the returned value will never be used because r gets 0 in the last statement of the method. On the other hand *partialUse()* uses y to define its return value and since y is computed using the parameters b and c passed to *compute()* the set of strongly live variables at method-entry is $\{b, c\}$.

5.4.2 Multiple Callers – Recursion

Next we consider the program in Listing 5.5. The method *exp_by_squaring(x, n)* recursively computes x^n by squaring partial results. We translated the pseudo code from Wikipedia[3] to Silver. If the return variable of *exp_by_squaring()* is assumed to be strongly live then x and n will also be strongly live at the method-entry. Note that n will always be live at method-entry no matter if r is live or not at method-exit because n is used in the condition of *if* statements. The program has two entry-methods where *entry1()* uses the result computed by *exp_by_squaring()* and *entry2()* does not.

[3] https://en.wikipedia.org/wiki/Exponentiation_by_squaring

```
1  method entry1(x: Int, y: Int) returns (r: Int) {
2    r := exp_by_squaring(x, y)
3  }
4
5  method entry2(i: Int, j: Int) returns (r: Int) {
6    var x: Int := 0
7    x := exp_by_squaring(i, j)
8    r := 0
9  }
10
11 method exp_by_squaring(x: Int, n: Int)
12   returns(r: Int) {
13   var modCheck: Int := 0
14   if(n<0) {
15     r := exp_by_squaring(1/x, -n)
16   } elseif(n == 0) {
17     r := 1
18   } elseif(n == 1) {
19     r := x
20   } else {
21     modCheck := n / 2 * 2
22     if(modCheck == n) {          // is even
23       r := exp_by_squaring(x * x, n / 2)
24     } else {                     // is odd
25       r := exp_by_squaring(x * x, (n-1) / 2)
26       r := r * x
27     }
28   }
29 }
```

Listing 5.5: Multiple entry-methods that compute x^y respectively i^j.

Table 5.7 shows the analysis results for the recursive program. For the context-insensitive case *k=0* it is expected that both callers get the same result back from the callee, that is, all parameters of *exp_by_squaring()* are considered to be strongly live at method-entry and therefore the callers parameters passed to the callee are considered to be strongly live too. In the context-sensitive case we expected that in the entry-state of *entry2()* only the parameter *j* is strongly live because the callee parameter *n* is always strongly live. Unfortunately the analysis result does not get more precise for increasing *k* because the information of the two callers will always be joined and the analysis time grows exponentially. We did not run the analysis for call-string bounds higher than 5. When the method *entry1()* is removed Sample

provides the precise analysis result with only *j* being strongly live at the entry-state of *entry2()*.

Strongly Live Variables at entry-state			
k	Time (s)	*entry1()*	*entry2()*
0	< 2	$\{x, y\}$	$\{i, j\}$
2	< 2	$\{x, y\}$	$\{i, j\}$
4	21	$\{x, y\}$	$\{i, j\}$
5	477	$\{x, y\}$	$\{i, j\}$
None	–	–	–

Table 5.7: Strongly live variable analysis results for the multiple caller – recursion experiment.

5.4.3 Review

The test results in this section show that the framework can be instantiated for non-numerical domains and that the backward implementation of call-strings works as expected. The non-recursive test case showed the expected results for the context-sensitive and context-insensitive analyses for all entry-methods (Section 5.4.1). In the recursive test case in Section 5.4.2 we do not get the precise result in which only variable *j* is strongly live at the entry of method *entry2()*. This is because we need to limit the call-string length due to the recursion. If the call-string length limit is reached, the two calling methods will share the result of a call with common call-string suffix and therefore the second parameter of *entry2()* is considered to be strongly live too.

Drawbacks / Future Work

The recursive test case points out two drawbacks which could be addressed in future work.

Only analyse one entry-method at a time. It could make sense for future work to implement a top-down analysis that only starts at one entry-method at a time instead of all at the same time like our implementation does. If only *entry2()* in Listing 5.5 is considered without *entry1()* then the analysis yields the precise result. This may be useful for other analyses too.

Limit call-string construction. The measured time in Table 5.7 shows that it grows exponentially when increasing the call-string. This is, because every possible valid call-string combination is built until the call-string length limit is reached. For live variable analysis this is frustrating because the size of the

lattice is finite. In method *exp_by_squaring()* of Listing 5.5 the liveness of only three variables needs to be tracked and the lattice therefore has the same shape as the lattice in Figure 3.10. There are eight possible transfer-values (elements of the power set $\mathcal{P}(\{x,\ r,\ n\})$) that can be propagated but the interpreter grows the call-string until the limit is reached no matter if the same transfer-value has previously been propagated to the same method or not. Khedker and Karkare [14] propose an extension of call-strings that recognizes when the same information is propagated again and again and it therefore would not be necessary to continue with recursive calls up to the call-string bound. Possible future work could be to add such an extension to our call-string implementation.

5.5 Specification Inference

In this section we evaluate the bottom-up and top-down specification inference. The top-down inference can be started by running the Scala program *IntegerOctagonInference*. The bottom-up inference is implemented in the Scala object *InterproceduralIntegerOctagonBottomUpInference*. All analyses have been run with *SystemParameters.callStringLength=5* because our tests so far have shown that the analysis result usually does not get more precise while the time necessary to run the analysis increases exponentially.

5.5.1 Program four()

Listing 5.8 shows the inferred specification by top-down and bottom-up analyses of a program that computes the number 4 by incrementing a variable three times. The bottom-up inferred specification is useful and precisely describes the effect of the two methods. The top-down inferred specification on the other hand is less useful because only implications are inferred for the *increment()* method. But it may be useful for documentation purposes because the implications show actual calls seen during analysis. The implication *ensures i == 2 ==> r == 3* would for example tell us that somewhere during the analysis a call to *increment* with argument $i = 2$ was encountered.

5.5.2 Ackermann Function

We used the Ackermann function[4] implemented as *ack()* in Listing 5.9 to compare the specification inferred by Sample to what the Interproc analyzer [11] infers for the same program. The implementation below is a translation of Interproc's *ack()* example program to Silver. The highlighted postconditions show the specification inferred by the bottom-up inference. Interproc uses an *assume* command to ensure that *ack()* is only called with parameters

[4] https://en.wikipedia.org/wiki/Ackermann_function

```
method four() returns (r: Int)
  ensures r == 4
{
  var l: Int
  l := 1
  l := increment(l)
  l := increment(l)
  l := increment(l)
  r := l
}

method increment(i: Int) returns (r: Int)
  ensures i - r == -1
{
  r := i + 1
}
```

Listing 5.6: Bottom-up

```
method four() returns (r: Int)
  ensures r == 4
{
  var l: Int
  l := 1
  l := increment(l)
  l := increment(l)
  l := increment(l)
  r := l
}

method increment(i: Int) returns (r: Int)
  ensures i == 1 ==> i == 1
  ensures i == 1 ==> r == 2
  ensures i == 2 ==> i == 2
  ensures i == 2 ==> r == 3
  ensures i == 3 ==> i == 3
  ensures i == 3 ==> r == 4
{
  r := i + 1
}
```

Listing 5.7: Top-down

Listing 5.8: Inferred specification for the *four()* program.

$0 \leq x, 0 \leq y$. To simulate the same we added a precondition at line 21 to the program and then let Sample infer the postconditions. Without the precondition Sample infers the weaker postcondition *ensures b - r <= -1* for the *main()* method.

```
1  method ack(x: Int, y: Int) returns (res: Int)
2    ensures 1 <= res - y
3  {
4    var t: Int
5    var t1: Int
6    if (x <= 0) {
7      res := y + 1
8    } elseif (y <= 0) {
9      t1 := x - 1
10     t := 1
11     res := ack(t1, t)
12   } else {
13     t1 := y - 1
14     t := ack(x, t1)
15     t1 := x - 1
16     res := ack(t1, t)
17   }
18 }
19
20 method main(a: Int, b: Int) returns (r: Int)
21   requires a >= 0 && b >= 0
```

```
22     ensures 0 <= b
23     ensures b - r <= -1
24     ensures 1 <= b + r
25     ensures 0 <= a + b
26     ensures 0 <= a
27     ensures 1 <= r
28     ensures 1 <= a + r
29   {
30     r := ack(a, b)
31   }
```

Listing 5.9: Bottom-up inferred specification for the Ackermann function. Highlighted postconditions have been inferred by Sample.

The inferred postconditions in Listing 5.9 match the analysis result of the Interproc analyzer when run with default settings and octagon abstract domain. Specification inference was also run top-down with the result in Listing A.4 in the Appendix. Both top-down and bottom-up inference completed within less than 20 seconds. The inferred specification is correct and can be verified. The programs were verified using the online tools of the Viper[5].

5.5.3 Program getElementOrLast()

Chapter 4 introduced the *getElementOrLast()* program that bounded the index *pos* before accessing the sequence *xs*. Listing 5.10 shows an adapted version of this program where *upperBound()* is implemented using a loop and a helper method. The highlighted invariant and postconditions show the inferred specification. Inference completed within seconds and both top-down and bottom-up approaches infer the same invariant and postconditions. Using the inferred specification the Viper online verifier was able to verify that access to *xs* is within bounds.

```
1  method getElementOrLast(pos: Int, xs: Seq[Int])
2  returns (r: Int)
3  {
4    var bounded: Int
5    bounded := upperBound(pos, |xs| - 1)
6    if (bounded >= 0) {
7      r := xs[bounded]
8    } else {
9      r := -1
10   }
```

[5] http://viper.ethz.ch/examples/blank-example.html

```
11 }
12
13 method upperBound(n: Int, upper: Int)
14 returns (r: Int)
15   ensures r - upper <= 0
16   ensures 0 <= n - r
17 {
18   r := n
19   while (r > upper)
20     invariant 0 <= n - r
21   {
22     r := decrement(r)
23   }
24 }
25
26 method decrement(i: Int) returns (r: Int)
27   ensures i - r == 1
28 {
29   r := i - 1
30 }
```

Listing 5.10: Inferred specification (highlighted) for the getElementOrLast() program. Top-down and bottom-up inference result in the same specification.

5.5.4 Strongly Connected Component in Callgraph

So far bottom-up inference has proven to be more useful because it provides generally valid statements about all procedures whereas top-down only provides implications for invariants and postconditions for called methods. Listing 5.13 provides a hand-crafted program where the entry-methods form a strongly connected component in the callgraph. The program itself does nothing useful but it is guaranteed to terminate when one of the entry-methods *foo()* or *bar()* is called with any argument. All postconditions in Listing 5.13 have been inferred and pass verification.

For this program top-down inference is able to infer a stronger postcondition for the method *bar()*. It infers an upper and lower bound for *i* whereas bottom-up only infers a lower bound. In this case top-down provides a stronger postcondition for *bar()* because it is, together with *foo()*, at the top of the callgraph. If another method were to exist in the program that calls *bar()* and is neither called by *foo()* or *bar()* then top-down inference would only provide implications as postconditions of *bar()*.

```
method foo(b: Int) returns (i: Int)
  ensures i <= 22
  ensures −1 <= b − i
{
  if (b > 20) {
    i := bar(21)
  } else {
    i := baz(b)
  }
}

method bar(z: Int) returns (i: Int)
  ensures i <= 22
{
  if (z == 21) {
    i := baz(z)
  } else {
    i := foo(10)
  }
}

method baz(x: Int) returns (r: Int)
  ensures r − x == 1
{
  r := x + 1
}
```

Listing 5.11: Bottom-up

```
method foo(b: Int) returns (i: Int)
  ensures i <= 22
  ensures −1 <= b − i
  ensures b == 10 ==> b == 10
  ensures b == 10 ==> i == 11
{
  if (b > 20) {
    i := bar(21)
  } else {
    i := baz(b)
  }
}

method bar(z: Int) returns (i: Int)
  ensures 11 <= i
  ensures i <= 22
  ensures z == 21 ==> z == 21
  ensures z == 21 ==> i == 22
{
  if (z == 21) {
    i := baz(z)
  } else {
    i := foo(10)
  }
}

method baz(x: Int) returns (r: Int)
  ensures x <= 20 ==> x <= 20
  ensures x <= 20 ==> r <= 21
  ensures x <= 20 ==> r + x <= 41
  ensures x <= 20 ==> r − x == 1
  ensures x == 21 ==> x == 21
  ensures x == 21 ==> r == 22
  ensures x == 10 ==> x == 10
  ensures x == 10 ==> r == 11
{
  r := x + 1
}
```

Listing 5.12: Top-down

Listing 5.13: All postconditions have been inferred. The highlighted postconditions show where top-down is able to infer a stronger postcondition.

5.5.5 Review

The test results show that the top-down inference may be useful if one is interested in strong postconditions of (only) the entry-methods of a program. For the *bar()* method in Section 5.5.4 and the *double()* method introduced in the previous chapter in Section 4.3.4 the top-down inferred postcondition of the entry-method was stronger than for bottom-up inference. But the drawback of top-down analysis is, that for all called methods only partial specifications are inferred. Bottom-up inference on the other hand may infer a weaker postcondition for entry-methods but it provides useful information for all methods of the program. Another advantage of the bottom-up approach is, that it is a modular analysis. Big programs can be analysed as components (of the condensed callgraph) and the result, meaning the post-

conditions of a callee, can be reused as the result of a method call in all callers. The top-down inference on the other hand will have to analyse each caller for each method call again.

5.6 Evaluation Review

The results in this chapter show that the generic framework can be instantiated for different analyses in forward or backward direction. For numerical analyses using the integer octagon abstract domain the bottom-up inference achieves the same results as the Interproc analyzer [11]. This is also the case for context-sensitive top-down analyses of procedures whose callers passed in any value (e.g. $n = \top$ for Fibonacci). But top-down analysis using octagons is less useful when the call-string length is limited, which is necessary for recursive procedures, and the analysed program calls the caller with specific arguments (e.g. $n = 7$ for Fibonacci). An analysis and conclusions of this can be found in Section 5.3.4. Due to this, we believe octagons are most useful in our implementation when either used for bottom-up analyses or top-down without restricting the call-string length. Another result seen in all three evaluations is that even for programs consisting of only a couple of methods the analysis gets impractical in terms of running time for call-string bounds greater than 3 or 4. In Section 5.3 we also saw that for procedures analysed without an assumption about their parameter values the analysis results usually do not improve when increasing the call-string bound above 1 or 2. Please refer to Sections 5.3.4, 5.4.3 and 5.6 for a more detailed review of the three evaluations.

Chapter 6

Conclusions

In this thesis, we have presented a generic interprocedural static analysis framework which adds support for interprocedural analyses of Silver programs to Sample [2], a static analyser developed at the Chair of Programming Methodology at ETH Zurich.

Our analysis is based on Sharir and Pnueli's call-string approach [21] and the framework is customisable by setting the widening limit and configuring an upper bound for the call-string length for *call-string suffix approximation* proposed by Sharir and Pnueli. Our approach adds support for procedures called with arguments and recursion with local variables to call-strings (Chapter 2).

The framework is generic with respect to the static analysis problem and we provide instantiations for forward and backward analyses. In forward direction we provide implementations for analyses using the integer interval and integer octagon abstract domains. An instantiation of a backward analysis is provided as a variant of live variable analysis (Chapter 3). For the implementation we made a simplifying assumption that all information that is necessary to analyse a called method is only related to its parameters or return variables. This can easily be adapted if necessary, like for example, when global variables should be be taken into account too.

Functionality to create whole-program top-down analyses or modular bottom-up analyses is provided. Developers can instantiate their own analysis by providing the necessary framework parameters (Section 2.5.3). To implement an analysis we refer to our instantiations that provide an example of what has to be implemented (Section 3.4).

The new interprocedural analysis framework was used to implement specification inference for numerical programs (Chapter 4). We provide a top-down and bottom-up specification inference using integer octagons. Fur-

thermore, we have implemented an interface for Viper IDE[1] to infer specifications of programs that are being edited in the IDE.

Finally, the results of the evaluation show that the generic framework can successfully be instantiated for different interprocedural analyses in forward or backward direction. In our tests of the numerical analyses we achieved the same analysis results as the Interproc analyzer [11] using standard settings and octagon domain (Chapter 5).

6.1 Future Work

In this section we discuss possible future work.

Generalise the assumption on transfer-values. The current implementation makes the simplifying assumption that everything necessary to analyse a called procedure can be passed into it using its parameters and return variables. We call this information a transfer-value. A more generic implementation could delegate the creation of the transfer-value to users of the framework instead of creating transfer-values using this assumption. This may be useful if a specific analysis wants to provide more information than only call-string and parameters to a callee.

Only analyse one entry-method at a time. Our implementation of the top-down analysis runner analyses all entry-methods during the same analysis. In case of a limited call-string length results may be reused for multiple callers resulting in a less precise analysis result (Section 5.4.3). It could be useful to provide an analysis runner that runs multiple top-down analyses starting at each possible entry-point of the program once.

More powerful numerical domain for specification inference. It could be useful to instantiate the specification inference with a more powerful numerical abstract domain like the polyhedra abstract domain [7]. As soon as the abstract domain is available for an intraprocedural analysis in Sample, the instantiation for interprocedural specification inference is straightforward.

Limit call-string construction. For recursive procedures the analysis explores all possible call-strings until the length limit is reached. Especially for finite domains it is possible that the recursive procedure is analysed repeatedly with the same in-state (see Section 5.4.3) but changing call-strings. Khedker and Karkare [14] propose an extension of call-strings that recognizes when the same information is propagated again and again and it therefore would not be necessary to continue with recursive calls up to the

[1] https://bitbucket.org/viperproject/viper-ide

full call-string length limit. Possible future work could be to add such an extension to our call-string implementation.

6.2 Acknowledgements

I would like to thank my supervisors Dr. Caterina Urban and Jérôme Dohrau for their great support throughout my thesis. Their advice in our weekly meetings was an immense help. I am also very grateful for all the time they invested to provide valuable feedback on the written report, do code-reviews or help coordinate with other projects when something out of my hands interfered with Sample. My thanks go to Prof. Dr. Peter Müller for giving me the opportunity to work on this thesis. Finally, I would like to express my gratitude to my girlfriend and family for their constant support and patience.

Appendix A

Appendix

A.1 Bottom-Up Inferred Specification

```
1  method main() returns (r: Int)
2    ensures 0 <= r
3  {
4    var i: Int
5    i := 5
6    r := double(i)
7  }
8
9  method double(i: Int) returns (r: Int)
10   ensures 0 <= r
11 {
12   var l: Int
13   l := i
14   r := 0
15   while (l > 0)
16     invariant 0 <= r
17     invariant 0 <= i - l
18   {
19     r := r + 2
20     l := l - 1
21   }
22 }
```

Listing A.1: Bottom-up inferred specification.

A.2 Top-Down Inferred Specification

```
1  method main() returns (r: Int)
2    ensures 5 <= r
3  {
4    var i: Int
5    i := 5
6    r := double(i)
7  }
8
9  method double(i: Int) returns (r: Int)
10   ensures i == 5 ==> 5 <= r
11   ensures i == 5 ==> i == 5
12 {
13   var l: Int
14   l := i
15   r := 0
16   while (l > 0)
17     invariant i == 5 ==> l <= 5
18     invariant i == 5 ==> 5 <= l + r
19     invariant i == 5 ==> 0 <= r
20     invariant i == 5 ==> l - r <= 5
21     invariant i == 5 ==> i == 5
22   {
23     r := r + 2
24     l := l - 1
25   }
26 }
```

Listing A.2: Top-down inferred specification.

A.3 Running an Analysis

```
1  sample@9eda8cb11fcc:~/workspace/sample$ sbt shell
2  [info] Loading project definition from /home/sample
      /workspace/sample/project
3  ...
4  > project sample-silver
5  ...
6  > run-main ch.ethz.inf.pm.sample.abstractdomain.
      InterproceduralIntegerIntervalAnalysis src/test/
      resources/silver/example.sil
7  ...
8  [info]
9  [info] ******************
10 [info] * AnalysisResult *
11 [info] ******************
12 [info]
13 ...
14 [success] Total time: 11 s, completed Aug 9, 2017
      1:49:04 PM
```

Listing A.3: Commands to run an interprocedural integer intervals analysis using *sbt shell*

A.4 Top-Down inferred Specification for the Ackermann Function

```
1  method ack(x: Int, y: Int) returns (res: Int)
2    ensures 1 <= x && y == 0 ==> 1 <= x
3    ensures 1 <= x && y == 0 ==> 3 <= res + x
4    ensures 1 <= x && y == 0 ==> 1 <= res - x
5    ensures 1 <= x && y == 0 ==> 2 <= res
6    ensures 1 <= x && y == 0 ==> y == 0
7    ensures 0 <= x && y == 1 ==> y == 1
8    ensures 0 <= x && y == 1 ==> 0 <= x
9    ensures 0 <= x && y == 1 ==> 2 <= res
10   ensures 0 <= x && y == 1 ==> 2 <= res - x
11   ensures 0 <= x && y == 1 ==> 2 <= res + x
12   ensures 1 <= x && 0 <= y && 1 <= x + y ==> 1 <= x
13   ensures 1 <= x && 0 <= y && 1 <= x + y ==> 0 <= y
14   ensures 1 <= x && 0 <= y && 1 <= x + y ==> 1 <= x + y
15   ensures 1 <= x && 0 <= y && 1 <= x + y ==> 2 <= res - y
16   ensures 1 <= x && 0 <= y && 1 <= x + y ==> 2 <= res + y
17   ensures 1 <= x && 0 <= y && 1 <= x + y ==> 1 <= res - x
18   ensures 1 <= x && 0 <= y && 1 <= x + y ==> 2 <= res
19   ensures 1 <= x && 0 <= y && 1 <= x + y ==> 3 <= res + x
20   ensures 0 <= x && 2 <= y && 2 <= x + y && x - y <= -2 ==> 3 <= res - x
21   ensures 0 <= x && 2 <= y && 2 <= x + y && x - y <= -2 ==> 3 <= res + x
22   ensures 0 <= x && 2 <= y && 2 <= x + y && x - y <= -2 ==> 2 <= x + y
23   ensures 0 <= x && 2 <= y && 2 <= x + y && x - y <= -2 ==> 5 <= res + y
24   ensures 0 <= x && 2 <= y && 2 <= x + y && x - y <= -2 ==> 0 <= x
25   ensures 0 <= x && 2 <= y && 2 <= x + y && x - y <= -2 ==> 1 <= res - y
26   ensures 0 <= x && 2 <= y && 2 <= x + y && x - y <= -2 ==> 2 <= y
27   ensures 0 <= x && 2 <= y && 2 <= x + y && x - y <= -2 ==> x - y <= -2
28   ensures 0 <= x && 2 <= y && 2 <= x + y && x - y <= -2 ==> 3 <= res
29   ensures 1 <= x && 2 <= y && 3 <= x + y && x - y <= -1 ==> x - y <= -1
30   ensures 1 <= x && 2 <= y && 3 <= x + y && x - y <= -1 ==> 2 <= y
31   ensures 1 <= x && 2 <= y && 3 <= x + y && x - y <= -1 ==> 6 <= res + y
32   ensures 1 <= x && 2 <= y && 3 <= x + y && x - y <= -1 ==> 3 <= x + y
33   ensures 1 <= x && 2 <= y && 3 <= x + y && x - y <= -1 ==> 4 <= res
34   ensures 1 <= x && 2 <= y && 3 <= x + y && x - y <= -1 ==> 2 <= res - y
35   ensures 1 <= x && 2 <= y && 3 <= x + y && x - y <= -1 ==> 3 <= res - x
36   ensures 1 <= x && 2 <= y && 3 <= x + y && x - y <= -1 ==> 5 <= res + x
37   ensures 1 <= x && 2 <= y && 3 <= x + y && x - y <= -1 ==> 1 <= x
38   ensures 1 <= x && 1 <= y && 2 <= x + y && x - y <= 0 ==> 4 <= res + y
39   ensures 1 <= x && 1 <= y && 2 <= x + y && x - y <= 0 ==> 3 <= res
40   ensures 1 <= x && 1 <= y && 2 <= x + y && x - y <= 0 ==> 2 <= x + y
41   ensures 1 <= x && 1 <= y && 2 <= x + y && x - y <= 0 ==> 2 <= res - y
42   ensures 1 <= x && 1 <= y && 2 <= x + y && x - y <= 0 ==> 1 <= y
43   ensures 1 <= x && 1 <= y && 2 <= x + y && x - y <= 0 ==> 2 <= res - x
44   ensures 1 <= x && 1 <= y && 2 <= x + y && x - y <= 0 ==> x - y <= 0
45   ensures 1 <= x && 1 <= y && 2 <= x + y && x - y <= 0 ==> 1 <= x
46   ensures 1 <= x && 1 <= y && 2 <= x + y && x - y <= 0 ==> 4 <= res + x
47   ensures 1 <= x && 0 <= y && 1 <= x + y && x - y <= 1 ==> 2 <= res - y
48   ensures 1 <= x && 0 <= y && 1 <= x + y && x - y <= 1 ==> 2 <= res + y
49   ensures 1 <= x && 0 <= y && 1 <= x + y && x - y <= 1 ==> 3 <= res + x
50   ensures 1 <= x && 0 <= y && 1 <= x + y && x - y <= 1 ==> 1 <= x + y
51   ensures 1 <= x && 0 <= y && 1 <= x + y && x - y <= 1 ==> 2 <= res
52   ensures 1 <= x && 0 <= y && 1 <= x + y && x - y <= 1 ==> 1 <= x
53   ensures 1 <= x && 0 <= y && 1 <= x + y && x - y <= 1 ==> 0 <= y
54   ensures 1 <= x && 0 <= y && 1 <= x + y && x - y <= 1 ==> 1 <= res - x
55   ensures 1 <= x && 0 <= y && 1 <= x + y && x - y <= 1 ==> x - y <= 1
56   ensures x == 0 && y == 1 ==> x == 0
57   ensures x == 0 && y == 1 ==> y == 1
58   ensures x == 0 && y == 1 ==> res == 2
59   ensures 0 <= x && 3 <= y && 3 <= x + y && x - y <= -3 ==> 7 <= res + y
60   ensures 0 <= x && 3 <= y && 3 <= x + y && x - y <= -3 ==> x - y <= -3
61   ensures 0 <= x && 3 <= y && 3 <= x + y && x - y <= -3 ==> 3 <= x + y
62   ensures 0 <= x && 3 <= y && 3 <= x + y && x - y <= -3 ==> 1 <= res - y
63   ensures 0 <= x && 3 <= y && 3 <= x + y && x - y <= -3 ==> 4 <= res - x
64   ensures 0 <= x && 3 <= y && 3 <= x + y && x - y <= -3 ==> 4 <= res + x
65   ensures 0 <= x && 3 <= y && 3 <= x + y && x - y <= -3 ==> 4 <= res
66   ensures 0 <= x && 3 <= y && 3 <= x + y && x - y <= -3 ==> 3 <= y
67   ensures 0 <= x && 3 <= y && 3 <= x + y && x - y <= -3 ==> 0 <= x
68   ensures 1 <= x && 0 <= y && 1 <= x + y && x - y <= 2 ==> 1 <= x
69   ensures 1 <= x && 0 <= y && 1 <= x + y && x - y <= 2 ==> 1 <= x + y
```

```
70   ensures 1 <= x && 0 <= y && 1 <= x + y && x − y <= 2 ==> 1 <= res − x
71   ensures 1 <= x && 0 <= y && 1 <= x + y && x − y <= 2 ==> 3 <= res + x
72   ensures 1 <= x && 0 <= y && 1 <= x + y && x − y <= 2 ==> 2 <= res − y
73   ensures 1 <= x && 0 <= y && 1 <= x + y && x − y <= 2 ==> 2 <= res + y
74   ensures 1 <= x && 0 <= y && 1 <= x + y && x − y <= 2 ==> 2 <= res
75   ensures 1 <= x && 0 <= y && 1 <= x + y && x − y <= 2 ==> 0 <= y
76   ensures 1 <= x && 0 <= y && 1 <= x + y && x − y <= 2 ==> x − y <= 2
77   ensures 0 <= x && 4 <= y && 4 <= x + y && x − y <= −4 ==> 5 <= res + x
78   ensures 0 <= x && 4 <= y && 4 <= x + y && x − y <= −4 ==> 5 <= res − x
79   ensures 0 <= x && 4 <= y && 4 <= x + y && x − y <= −4 ==> 9 <= res + y
80   ensures 0 <= x && 4 <= y && 4 <= x + y && x − y <= −4 ==> 5 <= res
81   ensures 0 <= x && 4 <= y && 4 <= x + y && x − y <= −4 ==> x − y <= −4
82   ensures 0 <= x && 4 <= y && 4 <= x + y && x − y <= −4 ==> 4 <= x + y
83   ensures 0 <= x && 4 <= y && 4 <= x + y && x − y <= −4 ==> 1 <= res − y
84   ensures 0 <= x && 4 <= y && 4 <= x + y && x − y <= −4 ==> 0 <= x
85   ensures 0 <= x && 4 <= y && 4 <= x + y && x − y <= −4 ==> 4 <= y
86   ensures 1 <= x && 3 <= y && 4 <= x + y && x − y <= −2 ==> 8 <= res + y
87   ensures 1 <= x && 3 <= y && 4 <= x + y && x − y <= −2 ==> 4 <= x + y
88   ensures 1 <= x && 3 <= y && 4 <= x + y && x − y <= −2 ==> 2 <= res − y
89   ensures 1 <= x && 3 <= y && 4 <= x + y && x − y <= −2 ==> x − y <= −2
90   ensures 1 <= x && 3 <= y && 4 <= x + y && x − y <= −2 ==> 1 <= x
91   ensures 1 <= x && 3 <= y && 4 <= x + y && x − y <= −2 ==> 6 <= res + x
92   ensures 1 <= x && 3 <= y && 4 <= x + y && x − y <= −2 ==> 5 <= res
93   ensures 1 <= x && 3 <= y && 4 <= x + y && x − y <= −2 ==> 4 <= res − x
94   ensures 1 <= x && 3 <= y && 4 <= x + y && x − y <= −2 ==> 3 <= y
95   ensures 0 <= x && 5 <= y && 5 <= x + y && x − y <= −5 ==> x − y <= −5
96   ensures 0 <= x && 5 <= y && 5 <= x + y && x − y <= −5 ==> 6 <= res
97   ensures 0 <= x && 5 <= y && 5 <= x + y && x − y <= −5 ==> 6 <= res − x
98   ensures 0 <= x && 5 <= y && 5 <= x + y && x − y <= −5 ==> 6 <= res + x
99   ensures 0 <= x && 5 <= y && 5 <= x + y && x − y <= −5 ==> 5 <= y
100  ensures 0 <= x && 5 <= y && 5 <= x + y && x − y <= −5 ==> 1 <= res − y
101  ensures 0 <= x && 5 <= y && 5 <= x + y && x − y <= −5 ==> 5 <= x + y
102  ensures 0 <= x && 5 <= y && 5 <= x + y && x − y <= −5 ==> 0 <= x
103  ensures 0 <= x && 5 <= y && 5 <= x + y && x − y <= −5 ==> 11 <= res + y
104  ensures 1 <= x && 4 <= y && 5 <= x + y && x − y <= −3 ==> 6 <= res
105  ensures 1 <= x && 4 <= y && 5 <= x + y && x − y <= −3 ==> 10 <= res + y
106  ensures 1 <= x && 4 <= y && 5 <= x + y && x − y <= −3 ==> 2 <= res − y
107  ensures 1 <= x && 4 <= y && 5 <= x + y && x − y <= −3 ==> 7 <= res + x
108  ensures 1 <= x && 4 <= y && 5 <= x + y && x − y <= −3 ==> x − y <= −3
109  ensures 1 <= x && 4 <= y && 5 <= x + y && x − y <= −3 ==> 5 <= x + y
110  ensures 1 <= x && 4 <= y && 5 <= x + y && x − y <= −3 ==> 1 <= x
111  ensures 1 <= x && 4 <= y && 5 <= x + y && x − y <= −3 ==> 4 <= y
112  ensures 1 <= x && 4 <= y && 5 <= x + y && x − y <= −3 ==> 5 <= res − x
113  ensures 0 <= x && 6 <= y && 6 <= x + y && x − y <= −6 ==> 0 <= x
114  ensures 0 <= x && 6 <= y && 6 <= x + y && x − y <= −6 ==> 7 <= res − x
115  ensures 0 <= x && 6 <= y && 6 <= x + y && x − y <= −6 ==> 7 <= res + x
116  ensures 0 <= x && 6 <= y && 6 <= x + y && x − y <= −6 ==> 6 <= x + y
117  ensures 0 <= x && 6 <= y && 6 <= x + y && x − y <= −6 ==> 7 <= res
118  ensures 0 <= x && 6 <= y && 6 <= x + y && x − y <= −6 ==> x − y <= −6
119  ensures 0 <= x && 6 <= y && 6 <= x + y && x − y <= −6 ==> 13 <= res + y
120  ensures 0 <= x && 6 <= y && 6 <= x + y && x − y <= −6 ==> 1 <= res − y
121  ensures 0 <= x && 6 <= y && 6 <= x + y && x − y <= −6 ==> 6 <= y
122  ensures 1 <= x && 5 <= y && 6 <= x + y && x − y <= −4 ==> 12 <= res + y
123  ensures 1 <= x && 5 <= y && 6 <= x + y && x − y <= −4 ==> 2 <= res − y
124  ensures 1 <= x && 5 <= y && 6 <= x + y && x − y <= −4 ==> 6 <= x + y
125  ensures 1 <= x && 5 <= y && 6 <= x + y && x − y <= −4 ==> 8 <= res + x
126  ensures 1 <= x && 5 <= y && 6 <= x + y && x − y <= −4 ==> 5 <= y
127  ensures 1 <= x && 5 <= y && 6 <= x + y && x − y <= −4 ==> x − y <= −4
128  ensures 1 <= x && 5 <= y && 6 <= x + y && x − y <= −4 ==> 6 <= res − x
129  ensures 1 <= x && 5 <= y && 6 <= x + y && x − y <= −4 ==> 7 <= res
130  ensures 1 <= x && 5 <= y && 6 <= x + y && x − y <= −4 ==> 1 <= x
131  ensures 0 <= x && 7 <= y && 7 <= x + y && x − y <= −7 ==> 8 <= res
132  ensures 0 <= x && 7 <= y && 7 <= x + y && x − y <= −7 ==> 7 <= x + y
133  ensures 0 <= x && 7 <= y && 7 <= x + y && x − y <= −7 ==> 8 <= res − x
134  ensures 0 <= x && 7 <= y && 7 <= x + y && x − y <= −7 ==> 8 <= res + x
135  ensures 0 <= x && 7 <= y && 7 <= x + y && x − y <= −7 ==> x − y <= −7
136  ensures 0 <= x && 7 <= y && 7 <= x + y && x − y <= −7 ==> 1 <= res − y
137  ensures 0 <= x && 7 <= y && 7 <= x + y && x − y <= −7 ==> 15 <= res + y
138  ensures 0 <= x && 7 <= y && 7 <= x + y && x − y <= −7 ==> 0 <= x
139  ensures 0 <= x && 7 <= y && 7 <= x + y && x − y <= −7 ==> 7 <= y
140  ensures 1 <= x && 6 <= y && 7 <= x + y && x − y <= −5 ==> 8 <= res
141  ensures 1 <= x && 6 <= y && 7 <= x + y && x − y <= −5 ==> 2 <= res − y
142  ensures 1 <= x && 6 <= y && 7 <= x + y && x − y <= −5 ==> 6 <= y
143  ensures 1 <= x && 6 <= y && 7 <= x + y && x − y <= −5 ==> x − y <= −5
```

```
144     ensures 1 <= x && 6 <= y && 7 <= x + y && x − y <= −5 ==> 14 <= res + y
145     ensures 1 <= x && 6 <= y && 7 <= x + y && x − y <= −5 ==> 7 <= x + y
146     ensures 1 <= x && 6 <= y && 7 <= x + y && x − y <= −5 ==> 7 <= res − x
147     ensures 1 <= x && 6 <= y && 7 <= x + y && x − y <= −5 ==> 9 <= res + x
148     ensures 1 <= x && 6 <= y && 7 <= x + y && x − y <= −5 ==> 1 <= x
149     ensures 0 <= x && 8 <= y && 8 <= x + y && x − y <= −8 ==> 8 <= x + y
150     ensures 0 <= x && 8 <= y && 8 <= x + y && x − y <= −8 ==> 9 <= res − x
151     ensures 0 <= x && 8 <= y && 8 <= x + y && x − y <= −8 ==> 9 <= res + x
152     ensures 0 <= x && 8 <= y && 8 <= x + y && x − y <= −8 ==> 8 <= y
153     ensures 0 <= x && 8 <= y && 8 <= x + y && x − y <= −8 ==> 1 <= res − y
154     ensures 0 <= x && 8 <= y && 8 <= x + y && x − y <= −8 ==> 9 <= res
155     ensures 0 <= x && 8 <= y && 8 <= x + y && x − y <= −8 ==> x − y <= −8
156     ensures 0 <= x && 8 <= y && 8 <= x + y && x − y <= −8 ==> 0 <= x
157     ensures 0 <= x && 8 <= y && 8 <= x + y && x − y <= −8 ==> 17 <= res + y
158     ensures 1 <= x && 8 <= y && 9 <= x + y && x − y <= −7 ==> 2 <= res − y
159     ensures 1 <= x && 8 <= y && 9 <= x + y && x − y <= −7 ==> 8 <= y
160     ensures 1 <= x && 8 <= y && 9 <= x + y && x − y <= −7 ==> 1 <= x
161     ensures 1 <= x && 8 <= y && 9 <= x + y && x − y <= −7 ==> 10 <= res
162     ensures 1 <= x && 8 <= y && 9 <= x + y && x − y <= −7 ==> 9 <= res − x
163     ensures 1 <= x && 8 <= y && 9 <= x + y && x − y <= −7 ==> 11 <= res + x
164     ensures 1 <= x && 8 <= y && 9 <= x + y && x − y <= −7 ==> x − y <= −7
165     ensures 1 <= x && 8 <= y && 9 <= x + y && x − y <= −7 ==> 18 <= res + y
166     ensures 1 <= x && 8 <= y && 9 <= x + y && x − y <= −7 ==> 9 <= x + y
167     ensures 0 <= x && 10 <= y && 10 <= x + y && x − y <= −10 ==> 10 <= x + y
168     ensures 0 <= x && 10 <= y && 10 <= x + y && x − y <= −10 ==> 11 <= res
169     ensures 0 <= x && 10 <= y && 10 <= x + y && x − y <= −10 ==> 10 <= y
170     ensures 0 <= x && 10 <= y && 10 <= x + y && x − y <= −10 ==> x − y <= −10
171     ensures 0 <= x && 10 <= y && 10 <= x + y && x − y <= −10 ==> 21 <= res + y
172     ensures 0 <= x && 10 <= y && 10 <= x + y && x − y <= −10 ==> 1 <= res − y
173     ensures 0 <= x && 10 <= y && 10 <= x + y && x − y <= −10 ==> 11 <= res + x
174     ensures 0 <= x && 10 <= y && 10 <= x + y && x − y <= −10 ==> 11 <= res − x
175     ensures 0 <= x && 10 <= y && 10 <= x + y && x − y <= −10 ==> 0 <= x
176     ensures 0 <= x && 0 <= y && 0 <= x + y ==> 0 <= x
177     ensures 0 <= x && 0 <= y && 0 <= x + y ==> 0 <= y
178     ensures 0 <= x && 0 <= y && 0 <= x + y ==> 1 <= res + y
179     ensures 0 <= x && 0 <= y && 0 <= x + y ==> 1 <= res − y
180     ensures 0 <= x && 0 <= y && 0 <= x + y ==> 1 <= res
181     ensures 0 <= x && 0 <= y && 0 <= x + y ==> 0 <= x + y
182     ensures 0 <= x && 0 <= y && 0 <= x + y ==> 1 <= res − x
183     ensures 0 <= x && 0 <= y && 0 <= x + y ==> 1 <= res + x
184   {
185     var t: Int
186     var t1: Int
187     if (x <= 0) {
188       res := y + 1
189     } elseif (y <= 0) {
190       t1 := x − 1
191       t := 1
192       res := ack(t1, t)
193     } else {
194       t1 := y − 1
195       t := ack(x, t1)
196       t1 := x − 1
197       res := ack(t1, t)
198     }
199   }
200
201   method main(a: Int, b: Int) returns (r: Int)
202     requires a >= 0 && b >= 0
203     ensures 0 <= b
204     ensures b − r <= −1
205     ensures 1 <= b + r
206     ensures 0 <= a + b
207     ensures 0 <= a
208     ensures 1 <= r
209     ensures a − r <= −1
210     ensures 1 <= a + r
211   {
212     r := ack(a, b)
213   }
```

Listing A.4: Top-down inferred specification for the Ackermann function. The highlighted precondition at line 202 was added to the program. All postconditions have been inferred by Sample.

A.5 Viper Protocol Message with Inferred Specification

```
{
  "type":"SpecificationInference",
  "file":"/silver/double.vpr",
  "preconditions":[
    {
      "position":"2:1",
      "specifications":[]
    },
    {
      "position":"9:1",
      "specifications":[]
    }
  ],
  "postconditions":[
    {
      "position":"2:1",
      "specifications":[
        "ensures 0 <= r"
      ]
    },
    {
      "position":"9:1",
      "specifications":[
        "ensures 0 <= r"
      ]
    }
  ],
  "invariants":[
    {
      "position":"14:2",
      "specifications":[
        "invariant 0 <= r",
        "invariant 0 <= i - 1"
      ]
    }
  ]
}
```

Listing A.5: A Viper protocol messages sent to the IDE with inferred postconditions and invariants.

A.6 Viper Protocol Error Message


```
{
  "type":"Error",
  "file":"/silver/double.vpr",
  "errors":[
    {
      "start":"2:9",
      "end":"2:13",
      "tag":"sample.error.inferenceOmitted",
      "message":"Inference omitted since some \
        specifications already exist."
    }
  ]
}
```


Listing A.6: Error message sent to Viper IDE. Start and end denote where in the source code a specification already exists.

Bibliography

[1] Lucas Brutschy, Pietro Ferrara, and Peter Müller. Static analysis for independent app developers. *ACM SIGPLAN Notices*, 49(10):847–860, 2014.

[2] Chair of Programming Methodology, ETH Zurich. Sample project page. http://www.pm.inf.ethz.ch/research/sample.html. [Online; accessed 31-July-2017].

[3] Chair of Programming Methodology, ETH Zurich. Silver project page. http://www.pm.inf.ethz.ch/research/viper.html.

[4] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, 1976.

[5] Patrick Cousot and Radhia Cousot. 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*, pages 238–252. ACM, 1977.

[6] Patrick Cousot and Radhia Cousot. Compositional separate modular static analysis of programs by abstract interpretation. In *Proc. SSGRR*, pages 6–10, 2001.

[7] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.

[8] Deepak D'Souza. Interprocedural analysis: Sharir-pnueli's call-strings approach lecture at Department of Computer Sci-

ence and Automation, Indian Institute of Science, Bangalore. http://www.cse.psu.edu/~trj1/cse598-f11/slides/interprocedural-call-strings-2010.pdf. [Online; accessed 24-July-2017].

[9] Pietro Ferrara and Peter Müller. Automatic inference of access permissions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 202–218. Springer, 2012.

[10] Robert Giegerich and Ulrich Möncke. Invariance of approximative semantics with respect to program transformations. In *GI—11. Jahrestagung*, pages 1–10. Springer, 1981.

[11] Bertrand Jeannet, G Lalire, and M Argoud. The interproc analyzer. http://pop-art.inrialpes.fr/interproc/interprocweb.cgi. [Online; accessed 09-August-2017].

[12] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification*, pages 661–667. Springer, 2009.

[13] Ruben Kälin. Advanced features for an integrated verification environment. Master's thesis, ETH Zürich, 2016.

[14] Uday Khedker and Bageshri Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In *Compiler Construction*, pages 213–228. Springer, 2008.

[15] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.

[16] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[17] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[18] Liat Ramati and Dor Nir. Interprocedral analysis class notes program analysis course given by prof. Mooly Sagiv, fourth lecture given by Noam Rinetzky. http://www.cs.tau.ac.il/~msagiv/courses/pa07/Interprocedural%20Analysis4.pdf, 2007. [Online; accessed 24-July-2017].

[19] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[20] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[21] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. 1978.

[22] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, volume 9, pages 148–172. Springer, 2009.

[23] Caterina Urban. *Static analysis by abstract interpretation of functional temporal properties of programs*. PhD thesis, École Normale Supérieure, 2015.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Interprocedural Static Analysis by Abstract Interpretation

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Rindisbacher	Flurin

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zürich, August 15th 2017

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.