# Disjunctive Invariants for Modular Static Analysis

**Corneliu Popeea**

***(B.Sc., University Politehnica of Bucharest, Romania)***

**A THESIS SUBMITTED**

**FOR THE DEGREE OF DOCTOR OF PHILOSOPHY**

**DEPARTMENT OF COMPUTER SCIENCE**

**NATIONAL UNIVERSITY OF SINGAPORE**

**July 2008**

**Disjunctive Invariants for Modular Static Analysis**

Approved by:

Date Approved _____

# ACKNOWLEDGEMENTS

I am deeply grateful to all the people who have helped me progress through my PhD years. First of all, I would like to thank my advisor Chin Wei Ngan who initiated me in the world of research and contaminated me with the passion to pursue challenging goals. His enthusiasm and continuous support helped me overcome those dispirited moments, while his many insightful advices showed me the way ahead. After six years of collaboration, he will always be an inspiration for me.

I wish to express my gratitude to Khoo Siau Cheng, Thomas Henzinger, Joxan Jaffar and Abhik Roychoudhury for accepting to be in my thesis committee and for their helpful comments on my work. I am also indebted to Martin Sulzmann, Martin Rinard, Zhenjiang Hu, Robert Glück, Xavier Rival, Francesco Logozzo and Andrew Santosa for advices given over the years on research and my thesis work.

During my graduate studies, I was fortunate to work with amazing colleagues who had influenced me both inside and outside of my research. I am grateful to Florin, Dana, Razvan, Saswat, Meng, Ping, Kenny, Huu Hai, Shengchao, Stefan, Cristina, David, Raluca, Razvan, Ana, Beatrice, Alexandru, Cristian, Mihai, Mariuca, Mihai, Ioana, Andrei, Claudia and Marian for generously sharing their knowledge with me and for making the PLS lab (my second home!) a cheerful place.

Living in Singapore was a new experience for me and I am thankful especially to Aurbind, Gabriel, Diana, Tudor, Nandini, Lei Yang, Sara, Dmitry, Aleksandar, Alberto, Trang, Ryan, Olaf and most of all Cristina for helping me enjoy my time here. This exciting experience and a complete change of mind would not be possible without you. My life before Singapore was profoundly touched by my happy and heartwarming friends and I hope our paths in life will cross again. Finally, I want to thank my parents and my dear sister for educating me and for always believing in me.

*Your bounty is beyond my speaking;*

*But though my mouth be dumb, my heart shall thank you.* (Nicholas Rowe)

# TABLE OF CONTENTS

**ACKNOWLEDGEMENTS** . . . . . **v**

**SUMMARY** . . . . . **xi**

**LIST OF FIGURES** . . . . . **xiii**

**I INTRODUCTION** . . . . . **1**

1.1 Background . . . . . 1

1.2 Contributions of this Thesis . . . . . 4

1.2.1 Disjunctive Abstract Domain . . . . . 4

1.2.2 Deriving Preconditions for Modular Static Analysis . . . . . 9

1.2.3 Dual Static Analysis . . . . . 12

1.3 Thesis Overview . . . . . 14

**II CORE LANGUAGE, SEMANTICS AND ABSTRACTION** . . . . **17**

2.1 Syntax of Imp Language . . . . . 17

2.1.1 Other Language Features . . . . . 19

2.2 Concrete (Operational) Semantics . . . . . 20

2.3 Abstract Semantics . . . . . 22

2.3.1 Overview . . . . . 22

2.3.2 Forward Reasoning Rules . . . . . 23

2.3.3 Fixed Point Approximation . . . . . 26

2.3.4 Method Summary . . . . . 28

**III DISJUNCTIVE FIXED-POINT ANALYSIS** . . . . . **31**

3.1 Background . . . . . 31

3.2 Overview of Fixed-Point Analysis . . . . . 32

3.2.1 Computing Fixed-Points in the Polyhedron Abstract Domain . 35

3.2.2 Computing Fixed-Points in a Disjunctive Abstract Domain . . . 35

3.3 Disjunctive Abstract Domain . . . . . 37

3.3.1 Planar Affinity and Selective Hulling . . . . . 37

3.3.2 Widening Operator . . . . . 41

3.3.3 Higher-Order Planar Affinity Measure . . . . . 44

3.4 Forward Reasoning Rules . . . . . 44

3.4.1 Verification of Preconditions . . . . . 46

3.5 Mixing Boolean and Integer Constraints . . . . . 47
3.6 Experimental Results . . . . . 49
3.7 Examples . . . . . 51
3.7.1 McCarthy's 91 Function . . . . . 52
3.7.2 Quicksort Example . . . . . 52
3.7.3 Fast Fourier Transform Example . . . . . 54
3.8 Correctness . . . . . 55
3.8.1 Termination of Analysis . . . . . 56
3.8.2 Soundness of Postcondition Inference . . . . . 57
3.8.3 Necessary Precondition . . . . . 57
3.8.4 Totally-Safe Program . . . . . 58
3.9 Related Work . . . . . 59
3.10 Summary . . . . . 60

**IV DERIVING PRECONDITIONS FOR MODULAR STATIC ANALYSIS . . . . . 61**

4.1 Overview . . . . . 62
4.2 An Imperative Language . . . . . 66
4.2.1 Target Language . . . . . 66
4.3 Type Inference Rules . . . . . 70
4.4 Recursion Analysis . . . . . 73
4.4.1 Deriving Postcondition . . . . . 73
4.4.2 Deriving Recursive Invariant . . . . . 75
4.4.3 Deriving Precondition . . . . . 76
4.5 Array Indirections . . . . . 77
4.6 Deriving Smaller Formulae . . . . . 79
4.7 Flexivariant Specialization . . . . . 80
4.8 Experimental Results . . . . . 84
4.9 Correctness . . . . . 87
4.10 Related Work . . . . . 87
4.11 Summary . . . . . 89

**V DUAL STATIC ANALYSIS . . . . . 91**

5.1 Introduction . . . . . 91
5.2 Our Approach . . . . . 93

5.3 Summarizing Dual Over-approximations . . . 96
5.3.1 Forward Reasoning Rules . . . 97
5.3.2 Fixed-Point Analysis . . . 101
5.4 Further Improvements . . . 103
5.4.1 Precise Error Tracing . . . 103
5.4.2 Non-Termination as Bugs . . . 104
5.4.3 Alias Analysis for Heap-Allocated Objects . . . 106
5.5 Experimental Results . . . 107
5.5.1 Comparison with BLAST . . . 109
5.5.2 Comparison with SYNERGY . . . 110
5.5.3 Examples from Verisec Benchmark . . . 112
5.5.4 Beyond Safety to Memory Bounds Inference . . . 113
5.6 Correctness of the Dual Static Analysis . . . 114
5.6.1 Consistency between Static and Dynamic Semantics . . . 115
5.6.2 Proof Methodology . . . 116
5.6.3 Main Theorem and Its Proof . . . 117
5.6.4 Soundness of the Fixed-Point Analysis . . . 125
5.6.5 Corollaries of the Main Theorem . . . 127
5.7 Related Work . . . 128
5.8 Summary . . . 133

**VI CONCLUSIONS . . . 135**

6.1 Main Results . . . 135
6.2 Future Work . . . 135

**BIBLIOGRAPHY . . . 139**

**INDEX . . . 151**

# SUMMARY

In this thesis we study the application of modular static analysis to prove program safety and detection of program errors. In particular, we shall consider imperative programs that rely on numerical invariants. The analyses proposed in this thesis benefit from the relationship between a modular analyzer and a precise disjunctive domain. A modular analyzer requires a precise abstract domain to reason about the symbolic method inputs. On the other hand, a modular analyzer has a local scope and therefore favors more complex invariants than those that are usually involved in global analyzers.

The thesis makes three main contributions. Firstly, to handle the challenges of disjunctive analyses, we introduce the notion of affinity to characterize how closely related is a pair of disjuncts. Finding related elements in the conjunctive (base) domain allows the formulation of precise hull and widening operators lifted to the disjunctive (powerset extension of the) base domain. We have implemented a static analyzer based on the disjunctive polyhedral analysis where the relational domain and the proposed operators can progressively enhance precision at a reasonable cost.

Secondly, we designed a modular analyzer that combines forward and backward analyses. The forward analysis aims to infer method postconditions, but it also discovers invariants that are useful in the backward derivation of sufficient preconditions. To increase the efficiency of the analysis, we designed a technique to strengthen preconditions and trade precision for speed. Rather than deriving one program precondition for proving program safety, our analysis derives individual preconditions for each check and goes one step further by performing aggressive optimizations of checks.

Our final objective is to support either a proof of the absence of bugs in the case of a valid program or bug finding in the case of a faulty program. We propose a dual static analysis that is designed to track concurrently two over-approximations: the success and the failure outcomes. The overlap between the two outcomes signifies imprecision in analysis and can be used to guide abstraction refinement. More interestingly, due to the

concurrent computation of outcomes, we can identify two significant input conditions: a never-bug condition that implies safety for inputs that satisfy it and a must-bug condition that characterizes inputs that lead to true errors in the execution of the program. As a result, our analysis can identify a part of the alarms as being true errors and reduces the manual effort of analyzing alarms to a smaller group of may-bugs.

# LIST OF FIGURES


1.1 Simple example . . . . . 5
1.2 Program state approximated at each label via forward analysis . . . . . 6
1.3 Program state approximated at each label via backward analysis . . . . 7
1.4 Example for interprocedural analysis . . . . . 10
2.1 Syntax of the language IMP . . . . . 18
2.2 Example of loop translation . . . . . 20
2.3 Operational semantics . . . . . 21
2.4 Overview of analysis framework . . . . . 22
2.5 Syntax of formulae and constraint abstractions . . . . . 24
2.6 Simple recursive example . . . . . 26
3.1 Pairs of disjuncts with similar Hausdorff distance . . . . . 41
3.2 Forward reasoning rules . . . . . 46
3.3 Statistics for postcondition inference . . . . . 49
3.4 Statistics for check elimination . . . . . 51
3.5 McCarthy's 91 function . . . . . 52
3.6 Quicksort example . . . . . 53
3.7 FFT example . . . . . 55
3.8 Inner loop of FFT example . . . . . 56
4.1 Inference and specialization : An example . . . . . 64
4.2 Inferred $\text{IMP}_I$ language . . . . . 67
4.3 Type inference rules . . . . . 69
4.4 Summation program . . . . . 75
4.5 Statistics for array bound checks elimination . . . . . 85
5.1 Classifying bugs . . . . . 95
5.2 Forward reasoning rules . . . . . 98
5.3 Adding `LOOP` to the bug classification . . . . . 105
5.4 Statistics for a set of array-based programs without bugs . . . . . 108
5.5 Buggy codes . . . . . 108
5.6 Comparison with Blast . . . . . 109
5.7 Examples from the SYNERGY paper . . . . . 110
5.8 Memory bounds estimation . . . . . 114

5.9 Classification of various analyses for proving safety and finding bugs . . 129

# CHAPTER I

# INTRODUCTION

During the last century computer-driven systems have become increasingly important in our daily lives. Their reliance on software systems implies that any fault in the software may cause the entire system to misbehave. Use of flight control systems coupled with digital computers has been adopted under the concept "fly-by-wire" on board of Airbus and Boeing planes and similarly the "drive-by-wire" initiative promises to be followed by the automotive industry. Two extreme failure examples are a spacecraft explosion caused by a floating-point conversion error (Ariane 5 failure, 1996 [99]) and the Mars Climate Orbiter crash in 1999 caused by an incorrect conversion between Imperial units and metric units [143]. Errors are even more obvious in software that is less safety-critical. "Blue screens of deaths" or "segmentation faults" are errors that cause frustration and countless hours of lost productivity. It was estimated that the costs of having an inadequate infrastructure for software testing in the United States are between USD 22.2 billion and USD 59.5 billion, approximately 0.2 to 0.6 percent of the country's Gross Domestic Product (GDP) [133]. The tremendous cost of software errors has led to increasing interest in methods for automatic analysis and software verification.

## 1.1 Background

Before dissecting the various techniques for program analysis, we should recognize that a related approach for preventing software errors is to design programming language of increasing sophistication, where (some kinds of) programmer errors are fully prevented [48, 86, 31]. While this approach is worthy to follow, it does not promise to solve the class of errors that exist in the current generation of software. Many software applications are written using the C programming language that was developed in the seventies. The widespread adoption of the C language is due to its flexibility as it encourages programmers to take control over low-level access to memory and allows constructs

that map efficiently to machine instructions. To quote from Bjarne Stroustrup, "C makes it easy to shoot yourself in the foot". Unfortunately, this ease of committing errors becomes prevalent in the current software systems when considering their ever increasing complexity.

To solve these difficulties, various techniques promise to provide automated support for detection of software errors ranging from testing to static analysis. *Systematic testing* or concrete state space exploration [59, 111] attempts to search through all the feasible paths of the programs. While testing can uncover serious errors, it cannot offer guarantees that no more bugs remain undiscovered in the program. On the other hand, *static analysis* [37, 43] uses abstraction to interpret exhaustively all the feasible program paths and compute which are the concrete program states that can be reached by execution. Static analysis sidesteps the undecidable problem of computing the reachable concrete states by using over-approximation to (potentially) reachable abstract states. Due to approximation, static analysis may report false positives that are possible bugs that do not exist in practice. Regardless of the inherent imprecision in static analysis, many software properties have been verified and we list only some of the most impressive results obtained:

- Verification against runtime errors including out-of-bound array accesses: ASTRÉE [12] for the flight control software of Airbus A340 and A380; C Global Surveyor [145] for the flight software of the Mars Path-Finder and the Deep Space 1 missions; ESPX [76] for preventing buffer overflows in future versions of Microsoft products.

- Ensuring proper usage of resources: ESP [46] for verifying file I/O in the gcc compiler; Saturn [148] for finding incorrect usage of lock related functions through the Linux kernel.

- Verifying consistency of complex data structure operations including circular linked lists, sorted linked lists, priority queues, red-black trees [115, 18].

- Finding errors in critical device drivers: Slam [5], Blast [80] and Static Driver Verifier [6] for safety properties; Terminator [34, 32] and Mutant [9] for liveness properties (e.g. termination).

From the properties enumerated previously, our study will focus on checking for out-of-bound array accesses as an example of assertion checking. We aim to provide a *sound* static analysis, meaning that if a property of a program is verified, then the program execution is guaranteed not to violate that property. To achieve soundness, we shall be content with the possibility of reporting alarms that may be false positives. However, to minimize their number, we emphasize on the *precision* of our analysis and on the *modularity* principle as a way to achieve precision regardless of the size of the program to analyze. This principle of performing separate local analyses and composing their results was also argued by Cousot and Cousot in their paper on "Modular Static Program Analysis" [40]:

> *"The central idea is that of compositional separate static analysis of program parts where very large programs are analyzed by analyzing parts (such as methods) separately and then by composing the analyses of these program parts to get the required information on the whole program. Components can be analyzed with a high precision whenever they are chosen to be small enough."*

In their paper [40], Cousot and Cousot give some guidelines for designing modular static analyses. The first proposal, the *worst-case separate analysis* consists in considering that absolutely no information is known on the interfaces of program parts. As a second approach, the *separate analysis with user-provided interfaces* asks the user to provide information about the interfaces of each program part. While pragmatic, these proposals give up precision in the first case and automation in the second case. The current thesis focuses on the third proposal, the *symbolic relational separate analysis*. We attempt to study its applicability to the analysis of software errors and to realize its potential. With this proposal, each program part is analyzed by giving symbolic names to all external objects used or modified in that part. The interaction between the local and the external objects is recorded symbolically and represents a *summary* or interface of the current program part. Wherever that program part is invoked, the computed summary can be used and the analysis proceeds without sacrificing precision. This analysis has smaller local scope and therefore can be very precise.

## 1.2 Contributions of this Thesis

This thesis makes three main contributions. Firstly, we present a *new disjunctive abstract domain* meant to enhance analysis precision at a reasonable cost. Secondly, we propose a *modular technique for deriving preconditions* sufficient to guarantee program safety. With these two techniques, we are able to derive both postconditions and preconditions and realize a completely modular analyzer for proving program safety. Since an analyzer that aims to prove program safety may report alarms that correspond in part to false positives, there is a need to (manually) classify the feasibility of alarms. Our third proposal is a *dual static analysis* that can identify (automatically) a part of the alarms as being true errors. More specifically, the dual static analysis identifies both a never-bug condition that implies program safety and a must-bug condition that leads to true errors (modulo program termination). The imprecision of static analysis results in a may-bug condition that characterizes inputs leading to either possible errors or false positives.

After we briefly described our contributions, we will show examples to motivate our goals and list the challenges that need to be overcome.

### 1.2.1 Disjunctive Abstract Domain

We shall highlight various techniques to discover static invariants that hold at each program point. A static invariant can be computed either using a forward or a backward analysis. A *forward* derivation traverses the statements of the program in the same order in which they execute, while a *backward* derivation does the traversal in the reverse order. In a forward analysis, a static invariant is usually computed assuming any inputs to the program. Therefore it represents an *over-approximation* of the concrete program state computed using particular program inputs. While including some unfeasible concrete states results in an over-approximating invariant, it is also possible to exclude some feasible states and compute an *under-approximating* invariant. [1]

The discovered invariants are used to prove the safety of either implicit checks (e.g. array bound checks where the index used to access an array must be within the bounds

[1] The crossbreeding of over/under approximations with forward/backward derivations leads to four analyses akin to the classical dataflow analyses [117, Section 2.3].

of the array) or explicit checks introduced using the language construct `assert`. To *prove safety* of checks, a forward analysis compares the computed invariant at the check point with the check to be satisfied. We will illustrate this process using the following example adapted from [136]:

```
void foo (int x) {
   l0 : int y; bool b;
   l1 : if (x > 0) { y = x; }
   l2 : else { y = −x;
   l3 : }
   l4 : b = (y > 10);
   l5 : assert(b ⇒ (x < −10 ∨ 10 < x));
}
```

**Figure 1.1:** Simple example

The method `foo` computes the absolute value of the integer parameter `x` and assigns it to the variable `y`. The boolean flag `b` is set to `true` when the value of the variable `y` is bigger than `10`. It can easily be observed that the assertion at label $l_5$ is satisfied for any execution of the program and for any value of the parameter `x`. We will illustrate two facts that may influence the ability of a static analyzer to prove this assertion: firstly, the precision of the abstract domain and secondly, whether the invariants are derived using a forward or a backward analysis.

The abstract domain defines a class of properties that can be used to compute and represent program invariants. There exists a wide variety of abstract domains that could be used to capture the numerical properties that interest us. These domains balance expressive power with computational cost. Two seminal papers introduced the abstract domains of intervals [36] and polyhedra [43]. The interval abstract domain is able to capture invariants stating that the variable `y` is positive at label $l_2$ (after the assignment statement is executed). In general, the form of interval invariants is a conjunction of constraints of the form $\pm x \leq c$, where `x` is some program variable while `c` is a constant. Analysis using the interval domain is efficient, but it loses precision by not capturing constraints that relate multiple program variables. [2] The polyhedron abstract domain is able to capture a more general form of invariants, as conjunctions of linear inequalities

[2]For this reason, the interval domain is called *non-relational* .

relating multiple variables: $\mathtt{a_1x_1}+\ldots\mathtt{a_nx_n} \leq \mathtt{c}$. In this format, program variables are represented by $\mathtt{x_i}$, while $\mathtt{a_i}$ and $\mathtt{c}$ are arbitrary constants.

| Label | Conjunctive (polyhedron) domain | Disjunctive (polyhedron) domain |
|---|---|---|
| $l_0$ | $\top$ | $\top$ |
| $l_1$ | $x{>}0 \wedge y{=}x$ | $x{>}0 \wedge y{=}x$ |
| $l_2$ | $x{\leq}0 \wedge y{=}{-}x$ | $x{\leq}0 \wedge y{=}{-}x$ |
| $l_3$ | $y{\geq}x \wedge y{\geq}{-}x$ | $x{>}0 \wedge y{=}x \ \vee$<br>$x{\leq}0 \wedge y{=}{-}x$ |
| $l_4$ | $y{\geq}x \wedge y{\geq}{-}x$ | $b{=}\mathtt{true} \wedge y{>}10 \wedge y{=}x \ \vee$<br>$b{=}\mathtt{true} \wedge y{>}10 \wedge y{=}{-}x \ \vee$<br>$b{=}\mathtt{false} \wedge y{\leq}10 \wedge x{>}0 \wedge y{=}x \ \vee$<br>$b{=}\mathtt{false} \wedge y{\leq}10 \wedge x{\leq}0 \wedge y{=}{-}x$ |
| $l_5$ | check cannot be proven | check can be proven |

**Figure 1.2:** Program state approximated at each label via forward analysis with conjunctive and disjunctive abstract domains

For our example from Figure 1.1, the invariants obtained using a forward analysis with the conjunctive polyhedron domain are shown in the second column of Figure 1.2. The forward derivation starts at label $\mathtt{l_0}$ with a special element $\top$, meaning that all the program variables, $\mathtt{x}, \mathtt{y}$ and $\mathtt{b}$ are unconstrained. At label $\mathtt{l_1}$, after the evaluation of the assignment from the first branch, an invariant would capture the formula ($\mathtt{x{>}0} \wedge \mathtt{y{=}x}$). Abstract domains like the interval or the polyhedron are imprecise when capturing invariants at join points that follow, for example, conditional statements. In particular, at label $\mathtt{l_3}$, the invariant after the conditional statement represents the over-approximation of the invariants from the two branches. The invariant at label $\mathtt{l_3}$ computed with the interval domain is determined to be $\mathtt{y{\geq}0}$. With the increased precision of the polyhedron domain, a refined invariant could be obtained at label $\mathtt{l_3}$: ($\mathtt{y{\geq}x} \wedge \mathtt{y{\geq}{-}x}$). Neither of these two invariants is precise enough to capture the fact that the variable $\mathtt{y}$ represents the absolute value of the variable $\mathtt{x}$. At the next step, the invariant at label $\mathtt{l_4}$ is not able to capture the relation between the variables $\mathtt{x}, \mathtt{y}$ and $\mathtt{b}$. Consequently, the check at label $\mathtt{l_5}$ cannot be proven.

A numerical abstract domain can be refined by adding elements that allow disjunctive properties to be represented precisely. This refinement can be done systematically [39] and results in a *powerset extension* of the base conjunctive domain. After illustrating the forward analysis with a conjunctive domain, we will compare it with the derivation shown

in the last column of Figure 1.2 based on a disjunctive domain. While the invariants at labels $\texttt{l}_0, \texttt{l}_1$ and $\texttt{l}_2$ are unchanged, the disjunctive domain captures more precisely the state at label $\texttt{l}_3$. The invariant captures each branch via a separate disjunct as $((\texttt{x}{>}0 \wedge \texttt{y}{=}\texttt{x}) \vee (\texttt{x}{\leq}0 \wedge \texttt{y}{=}{-}\texttt{x}))$. Further, the conditional assignment at label $\texttt{l}_4$ assigns a true value to `b` provided the value of `y` is bigger than `10`; otherwise `b` is assigned `false`. Consequently, the invariant at label $\texttt{l}_4$ captures the relations between the variables `x`, `y` and `b` with four disjuncts. Using this invariant, the check $(\texttt{b} \Rightarrow (\texttt{x}{<}{-}10 \vee 10{<}\texttt{x}))$ from label $\texttt{l}_5$ can be proven: the two disjuncts where `b` is `true` satisfy the property that the absolute value of `x` is bigger than `10`.

While we illustrated the need for disjunctive invariants using forward analysis, we can argue that a disjunctive abstract domain is also useful for backward analysis. In fact, the example from Figure 1.1 was used in [136] to show that the backward analysis can be more effective than a forward analysis. For this particular example, the backward analysis can indeed prove the check using a conjunctive domain!

| Label | Over-approximation from $\neg(chk)$ | Under-approximation from $chk$ |
|---|---|---|
| $l_5$ | $b{=}\texttt{true} \wedge -10{\leq}x{\leq}10$ | $b{=}\texttt{false} \ \vee$<br>$b{=}\texttt{true} \wedge x{<}{-}10 \ \vee$<br>$b{=}\texttt{true} \wedge x{>}10$ |
| $l_4$ | $y{>}10 \wedge -10{\leq}x{\leq}10$ | $y{<}10 \ \vee$<br>$y{>}10 \wedge x{<}{-}10 \ \vee$<br>$y{>}10 \wedge x{>}10$ |
| $l_3$ | (same as $l_4$) | (same as $l_4$) |
| $l_2$ | $\bot$ $(x{<}{-}10 \wedge -10{\leq}x{\leq}10)$ | $\top$ $(x{\geq}{-}10 \ \vee \ x{<}{-}10)$ |
| $l_1$ | $\bot$ $(x{>}10 \wedge -10{\leq}x{\leq}10)$ | $\top$ $(x{\leq}10 \ \vee \ x{>}10)$ |
| $l_0$ | $\bot$ (check can be proven) | $\top$ (check can be proven) |

**Figure 1.3:** Program state approximated at each label via backward analysis

The second and third columns from Figure 1.3 show the invariants that are inferred using backward analysis. This analysis starts with the location of the check to be proven $\texttt{l}_5$, where error states and safe states can be clearly identified. The error states correspond to the negation of the check $(\texttt{b}{=}\texttt{true} \wedge -10{\leq}\texttt{x}{\leq}10)$, while the safe states correspond to the check formula, written in disjunctive normal form as $(\texttt{b}{=}\texttt{false}) \vee (\texttt{b}{=}\texttt{true} \wedge \texttt{x}{<}{-}10) \vee (\texttt{b}{=}\texttt{true} \wedge \texttt{x}{>}10)$.

An over-approximating backward analysis starts with the error states and aims to

prove that no program inputs *may* lead to the error states (a may analysis). The error state from label $\texttt{l}_5$ may be reached from label $\texttt{l}_4$ provided that ($\texttt{y}{>}10 \wedge -10{\leq}\texttt{x}{\leq}10$). The backward derivation continues with the two conditional branches. Contradictions are obtained on each branch, represented with the special element $\bot$ that stands for the unsatisfiable invariant `false`. To reach the error state from the label $\texttt{l}_2$, it is necessary that `x` satisfies the following constraint: ($\texttt{x}{<}-10 \wedge -10{\leq}\texttt{x}{\leq}10$), which is impossible since this constraint is unsatisfiable. Similarly for the other branch, at label $\texttt{l}_1$ the constraint ($\texttt{x}{>}10 \wedge -10{\leq}\texttt{x}{\leq}10$) cannot be satisfied by any value for the input variable `x`. Since the error states cannot be reached from any of the two conditional branches, they cannot be reached from the beginning of the method at label $\texttt{l}_0$ and thus, the check is proven safe.

Despite the success in proving this particular example starting with the error states, conjunctive invariants may not be sufficient in general for backward analysis. This fact is illustrated using the combination between backward analysis and under-approximation as pioneered by Suzuki and Ishihata [144]. For this derivation, the starting invariant at label $\texttt{l}_5$ is the check to be proven. The aim of this under-approximating analysis is to prove that all inputs *must* lead to the safe states (a must analysis). For the current example, such an analysis fails when using a conjunctive domain: it computes only imprecise under-approximations at each program point (the `false` invariant). On the other hand, the third column of Figure 1.3 shows that the check can be proven if disjunctive invariants are available for analysis. As a summary to the various forward/backward analyses presented, we argue the general usefulness of a disjunctive abstract domain.

While a disjunctive domain is able to prove more checks due to its increased precision, the precision comes with a higher cost that hindered the adoption of disjunctive abstract domains. A disjunctive domain has an exponential number of elements when compared to the base conjunctive domain. Another potential problem shows in the case of recursive programs, where, unlike the simple non-recursive example from Figure 1.1, the disjunctive analysis may result in an unbounded number of disjuncts. One well-known approach to control the number of disjuncts during analysis is to use a domain where the number of disjuncts is syntactically bounded. In this setting, the challenge is to find appropriate disjuncts that can be merged without (evident) losses in precision.

Another challenge is to ensure the termination of analysis, by adapting the techniques based on widening operators from the conjunctive [37, 43] to the disjunctive setting.

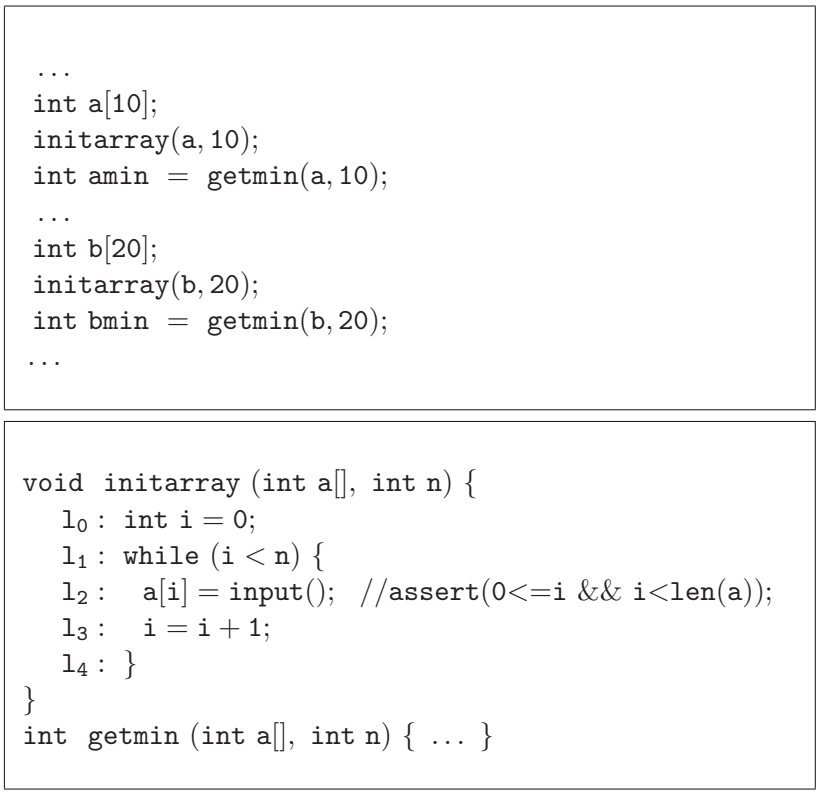
To handle these challenges in disjunctive analysis, we introduce in Chapter 3 the notion of *affinity* to characterize how closely related is a pair of disjuncts. Finding related elements in the conjunctive (base) domain allows the formulation of precise hull and widening operators lifted to the disjunctive (powerset extension of the) base domain. We have implemented a static analyzer based on the disjunctive polyhedral analysis where the relational domain and the proposed operators can progressively enhance precision at a reasonable cost.

### 1.2.2 Deriving Preconditions for Modular Static Analysis

A main challenge for static analysis is its extension from the *intraprocedural* setting used for our previous examples to *interprocedural analysis* where calls to procedures can be handled precisely and efficiently. A simple approach for solving this challenge is to assume method specifications in the form of pre- and post-conditions are available in the program. In this case, the intraprocedural analysis can be extended straightforward to the interprocedural setting. For the situation when pre- and post-conditions are not available, there are two main approaches to analyzing the program.

The *global* approach attempts to mimic the execution of a program in the order in which method calls and return instructions are processed. The analysis starts with the "main" method and traverses the call graph in *top-down* order. This approach has the advantage of exploiting the context (program invariant) of a call to the method `mn` when analyzing the callee, the method `mn`. Unfortunately, this approach discards the natural boundaries from the method declarations since invariants combine information from the call context with information local to the callee. This may lead to an explosion in the size of the invariants: elaborate techniques are reported in Astrée to process precisely and efficiently invariants relating tens of thousands of variables [12]. We will illustrate this global approach to interprocedural analysis using the program fragment shown in Figure 1.4. In this program, the method at the top of the figure declares two arrays, initializes them and then computes the minimum element from each array.

A typical global static analyzer derives invariants for a method in each of its call contexts.



```
...
int a[10];
initarray(a, 10);
int amin = getmin(a, 10);
...
int b[20];
initarray(b, 20);
int bmin = getmin(b, 20);
...
```

```
void initarray (int a[], int n) {
   l0 : int i = 0;
   l1 : while (i < n) {
   l2 :   a[i] = input();  //assert(0<=i && i<len(a));
   l3 :   i = i + 1;
   l4 : }
}
int getmin (int a[], int n) { ... }
```

**Figure 1.4:** Example for interprocedural analysis

In our example, the method `initarray` is analyzed twice, corresponding to the call contexts of `initarray(a, 10)` and `initarray(b, 20)`. In each call context, the assertion at label $l_2$ is proven safe, but the effort of proving the assertion is duplicated. Furthermore, the invariants that need to be inferred inside the method `initarray` could be larger than required since they take into account all the information from the call context.

A second alternative for analyzing interprocedural programs is a *modular* approach that performs local analysis within the boundaries of a method. A method `mn` is analyzed without assuming anything about its call contexts and an abstraction of the relation between values at the entry and at the method exit is computed [38, 141]. This abstraction represents a *summary* of the method body and therefore this approach is also denoted as *summary-based analysis* [118, 148]. Throughout this thesis, we will use the attributes *summary-based* and *modular* interchangeably for such an approach to analysis.

One advantage of this modular approach is that it minimizes the analysis redundancy, since the method summary is computed once and used at all the call sites. The

summary-based approach promises more scalability, since it only computes smaller, local invariants. The summary-based analysis processes the methods from a program in *bottom-up* order of the call graph. Recursion requires simultaneous handling of methods in the same strongly connected component of the program call graph. While the modular approach promises more scalability than a global approach, it poses an important challenge. Analyzing a method body without the information provided by the call context requires a more complex abstract domain. For example, reasoning about the code of `initarray` method when the array `a` and the variable `n` have symbolic (rather than fixed) values is more challenging.

There are many related works that use modularity in static analysis. We give a brief account of these related works, starting with the general approaches to interprocedural analysis proposed by Cousot-Cousot [38] and Sharir-Pnueli [141]. Reps et al [132] showed how to do precise interprocedural analysis with finite abstract domains and distributive transfer functions. Modular aliasing analyses have also been proposed [19, 21] and recently Yorsh et al [152] showed how to combine finite typestate with aliasing analysis. Closer to our focus on numerical properties, Müller-Olm et al [109] and Gulwani-Tiwari [69] presented precise interprocedural analyses with linear equalities. Precise analysis can be obtained only for programs that have non-deterministic conditionals; other classes of conditionals make the precise problem undecidable even for a finite height domain like that of linear equalities. Predicate abstraction domain is another instance of a finite domain that has been extended to a polymorphic [7] or compositional setting [85]. Other numerical abstract domains employed in interprocedural analysis are linear congruences [110] and polyhedra analysis [140, 63]. Seidl et al [140] have focused on analysis with a restricted form of polyhedra (simplices) that can be implemented efficiently, while Gopan-Reps [63] recently proposed an analysis for summarizing low-level libraries.

Our proposal shares the modularity principle with all these works, but we apply it in a more fine-grained abstract domain, that of disjunctive polyhedra. Furthermore, most summary-based analyses have focused on computing an over-approximation for each method body, including in this category summary-based alias analyses [19, 21]. On the other hand, in addition to over-approximations we also derive under-approximations modularly in the form of preconditions *sufficient* to guarantee the safety

of the method body. There has been considerably less success in deriving modular under-approximations. Recent efforts independent of our work include [108] in the context of modular assertion checking and [152] proposing a framework for generating procedure summaries restricted to finite-height abstract domains. In contrast to sufficient preconditions, another class of analyses generates preconditions *necessary* (but not sufficient) for ensuring the safety of a method [15, 123].

For the example from Figure 1.4, our modular analysis generates a summary for the `initarray` method that includes a precondition ($\texttt{n} \leq \texttt{len}(\texttt{a})$) sufficient for guaranteeing the safety of the assertion at label $\texttt{l}_2$. Compared to the global analysis requirement of re-analyzing the method's body for each call context, with a modular analysis it is only needed to check the precondition ($\texttt{n} \leq \texttt{len}(\texttt{a})$) at each call site.

Our proposal benefits from the symbiotic relationship between a modular analyzer and a precise disjunctive domain. Firstly, a modular analyzer requires a precise abstract domain to reason about the symbolic method inputs. On the other hand, a modular analyzer has a local scope and therefore favors more complex invariants than those that are usually involved in global analyzers. Based on this observation, we designed a modular analyzer that combines forward and backward analyses and can be practical and precise. The forward analysis aims to infer method postconditions, but it also discovers invariants that are useful in the backward derivation of sufficient preconditions.

### 1.2.3 Dual Static Analysis

Static analysis uses abstraction on program states to prove program safety. Due to approximation, static analysis may report *false positives* that are possible bugs that do not exist in practice. High incidents of false positives can make static analysis tools impractical to use for finding and eliminating bugs. Manual inspection of alarms (possible bugs) can be a very time-consuming process and may take several days even for simple alarms in a large program [136].

Since finding all errors and establishing program safety has proven difficult, from a pragmatic angle researchers have been concerned with methods to find some of the errors from a program. Traditionally, program testing [89] has been used for detecting faulty programs. More recently, model checking or concrete state space exploration

[59, 111] has been successfully applied to detect the presence of program errors. As this systematic testing may not terminate in a reasonable amount of time, a limit is set in practice on the number of paths that are covered. On the whole, the bugs that are discovered are sound (they are guaranteed to occur in some concrete execution), but some bugs may remain undiscovered.

Synergistic approaches for both proving safety and finding bugs usually rely on a combination of over and under approximation. Model checking based on abstraction refinement is often referred as CEGAR (counterexample-guided abstraction refinement) [29] and tools like SLAM [5] or BLAST [80] are based on this paradigm. In a first step, SLAM and BLAST perform a forward-directed overapproximating search for possible bugs. If no bugs are found, then the safety of the program has been proven. Otherwise, starting with a possible bug, a counterexample trace is analyzed backward via symbolic reasoning in order to derive its weakest liberal precondition. If the counterexample is shown to be feasible, then a true bug is reported. If the counterexample is shown to be infeasible or spurious, the abstraction is refined and the search process is iterated. Due to the infiniteness of the concrete state space, the process may not converge while continuously refining the abstract domain. Only true bugs are reported provided that the backward analysis is complete, for example in the context of the $\texttt{ACTL}^*$ fragment of Computational Tree Logic [29] or of the theory of linear arithmetic with uninterpreted functions [80]. However, in general the backward analysis is incomplete: in the presence of division operators represented as uninterpreted functions the analysis may report false positives.

To support the automatic analysis of false positives arisen from static analysis, we propose a *dual static analysis* that is designed to track concurrently two over-approximations: the success and the failure outcomes. The overlap between the two outcomes signifies imprecision in analysis and can be used to guide abstraction refinement. More interestingly, due to the concurrent computation of outcomes, we can identify two significant input conditions: a *never-bug* condition that implies safety for inputs that satisfy it and a *must-bug* condition that characterizes inputs that lead to true errors in the execution of the program. As a result, our analysis can identify a part of the alarms as being true errors and reduces the manual effort of analyzing alarms to a smaller group

of *may-bugs*.

## 1.3 Thesis Overview

The aim of this thesis is to investigate techniques that would broaden the class of applications to which modular static analyses approaches are applicable.

Firstly, to handle the challenges of disjunctive analyses, we introduce in Chapter 3 the notion of *affinity* to characterize how closely related are two disjuncts. Finding related elements in the conjunctive (base) domain allows the formulation of precise hull and widening operators lifted to the disjunctive (powerset extension of the) base domain. We have implemented a static analyzer based on the disjunctive polyhedral analysis where the relational domain and the proposed operators can progressively enhance precision at a reasonable cost. This chapter is based on a paper that was first presented at the 11th Annual Asian Computing Science Conference - ASIAN 2006: Corneliu Popeea and Wei-Ngan Chin - "Inferring Disjunctive Postconditions" [123].

Secondly, our proposal from Chapter 4 exploits the relationship between modular analysis and a precise disjunctive domain. We designed a modular analyzer that combines forward and backward analyses and can be practical and precise. The forward analysis aims to infer method postconditions, but it also discovers invariants that are useful in the backward derivation of sufficient preconditions. To increase the efficiency of the analysis, we designed a technique to strengthen preconditions and trade precision for speed. Rather than deriving one program precondition for proving program safety, our analysis derives individual preconditions for each check and goes one step further by performing aggressive optimizations of checks. We formalise our technique as a dependent type system that uses type annotations for communicating information between the inference and the optimization phases. This work was presented at the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation - PEPM 2008: Corneliu Popeea, Dana N. Xu, Wei-Ngan Chin - "A Practical and Precise Inference and Specializer for Array Bound Checks Elimination" [127].

Finally, we aim to support either a proof of the absence of bugs in the case of a valid program or bug finding in the case of a faulty program. In Chapter 5, we propose a *dual static analysis* that is designed to track concurrently two over-approximations:

the success and the failure outcomes. The overlap between the two outcomes signifies imprecision in analysis and can be used to guide abstraction refinement. More interestingly, due to the concurrent computation of outcomes, we can identify two significant input conditions: a *never-bug* condition that implies safety for inputs that satisfy it and a *must-bug* condition that characterizes inputs that lead to true errors in the execution of the program. As a result, our analysis can identify a part of the alarms as being true errors and reduces the manual effort of analyzing alarms to a smaller group of *may-bugs*. This chapter is an extension of the paper: Corneliu Popeea, Wei-Ngan Chin - "Dual Static Analysis" [125].

Before presenting the details of these extensions of static analysis, the next chapter presents the Imp language, a subset of the C language. By focusing on a smaller language, this thesis will introduce the key technical difficulties in modular static analysis. The experimental results that will be described subsequently handle a larger subset of the C language.

# CHAPTER II

# CORE LANGUAGE, SEMANTICS AND ABSTRACTION

The purpose of this chapter is to introduce the core language that we use for analysis, called Imp, and give some preliminary notions needed for the further technical developments. We will first introduce the syntax of Imp in Section 2.1 and its concrete semantics in Section 2.2. In Section 2.3, we will describe our analysis framework and show how to compute an abstract semantics for the Imp language based on a traditional conjunctive domain. We will explain the main features of our analysis framework: *forward reasoning rules* used to extract *constraint abstractions* and how to compute an approximate solution of these constraints using a *fixed point process*.

## 2.1 Syntax of Imp Language

In this section we will describe a small language, called Imp, that is a first-order sequential imperative language. This language retains only few constructs from the better-known C language and its purpose is to make program analyses easier to formulate and prove soundness. Despite its simplicity, the language can be used to encode recursive methods over data-types like integers. Therefore, the language is Turing-complete and interesting properties over programs written in Imp are undecidable in general [134]. This fact makes static program analyses both interesting and challenging.

The syntax of our Imp language is shown in Figure 2.1. A program $P$ written in this language consists of a set of methods, either user-defined or primitive methods. All methods have a return type and a list of parameters; each parameter has an optional `ref` keyword, a type and a name. The `ref` keyword indicates the parameter passing mechanism: when `ref` appears in the parameter declaration, then the parameter is meant to be passed-by-reference (any change in its value is visible to the caller); when the `ref` keyword is missing, the parameter is passed-by-value and any change to its value in the callee is not reflected to the caller.

| | | |
|---|---|---|
| $P$ | ::= $prim^*$ $meth^*$ | (program) |
| $prim$ | ::= $t$ $mn$ (([`ref`] $t$ $v$)$^*$) `where` $\Phi$ | (primitive method) |
| $meth$ | ::= $t$ $mn$ (([`ref`] $t$ $v$)$^*$) $\{e\}$ | (user-defined method) |
| $t$ | ::= `bool` \| `int` \| `float` \| `void` \| `t[]` | (type) |
| $e$ | ::= $v$ \| $k$ \| $v$:=$e$ \| $e_1; e_2$ \| $l : mn(v^*)$ \| $t\ v\ ;\ e$ | (expression) |
| | \| `if` $v$ `then` $e_1$ `else` $e_2$ \| $l$ : `error` | |
| $k$ | ::= `true` \| `false` \| $k^{\texttt{int}}$ \| $k^{\texttt{float}}$ \| () | (constant) |

**Figure 2.1:** Syntax of the language Imp

Types represented by $t$ can be either basic types or array type. User-defined method declarations include a method body represented by an expression $e$. The Imp language is expression-oriented and uses a normalised form: only variables are allowed as arguments to a method call or a conditional test. This normalization can be done with the help of a simple pre-processor and for brevity we may show examples in a form that is not normalized. Expression forms include assignment, sequence of expressions, method call, local variable declaration, conditional and an `error` construct.

The `error` construct allows the program to terminate in case of an error situation. Statically computing the conditions under which the `error` constructs are guaranteed to be unreachable reduces to proving program safety and is the focus of this thesis. The `error` construct is prefixed by a label $l$ that uniquely identifies it. A more conventional way to check that a condition holds is to use `assert e`. This form is equivalent to a conditional expression in our language: `if e then` () `else error`.

An important feature of our core language is the presence of primitive method declarations that lack a method body, but are instead given a symbolic description (summary) $\Phi$. Primitive methods can be used to encode the following:

- Various operators for integer values (`plus`,`minus`, `multiply`,`divide`), for bool values (`or`, `and`, `not`) and for comparison operators (`lt` or $<$, `lte` or $\leq$).

- Potentially unsafe operators that handle array values. The primitive `newarr` returns an array value, `len` returns the length of its array parameter. The primitive `sub` returns an array element from a specified index, while `assign` updates the specified array element with a given value. For example, an array access `a`[`i`] is viewed as `sub`(`a`, `i`), while an array update `a`[`i`]:=`v` is converted to the primitive

call `assign`(`a`, `i`, `v`).

- Calls to (external) library methods for which the analysis does not have access to the code that implements the library.

In most of the cases, the description $\Phi$ of the primitive operators can be automatically derived. This formula may include a safety precondition (for example, bound checks for array operations), or simply represent the input-output relation (for primitive numerical operations like `plus` or `multiply`). The syntax of the formulae depends on the underlying constraint solver. Throughout this thesis we will use various fragments of Presburger arithmetic [128] and introduce their syntax in Section 2.3.

### 2.1.1 Other Language Features

Other core languages have been proposed for analysing C-like languages and we enumerate some of them here: Ckit [8], CIL [114], CoreC [151], Core-Expressions [119]. The goal of these projects is to capture a large subset of the C language. Our goal is not so ambitious and we only discuss informally how some language features can be analyzed in the context of Imp. More details on supporting these and other features can be found in our technical report on a core language named $\mu$CIL [124].

**Loops** : Both `while` and `for` loops from C programs can be converted to tail-recursive methods, where variables used inside loops are promoted as method parameters. Variables used inside a loop may change value and the change has to be reflected across the loop body. For this purpose, the method parameters are passed by reference, using the `ref` keyword from Imp. Our translation always ensures that the `ref` parameters are all different and non-aliased. Constructs that interrupt the normal control-flow in a loop, like `continue`, `break` and `return`, can also be handled by our translation to tail-recursive methods. To illustrate the loop conversion, we use a simple example that assigns 0 to those elements in the array $a$ from the range $i$ to 1 (see Figure 2.2).

**Global variables** : Initialization of global variables can be moved to a newly-created function `globinit`. Global variables can then be made available to each method as parameters.

| | |
|---|---|
| `while (i > 0) {`<br>`  assign(a, i, 0);`<br>`  i:=i−1 }` | `void g(int[] a, ref int i)`<br>`{ if (i≤0) then ()`<br>`  else { assign(a, i, 0); i:=i−1; g(a, i) } }` |

**Figure 2.2:** Example of loop translation

**Floating-point values** : Even though float values are supported in Imp, this does not imply that our program analyses will handle them precisely. The constraint language that we use is based on the integer domain and does not capture values of float variables. Specialized techniques for handling rounding errors in floating-point computations have been proposed for static analysis in [102, 105].

**Integer overflow** : C programs do not manipulate perfect integers, but bounded-domain machine integers. Solutions to handle integer overflow detection have been proposed elsewhere via specific abstract domains [106, Chapter 7].

**Structure values and aliasing** : We have made some explorations on how to analyse structure values in the context of verification of functional and object-based programs [122, 25]. However, the subsequent chapters of this thesis do not discuss these language features.

Imp language is meant to facilitate program analysis of first-order sequential imperative programs. Therefore, it does not support features like higher-order functions or concurrency primitives.

## 2.2 Concrete (Operational) Semantics

In this section, we define a small-step operational semantics for our core imperative language. Our machine configuration is being represented by $\langle s, e\rangle$ where $s$ denotes the current stack and $e$ denotes the current program code.

$$\begin{aligned} Stack: &\quad s \in Stack = Var \rightarrow_{fin} Value \\ Values: &\quad \delta \in Value = Float \uplus Int \uplus Bool \uplus Void \end{aligned}$$

Each reduction step can then be formalised as a small-step transition of the following form: $\langle s, e\rangle \hookrightarrow \langle s_1, e_1\rangle$. The rules are standard and presented in Figure 2.3. As an

$$\frac{}{\langle s, v\rangle \hookrightarrow \langle s, s(v)\rangle}\ [\textbf{D–VAR}] \qquad \frac{\delta = \mathit{default}(t) \quad \mathit{fresh}\ x \quad \rho = [v \mapsto x]}{\langle s, t\ v; e\rangle \hookrightarrow \langle [x \mapsto \delta] + s, \texttt{ret}(x, \rho e)\rangle}\ [\textbf{D–BLK}] \qquad \frac{}{\langle s, l : \texttt{error}\rangle \hookrightarrow \langle s, \bot\rangle}\ [\textbf{D–ERROR}]$$

$$\frac{mn \in \mathit{Primitives} \quad \langle s', \delta\rangle = \mathit{exec}\langle s, l : mn(v_1, ..v_n)\rangle}{\langle s, l : mn(v_1, ..v_n)\rangle \hookrightarrow \langle s', \delta\rangle}\ [\textbf{D–PRIM}]$$

$$\frac{t_0\ mn((\texttt{ref}\ t_i\ w_i)_{i=1}^{m-1}, (t_i\ w_i)_{i=m}^{n})\{e\} \quad s' = [w_i \mapsto s(v_i)]_{i=m}^{n} + s}{\langle s, l : mn(v_1, ..v_n)\rangle \hookrightarrow \langle s', \texttt{ret}(\{w_i\}_{i=m}^{n}, [v_i/w_i]_{i=1}^{m-1} e)\rangle}\ [\textbf{D–CALL}]$$

$$\frac{s(v) = \texttt{true}}{\langle s, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rangle \hookrightarrow \langle s, e_1\rangle}\ [\textbf{D–IF–1}] \qquad \frac{s(v) = \texttt{false}}{\langle s, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rangle \hookrightarrow \langle s, e_2\rangle}\ [\textbf{D–IF–2}]$$

$$\frac{}{\langle s, v := \delta\rangle \hookrightarrow \langle s[v \mapsto \delta], ()\rangle}\ [\textbf{D–ASSIGN–1}] \qquad \frac{}{\langle s, \delta; e_2\rangle \hookrightarrow \langle s, e_2\rangle}\ [\textbf{D–SEQ–1}] \qquad \frac{}{\langle s, \texttt{ret}(v^*, \delta)\rangle \hookrightarrow \langle s - \{v^*\}, \delta\rangle}\ [\textbf{D–RET–1}]$$

$$\frac{\langle s, e\rangle \hookrightarrow \langle s', e'\rangle}{\langle s, v := e\rangle \hookrightarrow \langle s', v := e'\rangle}\ [\textbf{D–ASSIGN–2}] \qquad \frac{\langle s, e_1\rangle \hookrightarrow \langle s', e_1'\rangle}{\langle s, e_1; e_2\rangle \hookrightarrow \langle s', e_1'; e_2\rangle}\ [\textbf{D–SEQ–2}] \qquad \frac{\langle s, e\rangle \hookrightarrow \langle s', e'\rangle}{\langle s, \texttt{ret}(v^*, e)\rangle \hookrightarrow \langle s', \texttt{ret}(v^*, e')\rangle}\ [\textbf{D–RET–2}]$$

$$\frac{\langle s, e\rangle \hookrightarrow \langle s', \bot\rangle}{\langle s, v := e\rangle \hookrightarrow \langle s', \bot\rangle}\ [\textbf{D–ASSIGN–3}] \qquad \frac{\langle s, e_1\rangle \hookrightarrow \langle s', \bot\rangle}{\langle s, e_1; e_2\rangle \hookrightarrow \langle s', \bot\rangle}\ [\textbf{D–SEQ–3}] \qquad \frac{\langle s, e\rangle \hookrightarrow \langle s', \bot\rangle}{\langle s, \texttt{ret}(v^*, e)\rangle \hookrightarrow \langle s', \bot\rangle}\ [\textbf{D–RET–3}]$$

**Figure 2.3:** Operational semantics

example, a conditional expression is evaluated depending on the test value. If the value is a boolean constant, then either the rule [**D–IF–1**] (if the constant is `true`) or the rule [**D–IF–2**] applies. If the value is not of boolean type, then in principle the execution would be stuck with a type error. We rely on the fact that the source program is well-typed and such errors cannot occur. For any given complete execution, we expect one of three possible outcomes: $\langle s, e\rangle \hookrightarrow^* \langle s_1, \delta\rangle$ for success, $\langle s, e\rangle \hookrightarrow^* \langle s_1, \bot\rangle$ for failure, or $\langle s, e\rangle \not\hookrightarrow^*$ for non-termination.

To ease the formulation of the correctness relation between the concrete and the abstract semantics, we extend the source language with a construct to represent the

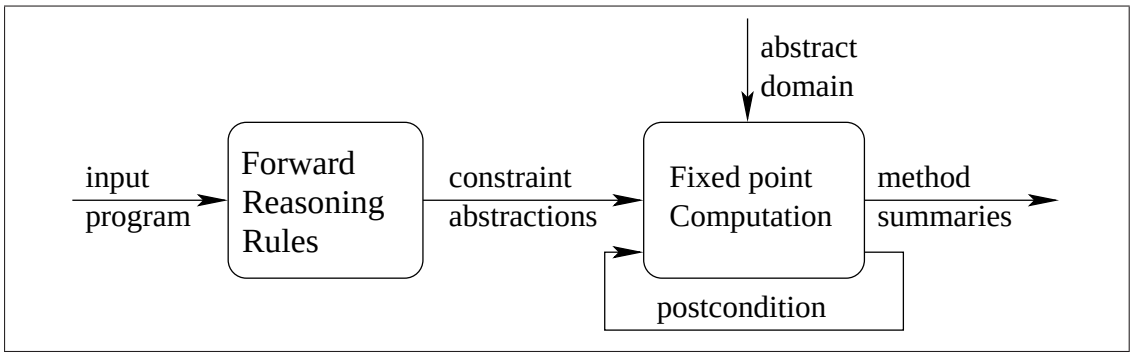



**Figure 2.4:** Overview of analysis framework

intermediate result of a method call: the evaluation of the expression $\texttt{ret}(v^*, e)$ proceeds first with the method's body $e$ (rule [**D–RET–2**]) and, after its reduction to a value, the parameters passed by value $v^*$ are removed from the current stack (rule [**D–RET–1**]). If the evaluation of the body reaches an error, then the rule [**D–RET–3**] throws the error to the caller.

## 2.3 Abstract Semantics

The operational semantics introduced in the previous section is not suitable for automated static analysis due to its undecidability. We will use the abstract interpretation framework [37] to compute an approximation of this semantics, a sound and decidable abstract semantics for Imp programs. This section presents an overview of our approach and an introduction to our modular static analysis framework based on forward reasoning rules, constraint abstractions and fixed point approximation.

### 2.3.1 Overview

An overview of our analysis framework is shown in Figure 2.4. The input to our analysis is an Imp program, consisting of a group of methods that are first analysed to obtain a call graph. A call graph contains one directed edge from each caller method to the corresponding callee method. Recursive methods are represented as a strongly connected component (SCC) in the graph.

Our analysis traverses the call graph in reverse topological order (bottom up) and each method (or group of methods) is analysed assuming unknown initial values. Specifically, each method is passed to the forward reasoning process and an intermediate constraint representation is derived in the form of a *constraint abstraction*. If the method

is non-recursive, then a method summary (that includes a postcondition and a precondition) can be immediately derived. If the method is recursive, then the constraint abstraction is passed on to a fixed point approximation process parameterized by an abstract domain. In the case of mutually-recursive methods, the fixed point process is done simultaneously for the corresponding constraint abstractions.

Most summary-based analyses compute an over-approximation of the set of reachable states: summary-based alias analyses [19, 75] or analyses computing method postconditions [38, 123]. For software verification purposes, it is required to compute also an under-approximation for method precondition. In the case of tail recursive methods (including loops), the precondition can be derived from the over-approximation phase without an additional fixed point computation. However, more general recursion patterns complicate the fixed point process. To support general recursion, our analysis uses a feedback loop since the postcondition is required for a second fixed point computation. After a summary is derived for the current method, the analysis proceeds with the next method in the call graph order. An inspirational line of works for our developments is the generic framework for inference of size relations in functional programs proposed by Chin, Khoo et al [23, 24, 25].

### 2.3.2 Forward Reasoning Rules

The goal of the forward reasoning process is to collect from each method a constraint abstraction that is amenable to fixed point computation. This process is built around a static judgement with roots in Hoare logic [82, 1]. Given a formula $\phi_1$ describing the current state and an expression $e$, the judgement derives a formula $\phi_2$ describing the state after the expression $e$ is evaluated:

$$\vdash \{\phi_1\}\, e\, \{\phi_2\}$$

The computation of $\phi_2$ assuming a given formula $\phi_1$ is what gives the forward character to the reasoning process. Next, we will explain two aspects concerning the formulae $\phi_1$ and $\phi_2$: their syntactic form and their semantic meaning.

The syntactic form of $\phi$ is based on the first order theory of linear arithmetic (Presburger arithmetic) with support for recursive constraint abstractions (see Figure 2.5).

The choice of the domain is influenced by the numerical properties that we want to capture. A set of recursive constraint abstractions is denoted by $Q$ and amenable to fixed point computation as explained in more detail later. The existential quantifier $\exists v \cdot \phi$ is used for eliminating intermediate variables when computing postconditions, while the universal quantifier $\forall v \cdot \phi$ is used for eliminating intermediate variables when computing preconditions (see Chapter 4). The operator $\neg\phi$ is used in Chapter 5 to derive the complement of a formula $\phi$. An equality constraint $(a_1v_1+\cdots+a_nv_n = a)$ can be represented as a conjunction of constraints: $(a_1v_1+\cdots+a_nv_n \leq a \wedge a_1v_1+\cdots+a_nv_n \geq a)$. A strict inequality $(a_1v_1+\cdots+a_nv_n < a)$ can also be represented using an axiom from integer arithmetic: $(a_1v_1+\cdots+a_nv_n+1 \leq a)$. We also use the syntactic shorthands $\texttt{true} = s \wedge \neg s$ and $\texttt{false} = s \vee \neg s$ for some predicate $s$.

| | | |
|---|---|---|
| $\phi$ | $::= s \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid$ $\exists v \cdot \phi \mid \forall v \cdot \phi \mid \neg\phi \mid q\langle v^* \rangle$ | (formula) |
| $s$ | $::= a_1v_1+\cdots+a_nv_n \leq a$ | (linear inequality) |
| $Q$ | $::= \{(\ q\langle v^* \rangle = \phi)^*\}$ | (constraint abstraction) |
| $q, v$ | $\in VAR$ | (identifier) |
| $a$ | $\in Z$ | (integer constant) |

**Figure 2.5:** Syntax of formulae and constraint abstractions

For the non-recursive integer constraints, we make use of a complete decision procedure for Presburger arithmetic implemented in the Omega Test [129]. The Omega Test is an extension of Fourier-Motzkin variable elimination to integer arithmetic. Despite its doubly-exponential worst case complexity, the Omega Test has been shown to be efficient in practice [129, 131].

The syntax of $\phi$ formulae gives us some flexibility in choosing the abstract domain used in the fixed point process. For fixed point computation, we have experimented with various subdomains of the theory of linear arithmetic like the conjunctive domains of octagons [104] and polyhedra[43] and the disjunctive domain of polyhedra [123].

The semantic meaning of a formula $\phi$ is an abstraction for a set of traces [35, 30, 135]. Such a transition formula ranges over two sets of logical variables: unprimed variables represent the values of program variables at the beginning of the current method, named prestate; primed variables represent the values of program variables at some program

point in the method (poststate). A simple example of an assignment expression follows:

$$\vdash \{x'{=}x \wedge y'{=}y\}\, x = y{+}1\, \{x'{=}y{+}1 \wedge y'{=}y\}$$

The input formula does not assume anything about the initial values of the variables $x$ and $y$. Additionally, the values of the variables at the beginning of a method are unchanged: $(x'{=}x \wedge y'{=}y)$. After the assignment expression is abstractly evaluated, the postcondition formula describes the state after the assignment relative to the state at the beginning of the method. The trace semantics forms the basis of our modular analysis. It allows individual reasoning about each method and a subsequent composition between the method abstraction and its call contexts. For example, if the call context would initialize $x$ and $y$ to the values 10 and respectively 20, then the state after the assignment can be resolved using trace composition to the formula $(x'{=}21 \wedge y'{=}20)$.

The most important rule in our forward reasoning process is the one that handles a method declaration and computes the method postcondition $\phi_{po}$:

$$\frac{\begin{array}{c}[\textbf{METH}] \\ W{=}\{v_i\}_{i=1}^{n} \quad \vdash \{\mathit{nochange}(W)\}\, e\, \{\phi\} \\ X{=}\{v_1, .., v_n, \mathbf{res}, v'_1, .., v'_{m-1}\} \quad V{=}\{v'_i\}_{i=m}^{n} \quad Q{=}\{mn\langle X\rangle = \exists V \cdot \phi\} \\ \phi_{po} = \mathit{fix}(Q)\end{array}}{\vdash t_0\ mn((\texttt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \{e\} \Rightarrow \phi_{po}}$$

The first line of the rule uses the expression judgement to traverse the method body $e$, where $W$ is a set of logical variables representing the inputs to the method $mn$. Using the (possibly recursive) postcondition $\phi$, a constraint abstraction $Q$ is constructed for the current method: $mn\langle X\rangle = \exists V \cdot \phi$. It has as arguments the variables from $X$, both inputs and outputs of the method. Updates to the parameters that are passed by value $V$ should not be visible in the postcondition and thus are existentially quantified. A non-recursive constraint abstraction $mn\langle X\rangle = \phi$ can be seen as a function which when given some variables $Y$ applies the substitution $[X \rightarrow Y]$ on the constraint $\phi$ [73]. In the case when the constraint abstraction is recursive, the third line of the [**METH**] rule invokes a fixed point process. The fixed point process computes an approximation to the least fixed point of the constraint abstraction function. This approximation represents the postcondition $\phi_{po}$ of the original method `mn`.

**Example 2.1 :** We illustrate the process of collecting a constraint abstraction using the simplest possible recursive method shown in Figure 2.6. The method `mn` is given an integer argument `x` that is decremented to `0` in case it is positive, otherwise being left unchanged. For exposition purposes, the example uses a recursive method and a parameter that is passed by reference.

```
void mn (ref int x) {
   if (x > 0) then {
    x := x − 1;
    mn(x);
   } else { () }
}
```

**Figure 2.6:** Simple recursive example

The intended semantics of the method `mn` is represented as a constraint in terms of the initial value of the parameter `x` and the latest value of the parameter `x′` as follows: $(x{\leq}0 \wedge x'{=}x) \vee (x{>}0 \wedge x'{=}0)$. The first step in computing this semantics is to collect a constraint abstraction which is close to the syntactic definition of the method `mn`:

$$mn(x, x') = (x{\leq}0 \wedge x'{=}x) \ \vee \ (x{>}0 \wedge \exists x_1 \cdot (x_1{=}x{-}1 \wedge mn(x_1, x'))) \qquad (2.1)$$

This view of the method `mn` has two parameters, namely the logical variables `x` and `x′`.

Next, we will show how to compute an approximation for the fixed point of the constraint abstraction.

### 2.3.3 Fixed Point Approximation

We briefly review the method based on Kleene's fixed point iteration and its application to the polyhedron abstract domain [43]. Let $(\mathcal{L}, \leq)$ be a complete lattice, and denote by $(\mathcal{P}, \Rightarrow)$ the lattice of polyhedra. We write $\bot$ for its least element (in $\mathcal{P}$, the empty polyhedron or its representation, the formula `false`), and $\top$ for its greatest element (in $\mathcal{P}$, the entire n-dimensional space or its representation, the formula `true`). The least upper bound and the greatest lower bound operations in the lattice of polyhedra are, respectively, the convex polyhedral hull and the set intersection, the first being denoted by $\oplus$. A function $f$ that is a self-map of a complete lattice is *monotone* if $x \leq y$ implies $f(x) \leq f(y)$. In particular, the constraint abstraction functions derived

by our analysis are monotone self-maps of the polyhedra lattice.

The *least fixed point* of a monotone function $f$ can be obtained by computing the ascending chain $f_0 = \bot$, $f_{n+1} = f(f_n)$, with $n \geq 0$. If the chain becomes stationary, i.e., if $f_m = f_{m+1}$ for some $m$, then $f_m$ is the least fixed point of $f$. In the case of a lattice infinite in height (as the lattice of polyhedra), an ascending chain may be infinite, and a widening operator must be used to ensure convergence. A widening operator $\nabla$ is a binary operator to ensure that the iteration sequence $f_0 = \bot$, $f_{k+1} = f(f_k)$ followed by $f_{n+1} = f_n \nabla f(f_n)$, with $n > k$, converges. In this case, the limit of the sequence is known as a *post fixed point* of $f$. A *post fixed point* is a sound approximation of the least fixed point, and the criterion to verify that $x$ is a post fixed point for $f$ is that $x \geq f(x)$. For the polyhedron domain, the standard widening operator was introduced in [43]. Intuitively, the result of the widening $\phi_1 \nabla \phi_2$ is obtained by removing from $\phi_1$ those conjuncts that are not satisfied by the next iteration $\phi_2$.

The post fixed point result represents the method postcondition, a conservative representation of the method transfer function. As mentioned before, the fixed point process is parametric in the abstract domain. Consequently, the method postcondition approximates more closely the concrete semantics when the abstract domain is more fine-grained.

For fixed point computation, the constraint abstraction can be viewed as a function that takes an abstract element in some lattice $\mathcal{L}$ and returns another abstract element. Compared to the syntactic view from Equation 2.1, we now use a more semantic view for the constraint abstraction function:

$$mn(\phi) = (x \leq 0 \wedge x' = x) \ \vee \ (x > 0 \wedge \exists x_1 \cdot (x_1 = x - 1 \wedge \phi[x \rightarrow x_1, x' \rightarrow x'])) \tag{2.2}$$

The argument $\phi$ represents an abstraction of the traces from the beginning to the end of the method (a constraint in terms of $x$ and $x'$). Using the least element of the lattice `false`($\bot$), the abstract element $mn_1$ is computed as $mn$(`false`), considering that the recursive branch is never executed. The next iteration computes $mn_2$ as $mn(mn_1)$ and the iteration process continues further. The element $mn_1$ represents the abstraction of traces through the method $mn$ for 0 recursive calls. The result of the next iteration $mn_2$ represents the abstraction of traces through the method $mn$ for 1 recursive call. At the limit, the least fixed point of the constraint abstraction function $mn$ represents

the abstraction of traces through the method *mn* for all possible recursive calls. As might be expected, the least fixed point may or may not be computable, depending on the abstract domain that is used. For the conjunctive polyhedra domain, the computation yields the following post fixed point: $(x'{\leq}x \wedge x'{\leq}0)$. The disjunctive abstract domain that will be introduced in Chapter 3 computes a more precise post fixed point: $(x{\leq}0 \wedge x'{=}x) \vee (x{>}0 \wedge x'{=}0)$.

We also note that the constraint abstraction body can be written using the *least fixed point operator* from a fixpoint calculus [53, 142]. In such a formalism, the method postcondition can be denoted as follows:

$$\phi_{po} = \mu X \cdot ((x{\leq}0 \wedge x'{=}x) \ \vee \ (x{>}0 \wedge \exists x_1 \cdot (x_1{=}x{-}1 \wedge X[x{\rightarrow}x_1, x'{\rightarrow}x'])))) \qquad (2.3)$$

Finally, we observe that the recursive method `mn` is tail-recursive and therefore it can be written simply as a loop. One of the advantages of our approach based on trace semantics and constraint abstraction formalism is that it analyzes more general patterns of recursion with the same algorithm used for tail-recursive methods (including loops).

### 2.3.4 Method Summary

Other than the derivation of postcondition, method summaries used in software verification require additional computations. We highlight three kinds of method summaries. Algorithms towards their inference will be formalized in subsequent chapters.

- The inference of a method produces a summary composed of a postcondition and a precondition *necessary for safety*. The precondition necessary for safety can be trivially derived from the postcondition. Despite the simplicity of this precondition inference, this approach can help eliminate a considerable number of checks in our experiments. However, the precondition derived in this manner requires a separate verification stage to ensure its soundness. We will formalize this approach in Chapter 3.

- The inference of a method produces a summary composed of a postcondition and a precondition *sufficient for safety*. We use an additional fixed point computation, in order to derive preconditions sufficient for safety (for the case of general recursion). Individual preconditions are derived for each check, which makes them useful not

only for proving safety, but also for aggressive optimization of the checks. The computation of such method summaries will be described in Chapter 4.

- The inference of a method produces a summary composed of a postcondition and a precondition *necessary for error*. This derivation is useful for proving safety and also for finding true errors that are *guaranteed* to occur during program execution (modulo termination). The same technique has potential in guiding abstraction refinement and for identifying preconditions *sufficient for non-termination*. These aspects will be further elaborated in Chapter 5.

# CHAPTER III

# DISJUNCTIVE FIXED-POINT ANALYSIS

Polyhedral analysis [43] is an abstract interpretation used for automatic discovery of invariant linear inequalities among numerical variables of a program. Convexity of this abstract domain allows efficient analysis but also loses precision via convex-hull and widening operators. To selectively recover the loss of precision, sets of polyhedra (disjunctive elements) may be used to capture more precise invariants. However a balance must be struck between precision and cost.

In this chapter, we introduce the notion of affinity to characterize how closely related is a pair of polyhedra. Finding related elements in the polyhedron (base) domain allows the formulation of precise hull and widening operators lifted to the disjunctive (powerset extension of the) polyhedron domain. We have implemented a modular static analyzer based on the disjunctive polyhedral analysis where the relational domain and the proposed operators can progressively enhance precision at a reasonable cost.

## 3.1 Background

Abstract interpretation [37, 39] is a technique for approximating a basic analysis, with a refined analysis that sacrifices precision for speed. Abstract interpretation relates the two analyses using a Galois connection between the two corresponding property lattices. The framework of abstract interpretation has been used to automatically discover program invariants. For example, numerical invariants can be discovered by using numerical abstract domains like the interval domain [36] or the polyhedron domain [43]. Such convex domains are efficient and their elements represent conjunctions of linear inequality constraints.

Abstract domains can be designed incrementally based on other abstract domains. The powerset extension of an abstract domain [39, 58] refines the abstract domain by adding elements that allow disjunctions to be represented precisely. Unfortunately, analyses using powerset domains can be exponentially more expensive compared to analyses

on the base domain. One well-known approach to control the number of disjuncts during analysis is to use a powerset domain where the number of disjuncts is syntactically bounded. In this setting, the challenge is to find appropriate disjuncts that can be merged without (evident) losses in precision. Our work was done at the same time with (and independently from) a related technique for disjunctive static analysis that has been proposed and implemented in [138]. Their analysis is formulated for a generic numerical domain and an heuristic function based on the Hausdorff distance is used to merge related disjuncts. Besides combining related disjuncts, recent interest has been shown in tackling another difficulty in disjunctive analysis, that of defining a convergent widening operator [2, 68].

In this chapter, we develop a novel technique for *selective hulling* to obtain precise fixed-points via disjunctive inference. Our framework uses a fixed-point algorithm guided by an affinity measure to find and combine disjuncts that are related. We also develop a precise widening operator on the powerset domain by using a similar affinity measure. We have built a prototype system to show the utility of the inferred postconditions and the potential for tradeoff between precision and analysis cost.

This chapter is organized as follows: an overview of our method with a running example is presented in Section 3.2. The proposed disjunctive abstract domain is detailed in Section 3.3 by introducing an affinity measure, a selective hull and a widening operator. Section 3.4 introduces a set of reasoning rules that collect a (possibly recursive) constraint abstraction from each method/loop to be analyzed. Those recursive constraint abstractions are the subject of disjunctive fixed-point analysis. Section 3.5 shows how boolean constraints can be handled in our framework. Our experimental results and interesting examples are presented in Section 3.6 and Section 3.7. Section 3.8 argues the correctness of our analysis, while Section 3.9 presents related work.

## 3.2 Overview of Fixed-Point Analysis

To provide an overview of our method, we will consider the following example.

```
x:=0;upd:=False;
while (x < N) do {
  if (randBool()) then {
```

```
    l:=x;upd:=True
  } else { () };
  x:=x+1 }
```

This program computes the index $l$ of a specific element in an array of size $N$. The array content has been abstracted out and only the updates to the index variables $l$ and $x$ have been retained. The call to the method *randBool* abstracts whether the current element indexed by $x$ is found to satisfy the search criterion. Whenever the criterion is satisfied, the index variable $l$ is updated, as well as the boolean flag *upd*. An assertion at the end of the loop could check that, whenever an element has been found ($upd{=}true$), its index $l$ is a valid index of the array ($0{\leq}l{<}N$). The aim of our static analysis is to infer disjunctive invariants that can help prove such properties.

A static analysis can be formulated as a *state-based* analysis: guided by the program state at the beginning of the loop, it computes the loop postcondition as a program state approximation [43, 68, 138]. As an alternative, our method is related to *trace-based* analysis [30] and computes the loop summary as a transition formula from the prestate (before the loop) to the poststate (after the loop body).

Our analysis is formulated in two stages. Firstly, it collects a constraint abstraction from the method/loop body to be analyzed. This abstraction can be viewed as an intermediate form and is related to the constraint abstraction introduced in [73]. As a second step, an iterating process will find the fixed-point for the constraint abstraction function.

For the running example, the constraint abstraction named $wh$ represents the input-output relation between the loop prestate (in terms of $X$, the unprimed variables $x, N, l, upd$) and the loop poststate (in terms of $X'$, the primed variables $x', N', l', upd'$).

$$\begin{array}{rl} wh(X,X') = & ((nochange(X) \wedge x'{<}N')\circ_{\{l,upd\}} \\ & (l'{=}x \wedge upd'{=}1 \vee nochange(l,upd))\circ_{\{x\}} \\ & (x'{=}x{+}1)\circ_X wh(X,X')) \\ \vee & (nochange(X) \wedge x'{\geq}N') \end{array}$$

The *nochange* operator is a special transition where original and primed variables are made equal: $nochange(\{\}) =_{df}$ `true`; $nochange(\{x\}{\cup}X) =_{df} (x'{=}x){\wedge}nochange(X)$. The

composition operator ($\phi_1 \circ_W \phi_2$) is left-associative and composes the input-output relations $\phi_1$ and $\phi_2$ updating $W$ variables as specified by $\phi_2$ formula. Formally:

**Definition 3.1 (Compose with Update).** *Given $\phi_1, \phi_2$, and the set of variables to be updated $X=\{x_1, \ldots, x_n\}$, the composition operator $\circ_X$ is defined as:*

$$\phi_1 \circ_X \phi_2 =_{df} \exists\, r_1..r_n \cdot \rho_1\, \phi_1 \wedge \rho_2\, \phi_2$$

where $r_1, \ldots, r_n$ are fresh variables;

$$\rho_1 = [x'_i \mapsto r_i]_{i=1}^n \; ; \rho_2 = [x_i \mapsto r_i]_{i=1}^n$$

*Note that $\rho_1$ and $\rho_2$ are substitutions that link each latest value of $x'_i$ in $\phi_1$ with the corresponding initial value $x_i$ in $\phi_2$ via a fresh variable $r_i$.*

With these two operators, the effects of the loop sub-expressions are composed to obtain the effect of the entire loop body. The 1st line of the constraint abstraction corresponds to the loop test that is satisfied. The 2nd line stands for the body of the conditional expression from the loop. Note that the boolean constants `false` and `true` are modeled as integers 0 and 1. The 3rd line represents the assignment that increments $x$ by 1 composed with the effect of subsequent loop iterations (the occurrence of the $wh$ constraint abstraction). The 4th and last line stands for the possibility that the loop test is not satisfied.

After some simplifications, the constraint abstraction reduces to:

$$\begin{aligned} wh(X, X') = \exists X_1 \cdot ( \quad & (x_1{=}x{+}1 \wedge N_1{=}N \wedge l_1{=}x \wedge upd_1{=}1 \wedge wh(X_1, X')) \\ \vee \quad & (x_1{=}x{+}1 \wedge N_1{=}N \wedge l_1{=}l \wedge upd_1{=}upd \wedge wh(X_1, X')) \\ \vee \quad & (x'{=}x \wedge N'{=}N \wedge l'{=}l \wedge upd'{=}upd \wedge x'{\geq}N')) \end{aligned}$$

where $X_1$ denotes the local variables $(x_1, N_1, l_1, upd_1)$.

The analysis goal is then to compute a fixed-point approximation for the constraint abstraction function. This function takes as argument a transition depending on $X, X'$ and its result is also expressed as a transition dependent on the same variables. Both transitions can either be approximated by polyhedra or, more precisely, by sets of polyhedra. The first case is akin to the polyhedral analysis from [43] and is reviewed next. For the second case, we will use our running example to show how to compute a disjunctive loop postcondition.

### 3.2.1 Computing Fixed-Points in the Polyhedron Abstract Domain

We will use the method based on Kleene's fixed-point iteration applied to the polyhedron abstract domain (see Section 2.3.3 for basic notations and a simpler example) for our running example. The fixed-point iteration starts with the least element of the abstract domain represented by the *false* formula. The first approximation $wh_1$ is a transition formula that considers that the loop test fails and the loop body is never executed:

$$wh_1 = (x'{=}x \wedge N'{=}N \wedge l'{=}l \wedge upd'{=}upd \wedge x'{\geq}N')$$

The next iteration is a three-disjunct formula that cannot be represented in the polyhedron domain. An approximation for the disjunctive formula is computed using the convex hull operator. A formula in disjunctive normal form $\phi = \vee_{i=1}^{n} d_i$ can be viewed as a set of disjuncts: $set_d(\phi) = \{d_i\}_{i=1}^{n}$. We use either infixed or prefixed operators on these disjuncts. For example, given $\phi{=}d_1{\vee}d_2$ then $\oplus(\phi) = \oplus(\{d_1, d_2\}) = d_1{\oplus}d_2$.

$$\begin{aligned}
wh_2 = \;& (x'{=}x{+}1 \wedge N'{=}N \wedge l'{=}x \wedge upd'{=}1 \wedge x'{\geq}N') \\
\vee \;& (x'{=}x{+}1 \wedge N'{=}N \wedge l'{=}l \wedge upd'{=}upd \wedge x'{\geq}N') \\
\vee \;& (x'{=}x \wedge N'{=}N \wedge l'{=}l \wedge upd'{=}upd \wedge x'{\geq}N') \\
wh_2' = \;& \oplus(wh_2) = (x{\leq}x'{\leq}x{+}1 \wedge N'{=}N \wedge x'{\geq}N) \\
wh_3' = \;& \oplus(wh_3) = (x{\leq}x'{\leq}x{+}2 \wedge N'{=}N \wedge x'{\geq}N)
\end{aligned}$$

The iterating sequence will not converge since the inequality $x'{\leq}x$ will be translated at the following iterations into $x'{\leq}x{+}1$, $x'{\leq}x{+}2$ and so on. Convergence is ensured by the widening operator which simplifies as follows:

$$wh_3'' = wh_2' \nabla wh_3' = (x{\leq}x' \wedge N'{=}N \wedge x'{\geq}N)$$

This result proves to be a post fixed point for the $wh$ function. However, the result is rather imprecise as it does not capture any information about the value of $l$ or the flag $upd$ at the end of the loop. Intuitively, such information was present in $wh_2$ and $wh_3$, but approximated by the convex hull operator to obtain $wh_2'$ and $wh_3'$. Next, we outline a method to compute disjunctive fixed-points able to capture this kind of information.

### 3.2.2 Computing Fixed-Points in a Disjunctive Abstract Domain

The two ingredients that we use to compute disjunctive fixed-points are counterparts to the convex hull and widening operators from the conjunctive case. Both operators

ensure a bound on the number of disjuncts allowed in the formulae.

We first propose a *selective hull* operator $\oplus_m$ parameterized by a constant $m$ that takes as argument a disjunctive formula and collapses these disjuncts into a result with at most $m$ disjuncts. The crux of this operator is an affinity measure to choose the two most related (affine) disjuncts from a disjunctive formula. Formally:

**Definition 3.2 (Selective Hull).** *Given $\phi = \vee_{i=1}^{n} d_i$, and let $d_i$,$d_j$ be the most related disjuncts as determined by their affinity, we define the selective hull operator as follows:*

$$\begin{aligned} \oplus_m(\phi) =_{df} \quad & \textit{if } n \leq m \textit{ then } \phi \\ & \textit{else } \oplus_m (set_d(\phi) \setminus \{d_i, d_j\} \cup \{d_i \oplus d_j\}) \end{aligned}$$

Note that the convex hull operator from the polyhedron domain $\oplus$ is equivalent to $\oplus_1$ since it reduces its disjunctive argument to a conjunctive formula with one disjunct. The affinity measure aims to quantify how close is the approximation $d_1 \oplus d_2$ from the disjunctive formula $d_1 \vee d_2$. Intuitively, it works by counting the number of inequalities (planes in the n-dimensional space) from the disjunctive formula that are preserved in the approximation $d_1 \oplus d_2$. Since it counts the number of inequalities (relations between variables), this affinity measure is able to handle the relational information captured by the formulae in the polyhedron domain.

As an example, consider $wh_2$ and $wh_3$ obtained previously. The result of selective hull with m=3 the bound on the number of disjuncts is computed as follows:

$$\begin{aligned} wh_2''' = \oplus_3(wh_2) = \quad & wh_2 \\ wh_3''' = \oplus_3(wh_3) = \quad & (x \leq x' \leq x+2 \wedge N'=N \wedge x \leq l' \leq x+2 \wedge upd'=1 \wedge x+2 \geq N) \\ \vee \quad & (x \leq x' \leq x+2 \wedge N'=N \wedge l'=l \wedge upd'=upd \wedge x+2 \geq N) \\ \vee \quad & (x'=x \wedge N'=N \wedge l'=l \wedge upd'=upd \wedge x \geq N) \end{aligned}$$

The second operator needed in the disjunctive abstract domain is a widening operator. We propose a similar affinity measure to find related disjuncts for pairwise widening. For the two disjunctive formulae $wh_2''' = (d_1 \vee d_2 \vee d_3)$ and $wh_3''' = (e_1 \vee e_2 \vee e_3)$, the most affine pairs will distribute the widening operator:

$$\begin{aligned} wh_2''' \nabla_3 wh_3''' = \quad & (d_1 \vee d_2 \vee d_3) \nabla_3 (e_1 \vee e_2 \vee e_3) = (d_1 \nabla e_1) \vee (d_2 \nabla e_3) \vee (d_3 \nabla e_3) \\ = \quad & (x'=N \wedge N'=N \wedge x \leq l' \leq N \wedge upd'=1) \\ \vee \quad & (x'=N \wedge N'=N \wedge l'=l \wedge upd'=upd \wedge x \leq N) \\ \vee \quad & (x'=x \wedge N'=N \wedge l'=l \wedge upd'=upd \wedge x > N) \end{aligned}$$

This result proves to be a post fixed point for the $wh$ function in the powerset domain. The first disjunct captures the updates to the variable $l$, thus $l'$ can safely be used as an index for the array of size $N$. The last two disjuncts capture the cases where, either the loop was executed but the *then* branch of the conditional has never been taken ($x{\leq}N \wedge upd'{=}upd$), or the loop has not been executed ($x{>}N$).

Note that our disjunctive fixed-point computation works not only for loops, but also for general recursion. Our analysis also supports mutual recursion where fixed-points are computed simultaneously for multiple constraint abstraction functions.

Since the computed fixed-point represents a transition, the analysis does not rely on a fixed initial state and can be implemented in a modular fashion. While modular analysis may expose more disjuncts (because no information is assumed about the initial state) and benefits more from our approach, disjunctive analysis has been shown to be also useful for global static analyses [68, 138].

## 3.3 Disjunctive Abstract Domain

Derived from the polyhedron abstract domain $(\mathcal{P}, \Rightarrow)$, we introduce a new disjunctive abstract domain $(\wp_m(\mathcal{P}), \Rightarrow)$ able to represent sets of polyhedra with fixed cardinality $m$. The partial order, the least and the greatest element are similar to those from the base polyhedron domain. Fixed-point analysis in the polyhedron domain [43] attempts to obtain a conjunctive formula result with the help of convex-hull and widening operators. A challenge for disjunctive fixed point inference is to apply hulling and widening selectively on related disjuncts whenever needed. We propose a planar affinity measure to be used by these two important operators in the disjunctive abstract domain.

### 3.3.1 Planar Affinity and Selective Hulling

In this section, we propose a qualitative measure called *affinity* to determine the suitability of two disjuncts for hulling. To identify disjuncts, we expect formulae obtained during fixed-point analysis to be in disjunctive normal form (DNF). For example, the simplification of a formula $\phi$ in our prototype is performed with the help of Omega library [88]. The result in DNF form $\phi = \vee_{i=1}^{n} d_i$ can be viewed as a set of disjuncts: $set_d(\phi) = \{d_i\}_{i=1}^{n}$. Each disjunct $d_i = \wedge_{j=1}^{m} c_{ij}$ is a conjunction of linear inequalities and equalities. It can be represented as a set of conjuncts $set_c(d_i) = \{c_{ij}\}_{j=1}^{m}$.

Measuring the affinity between disjuncts benefits from a canonical form of each disjunct. Representing such linear arithmetic formulae in canonical form can be done by removing constraints that are trivially redundant, syntactically redundant and semantically redundant [95]. The original algorithm is applicable to a set of linear inequalities. In addition, a linear equality can be split in two inequalities, while disequalities (or negative constraints) can also be handled by a more elaborate algorithm [96]. With Omega Library, the simplification of constraints from a disjunct can be done with various trade-offs between precision and efficiency. More expensive tests can ensure that redundant constraints are eliminated [88, page 24]. Alternative to the constraint form, polyhedra can be represented using the double-description method [107]. An advantage of this double representation is that some operators like the set intersection can be computed efficiently in constraint form, while other operators like the convex hull can be computed efficiently in generator form [43, 3]. For our purposes, we use only the constraint form and rely on Omega for efficient simplification at the expense of some redundancy in formulae.

In order to obtain the affinity between two disjuncts $\phi_1$ and $\phi_2$, we have to compute two main expressions (i) $\phi_{hull} = \phi_1 \oplus \phi_2$ and (ii) $\phi_{diff} = \phi_{hull} \wedge \neg(\phi_1 \vee \phi_2)$. Furthermore, we also require a heuristic function *heur* that indicates how closely related is the approximation $\phi_1 \oplus \phi_2$ from the original formula $\phi_1 \vee \phi_2$. With this, we can formally define the affinity measure using:

**Definition 3.3 (Affinity Measure for Hulling).** *Given a function heur that returns a value in the range 1..99, the affinity measure can be defined as:*

$$\begin{aligned} hull\_affin(\phi_1, \phi_2) =_{\texttt{df}} \quad & \texttt{if } \phi_{diff} = \texttt{false then 100} \\ & \texttt{else if } \phi_{hull} = \texttt{true then 0} \\ & \texttt{else } heur(\phi_1, \phi_2) \end{aligned}$$

The precise extreme (100) indicates that the convex-hull operation is exact without any loss of precision. The imprecise extreme (0) indicates that the convex-hull operation is inexact and yields the weakest possible formula `true`. In between these two extremes, we will use an affinity measure to indicate the closeness of the two terms by returning a value in the range 1..99.

This formulation of the affinity measure can be instantiated with various heuristic

functions. We propose the use of a planar affinity measure that computes the fraction of planes from the geometrical representation of the original formula that are preserved in the hulled approximation:

**Definition 3.4 (Planar Affinity for Hulling).** *Given two disjuncts $\phi_1, \phi_2$ and the convex-hull approximation $\phi_{hull} = \phi_1 \oplus \phi_2$, we first compute the set of conjuncts mset using the following: $mset = \{c \in (set_c(\phi_1) \cup set_c(\phi_2)) \mid \phi_{hull} \Longrightarrow c\}$. The planar affinity measure is shown below :*

$$p\text{-}heur(\phi_1, \phi_2) =_{df} \frac{|mset|}{|set_c(\phi_1) \cup set_c(\phi_2)|} * 98 + 1$$

The denominator $|set_c(\phi_1) \cup set_c(\phi_2)|$ represents the number of planes corresponding to the original formulae (from both polyhedra $\phi_1$ and $\phi_2$). Some of these planes are approximated by the hulling process, while others are preserved in the approximation $\phi_{hull}$. The number of preserved planes is represented by the cardinality of $mset$ and indicates the suitability of the two disjuncts for hulling.

**Example 3.1 :** To illustrate the use of this measure for selective hulling, consider the following disjunctive formula (obtained from the example on page 26):

$$F_3 = (x{\leq}0 \wedge x'{=}x) \vee (x{=}1 \wedge x'{=}0) \vee (x{=}2 \wedge x'{=}0)$$

Firstly, the three disjuncts (denoted respectively by $d_1, d_2$ and $d_3$) are converted to a canonical form. As with other operators on polyhedra (e.g. the standard widening operator from [77]), this minimal form requires that no redundant conjuncts are present and, furthermore, each equality constraint is broken into two corresponding inequalities as follows:

$$d_1 = (x{\leq}0 \wedge x'{\geq}x \wedge x'{\leq}x)$$
$$d_2 = (x{\geq}1 \wedge x{\leq}1 \wedge x'{\geq}0 \wedge x'{\leq}0)$$
$$d_3 = (x{\geq}2 \wedge x{\leq}2 \wedge x'{\geq}0 \wedge x'{\leq}0)$$

We compute three affinity values, one for each pair of disjuncts from $\phi$. Note that the cardinality of the set of conjuncts $(set_c(\phi_1) \cup set_c(\phi_2))$ is considered after removing

duplicate conjuncts that appear both in $\phi_1$ and $\phi_2$.

$$d_1 \oplus d_2 = (x'{\leq}x \wedge x'{\leq}0 \wedge x'{\leq}x{-}1) \qquad mset(d_1, d_2) = \{x'{\leq}x, x{\leq}1, x'{\leq}0\}$$
$$p\text{-}heur(d_1, d_2) = 3/7 * 98 + 1 = 43$$
$$d_1 \oplus d_3 = (x'{\leq}x \wedge x'{\leq}0 \wedge x'{\leq}x{-}2) \qquad mset(d_1, d_3) = \{x'{\leq}x, x{\leq}2, x'{\leq}0\}$$
$$p\text{-}heur(d_1, d_3) = 3/7 * 98 + 1 = 43$$

$$d_2 \oplus d_3 = (x{\geq}1 \wedge x{\leq}2 \wedge x'{\geq}0 \wedge x'{\leq}0) \quad mset(d_2, d_3) = \{x{\geq}1, x{\leq}2, x'{\geq}0, x'{\leq}0\}$$
$$p\text{-}heur(d_2, d_3) = 4/6 * 98 + 1 = 66$$

Based on these affinities, the most related pair of disjuncts is $\{d_2, d_3\}$. Even more, this pair of disjuncts satisfy the exact test from Definition 3.3 (there is no loss of precision by hulling $d_2$ and $d_3$) and therefore their affinity is 100. Computing the affinities between each pair of disjuncts has in general a quadratic cost in the number of disjuncts and we represent the results using a diagonal matrix of affinities:

| | $d_1$ | $d_2$ | $d_3$ |
|---|---|---|---|
| $d_1$ | - | 43 | 43 |
| $d_2$ | - | - | 100 |
| $d_3$ | - | - | - |

The above matrix is used to choose the two most affine disjuncts for hulling, $d_2$ and $d_3$. If this operation leaves more disjuncts than the allowed bound $m$, then other disjuncts are subsequently chosen for hulling. For the current example, the selective hull of $\phi$ captures a precise relation between $x$ and $x'$ and is computed as follows:

$$\oplus_2(F_3) = \oplus_2(d_1 \vee d_2 \vee d_3) = d_1 \vee (d_2 \oplus d_3) = (x{\leq}0 \wedge x'{=}x) \vee (x{\geq}1 \wedge x{\leq}2 \wedge x'{=}0)$$

Related to our affinity measure, Sankaranarayanan et al [138] have concurrently introduced a heuristic function that uses the Hausdorff distance to measure the distance between the geometrical representations of two disjuncts. The Hausdorff distance is a commonly used measure of distance between two sets. Given two polyhedra, $P$ and $Q$, their Hausdorff distance can be defined using the following function: $h\text{-}heur(P, Q) =_{df} max_{x \in P}\{min_{y \in Q}\{d(x, y)\}\}$ where $d(x, y)$ is the Euclidian distance between two points $x$ and $y$. This heuristic was deemed as hard to compute in [138] and, as an alternative, a range-based Hausdorff heuristic was used.

Because it reduces the relational constraints among variables to non-relational ranges bounded by constants, we can argue that a range-based heuristic is less suitable for a relational abstract domain like the polyhedron domain. Furthermore, we present an intuitive argument why such a distance based heuristic is less appropriate.

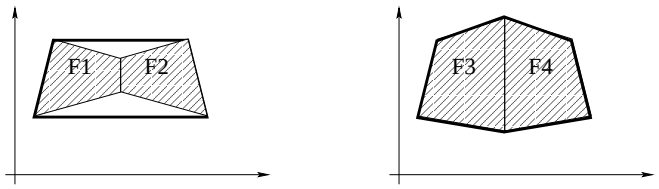



**Figure 3.1:** Pairs of disjuncts with similar Hausdorff distance

The pairs of disjuncts {F1,F2} and {F3,F4} from Figure 3.1 may have similar *h-heur* values; on the other hand, the affinity based on *p-heur* precisely indicates that the second pair {F3,F4} is more suited for hulling. In Section 3.6, we will compare experimentally these two heuristic functions when inferring postconditions for a suite of benchmark programs.

### 3.3.2 Widening Operator

The standard widening operator for the convex polyhedron domain was introduced in [43]. For disjunctive fixed point inference, a (powerset) widening operator for sets of polyhedra is required. Given two disjunctive formulae $\phi_1$ and $\phi_2$, the challenge is to find pairs of related disjuncts $\{d_i, e_i\}$ ($d_i \in \phi_1$, $e_i \in \phi_2$) such that the result of widening $d_i$ wrt $e_i$ is as precise as possible.

For this purpose, Bagnara et al [2] introduced a framework to lift a widening operator over a base domain to a widening operator over its powerset domain. The strategy used by the powerset widening based on a connector starts by joining (connecting) elements in $\phi_2$ to ensure that each such connected element approximates some element from $\phi_1$. Secondly, it chooses related pairs $\{d_i, e_i\}$ based on the logical implication relation, where $d_i \Rightarrow e_i$. Mostly concerned with convergence guarantees for widening operators, the framework from [2, 4] does not give a recipe for defining connector operators able to find related disjuncts. Later, the generic widening operator definition was instantiated for disjunctive polyhedral analysis by Gulavani et al in [68]. However, their proposal uses a connector operator that relies on the ability to find one minimal element from a set of

polyhedra. In general, the most precise result cannot be guaranteed by a deterministic algorithm, since the polyhedron domain is partially ordered. To overcome this problem, we propose an affinity measure to find related disjuncts for pairwise widening.

The strategy that we adopt for widening is to choose related pairs $\{d_i, e_i\}$ based on their affinity. After pairwise widening, we subject the result to a selective hull operation provided it contains more disjuncts than $\phi_1$. In general, there may be more disjuncts in $\phi_2$ than in $\phi_1$. A reason for non-convergence of the powerset widening operator is that some element from $\phi_2$ is not involved in any widening computation and included unchanged in the result. Our operator (similar to the connector-based widening) distributes each disjunct from the arguments $\phi_1$ and $\phi_2$ in a widening computation and thus ensures convergence. Formally, we define the widening operator as follows:

**Definition 3.5 (Widening Operator).** *Given two formulae* $\phi_1 = \bigvee_{i=1}^{m} d_i$ *and* $\phi_2 = \bigvee_{i=1}^{n} e_i$, *the powerset widening operator* $\nabla_m$ *is defined as follows:*

$$\phi_1 \nabla_m \phi_2 = \oplus_m(\{d_i \nabla e_i \mid d_i \in \phi_1, e_i \in \phi_2\})$$

*where* $d_i$ *is the best match for widening* $e_i$ *as found by the widen_affin measure.*

Similar to the affinity from Definition 3.3, the widen-affinity aims to find related disjuncts, but proceeds by indicating how closely related is the approximation $\phi_1 \nabla \phi_2$ from the original formula $\phi_1$:

**Definition 3.6 (Affinity Measure for Widening).** *Given two disjuncts* $\phi_1$, $\phi_2$ *and their widening* $\phi_{widen} = \phi_1 \nabla \phi_2$, *the affinity for widening is defined as:*

$$\begin{aligned} \mathit{widen\_affin}(\phi_1, \phi_2) = \quad & \texttt{if } \phi_{\texttt{widen}} \wedge \neg \phi_1 = \texttt{false then 100} \\ & \texttt{else if } \phi_{\texttt{widen}} = \texttt{true then 0} \\ & \texttt{else } \mathit{heur}(\phi_1, \phi_2) \end{aligned}$$

*The planar affinity measure from Definition 3.4 can be used for widening, provided we redefine mset to relate* $\phi_1, \phi_2$ *with the approximation* $\phi_{widen}$ *as follows:*

$$mset = \{c \in (set_c(\phi_1) \cup set_c(\phi_2)) \mid \phi_{widen} \Rightarrow c\}$$

**Example 3.2 :** To illustrate the use of this measure for disjunctive widening, consider the following two formulae obtained during successive iterations of fixed-point analysis

for the example on page 26:

$$F_3 = (x{\leq}0 \wedge x'{=}x) \vee (1{\leq}x{\leq}2 \wedge x'{=}0) \qquad (d_1 \vee d_2)$$
$$F_4 = (x{\leq}0 \wedge x'{=}x) \vee (1{\leq}x{\leq}3 \wedge x'{=}0) \qquad (e_1 \vee e_2)$$

After splitting the equality constraints, each disjunct can be written as follows:

$$d_1 = (x{\leq}0 \wedge x'{\leq}x \wedge x'{\geq}x)$$
$$d_2 = (1{\leq}x \wedge x{\leq}2 \wedge x'{\leq}0 \wedge x'{\geq}0)$$
$$e_1 = (x{\leq}0 \wedge x'{\leq}x \wedge x'{\geq}x)$$
$$e_2 = (1{\leq}x \wedge x{\leq}3 \wedge x'{\leq}0 \wedge x'{\geq}0)$$

We compute affinity values for each pair of disjuncts, one from $F_3$ and another from $F_4$:

$d_1 \nabla e_1 = (x{\leq}0 \wedge x'{\leq}x \wedge x'{\geq}x)$ $\quad$ $\mathit{mset}(d_1, e_1) = \{x{\leq}0, x'{\leq}x, x'{\geq}x\}$

$\mathit{p\text{-}heur}(d_1, e_1) = 3/3 * 98 + 1 = 99$

$d_1 \nabla e_2 = (x{\leq}0 \wedge x'{\leq}x)$ $\quad$ $\mathit{mset}(d_1, e_2) = \{x{\leq}0, x'{\leq}x\}$

$\mathit{p\text{-}heur}(d_1, e_2) = 2/7 * 98 + 1 = 29$

$d_2 \nabla e_1 = (x{\leq}2 \wedge x'{\leq}0 \wedge x'{\geq}0)$ $\quad$ $\mathit{mset}(d_2, e_1) = \{x{\leq}2, x'{\leq}0, x'{\geq}0\}$

$\mathit{p\text{-}heur}(d_2, e_1) = 3/7 * 98 + 1 = 43$

$d_2 \nabla e_2 = (1{\leq}x \wedge x'{\leq}0 \wedge x'{\geq}0)$ $\quad$ $\mathit{mset}(d_2, e_2) = \{1{\leq}x, x'{\leq}0, x'{\geq}0\}$

$\mathit{p\text{-}heur}(d_2, e_2) = 3/5 * 98 + 1 = 60$

These affinity values can be arranged in a matrix. The affinity between $d_1$ and $e_1$ is modified to 100, since the widening operator applied to them is precise (equivalent to $d_1$):

| | $d_1$ | $d_2$ |
|---|---|---|
| $d_1$ | 100 | 29 |
| $d_2$ | 43 | 60 |

The above matrix is used to choose the most affine disjuncts for widening. Accordingly, the widening operator is computed as follows:

$$F_3 \nabla_2 F_4 = (d_1 \vee d_2) \nabla_2 (e_1 \vee e_2) = (d_1 \nabla e_1) \vee (d_2 \nabla e_2) =$$
$$= (x{\leq}0 \wedge x'{=}x) \vee (x{>}0 \wedge x'{=}0)$$

To conclude this example, we observe that the result of the widening operator satisfies the post fixed point condition and therefore is a safe approximation for the postcondition of the method from Figure 2.6. Comparatively, the postcondition computed using a conjunctive abstract domain is less precise: $(x'{\leq}x \wedge x'{\leq}0)$.

### 3.3.3 Higher-Order Planar Affinity Measure

In the previous sections, we defined two affinity measures, one for hulling (Definition 3.3) and one for widening (Definition 3.6). Both affinity measures aim to predict the loss of precision induced by an abstract operator (e.g. hulling or widening) when applied to the two arguments $\phi_1$ and $\phi_2$. The accuracy of the prediction depends crucially on the heuristic function that is used (e.g. planar affinity or Hausdorff distance).

One important benefit of the planar affinity is that we can formulate it as a higher-order operator and use it to predict the loss of precision induced by an arbitrary abstract operator *op*:

$$\textit{p-heur}(op, \phi_1, \phi_2) =_{df} \frac{|\{c \in (set_c(\phi_1) \cup set_c(\phi_2)) \mid \phi_1 \ op \ \phi_2 \Longrightarrow c\}|}{|set_c(\phi_1) \cup set_c(\phi_2)|} * 98 + 1$$

With this definition, we can use the planar affinity for other abstract operators, like the greatest lower bound operator or the narrowing operator.

## 3.4 Forward Reasoning Rules

To complement the fixed-point analysis presented in the previous section, we propose a set of forward reasoning rules for collecting a constraint abstraction for each method/loop. Some primitive methods may lack a method body and be given instead a formula $\phi$: the given formula may include a safety precondition (for example, bound checks for array operations), or simply represent the input-output relation (for primitive numerical operations like *add* or *multiply*). The reasoning process is modular, starting with the methods at the bottom of the call graph.

We shall use the core imperative language introduced previously in Chapter 2. The constraint language is based on the theory of linear arithmetic and denoted by $\phi$ formulae. We shall assume that a type-checker exists to ensure that expressions and constraints used in a program are well-typed.

The rule [**meth**] associates each method $mn$ with a constraint abstraction of the same name. Namely, $mn(v^*, w^*) = \phi$, where $v^*$ covers the input parameters, while $w^*$ covers the method's output **res** and the primed variables from pass-by-reference parameters. The fixed point analysis outlined in the previous section is invoked by $\mathit{fix}(Q)$ and returns $\phi_{po}$, the input-output relation of the method. To derive suitable

postconditions, we shall subject each method declaration to the following rule:

$$[\underline{\textbf{METH}}]$$

$$\frac{W{=}\{v_i\}_{i=1}^{n} \quad V{=}\{v'_i\}_{i=m}^{n} \quad \vdash \{nochange(W)\}\, e\, \{\phi\} \qquad X{=}\{v_1,..,v_n,\mathbf{res},v'_1,..,v'_{m-1}\} \quad Q{=}\{mn(X)=\exists V\cdot\phi\} \quad \phi_{po}=\mathit{fix}(Q)}{\vdash t_0\ mn((\texttt{ref}\ t_i\ v_i)_{i=1}^{m-1},(t_i\ v_i)_{i=m}^{n})\ \texttt{where}\ mn(X)\{e\}\Rightarrow\phi_{po}}$$

The inference uses a set of Hoare-style forward reasoning rules of the following form $\vdash\{\phi_1\}\, e\, \{\phi_2\}$. Given a transition $\phi_1$ from the beginning of the current method/loop to the prestate before $e$'s evaluation, the judgement will derive $\phi_2$, a transition from the beginning of the current method/loop to the poststate after $e$'s evaluation. A special variable **res** is used to denote the result of method declaration as well as that of the current expression under program analysis. Due to our use of primed variables and existential linking of values that are passed around, the forward rules essentially derive an SSA-like translated formula. Each intermediate value is captured by a unique variable and the translated formula is purely declarative.

The reasoning rules are shown in Figure 3.2. The [**ASSIGN**] rule captures imperative updates with the help of the prime notation. The [**SEQ**] rule captures flow-sensitivity, while the [**IF**] rule captures path-sensitivity. The [**CALL**] rule accumulates the effect of the callee postcondition using the composition operator: $\phi\ \circ_W\ \phi_{po}$. This rule postpones the checking of the callee precondition to a later stage. The two rules [**METH**] and [**WHILE**] compute a postcondition (indicated to the right of the $\Rightarrow$ operator) which will be inserted in the code and used subsequently in the verification rules. The result of these rules is a definition for each constraint abstraction. As an example, consider:

```
void mnA(ref int x, int n) where (mnA(x,n,x'))
{ if x>n then x:=x-1;mnA(x,n) else () }
```

After applying the forward reasoning rules, we obtain the following constraint abstraction:

$$\texttt{mnA(x,n,x')}{=}(\texttt{x>n}\wedge(\exists\texttt{x1}\cdot\texttt{x1=x-1}\wedge\texttt{mnA(x1,n,x')}))\vee(\texttt{x}\leq\texttt{n}\wedge\texttt{x'=x})$$

Note that the forward rules can be used to capture the postcondition of any recursive method, not just for tail-recursive loops. For example, consider the following recursive method:

```
int mnB(int x) where (mnB(x,res)) { if x≤0 then 1 else x:=x-1;2+mnB(x) }
```

$$\frac{\phi_1 = (\phi \wedge \mathbf{res}{=}k)}{\vdash \{\phi\}\, k\, \{\phi_1\}}\ [\textsc{const}] \qquad \frac{\phi_1 = (\phi \wedge \mathbf{res}{=}v')}{\vdash \{\phi\}\, v\, \{\phi_1\}}\ [\textsc{var}]$$

$$\frac{\vdash \{\phi\}\, e\, \{\phi_1\} \quad \phi_2 = \exists \mathbf{res}\cdot(\phi_1 \circ_{\{v\}} v'{=}\mathbf{res})}{\vdash \{\phi\}\, v{:=}e\, \{\phi_2\}}\ [\textsc{assign}]$$

$$\frac{\vdash \{\phi\}\, e\, \{\phi_1\}}{\vdash \{\phi\}\, t\ v;\ e\, \{\exists v'\cdot\phi_1\}}\ [\textsc{blk}] \qquad \frac{\vdash \{\phi \wedge v'{=}1\}\, e_1\, \{\phi_1\} \quad \vdash \{\phi \wedge v'{=}0\}\, e_2\, \{\phi_2\}}{\vdash \{\phi\}\, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\, \{\phi_1 \vee \phi_2\}}\ [\textsc{if}]$$

$$\frac{\vdash \{\phi\}\, e_1\, \{\phi_1\} \quad \vdash \{\exists \mathbf{res}\cdot\phi_1\}\, e_2\, \{\phi_2\}}{\vdash \{\phi\}\, e_1; e_2\, \{\phi_2\}}\ [\textsc{seq}]$$

$$\frac{W{=}\{v_i\}_{i=1}^{m-1} \quad \mathit{distinct}(W) \quad t_0\ mn((\texttt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \texttt{where}\ \phi_{po}\ \{...\}}{\vdash \{\phi\}\, mn(v_1..v_n)\, \{\phi \circ_W \phi_{po}\}}\ [\textsc{call}]$$

$$\frac{X{=}\mathit{freevars}(v,e) \quad \vdash \{\mathit{nochange}(X) \wedge v'{=}1\}\, e\, \{\phi_1\} \quad \phi_2{=}(\phi_1 \circ_X wh(X,X')) \vee (\mathit{nochange}(X) \wedge v'{=}0) \quad Q{=}\{wh(X,X') = \phi_2\} \quad \phi_{po} = fix(Q)}{\vdash \{\phi\}\, \texttt{while}\ v\ \texttt{do}\ e\, \{\phi \circ_X \phi_{po}\} \Rightarrow \phi_{po}}\ [\textsc{while}]$$

**Figure 3.2:** Forward reasoning rules

Applying forward reasoning rules will yield the following constraint abstraction:

$$\texttt{mnB}(\texttt{x}, \mathbf{res}){=}(\texttt{x}{\leq}\texttt{0} \wedge \mathbf{res}{=}\texttt{1}) \vee (\texttt{x}{>}\texttt{0} \wedge (\exists \texttt{x1}, \texttt{r1}\cdot \texttt{x1}{=}\texttt{x}{-}\texttt{1} \wedge \texttt{mnB}(\texttt{x1}, \texttt{r1}) \wedge \mathbf{res}{=}\texttt{2}{+}\texttt{r1}))$$

The next step is to apply fixed point analysis on each recursive constraint abstraction. By applying disjunctive fixed point analysis, we can obtain:

$$\texttt{mnB}(\texttt{x}, \mathbf{res}){=}(\texttt{x}{\leq}\texttt{0} \wedge \mathbf{res}{=}\texttt{1}) \vee (\texttt{x}{\geq}\texttt{0} \wedge \mathbf{res}{=}\texttt{2}{*}\texttt{x}{+}\texttt{1})$$

### 3.4.1 Verification of Preconditions

Once a closed-form formula has been derived via fixed-point analysis, we shall return to checking the validity of preconditions that were previously skipped. The rules for verifying preconditions are similar to the forward rules for postcondition inference, with the exception of three rules, namely:

$$\frac{t_0\ mn((\texttt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \texttt{where}\ \phi_{po} \quad W{=}\{v_i\}_{i=1}^{m-1} \quad Z{=}\{\mathbf{res}, v'_1, .., v'_{m-1}\} \quad \phi_{pr}{=}\exists Z\cdot\phi_{po} \quad \phi \Longrightarrow [v_i{\mapsto}v'_i]_{i=1}^{n}\phi_{pr}}{\vdash \{\phi\}\, mn(v_1..v_n)\, \{\phi \circ_W \phi_{po}\}}\ [\textsc{verify-call}]$$

$$\frac{X = \mathit{freevars}(v,e) \quad \rho = X{\mapsto}X' \quad \phi_{pr} = \exists X'\cdot\phi_2 \quad \phi \implies \rho\phi_{pr} \quad \vdash \{\phi \wedge \rho\phi_{pr}\}\, e\, \{\phi'\}}{\vdash \{\phi\}\, \texttt{while}\ v\ \texttt{do}\ e\ \texttt{where}\ \phi_2\, \{\phi \circ_X \phi_2\}}\ [\textsc{verify-while}]$$

$$[\underline{\textbf{VERIFY}-\textbf{METH}}]$$
$$W{=}\{v_i\}_{i=1}^{n} \quad Z{=}\{\textbf{res}, v'_1, .., v'_{m-1}\}$$
$$\phi_{pr}{=}\exists Z{\cdot}\phi_{po} \quad \vdash \{\phi_{pr}{\wedge}nochange(W)\}\, e\, \{\phi\}$$
$$\vdash t_0\ mn((\texttt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \texttt{where}\ \phi_{po}\ \{e\}$$

The [**VERIFY–CALL**] rule checks that the precondition of each method call can be verified as *statically safe* by the current program state. If it cannot be proven statically safe, a run-time test will be inserted prior to the call site to guarantee the safety of the precondition during program execution. The precondition derived for recursive methods is meant to be also satisfied recursively. The [**VERIFY–METH**] rule ensures that each of its callees is either statically safe or has a runtime test inserted. The [**VERIFY–WHILE**] rule uses $X$ to denote the free variables appearing in the loop body; the substitution $\rho$ maps the unprimed to primed variables. This rule uses the loop formula $\phi_2$ to compute a precondition $\phi_{pr}$ necessary for the correct execution of the loop body. The precondition is checked for satisfiability using $\phi$, the state at the beginning of the loop. We refer to this new set of rules as *forward verification* rules. We define a special class of *totally-safe* programs, as follows:

**Definition 3.7 (Totally-Safe Program).** *A method is said to be* totally-safe *if the precondition derived from all calls in its method's body can be verified as statically safe. A program is* totally-safe *if all its methods are* totally-safe.

For each totally-safe program, we can guarantee that it never encounters any runtime error due to unsatisfied preconditions.

## 3.5 Mixing Boolean and Integer Constraints

In the preceding sections, we have focused largely on the (linear) arithmetic constraint domain. However, in our constraint sublanguage, we can provide support for boolean variables to be captured explicitly. In general, there are two ways to handle boolean constraints.

The simpler solution that we adopted is to map each boolean value into the integer domain and then allow an integer-based solver to handle the combined formulae. For example, we can map `false` to 0, `true` to 1 and ensure that each boolean variable $v$

be bounded by $0 \leq v \leq 1$ in the integer domain. Furthermore, the basic boolean operators would have their postcondition translated as follows (i) $not(v) \Rightarrow (\mathbf{res}=1-v)$, (ii) $or(v_1, v_2) \Rightarrow (\mathbf{res}=max(v_1, v_2))$ and (iii) $and(v_1, v_2) \Rightarrow (\mathbf{res}=min(v_1, v_2))$. The main advantage of this approach is that it allows a simple integration for both the boolean and integer constraints under a single framework. However, there are some problems with this approach. Firstly, the resulting constraint may have many disjunctions due to the use of *min* and *max* operators which are defined in terms of disjunctions. This can make such formulae more difficult to handle. Secondly, when performing fixed point inference, the integer-based solver may try to discover arithmetic relationships between boolean and integer variables that are typically meaningless. Thus, from the point of inference, a single domain may result in a more complicated analysis.

Another solution is to use a different boolean SAT solver to handle boolean constraints, and to allow this solver to coexist with the integer-based solver. Our proposal is to support a formula $\bigvee (\sigma \wedge \phi)^*$ where $\phi$ is an integer constraint in conjunctive form, while $\sigma$ is an arbitrary boolean formula. We call each disjunct of this form a mixed constraint. The formula may be subjected to a hulling operation to limit the size of the outermost disjunction. Formally, we define hulling for a pair of mixed constraints as follows:

$$(\sigma_1 \wedge \phi_1) \oplus_m (\sigma_2 \wedge \phi_2) =_{df} (\sigma_1 \vee \sigma_2) \wedge (\phi_1 \oplus_m \phi_2)$$

Note that approximation is carried out for integer constraint, while boolean constraint may be kept precise. We can now define an affinity measure for mixed constraints. We may give higher weightage to the affinity measure for boolean formulae as it is more precise than the heuristic used for integer constraints:

**Definition 3.8 (Affinity for Mixed Constraint).** *Let $k$ be a weightage ratio we give to prioritize boolean constraint over the integer constraint. We can define affinity for mixed constraints as follows:*

$$affin(\sigma_1 \wedge \phi_1, \sigma_2 \wedge \phi_2) =_{df} (affin(\sigma_1, \sigma_2) \times k + affin(\phi_1, \phi_2))/(k+1)$$

*Given a boolean formula, we use $\#\sigma$ to denote the number of truth assignments in $\sigma$ that can be obtained by a counting SAT algorithm. Let $\mathtt{mn}=min(\#\sigma_1, \#\sigma_2)$ and*

| Benchmark Programs | Source (lines) | Rec. constr. | m=1 (secs) | m=2 | | m=3 | | m=4 | | m=5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | (secs) | post | (secs) | post | (secs) | post | (secs) | post |
| binary search | 31 | 1 | 0.44 | 1.02 | 1 | - | - | - | - | - | - |
| bubble sort | 39 | 2 | 0.78 | 0.89 | 1 | - | - | - | - | - | - |
| init array | 5 | 1 | 0.17 | 0.24 | 1 | - | - | - | - | - | - |
| merge sort | 58 | 3 | 1.42 | 3.39 | 3 | 3.76 | 1 | 3.91 | 1 | 4.48 | 1 |
| queens | 39 | 2 | 1.89 | 2.41 | 2 | 2.48 | 1 | - | - | - | - |
| quick sort | 43 | 2 | 0.63 | 1.51 | 2 | 1.70 | 1 | - | - | - | - |
| FFT | 336 | 9 | 8.24 | 10.17 | 5 | 11.62 | 3 | 11.90 | 1 | 12.15 | 1 |
| LU Decomp. | 191 | 10 | 10.27 | 13.41 | 8 | 14.44 | 3 | - | - | - | - |
| SOR | 84 | 5 | 1.46 | 2.41 | 3 | 3.49 | 1 | 3.64 | 1 | - | - |
| Linpack | 903 | 25 | 28.14 | 33.23 | 20 | 35.04 | 2 | - | - | - | - |

**Figure 3.3:** Statistics for postcondition inference. Timings include precondition verification. ("-" signifies a time and post similar to those from the immediate lower value of m)

$\texttt{mx}=max(\#\sigma_1, \#\sigma_2)$ *We can define affinity of two boolean formulae* $\sigma_1$ *and* $\sigma_2$*, as follows:*

$$\begin{aligned} affin(\sigma_1, \sigma_2) \ =_{\texttt{df}} \ & \texttt{if } (\sigma_1 \wedge \neg\sigma_2) \texttt{ then } 100 \\ & \texttt{else if } \neg(\sigma_1 \wedge \sigma_2) \texttt{ then } 0 \\ & \texttt{else } 99 - ((\#(\sigma_1 \vee \sigma_2) - \texttt{mx}) \times 98/\texttt{mn}) \end{aligned}$$

With this extra definition for affinity, we may proceed to handle mixed constraints for disjunctive fixed point analysis using the same framework as that for integer constraints.

## 3.6 Experimental Results

We have implemented the proposed inference mechanisms with the goal of analyzing imperative programs. Our implementation includes a pre-processing phase to convert each C-like input program to our core language. The entire prototype system was built using Glasgow Haskell compiler [121] extended with the Omega constraint solving library [129, 88]. Our test platform was a Pentium 3.0 GHz system with 2GBytes main memory, running Fedora 4.

We tested our system on a set of small programs with challenging recursion, and also the Scimark and Linpack benchmark suites [112, 51]. Figure 3.3 summarizes the statistics obtained for each program. To quantify the analysis complexity of the benchmark programs, we counted the program size (column 2) and also the number of recursive methods and loops present in each program (column 3).

The main objective for building this prototype was to certify that the disjunctive analysis can be fully automated and that it gives more precise results compared to a conjunctive analysis. To this end, we experimented with different bounds on the number

$m$ of disjuncts allowed during fixed point analysis. For each value of $m$, we measured the analysis time and the number of methods for which the postcondition was *more precise* than using $(m-1)$ disjuncts. For each analyzed program, we detected a *bound* on the value of $m$: increasing $m$ over this bound does not yield more precision for the formulae. The analysis time remains constant for cases where $m$ is bigger than this bound, therefore the values beyond these bounds are marked with "-". Capturing a precise postcondition for algorithms like binary search, bubble sort, or init array was done with a value of $m$ equal to 2. We found that queens and quick sort require 3 disjuncts, while merge sort can be inferred by making use of 5 disjuncts.

After experimenting with different bounds for $m$ imposed by the user, we designed a heuristic to obtain automatically a bound useful for both precise and efficient analysis. Such a bound is computed separately for each recursive method. More specifically, the corresponding constraint abstraction is unrolled for a fixed number of times (3 in our implementation) and we count the number of non-adjacent disjuncts. Adjacent disjuncts, those with 100 affinity for hulling, are combined directly. This heuristic gives useful bounds for most of the methods and keeps the cost of analysis low by using more disjuncts only when needed. We found only few methods (in `FFT` and `Linpack` benchmarks) where, due to multiplication and division operators, a high bound is generated. For such cases, we normally force a high bound of 5 disjuncts, unless overridden by the user.

We also evaluated the usefulness of the disjunctive fixed point inference for static array bound check elimination. The results are summarized in the Figure 3.4. Column 2 presents the total number of checks (counted statically) that are present in the original programs. Columns 4 and 8 present the number of checks that cannot be proved safe by using conjunctive analysis ($m$=1) and, respectively, disjunctive analysis with $m$=5 and planar affinity. For comparison, column 6 shows results of analysis using the Hausdorff distance heuristic, where the number of checks not proven is greater than using planar affinity. While the planar affinity is usually more expensive (column 7) than the Hausdorff affinity (column 5) for our experiments, we found that the difference is minor considering the additional checks that can be proven safe using our planar affinity.

Using the planar affinity, the two programs bubble sort and init array were proven

| Benchmark Programs | Static Chks. | Conj.(m=1) | | Haus.(m=5) | | Plan.(m=5) | |
|---|---|---|---|---|---|---|---|
| | | (secs) | pre | (secs) | pre | (secs) | pre |
| binary search | 2 | 0.44 | 2 | 0.80 | 2 | 1.02 | 2 |
| bubble sort | 12 | 0.78 | 3 | 0.78 | 0 | 0.89 | 0 |
| init array | 2 | 0.17 | 2 | 0.22 | 0 | 0.24 | 0 |
| merge sort | 24 | 1.42 | 9 | 3.09 | 4 | 4.48 | 0 |
| queens | 8 | 1.89 | 4 | 2.37 | 2 | 2.48 | 2 |
| quick sort | 20 | 0.63 | 5 | 1.43 | 5 | 1.70 | 1 |
| FFT | 62 | 8.24 | 17 | 11.91 | 12 | 12.15 | 5 |
| LU Decomp. | 82 | 10.27 | 42 | 14.71 | 9 | 14.44 | 4 |
| SOR | 32 | 1.46 | 15 | 2.78 | 2 | 3.64 | 0 |
| Linpack | 166 | 28.14 | 92 | 31.99 | 65 | 35.04 | 52 |

**Figure 3.4:** Statistics for check elimination

totally safe with 2-disjunctive analysis. Merge sort and SOR exploited the precision of 4-disjunctive analysis for total check elimination. Even if not all the checks could be proven safe for queens, quick sort, FFT, LU and Linpack benchmarks, the number of potentially unsafe checks decreased gradually, for analyses with higher values of $m$. As a matter of fact, our focus in this chapter was to infer precise postconditions and we relied on a simple mechanism to derive preconditions. To eliminate more checks, we could use preconditions sufficient for safety in the style of [28, 127]. In the next chapter, we will propose a technique that is powerful enough to derive sufficient preconditions and eliminate all checks in this set of benchmarks. However, we stress that, either kind of prederivation we use, disjunctive analysis is needed for better check elimination.

In general, analysis with higher values for $m$ has the potential of inferring more precise formulae. The downside is that computing the affinities of $m$ disjuncts is an operation with quadratic complexity in terms of $m$ and may become too expensive for higher values of $m$. In practice, we found that the case ($m$=3) computes formulae sufficiently precise, with a reasonable inference time.

## 3.7 Examples

In this section, we give details on the analysis of some interesting examples. Firstly, we show how disjunctive fixed-point is able to obtain a precise result for the McCarthy's 91 function ($\texttt{res} = 91$). Secondly, we show the pivot partitioning method from the quicksort algorithm where we observe a gradual increase in precision for postconditions computed with 1, 2 and 3 disjuncts. Lastly, we show an example where a modular

analysis requires a more complex abstract domain compared to a global analysis with fixed values for the method parameters. This last example is extracted from the `FFT` benchmark and requires non-linear invariants.

### 3.7.1 McCarthy's 91 Function

The McCarthy's 91 function shown in Figure 3.5 returns the value 91 for all integer arguments smaller or equal with 101. For arguments $n$ bigger than 101, the function returns $(n - 10)$. Despite its simplicity, this example is challenging for static analysis and was described in the context of various conjunctive domains in [38, 15, 93].

```
int f91(int n) {
  if (n<=100) then {
    f91(f91(n+11))
  } else { n-10 }
}
```

**Figure 3.5:** McCarthy's 91 function

Our disjunctive fixed-point analysis computes the following intermediate results:

$$F_1 = (n{\geq}101 \wedge res{=}n{-}10)$$
$$F_2 = (n{\geq}101 \wedge res{=}n{-}10) \vee (n{=}100 \wedge res{=}91)$$
$$F_3 = (n{\geq}101 \wedge res{=}n{-}10) \vee (n{=}100 \wedge res{=}91) \vee (n{=}99 \wedge res{=}91)$$

The 2-disjunctive abstract domain is sufficiently fine-grained, since hulling all the disjuncts obtained from recursion is precise with 100% affinity. The postcondition computed matches the exact semantics of the code:

$$(n{\geq}101 \wedge res{=}n{-}10) \vee (n{\leq}100 \wedge res{=}91)$$

### 3.7.2 Quicksort Example

Figure 3.6 shows an excerpt from a quicksort algorithm. The method `partition` divides the elements of the array `a` between the indexes `l` and `h` into two partitions: elements smaller than the initial pivot value `a[l]` and elements greater than the pivot value. The parameters of the method `changeN` are `a`, `n`, `i`, `h` and `v`: `a` is the array to sort; `n` represents the last element smaller-or-equal than the pivot; `i` and `h` are the start and the end of

the sequence remaining to be compared with the pivot and `v` corresponds to the value of the pivot.

```
int partition(float[] a, int l, int h) {
  int v := a[l];
  int n := changeN(a,l,l+1,h,v);
  swap(a,l,n); n }

int changeN(float[] a, int n, int i, int h, float v) {
  if (i <= h) then {
    if (a[i] < v) then {
      swap(a,n+1,i);
      changeN(a,n+1,i+1,h,v)
    } else { changeN(a,n,i+1,h,v) }
  } else { n } }

void swap(float[] a, int i, int j) {
  float temp := a[i];
  a[i]:=a[j]; a[j]:=temp }
```

**Figure 3.6:** Quicksort example

We present the postconditions inferred for the recursive method `changeN` with increasing values of $m$ (1, 2 and 3). While a lower bound for the result of the method (`res`$\geq$`n`) can be discovered using conjunctive analysis, inference with 2 and 3 disjuncts are able to discover also upper bounds for `res`. The result `res` may be safely used as a valid index for the array parameter only with the relation discovered by the 3-disjunctive analysis: (`0`$\leq$`res`$<$`s`), where `s` is used to denote the size of array `a`.

$$\texttt{Postcondition with 1-disjunct}:$$

$$\texttt{changeN}(\texttt{s},\texttt{n},\texttt{i},\texttt{h},\texttt{res}) \quad =(\texttt{res}\geq\texttt{n})$$

$$\texttt{Postcondition with 2-disjuncts}:$$

$$\texttt{changeN}(\texttt{s},\texttt{n},\texttt{i},\texttt{h},\texttt{res}) \quad =(\texttt{h}<\texttt{i} \wedge \texttt{res}=\texttt{n})$$
$$\vee(0\leq\texttt{i}\leq\texttt{h}<\texttt{s} \wedge \texttt{n}\leq\texttt{res}\leq\texttt{n}+(\texttt{h}-\texttt{i})+1)$$

$$\texttt{Postcondition with 3-disjuncts}:$$

$$\texttt{changeN}(\texttt{s},\texttt{n},\texttt{i},\texttt{h},\texttt{res}) \quad =(\texttt{h}<\texttt{i} \wedge \texttt{res}=\texttt{n})$$
$$\vee(0\leq\texttt{i}\leq\texttt{h}<\texttt{s} \wedge \texttt{res}=\texttt{n}))$$
$$\vee(0\leq\texttt{i}\leq\texttt{h}<\texttt{s} \wedge 0\leq\texttt{n}+1<\texttt{s} \wedge \texttt{n}<\texttt{res}\leq\texttt{n}+(\texttt{h}-\texttt{i})+1 \wedge \texttt{res}<\texttt{s})$$

For array bound check elimination, the precondition obtained from the 2-disjunctive postcondition of the method `changeN` helps to prove statically-safe the array access to $\texttt{a}[\texttt{i}]$: $\phi_{\texttt{pre}} = (\texttt{h}<\texttt{i})\vee(0\leq\texttt{i}\leq\texttt{h}<\texttt{s})$, However, this precondition cannot guarantee the safety of the array accesses inside $\texttt{swap}(\texttt{a},\texttt{n}+1,\texttt{i})$ call. Consequently, in the optimized quicksort program, a runtime test is needed to protect this call to the method `swap`.

The precondition $\phi_{\texttt{pre}}$ may be hoisted out for runtime testing at some of its call sites. At first sight, the replacement of two checks $0\leq\texttt{i}<\texttt{s}$ of array access $\texttt{a}[\texttt{i}]$ by a more complex condition $(\texttt{h}<\texttt{i})\vee(0\leq\texttt{i}\leq\texttt{h}<\texttt{s})$ may potentially cause performance degradation during check hoisting. However, the array access occurs within a recursive method, and we are effectively replacing multiple runtime checks by a single hoisted check.

As mentioned previously, the derivation of *necessary* preconditions may be complemented by a *sufficient* precondition derivation technique. The following sufficient precondition can be derived (with the use of disjunctive fixed point analysis) and helps prove all checks in quicksort as statically safe: $(\texttt{h}<\texttt{i} \vee 0\leq\texttt{i}\leq\texttt{h}) \wedge (\texttt{h}<\texttt{a}) \wedge(0\leq\texttt{n}+1<\texttt{a}-(\texttt{h}-\texttt{i}))$

### 3.7.3 Fast Fourier Transform Example

The third example shows a complex excerpt from the `FFT` benchmark (see Figure 3.7). The code written in C language indicates with comments what are the values of the relevant parameters: $(len(data) = 2048)$ and $(N = 2048)$. An invariant for the outer loop is: $(0\leq bit<logn \wedge dual=2^{bit})$. Even with the help of this non-linear invariant,

```
void FFT_transform_internal (int N, double *data, int direction) {
  // len(data) is 2048
  int n = N / 2;    // N is 2048, n is 1024
  int dual = 1;
  int logn = int_log2(n);    // logn is 10
  for (bit = 0; bit < logn; bit++, dual *= 2) {
    // dual will have values of {1,2,4,8,16,32,64,128,256,512}
    ...
    for (b = 0; b < n; b += 2 * dual) {
      int i = 2 * b;
      int j = 2 * (b+dual);
      ...  data[j] ...
      ...  data[j+1] ...      // assert (0 <= j+1 < len(data))
      ...  data[i] ...
      ...  data[i+1] ...
    }
  }
}
```

**Figure 3.7:** FFT example

it is tricky to obtain the inner loop invariant that is useful in proving the safety of array checks: $(0 \leq b{+}dual < 1024)$. For example, the array check `data[j+1]` requires: $(0 \leq 2{*}(b{+}dual){+}1 < 2048)$.

We use this example to show one complex example of the checks that cannot be proven by our system. The verification task can be eased by specializing a copy of the inner loop for each value of the `dual` local variable. Even in this simpler case, analysis of the inner loop (shown in Figure 3.8 specialized for the value `2` of the local variable `dual`) requires an existential invariant: $(0{\leq}b{<}1024 \wedge \exists k \cdot (b{=}4k))$. Inferring this existential invariant is not possible using our disjunctive abstract domain, but would be an interesting future work. It requires the extension of our abstract domain with support for congruence relations introduced by Granger in [64, 65].

## 3.8 Correctness

In this section, we shall outline the proofs that state that our forward reasoning rules are correct in the following ways:

1. the forward analysis algorithm terminates (lemma 3.1).

```
for (b = 0; b < 1024; b += 4) {
  ...
  int j = 2 * (b+4);
  ...  data[j+1] ...
}
```

**Figure 3.8:** Inner loop of FFT example

2. its inferred postcondition is a safe approximation of the possible final program state (lemma 3.2).

3. each program that has been verified as *totally-safe* never fails due to primitive operations having unsatisfiable preconditions (lemma 3.4).

The first two aspects are directly related to the correctness of forward reasoning rules, while the third aspect is concerned with the safety of *totally-safe* program whose preconditions have been statically verified. We shall also highlight why it is sound to use the inferred postcondition of each method for strengthening its method's necessary precondition.

When used with both unprimed and primed variables, $\phi$ actually denotes a transition (or change) in abstract states. In this situation, we shall use two operators to distinguish between the original and final abstract states, as follows:

**Definition 3.9 (Prestate and Poststate).** *Given an abstract state transition $\phi$, its prestate $PreSt(\phi)$ captures the relation between unprimed variables of $\phi$. Correspondingly, its poststate $PostSt(\phi)$, captures the relation between primed variables of $\phi$.*

### 3.8.1 Termination of Analysis

As highlighted in Sections 3.4 and 3.3, our postcondition inference algorithm is being organised into two main stages. In the first stage, a constraint abstraction is built for each method. In the second stage, a fixed point analysis is applied to each constraint abstraction that is recursive. The first stage always terminates since the forward rules effectively perform a structural recursion over its input expression. Consequently, the termination of postcondition inference is solely dependent on the termination of fixed

point analysis. We can show the terminations for the disjunctive fixed point analysis, as follows:

**Lemma 3.1 (Termination of Disjunctive Fixed Point).** *Given an affinity measure, we can show that disjunctive fixed point with an upper bound on the number of disjuncts always terminates.*

**Proof** *: Related to the result of [43] in which widening is used to ensure that constraints encountered during conjunctive fixed point have at most finite variations. Due to the existence of an upper bound on the number of disjuncts, widening will eventually occur for a disjunctive formulae with respect to an earlier formulae with an identical number of disjuncts. This widening operation will be applied to the respective components (with closest affinity) of two disjunctive formulae. It will ensure that there are finite variations in the constraints encountered for each m-disjuncts formulae. Hence, there are at most finite variations of the disjunctive formulae. The choice of the affinity measure does not influence the termination of the fixed point analysis (only its precision).*

### 3.8.2 Soundness of Postcondition Inference

The poststate that we infer is a conservative approximation of the program state that we expect after executing the program. A related property is stated as Theorem 5.1, and we refer to its complete proof in Chapter 5.

**Lemma 3.2 (Soundness of Postcondition Inferred).** *Given an abstract state $\phi$ and an expression $e_1$, we may obtain a new poststate $\phi_1$ via our forward reasoning rules, as follows: $\vdash \{\phi\}\, e_1\, \{\phi_1\}$. This inference is sound as we can show that the following holds, namely : for all states $s_1$ consistent with $\phi_1$, if $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \delta\rangle$, then it must be the case that the pair of states $(s_1,\ [res \mapsto \delta] + s_n)$ is soundly approximated by $\phi_1$.*

### 3.8.3 Necessary Precondition

Postcondition inference expresses a possible abstract state and is conservative in that whenever its program code terminates, its resulting program state is always captured by the inferred postcondition. As a corollary, if an initial state $s_1$ is not covered by the prestate of the inferred postcondition, we can confirm that it fails either as $\langle s_1, e_1\rangle \hookrightarrow^* \langle \texttt{error}\rangle$ from runtime error or as $\langle s_1, e_1\rangle \not\hookrightarrow^*$ from non-termination. Thus,

strengthening the precondition of a method with the prestate of its method's inferred postcondition is safe to use for ruling out a class of definite errors, including those from non-termination. The following lemma shows that necessary precondition is always safe to use and never induces any false alarms.

**Lemma 3.3 (Definite Errors from each Necessary Precondition).** *Consider an inference* $\vdash \{\phi\}\, e_1\, \{\phi_1\}$ *and an initial abstract state* $\phi$ *such that* $\phi \Longrightarrow nochange(\mathcal{V}(e))$. *For each state* $s_1$ *consistent with* $\phi$, *but not consistent with* $\phi_1$, *it is never the case that* $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \delta\rangle$. *This means that either* $\langle s_1, e_1\rangle \hookrightarrow^* \langle \texttt{error}\rangle$ *or* $\langle s_1, e_1\rangle \not\hookrightarrow^*$ *does not terminate.*

**Proof** *: Follows as a corollary of Lemma 3.2 on the Soundness of Inferred Postcondition, where we require that* $s_1$ *is consistent with* $\phi_1$ *whenever there is a possibility that* $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \delta\rangle$.

*A related property in the context of Chapter 5 is stated by the Corollary 5.5.*

### 3.8.4 Totally-Safe Program

For programs that have been verified as totally-safe by our forward reasoning rules, we can guarantee that run-time errors from primitive operations can never occur using the following lemma:

**Lemma 3.4 (Totally-Safe Program).** *Given a program that has been shown to be* totally-safe *by the forward verification rules of Section 3.4, its execution will always either make progress or terminate with a value, but will never result in any runtime error.*

**Proof** *: Follows from a subject reduction condition that is ensured for each totally-safe program code. Given any pair* $\langle s_1, e_1\rangle$ *such that* $s_1$ *is consistent with* $\phi$. *If the judgement* $\vdash \{\phi\}\, e_1\, \{\phi_1\}$ *holds, we can always show that either* $e_1$ *is a value or* $\langle s_1, e_1\rangle \hookrightarrow \langle s_2, e_2\rangle$ *and that* $\vdash \{\phi_2\}\, e_2\, \{\phi_1'\}$ *holds where* $\phi_1' \Rightarrow \phi_1$. *This evaluation either terminates or makes another evaluation step, but it can never fail due to runtime error. The result can be proven by an induction over the small-step semantics of each totally-safe program code.*

*A related property in the context of Chapter 5 is stated by the Corollary 5.6.*

## 3.9 Related Work

Our analysis is potentially useful for software verification and for static analyses based on numerical abstract domains.

Program verification may be performed by generating verification conditions, where their validity implies that the program satisfies its safety assertions. Verification condition generators assume that loop invariants are present in the code, either annotated by the user or inferred automatically. Methods for loop invariant inference include the induction-iteration approach [144] and approaches based on predicate abstraction [56, 92]. Leino and Logozzo [98] designed a loop invariant computation that can be invoked on demand when an assertion from the analyzed program fails. The invariant that is inferred satisfies only a subset of the program's executions on which the assertion is encountered. Comparatively, our method infers a disjunctive formula that is valid for all the program's executions, with each disjunct covering some related execution paths. We achieve this modularly, regardless of any subsequent assertions. Thus, our results can be directly used in the inter-procedural setting.

Partitioning of the abstract domain was first introduced in [44]. Recently, Mauborgne and Rival [103] have given strategies for partition creation and demonstrated their feasibility through their use in Astrée static analyzer [12]. Like them, we make the choice of which disjunctions to keep at analysis time. However, the partitioning criterion is different. In their case, the control flow is used to choose which disjunctions to keep. Specifically, a token representing some conditions on the execution flow is attached to a disjunct, and formulae with similar tokens are hulled together. In our case, the partitioning criterion is based on a property of the disjuncts themselves, with the affinity measure aiming to hull together the most closely related disjuncts.

Various abstract numerical domains have been developed for static analysis based on abstract interpretation. The form of invariants to be discovered is determined by the chosen numerical domain: from the interval domain that is able to discover relations of the form $(\pm x \leq c)$, to the lattice of polyhedra that represents invariants of the form $(a_1x_1 + .. + a_nx_n \leq c)$, all these abstract domains represent conjunctions of linear inequalities. Our pre/post analysis is formalised in a manner that is independent of the abstract domain used. It can therefore readily benefit from advances in constraint

solving techniques for these numerical domains.

## 3.10 Summary

We have proposed a new method for inferring disjunctive postconditions. Our approach is based on the notion of selective hulling as a means to implement adjustable precision in our analysis. We introduced a simple but novel concept called affinity and showed that planar affinity is superior to a recently introduced method based on Hausdorff distance. We have built a prototype system for disjunctive inference and have proven its correctness. Our experiments demonstrate the utility of the disjunctive postconditions for proving a class of runtime checks safe at compile-time, and the potential for tradeoff between precision and analysis cost.

# CHAPTER IV

# DERIVING PRECONDITIONS FOR MODULAR STATIC ANALYSIS

Array bound check optimization has been extensively investigated over the last three decades [144, 43, 71], with renewed interests as recently as [13, 149, 52, 145, 116]. While the successful elimination of bound checks can bring about measurable efficiency gain, the importance of check optimization goes beyond this direct gain. In safety-oriented languages, such as Java, all bound violation must be faithfully reported under precise exception handling mechanism. Thus, check optimization is even more important for run-time efficiency under such constraints. For example, the code motion technique is severely hindered by potential array bound violations.

Most array optimization techniques (*e.g.* [144, 43, 147]) focus on the elimination of totally redundant checks. To achieve this, whole program analysis is carried out to propagate analysis information (*e.g.* availability) to each program point. Even for techniques that handle partially redundant checks, such as partial redundancy elimination (PRE)[14], the focus has been on either moving these checks or restructuring the control flows, but without exploiting path-sensitivity or interprocedural relational analysis. These features are important for supporting precise analyses.

In this chapter, we propose a practical approach towards array bound checks optimization that is both precise and efficient. Our approach is based on the derivation of a suitable precondition for each array check across the method boundary, followed by program specialization to eliminate array checks found to be redundant. Successful elimination of array checks depends on how accurately we are able to infer the states of the program variables. To achieve accuracy, we employ the disjunctive abstract domain developed in Chapter 3. We formalise our technique as a type system that uses type annotations for communicating information between the inference and the specialization phases. We use a form of dependent type [84, 147, 27] that can capture symbolic program states using a relational analysis. The key contributions described in this chapter

include:

- **Forward with Backward Combination :** We propose a novel combination of forward plus backward analysis that can be practical and precise. This combination performs the more expensive forward fix-point analysis only once per method, but proceeds to derive individual safety precondition for each check across procedural boundary. We provide the first formalization and implementation of this combination technique for an imperative language. (Sections 4.3 and 4.4)

- **Indirection Arrays :** Our approach can analyse the *bounds of elements* inside an array. This is important for eliminating array checks for a class of programs where indexes are kept inside indirection arrays (Section 4.5). Past techniques on array bound checks elimination have largely ignored this aspect.

- **Smaller Preconditions :** To obtain a practical analysis, we devise a new technique to make *formulae smaller* by suitable strengthening of preconditions (Section 4.6). This approach trades (some) precision for speed and has been vindicated by experiments with our prototype inference system.

- **Integration with Specializer:** We adopt a *summary-based approach* that gathers preconditions, postcondition and unsafe checks for each method. While summary-based techniques have already been proposed for a number of program analyses [19, 40, 148], their integration with program specializer is hardly investigated. We show how a *flexivariant* specializer could be used to insert runtime test for each array check that has been classified as unsafe (Section 4.7).

- **Prototype :** To confirm the viability of our approach, we have built a prototype inference and specializer system (Section 4.8).

## 4.1 Overview

A key feature of our approach is the three-way classification of checks. Given a method definition with a set of parameters $V$ and a set of checks $C$, our approach will classify each check ($c \in C$) that occurs at a location with a symbolic program state $s$, as follows:

- $c$ is *safe* if it is redundant under the program state $s$ at the location of this check. This holds if the following is valid:

$$(s \Rightarrow c)$$

- $c$ is *partially-safe* if it may become redundant under an extra condition. This holds if there exists a satisfiable precondition $pre$ (expressed in terms of variables from only $V$) such that:

$$(pre \wedge s \Rightarrow c) \tag{4.1}$$

  The precondition can be derived using $pre = (\forall L \cdot \neg s \vee c)$, where $L$ is the set of local variables, denoted by $vars(s, c) - V$. The function *vars* returns the free variables used in $s$ and $c$.

- $c$ is *unsafe*, if `false` is the only precondition that can be found to satisfy (4.1). In this case, the analysis will (conservatively) conclude that the check $c$ may fail at runtime.

Partially-safe checks are special in that they can be propagated across methods from callees to callers. This mechanism can further exploit the program states at callers' sites for the elimination of checks. While the above classification is general and may be applicable to any kind of checks, in this chapter we shall be focusing exclusively on array-related checks.

Let us highlight the above check classification using the `foo` example at the top of Figure 4.1. In this example, `randInt` returns a random integer, while `abs` converts each number into its positive counterpart. The set of parameters $V$ at method boundary is $\{\texttt{a}, \texttt{j}, \texttt{n}\}$ where `a` is an array with indices from `0` to `len(a)`$-$`1`. The `foo` method contains two array accesses at locations $\ell_1$ and $\ell_2$. The symbolic program states (`sps`) at these sites may be affected by the type invariants [1], conditionals, imperative updates and by prior calls. Computing the states for the method entry $\ell_0$ and the locations $\ell_1$ and $\ell_2$, we get:

[1] An example of a type invariant is that the size of an array `a`, denoted by `len(a)`, is positive (a design decision we took for our language).

```
Float foo(Float[] a, Int j, Int n)
 ℓ0:{ Float v=0.0; Int i=j+1;
      if (0<i<=n) then v=(ℓ1:a[i]) else ();
      Int m=abs(randInt());
      v+(ℓ2:a[m]) }

              ⇓ Inference

Float foo(Float[Int^s] a, Int^j j, Int^n n)
 where (j≤s−2) ∨ (n≤j ∧ j+1≥s) ;
    {ℓ1.H : (j≤s−2) ∨ (n≤j ∧ j+1≥s)}; {ℓ2.H} {···}

              ⇓ Specialization

Float foo(Float[Int^s] a, Int^j j, Int^n n)
 where (j≤s−2) ∨ (n≤j∧j+1≥s); (j≤s−2) ∨ (n≤j∧j+1≥s)
 ℓ0:{ Float v=0.0; Int i=j+1;
      if (0<i<=n) then v=(ℓ1:a[i])else ();
      Int m=abs(randInt());
      v+(if (m<len(a)) then ℓ2:a[m] else error) }
```

**Figure 4.1:** Inference and specialization : An example

$$\texttt{sps}(\ell_0) \;=\; \texttt{len(a)} > 0$$

$$\texttt{sps}(\ell_1) \;=\; \texttt{sps}(\ell_0) \wedge \texttt{i=j+1} \wedge \texttt{0<i<=n})$$

$$\texttt{sps}(\ell_2) \;=\; \texttt{sps}(\ell_0) \wedge \texttt{i=j+1} \wedge \texttt{m>=0}$$

Based on the earlier classification of checks, we can establish that the low-bound checks (at $\ell_1$ and $\ell_2$) are safe, since:

$$\texttt{sps}(\ell_1) \Rightarrow (\texttt{i} >= 0) \qquad \text{and} \qquad \texttt{sps}(\ell_2) \Rightarrow (\texttt{m} >= 0)$$

For the high-bound checks (denoted by $\ell_1$.H and $\ell_2$.H), we derive (the weakest) pre-conditions through universal quantification of the local variables, as follows:

$$\texttt{pre}(\ell_1.\texttt{H}) = \forall \texttt{i},\texttt{m} \cdot (\neg \texttt{sps}(\ell_1) \vee \texttt{i<len(a)})$$

$$= \forall \texttt{i},\texttt{m} \cdot (\neg(\texttt{len(a)>0} \wedge \texttt{i=j+1} \wedge \texttt{0<i<=n}) \vee \texttt{i<len(a)})$$

$$= \texttt{len(a)<=0} \vee (\texttt{j<=len(a)}-2 \wedge \texttt{1<=len(a)})$$

$$\vee(\texttt{1<=len(a)<=j+1} \wedge \texttt{n<=j})$$

$$\texttt{pre}(\ell_2.\texttt{H}) = \forall \texttt{i},\texttt{m} \cdot (\neg \texttt{sps}(\ell_2) \vee \texttt{m<len(a)})$$

$$= \forall \texttt{i},\texttt{m} \cdot (\neg(\texttt{len(a)>0} \wedge \texttt{i=j+1} \wedge \texttt{m>=0}) \vee \texttt{m<len(a)})$$

$$= \texttt{len(a)<=0}$$

These derived preconditions may be the weakest, but they do not take into account the type invariant and thus are larger than needed. The type invariant $\texttt{len(a)} > \texttt{0}$ can be used to simplify $\texttt{pre}(\ell_2.\texttt{H})$ to `false` and $\texttt{pre}(\ell_1.\texttt{H})$ to ($\texttt{j<=len(a)}-\texttt{2} \vee \texttt{n<=j} \wedge \texttt{j+1>=len(a)}$). The last formula contains a disjunct ($\texttt{j<=len(a)}-\texttt{2}$) for satisfying the check, and a second disjunct ($\texttt{n<=j} \wedge \texttt{j+1>=len(a)}$) for avoiding the check (when the conditional test is unsatisfiable). In general, the simplification may drop disjuncts that violate the type invariant ($\texttt{len(a)<=0}$) or remove conditions already present in the type invariant ($\texttt{len(a)>0}$). We perform each simplification of a formula $\phi_1$ under type invariant $\phi_2$ by the operation (*gist* $\phi_1$ *given* $\phi_2$) introduced in [130]. This *gist* operation yields a simplified term $\phi_3$ such that $\phi_3 \wedge \phi_2 \equiv \phi_1 \wedge \phi_2$. Informally, the operation (*gist* $\phi_1$ *given* $\phi_2$) returns the new information from $\phi_1$ given that $\phi_2$ holds.

While a goal of our analysis is to obtain weaker preconditions for precision, this might impact the scalability of our analysis. To obtain smaller (but stronger) preconditions, we apply a similar simplification based on the *gist* operation, but more aggressive. For example, simplifying $\texttt{pre}(\ell_1.\texttt{H})$ with respect to the program state of the check $\exists \texttt{i} \cdot \texttt{sps}(\ell_1)$ yields a smaller precondition ($\texttt{j<=len(a)}-\texttt{2}$) without the disjunct that allows avoiding the check. Our proposal trades off precision for performance and is crucial for overcoming the intractability of solving large Presburger arithmetic formulae.

One feature of our optimization is its formulation in two stages: *type inference followed by specialization*. The type inference stage processes methods in reverse topological order of the call graph. It computes post-states at each program point, classifies checks and propagates preconditions as new checks at each method boundary. It also marks all unsafe checks. These information are collected for each method declaration: a postcondition $\Delta$, a set of preconditions $\Phi$, a set of unsafe checks $\Upsilon$, and annotated types $\tau_0, .., \tau_k$.

$$\tau_0\ m\ (\tau_1\ v_1, \ldots, \tau_k\ v_k)\ \texttt{where}\ \Delta;\ \Phi;\ \Upsilon\ \{body\}$$

For example, after type inference on the `foo` method, we would obtain the method displayed in the middle of the Figure 4.1, where the unchanged method body is replaced by $\{\ldots\}$. During the actual inference, we use size variables instead of program variables. For example, size variables $s$, $j$ and $n$ denote $\texttt{len(a)}$, `j` and `n` respectively.

The inference result is then used by the specialization stage to insert runtime tests to guard unsafe checks and to derive *target programs that are well-typed*. Well-typed specialised methods are decorated with a postcondition $\Delta$ and a precondition $\phi_{pre}$:

$$\tau_0\ m\ (\tau_1\ v_1, \ldots, \tau_k\ v_k)\ \texttt{where}\ \Delta;\ \phi_{pre}\ \{body\}$$

The precondition $\phi_{pre}$ is a conjunction of checks from $\Phi$ that are guaranteed safe at each call site. For example, if $\texttt{pre}(\ell_1.\texttt{H})$ is found to be safe when analyzing the call sites of method `foo`, we can generate the specialised (and well-typed) method at the bottom of Figure 4.1. Note that $\Delta \equiv \phi_{pre}$ holds for this particular example, but in general the two formulae may be different. This is so as postcondition is computed using over-approximation, while precondition is computed using under-approximation. Moreover, postcondition may capture its method's result(s), but not so for precondition.

Well-typed programs are safe in that no array bound errors are ever encountered by any array access during program execution. This safety property is guaranteed by either the program context (for array checks $\ell_1.\texttt{L}$ and $\ell_2.\texttt{L}$), or the precondition of each method (for array check $\ell_1.\texttt{H}$) or the inserted runtime test (for $\ell_2.\texttt{H}$). In the rest of this chapter, we shall formalise a type inference system to derive well-typed programs for a core imperative language.

## 4.2 An Imperative Language

To formalise our type inference we use as source language Imp, as introduced in Chapter 2 (see Figure 2.1), where types, denoted by `t` do not have annotations. Imp has support for assignments, conditionals, local declarations, method calls, and multidimensional arrays. Typical language constructs, such as multi-declaration block, sequence, calls with complex arguments can be automatically translated to constructs in Imp. In addition, loops can be viewed as syntactic abbreviations for tail-recursive methods, and are supported by our analysis with the help of pass-by-reference parameters.

### 4.2.1 Target Language

The target of our inference system is a corresponding imperative language with dependent types where types may be annotated with size variables. For example, a boolean value can be denoted by $\texttt{Bool}^b$ where $b = 0$ represents `false` and $b = 1$ represents `true`;

$$
\begin{array}{rcl}
meth & ::= & \tau\ mn\ ((\texttt{[ref]}\ \tau\ v)^*)\ \texttt{where}\ \Delta;\Phi;\Upsilon\ \{e\} \\
prim & ::= & \tau\ mn\ ((\tau\ v)^*)\ \texttt{where}\ \Delta;\Phi;\texttt{C} \\
\tau,\hat{\tau} & ::= & \underline{\tau} \mid \underline{\tau}[\texttt{Int}^{s_1},\ldots,\texttt{Int}^{s_k}] \\
\underline{\tau} & \in & \textbf{PrimAnnType} \\
 & ::= & \texttt{Void} \mid \texttt{Int}^{s} \mid \texttt{Bool}^{s} \mid \texttt{Float} \\
\Phi & ::= & \{\ (l^+ : \phi)^*\ \}\ \text{(Labelled Preconditions)} \\
\Upsilon & ::= & \{\ (l^+)^*\ \}\ \text{(Unsafe Checks)} \\
\texttt{C} & ::= & \{\ (l^+ : e)^*\ \}\ \text{(Labelled Runtime Checks)} \\
\ell & \in & \textbf{Label} \\
\ell^+ & ::= & \ell \mid \ell_1.\cdots.\ell_n\ \text{(Label Sequences)} \\
\phi,\Delta & \in & \textbf{Formula}\ \text{(first order theory of linear arithmetic)}
\end{array}
$$

**Figure 4.2:** Inferred Imp$_I$ language

an integer value can be denoted by $\texttt{Int}^n$ with $n$ to denote its integer value, while $\texttt{Float}[\texttt{Int}^s]$ can denote an array of floats with $s$ elements. Input-output relation between size variables from method parameters and result is captured after the `where` keyword:

$\texttt{Int}^r$ `randInt() where true`;...

$\texttt{Int}^r$ `abs(Int`$^a$ `v)`

  `where` $(a{<}0{\wedge}r{=}{-}a \vee a{\geq}0{\wedge}r{=}a){\wedge}(a'{=}a)$;...

$\texttt{Int}^r$ `add(Int`$^a$ `x, Int`$^b$ `y)`

  `where` $(r{=}a{+}b) \wedge \textit{nochange}\{a,b\}$;...

$\texttt{Bool}^r$ `lessThan(Int`$^a$ `x, Int`$^b$ `y)`

  `where` $(a{<}b{\wedge}r{=}1 \vee a{\geq}b{\wedge}r{=}0){\wedge}\textit{nochange}\{a,b\}$;...

Note that `true` for `randInt` signifies that $r$ is unbounded. Also, non-trivial size relations can be supported through disjunctive formulae. The *prime* notation is used to denote the state of size variables at the end of the method. Parameter values that are unchanged across method calls are captured using the notation $nochange\{a,b\}{\equiv}(a'{=}a{\wedge}b'{=}b)$ as a shorthand for "no change in state". This no-change in state occurs mostly for parameters that are passed by value. Pass-by-reference parameters are also supported in our language using the `ref` keyword.

Figure 4.2 summarises a language with dependent type, called Imp$_I$, which is designed to be the target of our inference. Each method declaration captures three information: an input-output relation (postcondition) $\Delta$, a set $\Phi$ that contains a precondition for each partially-safe check, and a set of label sequences $\Upsilon$, each sequence representing the location of an unsafe check. The labels from $\Phi$ and $\Upsilon$ identify call sites from the body of the current method. This is enabled in our language since every method call is *uniquely labelled*. The suffix notation $s^*$ denotes a list of zero or more distinct syntactic terms separated by appropriate separators, while $s^+$ represents a list of one or more distinct syntactic terms.

For a non-recursive method `mn`, the triple $(\Delta, \Phi, \Upsilon)$ can be derived via inference of the method body (since the triple for each method called in `mn` are already inferred.) To support recursive methods, we make use of *constraint abstractions*. For each mutual-recursive method, we first derive a (recursive) constraint abstraction $\mathcal{Q}$ of the form $q\langle n^* \rangle = \phi$. These abstractions are used by fix-point computation to provide a sound and precise analysis for recursive methods. An adaptation of the fix-point approximation from [43] is detailed via examples in Section 4.4. Besides constraint abstractions, our language of constraints contains conjunctions and disjunctions of linear (in)equalities. We make use of a Presburger solver [129] (with support for universal and existential quantifications) to eliminate local variables or simplify formulae.

Primitive methods (denoted by *prim* in Figure 4.2) lack a method body and are instead annotated with a postcondition and a set of preconditions to support type inference. A primitive is also annotated with a set of runtime tests `C` for use by the specializer: if some precondition is not satisfied at a primitive call site, its corresponding runtime test is to be inserted. Array operations are implemented as calls to primitive methods. For example, 1-dimensional array operations with element type $\tau$ are shown below:

$\underline{\tau}[\texttt{Int}^r]$ `newarr`$(\texttt{Int}^s\ \texttt{s}, \underline{\tau}\ \texttt{v})$ `where` $(0<s \wedge r=s \wedge s'=s); \{\texttt{S}{:}\ s>0\}; \{\texttt{S}{:}\ \texttt{s}>0\}$

$\texttt{Int}^r$ `len`$(\underline{\tau}[\texttt{Int}^s]\ a)$ `where` $(r=s \wedge s'=s); \{\}; \{\}$

$\underline{\tau}$ $\texttt{sub}(\underline{\tau}[\texttt{Int}^s]\ \texttt{a}, \texttt{Int}^i\ \texttt{i})$

$\texttt{where}\ \ (0{\leq}i{<}s \wedge \mathit{nochange}\{i,s\});\ \{\texttt{L}\colon 0{\leq}i, \texttt{H}\colon i{<}s\}; \{\texttt{L}\colon 0{\leq}\texttt{i},\ \texttt{H}\colon \texttt{i}{<}\texttt{len}(\texttt{a})\}$

$\texttt{Void}\ \texttt{assign}(\underline{\tau}[\texttt{Int}^s]\ \texttt{a}, \texttt{Int}^i\ \texttt{i}, \underline{\tau}\ \texttt{v})$

$\texttt{where}\ \ (0{\leq}i{<}s \wedge \mathit{nochange}\{i,s\});\ \{\texttt{L}\colon 0{\leq}i,\ \texttt{H}\colon i{<}s\}; \{\texttt{L}\colon 0{\leq}\texttt{i}, \texttt{H}\colon \texttt{i}{<}\texttt{len}(\texttt{a})\}$

The primitive `newarr` returns a new array with all elements initialized to the value `v`, `len` returns the length of the array, `sub` returns an array element from the specified index `i`, while `assign` updates the specified array element with the value `v`. For example, an array access `a[i]` is (automatically) converted to `sub(a, i)`, while an array update `a[i] = v` is converted to the primitive call `assign(a, i, v)`.

$$\frac{\Gamma(v)=\tau \quad \tau_1=\mathit{fresh}(\tau) \quad \phi=\mathit{equate}(\mathit{prime}(\tau),\tau_1)}{V;\Gamma;\Delta\vdash v\rightsquigarrow v::\tau_1,\Delta\wedge\phi,\emptyset,\emptyset}\ [\textbf{Var}]$$

$$\frac{V;\Gamma;\Delta\vdash e\rightsquigarrow e_1::\tau_1,\Delta_1,\Phi,\Upsilon \quad \Gamma(v)=\tau \quad \Delta_2=\mathit{assign}(\Delta_1,\tau,\tau_1)}{V;\Gamma;\Delta\vdash v{:=}e\rightsquigarrow v{:=}e_1::\texttt{Void},\Delta_2,\Phi,\Upsilon}\ [\textbf{Var-Assign}]$$

$$\frac{\begin{array}{c}\Gamma(v)=\texttt{Bool}^b \quad V;\Gamma;\Delta\wedge(b'=1)\vdash e_1\rightsquigarrow e_3::\tau_1,\Delta_1,\Phi_1,\Upsilon_1\\ \tau=\mathit{fresh}(\tau_1) \quad V;\Gamma;\Delta\wedge(b'=0)\vdash e_2\rightsquigarrow e_4::\tau_2,\Delta_2,\Phi_2,\Upsilon_2\\ \rho_i=\mathit{rename}(\tau_i,\tau)\ \ \forall i\in\{1,2\} \quad \Delta_3=\rho_1\Delta_1\vee\rho_2\Delta_2 \quad e_5=\texttt{if}\ v\ \texttt{then}\ e_3\ \texttt{else}\ e_4\end{array}}{V;\Gamma;\Delta\vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rightsquigarrow e_5::\tau,\Delta_3,\Phi_1{\cup}\Phi_2,\Upsilon_1{\cup}\Upsilon_2}\ [\textbf{If}]$$

$$\frac{\begin{array}{c}\Gamma(v_i)=\tau_i\ \forall i\in 1..n \quad \tau=\ \mathit{fresh}(\hat{\tau}) \quad U=\bigcup_{i=1}^k V(\tau_i) \quad \ell=\mathit{fresh}()\\ (\hat{\tau}\ m(\hat{\tau}_1\ x_1,\ldots,\hat{\tau}_n\ x_n)\ \texttt{where}\ \Delta_m;\ \Phi_m;\cdots)\in P\cup P_m \qquad \Phi_m=\{(\ell_1^+:\phi_1),..,(\ell_k^+:\phi_k)\}\\ \rho=\mathit{rename}(\hat{\tau},\tau)\uplus\Sigma_{i=1}^n\{\mathit{rename}(\hat{\tau}_i,\tau_i)\} \qquad \mathit{pre}_i\equiv_s(\Delta\lll\rho\phi_i)\downarrow_V\\ \mathit{mkChk}(\mathit{pre}_i,\ell.\ell_i^+,\hat{\Phi}_i,\Upsilon_i)\ \forall i\in 1..k \qquad \hat{\Phi}=\bigcup_{i=1}^k\hat{\Phi}_i \quad \Upsilon=\bigcup_{i=1}^k\Upsilon_i\end{array}}{V;\Gamma;\Delta\vdash m(v_{1..n})\rightsquigarrow\ell:m(v_{1..n})::\tau,\Delta\circ_U\rho(\Delta_m),\hat{\Phi},\Upsilon}$$

$$\frac{\begin{array}{c}md=\texttt{t}\ m(\texttt{t}_1\ v_1,\ldots,\texttt{t}_k\ v_k)\ \{e\} \qquad \tau_i=\mathit{fresh}(\texttt{t}_i)\ \forall\ i=1..n \quad \tau=\mathit{fresh}(\texttt{t})\\ V=\bigcup_{i=1}^k V(\tau_i) \quad W=V\cup V(\tau) \qquad \Gamma=\{v_1:\tau_1,\ldots,v_k:\tau_k\}\\ \Delta_{init}=\mathit{init}(\Gamma) \quad V;\Gamma;\Delta_{init}\vdash e\ \rightsquigarrow\ e_1::\tau,\Delta,\Phi,\Upsilon \quad \mathcal{Q}=\{m\langle W\rangle=\Delta\}\end{array}}{\vdash_I md\rightsquigarrow\tau\ m(\tau_1\ v_1,\ldots\tau_k v_k)\ \texttt{where}\ m\langle W\rangle;\Phi;\Upsilon\{e_1\}\mid\mathcal{Q}}$$

$$\frac{\mathit{pre}\equiv\ \texttt{true}}{\mathit{mkChk}(\mathit{pre},\ell^+,\emptyset,\emptyset)}\ [\textbf{MkChk-1}] \qquad \frac{\mathit{pre}\equiv\ \texttt{false}}{\mathit{mkChk}(\mathit{pre},\ell^+,\emptyset,\{\ell^+\})}\ [\textbf{MkChk-2}]$$

$$\frac{\neg(\mathit{pre}\equiv\texttt{true}\vee\mathit{pre}\equiv\texttt{false})}{\mathit{mkChk}(\mathit{pre},\ell^+,\{\ell^+:\mathit{pre}\},\emptyset)}\ [\textbf{MkChk-3}]$$

**Figure 4.3:** Type inference rules

## 4.3 Type Inference Rules

Our inference system analyses and propagates state information so as to determine if an array check is *safe* and if a *precondition* is to be propagated to the *method boundary*. The type judgment for the entire program is $P_m \vdash_I P \rightsquigarrow P_I$. It derives a program $P_I \in \text{IMP}_I$ from a program $P \in \text{IMP}$ and a set of primitive declarations $P_m$.

The type judgement for expressions is specified as follows:

$$V; \Gamma; \Delta \vdash e \rightsquigarrow e_1 :: \tau, \Delta_1, \Phi, \Upsilon$$

Here $V$ is a set of size variables (called *boundary variables*) available at the boundary of the method in which the expression $e$ resides. $\Gamma$ is a type environment mapping program variables to their annotated types. The above judgement states that $e$ will be transformed into $e_1$ during the inference: the target expression $e_1$ will contain types annotated with fresh size-variables and labels that uniquely identify method calls. Both $e$ and $e_1$ have the same underlying type. Furthermore, successful evaluation of $e$ (and $e_1$) requires the validity of preconditions $\Phi$, and the inclusion of the runtime tests $\Upsilon$. Successful evaluation of $e$ also changes the program state from $\Delta$ to $\Delta_1$.

For convenience, our inference rules ensure that the size variables occurring in the annotated type $\tau$ are unique; *ie.,* $V(\tau) \cap V(\Gamma) = \emptyset$ where $FSV$ returns the set of free size variables found. Some of the interesting inference rules are specified in Figure 4.3. In these rules, we use $s = \mathit{fresh}()$ and $\ell = \mathit{fresh}()$ to generate a new size variable and a new label, respectively. For annotated types, $\hat{\tau} = \mathit{fresh}(\mathtt{t})$ (or $\hat{\tau} = \mathit{fresh}(\tau)$) returns a new type $\hat{\tau}$ with the same underlying type as $\mathtt{t}$ (or $\tau$), but annotated with fresh size variables. The function $\mathit{equate}(\tau_1, \tau_2)$ generates equality constraints for the corresponding size variables of its two arguments, assuming both arguments share the same underlying type. For example, we have $\mathit{equate}(\mathtt{int}^n, \mathtt{int}^{m'}) = (n = m')$. The function $\mathit{rename}(\tau_1, \tau_2)$ returns a mapping instead, e.g. $\mathit{rename}(\mathtt{int}^n, \mathtt{int}^{m'}) = (n \mapsto m')$. A conditional constraint is expressed as $\zeta_1 \lhd b \rhd \zeta_2 =_{df}$ if $b$ then $\zeta_1$ else $\zeta_2$. For the rest of this section, we highlight the important aspects of our inference system via examples.

### 4.3.1 Inferring Imperative Update

Let us consider an expression $\mathtt{v} := \mathtt{v} + \mathtt{u}$, a pre-state formula $\Delta = (m'{=}2{+}n' \wedge n'{=}5)$ and the type environment $\Gamma = \{\mathtt{u} :: \mathtt{Int}^m, \mathtt{v} :: \mathtt{Int}^n, \ldots\}$ . This example shows how the *prime*

notation is used to capture the latest values of size variables at each symbolic state. It also shows how updates are effected by a sequential composition operator, $\circ_X$, where $X$ denotes a set of size variables that are being updated.

The following depicts the inference step for assignment:

$$\frac{\Gamma(\mathtt{v}) = \mathtt{int}^n \quad \Gamma(\mathtt{u}) = \mathtt{int}^m \\ V;\Gamma;\Delta \vdash \mathtt{v}+\mathtt{u} \rightsquigarrow \mathtt{v}+\mathtt{u} :: \mathtt{int}^r, \Delta \wedge r = n'+m', \emptyset, \emptyset \\ \Delta_2 = \mathit{assign}(\Delta \wedge r = n'+m', \mathtt{int}^n, \mathtt{int}^r)}{V;\Gamma;\Delta \vdash \mathtt{v} := \mathtt{v}+\mathtt{u} \rightsquigarrow \mathtt{v} := \mathtt{v}+\mathtt{u} :: \mathtt{void}, \Delta_2, \emptyset, \emptyset}$$

The function *assign* performs the necessary sequential composition:

$$\begin{aligned} \mathit{assign}(\Delta, \tau, \tau_1) =_{def}\ & \mathit{let}\ X = V(\tau)\ ;\ Y = V(\tau_1) \\ & \mathit{in}\ \exists Y.(\Delta \circ_X \mathit{equate}(\mathit{prime}(\tau), \tau_1)) \end{aligned}$$

For our example, the correct post-state of the assignment can be computed as follows:

$$\begin{aligned} \Delta_2 =\ & \exists r \cdot ((\Delta \wedge r{=}n'{+}m')\ \circ_{\{n\}} (n'{=}r)) \\ =\ & \exists r \cdot ((m'{=}2{+}n' \wedge n'{=}5 \wedge r{=}n'{+}m')\ \circ_{\{n\}} (n'{=}r)) \\ =\ & \exists r \cdot (\exists n_0 \cdot m'{=}2{+}n_0 \wedge n_0{=}5 \wedge r{=}n_0{+}m' \wedge n'{=}r) \\ =\ & (m'{=}7 \wedge n'{=}m'{+}5) \end{aligned}$$

More formally, sequential composition is defined as:

$$\begin{aligned} & \phi_1 \circ_X \phi_2 =_{def} \exists R \cdot \rho_1(\phi_1) \wedge \rho_2(\phi_2) \\ & \text{where}\quad X = \{s_1, \ldots, s_n\} \text{ are size variables being updated} \\ & \qquad\ R = \{r_1, \ldots, r_n\} \text{ are fresh size variables} \\ & \qquad\ \rho_1 = \{s'_i \mapsto r_i\}_{i=1}^n \qquad \rho_2 = \{s_i \mapsto r_i\}_{i=1}^n \end{aligned}$$

### 4.3.2 Path Sensitive Inference

The [**If**] rule attempts to track the size constraint of conditionals with path sensitivity. The two conditional branches are distinguished by assuming the conditional-test result to be either 1 or 0, representing the `true` or the `false` value, respectively. Given $\mathtt{e} = \mathtt{if\ u\ then\ v\ else\ 5}$ and $\Gamma = \{\mathtt{v} :: \mathtt{int}^n, \mathtt{u} :: \mathtt{bool}^b\}$, the rule derives $\Delta_3$ combining via disjunction the inference results of both branches. We replace both $r_1$ and $r_2$ (the

resulting sizes from both branches) by the final resulting size $r$.

$$\frac{\begin{array}{c}\Delta_1 = \Delta \wedge (b'{=}1) \qquad \Delta_2 = \Delta \wedge (b'{=}0) \\ V; \Gamma; \Delta_1 \vdash \mathtt{v} \rightsquigarrow \mathtt{v} :: \mathtt{int}^{r_1}, \Delta_1 \wedge (r_1{=}n'), \emptyset, \emptyset \\ V; \Gamma; \Delta_2 \vdash \mathtt{5} \rightsquigarrow \mathtt{5} :: \mathtt{int}^{r_2}, \Delta_2 \wedge (r_2{=}5), \emptyset, \emptyset \\ \Delta_3 = \Delta \wedge ((b'{=}1 \wedge r{=}n') \vee (b'{=}0 \wedge r{=}5))\end{array}}{V; \Gamma; \Delta \vdash \mathtt{e} \rightsquigarrow \mathtt{e} :: \mathtt{int}^{r}, \Delta_3, \emptyset, \emptyset}$$

### 4.3.3 Precondition for Safety of Check

Precondition derivation is essential for the detection of safe checks across method boundaries. A check is proved safe when a call context implies the call's preconditions. Otherwise, the preconditions associated with a call are replaced by preconditions associated with its caller. The generated preconditions are expressed in terms of the boundary variables. The [**Call**] rule formalizes this process.

As an example, consider inferring a primitive call $\mathtt{sub(z, j)}$ under the type assumption $\Gamma = \{\mathtt{v} :: \mathtt{int}^v, \mathtt{z} :: \mathtt{Float}[\mathtt{int}^m], \mathtt{j} :: \mathtt{int}^j\}$ and the pre-state $\Delta = (m'{=}m \wedge m'{=}10 \wedge j'{=}v'{+}2 \wedge v'{=}v{+}1 \wedge v'{>}5)$. Furthermore, let the set of boundary variables $V$ be $\{v, m\}$ and $\mathtt{j}$ be a local variable. The two array-bound checks of the `sub` primitive, $0{\leq}i$ and $i{<}s$, are transformed into the following preconditions:

$$\begin{aligned} pre_1 &= (\Delta \approx\!\gg \rho(0 \leq i))\downarrow_V \equiv_s \mathtt{true} \\ pre_2 &= (\Delta \approx\!\gg \rho(i < s))\downarrow_V \equiv_s (v < 7) \end{aligned}$$

where $\rho = \{s \mapsto m, s' \mapsto m', i \mapsto j, i' \mapsto j'\}$. The substitution $\rho$ replaces the size variables associated with the formal parameters of `sub` with those from the actual parameters of the call. The new preconditions are obtained by simplifying ($\equiv_s$) the result of the operations ($\approx\!\gg$) and $\downarrow_V$. The operator $\approx\!\gg$ formulates the implication of an array-bound check by the corresponding calling context. It ensures that all size variables are expressed in terms of those of the call arguments, and primed variables are used in the post-state of the caller:

$$\Delta \approx\!\gg \phi =_{def} (\Delta \Rightarrow \rho(\phi)) \text{ where } \begin{array}{l}\rho = \{s_1 \mapsto s'_1, \ldots, s_n \mapsto s'_n\}; \\ \{s_1, \ldots, s_n\} = V(\phi)\end{array}$$

The operator $\downarrow_V$ projects a constraint to the boundary variable set $V$ through quantification of (size variables from) the local variables. These variables are universally quantified, so that the resulting precondition is strengthened (weakening via $\exists$ quantifier is unsound in this case):

$$\phi\downarrow_V =_{def} \forall W \cdot \phi \;\text{ where } W = FSV(\phi) - V.$$

After its derivation, each precondition is classified by the relation $mkChk(pre, A, B, C)$ to determine if the corresponding array bound check can be eliminated safely, be left as runtime check, or decided at a later stage (a partially-safe check). Here, $A$ is a label sequence leading to the specific bound-check, $B$ outputs the check if it is partially-safe, and $C$ outputs the label sequence identifying the check if it should be left at runtime. For the example above, we have $mkChk(pre_1, \ell.\mathtt{L}, \emptyset, \emptyset)$ and $mkChk(pre_2, \ell.\mathtt{H}, \{\ell.\mathtt{H} : pre_2\}, \emptyset)$, where $\ell$ is a new label associated with the call $\mathtt{sub}(\mathtt{z}, \mathtt{j})$. These $mkChk$ clauses indicate that the low-bound check is safe, while the upper-bound check is partially safe.

For recursive methods, we first employ a fixed-point computation to derive both the method postcondition and a recursive invariant. The invariant captures a size relation to relate the parameters of an arbitrary-nested recursive call with those of the first call. Once the postcondition and the invariant are determined, we can compute the program state at each program point and derive preconditions similarly to the non-recursive case. Details are given next.

## 4.4 Recursion Analysis

Our type inference rules effectively determine both a postcondition and a set of preconditions for non-recursive methods. For recursive methods, these rules derive a (recursive) constraint abstraction that can be analyzed via fix-point analysis. The analysis steps are: (i) determine a fix-point for the constraint abstraction, and derive the method postcondition, (ii) determine an invariant for the recursive calls, and (iii) derive preconditions for checks inside recursion.

### 4.4.1 Deriving Postcondition

The postcondition can be derived from a recursive constraint via a fix-point approximation procedure pioneered in [43] and adapted for a disjunctive domain in [138, 123]. Let

us consider a constraint abstraction of the form $q\langle n^*, r\rangle$ where $n^*$ denote inputs, while $r$ denotes its output. For simplicity and without loss of generality, let us assume we have a constraint abstraction with two recursive invocations of the following form.

$$q\langle n^*, r\rangle = \phi_0 \vee \phi_1[q\langle s^*, r_1\rangle, q\langle t^*, r_2\rangle]$$

Note that $\phi_1[\_, \_]$ is a formula with two holes containing the two recursive invocations, while $\phi_0$ is the base case. The fix-point of such an abstraction can be formalised by the following series:

$$q_0\langle n^*, r\rangle = \texttt{false}$$
$$q_{i+1}\langle n^*, r\rangle = \phi_0 \vee \phi_1[q_i\langle s^*, r_1\rangle, q_i\langle t^*, r_2\rangle]$$

For the above fix-point series to converge, we perform approximations via two techniques, known as *hulling* and *widening*.

Hulling approximates a set of disjuncts $\bigvee \phi_i$ with a conjunct $\phi$ such that $(\bigvee \phi_i) \Rightarrow \phi$. This process can be refined by hulling selectively a subset of closely-related disjuncts. We use the notion of affinity to characterize how closely related is a pair of disjuncts [123]. This selective hulling process is denoted by $\bigvee \phi_i \equiv_h \phi$.

Conjunctive widening takes a formula $\bigwedge \phi_i$ and drops (by replacing with `true`) those constraints $\phi_i$ that are changed compared to the previous step. To apply the widening operator to a disjunctive formula, we first look for pairs of disjuncts (from the current and the previous step) to widen and then apply the conjunctive widening on these pairs [123]. Let us denote widening by $\equiv_w$. We shall apply each fix-point approximation until we obtain a formula $q_p\langle n^*, r\rangle$ such that $q_{p+1}\langle n^*, r\rangle \Rightarrow q_p\langle n^*, r\rangle$. This test indicates that a post fix-point $q_p\langle n^*, r\rangle$ has been reached.

Consider the simple summation program from Figure 4.4, where the constraint abstraction obtained from our inference rules is also given. To obtain a closed-form post-condition, we apply fix-point analysis starting with `false`, the least element of the disjunctive polyhedron domain. Due to the use of widening, such fix-point approximation *always terminates*. For brevity, we display related constraints like $(j{-}1{\leq}i \wedge 0{\leq}i \wedge i{\leq}j)$ using the abbreviated form $(j{-}1, 0{\leq}i{\leq}j)$.

$\mathit{sumvec}_0\langle s, i, j\rangle$ = false

$\mathit{sumvec}_1\langle s, i, j\rangle$ = $(i{>}j) \vee (i{\leq}j \wedge 0{\leq}i{<}s \wedge (\exists i_1 \cdot i_1{=}i{+}1 \wedge \texttt{false}))$

**Methods with Postconditions:**

```
Float sumvec(Float[Int^s] a,Int^i i,Int^j j)
  where sumvec⟨s,i,j⟩,...
{ if i>j then 0.0 else {Int v= ℓ1:sub(a,i);
                  v+ℓ2:sumvec(a,i+1,j) } }
Float sum(Float[Int^s] a) where sum⟨s⟩,...
{ Int l=ℓ3:len(a); ℓ4:sumvec(a,0,l-1)  }
```

**Constraint Abstraction :**

$\mathit{sumvec}\langle s,i,j\rangle \equiv (i{>}j)\vee(i{\leq}j \wedge 0{\leq}i{<}s \wedge \mathit{sumvec}\langle s,i{+}1,j\rangle)$

**Figure 4.4:** Summation program

$$
\begin{aligned}
&= (i{>}j)\\
\mathit{sumvec}_2\langle s,i,j\rangle &= (i{>}j)\vee(i{\leq}j \wedge 0{\leq}i{<}s \wedge (\exists i_1{\cdot}i_1{=}i{+}1{\wedge}i_1{>}j))\\
&= (i{>}j)\vee(0{\leq}i{<}s \wedge i{=}j)\\
\mathit{sumvec}_3\langle s,i,j\rangle &= (i{>}j)\vee(i{\leq}j \wedge 0{\leq}i{<}s \wedge (\exists i_1{\cdot}i_1{=}i{+}1\\
&\quad \wedge\ (i_1{>}j \vee (0{\leq}i_1{<}s \wedge i_1 = j))))\\
&= (i{>}j)\vee(0{\leq}i{<}s{-}1 \wedge j{=}i{+}1)\vee(0{\leq}i{\leq}s{\wedge}i{=}j)\\
&\equiv_h\ (i{>}j)\vee(j{-}1, 0{\leq}i{\leq}j{<}s)\\
\mathit{sumvec}_4\langle s,i,j\rangle &= (i{>}j)\vee(i{\leq}j \wedge 0{\leq}i{<}s \wedge (\exists i_1{\cdot}i_1{=}i{+}1\\
&\quad \wedge\ (i_1{>}j \vee (0{\leq}i_1{<}s{-}1 \wedge j{=}i_1{+}1)\vee(0{\leq}i_1{\leq}s{\wedge}i_1{=}j)))))\\
&\equiv_h\ (i{>}j)\vee(j{-}2, 0{\leq}i{\leq}j{<}s)\\
&\equiv_w\ (i{>}j)\vee(0{\leq}i{\leq}j{<}s)\\
\mathit{sumvec}_5\langle s,i,j\rangle &= (i{>}j)\vee(i{\leq}j \wedge 0{\leq}i{<}s \wedge (\exists i_1{\cdot}i_1{=}i{+}1\\
&\quad \wedge\ (i_1{>}j \vee (0{\leq}i_1{\leq}j{<}s))))\\
&= (i{>}j)\vee(0{\leq}i{\leq}j{<}s)
\end{aligned}
$$

*Fix-Point Detected:* $\mathit{sumvec}_5\langle s,i,j\rangle \Rightarrow \mathit{sumvec}_4\langle s,i,j\rangle$

We reach the following fix-point in five iterations:

$$\mathit{sumvec}\langle s,i,j\rangle = (i{>}j) \vee (0{\leq}i{\leq}j{<}s)$$

### 4.4.2 Deriving Recursive Invariant

Within each recursive method, we may have checks that must be optimized. To deal with this, we compute another constraint, but this time, for just the input parameters (excluding the results of method). More specifically, we build a one-step size relation

to relate the parameters of the next recursive calls with those of the first call. This relation is then analysed via fix-point analysis to derive a multi-steps relation, known as *recursive invariant*. The latter can relate the parameters of an arbitrary recursive call with those of the first call.

One-step relation can be directly extracted from each recursive constraint abstraction. Given the earlier constraint abstraction with two recursive invocations, $q\langle n^*, r\rangle = \phi_0 \vee \phi_1[q\langle s^*, r_1\rangle, q\langle t^*, r_2\rangle]$, we can obtain a one-step relation, named $I$, that attempts to relate the input $n^*$ with that of its recursive call, $\hat{n}^*$, as shown below.

$$I\langle n^*, \hat{n}^*\rangle = \phi_1[\bigwedge(s = \hat{n})^*, q\langle t^*, r_2\rangle] \vee \phi_1[\bigwedge(t = \hat{n})^*, q\langle s^*, r_1\rangle]$$

With this relation, we can now apply fix-point analysis to obtain:

$$I_1\langle n^*, \hat{n}^*\rangle = I\langle n^*, \hat{n}^*\rangle$$
$$I_{i+1}\langle n^*, \hat{n}^*\rangle = I_i\langle n^*, \hat{n}^*\rangle \vee (\exists z^* \cdot I_i\langle n^*, z^*\rangle \wedge I\langle z^*, \hat{n}^*\rangle)$$

We derive the following recursive invariant via fix-point analysis:

$$sumvecI\langle s, i, j, \hat{s}, \hat{i}, \hat{j}\rangle = (\hat{s}{=}s) \wedge (\hat{j}{=}j) \wedge (0 \leq i < \hat{i} \leq s, j{+}1)$$

The recursive invariant is important for deriving safety preconditions of checks inside recursive methods, as elaborated next.

### 4.4.3 Deriving Precondition

Our inference can derive preconditions for checks inside recursion. Due to recursion, such checks may be encountered multiple times. We propose to separate out the check of the *first* recursive call from the checks of the *rest* of the recursive calls. The reason for this is that recursive invariant that we derive is applicable to all recursive calls, except the first. Consequently, the program state for the first check and the program state for the recursive checks are different. More specifically, consider a check $c$ labelled as $\ell$ at program context $s$ in a recursive method $m$ with invariant $i$. Its two preconditions can be derived as follows:

$$\texttt{preFst}(\ell) \;=\; \forall L \cdot (s \Rightarrow c) \;\texttt{where}\; L = \mathit{vars}(s, c) - V$$
$$\texttt{preRec}(\ell) \;=\; \forall L \cdot (s \wedge i \Rightarrow c) \;\texttt{where}\; L = \mathit{vars}(s, c, i) - V$$

For the *sumvec* example, we would derive two sets of preconditions, namely:

$$\texttt{preFst}(\ell_1.\texttt{L}) = (j{<}i){\vee}(0{\leq}i)$$
$$\texttt{preFst}(\ell_1.\texttt{H}) = (j{<}i){\vee}(i{<}s)$$
$$\texttt{preRec}(\ell_1.\texttt{L}) = \texttt{true}$$
$$\texttt{preRec}(\ell_1.\texttt{H}) = (j{<}s){\vee}(s{\leq}j{\wedge}i{\leq}{-}1){\vee}(s{\leq}j,i)$$

These preconditions are propagated to the caller of each *sumvec* call. Note that the precondition for (rest of the) recursive checks for $\ell_1.\texttt{L}$ is totally safe, but the first check of $\ell_1.\texttt{L}$ can be guarded by a condition $(j{<}i){\vee}(0{\leq}i)$. These different scenarios of array checks can be exploited by program specialization, so as to maximise the elimination of redundant checks whilst being mindful of the potential for code explosion. We describe such a specialization process in Section 4.7.

## 4.5 Array Indirections

There is a class of programs which has been largely ignored in past work on array bound checks elimination. This class of programs uses indexes that are stored in another array (*indirection array*). Array indirections are used intensively for implementing sparse matrix operations. For such matrices, only nonzero elements are stored; Additionally, the indices of these elements are kept inside an indirection array. Luján *et al* [101] proposed a solution to handle indirection arrays via a runtime mechanism. Our system handles indirection arrays and relies entirely on compile-time analysis.

To support programs with indirection arrays, the bounds of their elements will have to be captured using an additional size variable $a$ via a new annotated type for integer array $\texttt{Int}^a[\texttt{Int}^s]$. Precise tracking will allow us to analyse the indexes retrieved from such integer arrays. As the array elements are being changed by the `assign` primitive, their bounds may also change during program execution. Such size properties are therefore *mutable*. To handle them safely, we require the support of an alias analysis, such as the one proposed in [79], that could be used to identify may-aliases amongst the integer arrays.

In addition to alias annotation, the main extra machinery is a set of enhanced primitive declarations (preconditions and runtime tests are unchanged, so we replace them

for brevity with ...).

$$\texttt{Int}^a[\texttt{Int}^r]\ \texttt{newarr}(\texttt{Int}^s\ \texttt{s}, \texttt{Int}^v\ \texttt{v})$$

$$\quad \texttt{where}\ (0{<}s \wedge r{=}s \wedge a{=}v \wedge \mathit{nochange}\{s, v\}); \ldots$$

$$\texttt{Int}^r\ \texttt{sub}(\texttt{Int}^a[\texttt{Int}^s]\ \texttt{a}, \texttt{Int}^i\ \texttt{i})$$

$$\quad \texttt{where}\ \ (0{\leq}i{<}s \wedge r{=}a \wedge \mathit{nochange}\{i, s, a\}); \ldots$$

$$\texttt{Void}\ \texttt{assign}(\texttt{Int}^a[\texttt{Int}^s]\ \texttt{a}, \texttt{Int}^i\ \texttt{i}, \texttt{Int}^v\ \texttt{v})$$

$$\quad \texttt{where}\ \ (0{\leq}i{<}s \wedge (a'{=}v \vee a'{=}a) \wedge \mathit{nochange}\{i, s, v\}); \ldots$$

The array elements are updated by the `newarr` and `assign` primitives, and read by the `sub` primitive. In particular, the formula $(a'{=}v \vee a'{=}a)$ captures a weak update operation with a new approximation to the state of elements in the array. Furthermore, we may even track the relation between array indexes and their elements by using the annotated type $\texttt{Int}^{(i,a)}[\texttt{Int}^s]$ with a new size variable $i$ to denote index positions. By using primitives with such type declarations, we can selectively support increased precision for our analysis. Note that both the inference and the specializer work with the above indirection array primitives as well as with the array primitives without indirection from Section 4.2.1.

Let us illustrate how array indirections are analyzed via a simple example that initializes an array with a range of integer values:

```
Void initarr (Int^a[Int^s] a, Int^i i, Int^j j, Int^n n)
    where initarr⟨a, s, i, j, n⟩
{   if i>j then () else {a[i]=n; initarr(a,i+1,j,n+1)} }
```

Using the fix-point analysis described in Section 4.4, we can obtain the following postcondition which captures the initialization of the array elements:

$$\mathit{initarr}\langle a, s, i, j, n\rangle \ \equiv\ (i{>}j \wedge a'{=}a)\ \ \vee\ \ (0{\leq}i{\leq}j{<}s \wedge (a'{=}a \vee n{\leq}a'{\leq}n{+}j{-}i))$$

This postcondition captures an universal property about the elements of the array $a$. The constraint $(a'{=}a)$ indicates there is no change in the value of array elements. Intuitively, it can be expressed in the universal fragment of the theory of arrays [16] using the following constraint: $\forall ix \cdot (0 \leq ix < len(a) \Rightarrow a[ix]' = a[ix])$. The constraint

$(n \leq a' \leq n+j-i)$ captures both a lower and an upper bound for all the array elements, also expressible in the theory of arrays as follows:

$$\forall ix \cdot (0 \leq ix < len(a) \Rightarrow n \leq a[ix]' \leq n+j-i)$$

## 4.6 Deriving Smaller Formulae

An important property of program analysis is efficiency, and this is particularly so for an inference system based on Presburger arithmetic. Presburger arithmetic can give highly accurate analysis (with disjunctions and quantifiers) but has double-exponential complexity, namely $2^{2^{cn}}$ where $n$ is the size of its formulae. A summary-based analysis like ours brings about a smaller number of size variables at each method boundary than a global analysis approach. With this decrease, the main proviso for efficiency is to ensure that the pre and postconditions are kept small in size.

A major reason for large formulae is the presence of disjuncts related to the specification aggregation problem observed in [94]. To counter this effect, a derived postcondition can be weakened through the *hulling* of its disjuncts. However, applying a weakening process is unsound for preconditions! For preconditions, it is only safe to strengthen and we propose a new technique that improves the analysis efficiency at a low cost in precision. We perform the strengthening of the precondition $\phi_{pre}$ using the *gist* operation from the Omega library [130].

Given a check $c$ which occurs at a location with program state $s$ and local variables $V_L$, we have earlier derived the weakest precondition using $\texttt{pre} = (\forall V_L \cdot \neg s \vee c)$. This derived precondition is unsuitable due to the negation of a (possibly very large) program state formula $s$. To derive smaller preconditions, we may simplify `pre` using a valid state $s_1$ for which $(\exists V_L \cdot s) \Rightarrow s_1$ holds.

- One such $s_1$ that can be used is the *type invariant* `inv` at method entry. Let us refer to this technique of using (*gist* `pre` *given* `inv`) as *weak pre-derivation*.
- A second technique is to use $\exists V_L \cdot s$ itself. Let us refer to this technique as *strong pre-derivation*: it uses (*gist* `pre` *given* $\exists V_L \cdot s$) . This technique would strip off all the *avoidance conditions* from the derived precondition, which may result in some loss of precision.

- To recover this loss of precision, we also propose a third technique, called *selective prederivation*, which would first obtain a variant of $\exists V_L \cdot s$ that is weakened by removing conditional tests from $s$.

For example, consider a symbolic program state derived from the recursive `sumvec` method as follows: $\exists\, \hat{i} \cdot s{>}0 \wedge \hat{i} \leq j \wedge (0 \leq i < \hat{i} \leq s, j{+}1)$. After stripping off its conditional test, $\hat{i} \leq j$, we would obtain a weaker state:

$$\exists\, \hat{i} \cdot s{>}0 \wedge (0 \leq i < \hat{i} \leq s, j{+}1)$$

Simplifying the precondition of $(j{<}s) \vee (s \leq j \wedge i \leq -1) \vee (0 \leq i \wedge s \leq j, i)$ with this program state results in a much smaller precondition, namely $j{<}s$, that is obtained by both selective and strong prederivations. This is in contrast to $(j{<}s) \vee (s \leq j \wedge i \leq -1) \vee (s \leq j, i)$ that is obtained by weak prederivation.

In our experiments (see Section 4.8), we tested the three prederivation techniques. When compared to the weak prederivation technique, we were able to reduce the size of preconditions on average by 63.4% for selective prederivation and by 81.8% for strong prederivation. We found the selective prederivation to have a reasonable compromise between efficiency and precision. Furthermore, we achieved a significant reduction in the inference times needed by some larger programs which fail to complete in reasonable (allotted) time, otherwise!

## 4.7 Flexivariant Specialization

The objective of specialization is to place run-time tests (for unsafe checks) at their respective primitive operations with the objective that array operations become safe, *and* the array checks are done minimally. To this end, we specialize the existing method definitions with information about run-time tests.

To understand the effectiveness of various approaches to specializing method definitions, we examine the following example program:

| | | |
|---|---|---|
| `void main()` | `t2 p(···)` | `t1 q(···)` |
| { ··· | { ··· | { ··· |
| $\ell_5$ : `p(···)`; | $\ell_3$ : `q(···)`; | `v1`=($\ell_1$: `sub(a1, i1)`); |
| ··· | ··· | $\ell_2$: `assign(a2, i2, v1)`; |
| $\ell_6$ : `q(···)`} | $\ell_4$ : `q(···)`} | ···} |

Let us assume that the results of inference are as follows:

| Preconditions for `q` | | | |
|---|---|---|---|
| from $\ell_1$ | | from $\ell_2$ | |
| $\ell_1$.L | $\ell_1$.H | $\ell_2$.L | $\ell_2$.H |
| `true` | $\phi_1$ | `true` | $\phi_2$ |

| Preconditions for `p` | | | |
|---|---|---|---|
| from $\ell_3$ | | from $\ell_4$ | |
| $\ell_3.\ell_1$.H | $\ell_3.\ell_2$.H | $\ell_4.\ell_1$.H | $\ell_4.\ell_2$.H |
| `true` | $\phi_3$ | $\phi_4$ | `false` |

| Preconditions for `main` | | | |
|---|---|---|---|
| from $\ell_5$ | | from $\ell_6$ | |
| $\ell_5.\ell_3.\ell_2$.H | $\ell_5.\ell_4.\ell_1$.H | $\ell_6.\ell_1$.H | $\ell_6.\ell_2$.H |
| `true` | `true` | `false` | `false` |

This corresponds to the following inferred method headers with partially-safe and unsafe checks.

`t1 q(···) where` $\Delta_q$, $\{\ell_1.\mathtt{H} : \phi_1, \ell_2.\mathtt{H} : \phi_2\}, \{\}$

`t2 p(···) where` $\Delta_p$, $\{\ell_3.\ell_2.\mathtt{H} : \phi_3, \ell_4.\ell_1.\mathtt{H} : \phi_4\}, \{\ell_4.\ell_2.\mathtt{H}\}$

`void main() where` $\Delta_{main}$, $\{\}, \{\ell_6.\ell_1.\mathtt{H}, \ell_6.\ell_2.\mathtt{H}\}$

Thus, there are three unsafe checks that must be residualized at run-time, namely $\ell_4.\ell_2$.H, $\ell_6.\ell_1$.H and $\ell_6.\ell_2$.H. The other checks are either safe, or partially-safe with the possibility

of becoming safe using the context of the caller. An aggressive approach to eliminating checks is *polyvariant specialization*. This aims at creating *multiple* specialized methods for each method definition, such that each specialized version of a method has a different set of array checks being eliminated. Its application on our example program yields the following result:

```
void main()
{ ···
  p(···);
  ···
  q_3(···)}
```

```
t2 p(···) where .., φ3∧φ4
{ ···
  q_1(···);
  ···
  q_2(···)}
```

```
t1 q_1(···) where .., φ1∧φ2
{ ···
  v1 = (sub(a1, i1));
  assign(a2, i2, v1);
  ···}
```

```
t1 q_2(···) where .., φ1
{ ···
  v1 = sub(a1, i1);
  if (i2 < len(a2)) then
    assign(a2, i2, v1)
  else error ··· }
```

```
t1 q_3(···) where .., true
{ ···
  v1 = (if (i1 < len(a1))
        then sub(a1, i1)
        else error);
  if (i2 < len(a2)) then
    assign(a2, i2, v1)
  else error ··· }
```

Note that three versions of q have been created to handle its three calls under different calling contexts.

In this section, we propose a *flexivariant* program specialization scheme. As special cases, we can either support polyvariant or monovariant specializations. For polyvariance, we can achieve it by never attempting to weaken any of the configurations encountered. For monovariance, we can achieve it by weakening each configuration encountered to its most conservative variant with maximal unsafe checks. For this example, the monovariant case will weaken the configurations of both q_1 and q_2 to q_3. Even though q_3 is the weakest configuration, it still has two low bound checks eliminated.

A key feature of our flexivariant specialization scheme is its ability to trade-off optimization for a reduction in code size. Furthermore, it is possible to achieve such trade-offs with minimal loss in performance. For example, if it can be determined that

`q_1` configuration occurs *infrequently*, we may weaken it into `q_2` to save on code size with little loss in performance.

Flexivariant specialization of a program `P` into an optimized program `S` is declared as follows: $\triangleright_{flex}$ `P` $\rightharpoonup$ `S`. Specializing a method requires information about the set of runtime tests to which calls in the method body may lead. Thus, a specialized method can be identified by a triple comprising the original method name, a set of label sequences associated with the relevant runtime tests, and a new method name uniquely defined by the first two components of the triple. We call such a triple a *specialization signature* (or *signature* in short), and a set containing such signatures a *specialization cache* (or *cache* in short).

$$\begin{aligned} (m, \varsigma, \hat{m}) &\in \mathbf{SSig} = \mathbf{MName} \times \mathbf{LSet} \times \mathbf{MName} \\ \sigma, \sigma_Y, \sigma_N, \sigma\hat{}_N &\in \mathbf{SCache} = \mathcal{P}(\mathbf{SSig}) \\ \varsigma &\in \mathbf{LSet} = \mathcal{P}(\mathbf{Label}^+) \end{aligned}$$

The specialization of an expression is defined by:

$$\mathtt{P},\ \sigma,\ \varsigma\ \triangleright^{e}_{flex}\ \ e\ \rightharpoonup\ e_1,\ \sigma_N$$

The specialization cache $\sigma$ drives the process, while $\varsigma$ contains the checks to be residualized. New specialization points created during specialization are stored in $\sigma_N$. We highlight the most important specialization rules below.

An array operation is specialized in [**Spec–Prim**] by calling the respective primitive method without array checks under the condition that the combined runtime checks for this operation, $e_1$, is true.

$$\frac{\begin{array}{c}[\textbf{Spec–Prim}] \\ \tau\ m(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ \text{ where } \Delta, \Phi, C\ \in\ P_m \\ \rho = [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \\ e_1 = \bigwedge\{\rho\ e \mid\ \ell.c \in \varsigma \wedge (c : e) \in C\} \\ e_2 = \texttt{if}\ e_1\ \texttt{then}\ m(v_1, \ldots, v_k)\ \texttt{else error} \\ e_3 = m(v_1, \ldots, v_k) \lhd (e_1 = \texttt{true}) \rhd e_2 \end{array}}{\mathtt{P},\ \sigma,\ \varsigma\ \triangleright^{e}_{flex}\ \ (\ell : m(v_1, \ldots, v_k)) \rightharpoonup e_3,\ \emptyset}$$

Here, a label sequence of the form $\ell.c$ occurring in the set $\varsigma$ represents an array check to be residualized. Its code is available at the corresponding primitive method declaration.

Variable substitution is needed to residualize the code. All codes thus generated are combined as a conjunct, named $e_1$, which is then wrapped as a runtime test for the primitive call to $m$. If the runtime set is empty – signified by $e_1$ being `true` – the $m$ call will not be wrapped by a conditional.

Similarly, user-defined methods are specialized with respect to the set of runtime tests ([**Spec−Call$_1$**]). Weakening of configurations by $\mathcal{W}$ may enlarge this set of runtime tests. Specialization produces a signature for this specialized method if the latter has not been recorded in the current cache. Otherwise, it reuses the specialised method that has been recorded previously, as specified in [**Spec−Call$_2$**].

$$\frac{\begin{array}{c}[\textbf{Spec}-\textbf{Call}_1]\\ (\tau\ m(\tau_1\ x_1,\ldots\tau_k x_k)\ \texttt{where}\ \Delta,\Phi,\Upsilon\ \{e\})\in \texttt{P}\\ \varsigma_2=\mathcal{W}(m,\varsigma_1)\quad \varsigma_1=\{\ell^+\mid \ell_1.\ell^+\in\varsigma\}\cup\Upsilon\\ (m,\varsigma_2,\_)\notin\sigma\qquad m_s=genName(m,\varsigma_2)\end{array}}{\begin{array}{l}\texttt{P},\ \sigma,\ \varsigma\ \rhd^e_{flex}\ (\ell_1:m(v_1,\ldots,v_k))\\ \qquad\rightharpoonup\ m_s(v_1,\ldots,v_k),\ \{(m,\varsigma_1,m_s)\}\end{array}}$$

$$\frac{\begin{array}{c}[\textbf{Spec}-\textbf{Call}_2]\\ (\tau\ m(\tau_1\ x_1,\ldots\tau_k x_k)\ \texttt{where}\ \Delta,\Phi,\Upsilon\{e\})\in \texttt{P}\\ \varsigma_1=\{\ell^+\mid \ell_1.\ell^+\in\varsigma\}\cup\Upsilon\qquad (m,\mathcal{W}(m,\varsigma_1),m_s)\in\sigma\end{array}}{\texttt{P},\ \sigma,\ \varsigma\ \rhd^e_{flex}\ (\ell_1:m(v_1,\ldots,v_k))\rightharpoonup\ m_s(v_1,\ldots,v_k),\ \emptyset}$$

## 4.8 Experimental Results

On top of the disjunctive fixed-point analyzer (described in Chapter 3), we have constructed a modular inference system together with a program specializer. The output from our system was validated by a separate checking system that we have also built. The entire prototype system was written in Haskell and compiled using Glasgow Haskell compiler[121]. For constraint solving in the Presburger arithmetic domain, we used the Omega library [129, 88]. A web-demo of our system can be found at `http://loris-7.ddns.comp.nus.edu.sg/~popeeaco/imp/`.

We evaluated our prototype using small programs with challenging recursion and two numerical-intensive benchmarks: SciMark (`Fast Fourier Transform, LU decomposition,`

| Programs | Source (lines) | Static Checks | Checking (secs) | Inference (secs) | | | Static Checks Eliminated |
|---|---|---|---|---|---|---|---|
| | | | | Weak | Selective | Strong | |
| binary search | 31 | 2 | 0.17 | 1.84 | 1.81 | 1.79 | 100% |
| bubble sort | 39 | 12 | 0.43 | 1.55 | 1.51 | 1.47 | 100% |
| foo | 12 | 4 | 0.39 | 0.66 | 0.67 | 0.87 | 50%/75% |
| hanoi tower | 38 | 16 | 3.73 | 11.74 | 11.53 | 11.47 | 100% |
| merge sort | 58 | 24 | 7.70 | 11.21 | 16.01 | 13.07 | 100% |
| queens | 39 | 8 | 0.52 | 2.13 | 2.11 | 2.10 | 100% |
| quick sort | 43 | 20 | 0.38 | 1.92 | 1.92 | 1.76 | 100% |
| sentinel | 26 | 4 | 0.05 | 0.18 | 0.16 | 0.15 | 75% |
| sparse multiply | 46 | 12 | 3.27 | 22.61 | 17.37 | 7.09 | 100% |
| sumvec | 33 | 2 | 0.11 | 0.51 | 0.48 | 0.47 | 100% |
| FFT | 336 | 62 | 9.58 | * | 58.02 | 28.74 | 100% |
| LU Decomp. | 191 | 82 | 13.10 | 137.1 | 93.31 | 72.91 | 100% |
| SOR | 84 | 32 | 1.15 | 7.18 | 4.67 | 3.8 | 100% |
| Linpack | 903 | 166 | 42.26 | * | 360.1 | 162.2 | 100% |

**Figure 4.5:** Statistics for array bound checks elimination

`Successive Over-Relaxation`) [112] and Linpack [51]. Our test platform was a Pentium 2.8 GHz system with 1GByte main memory, running Red Hat Linux 9.0.

Our main objective was to show the viability and the precision of the system. Figure 4.5 summarises the statistics obtained for each program that we inferred. To quantify the analysis complexity of the benchmark programs, we counted the program size (column 2) and also the number of static checks present in each program (column 3). The time taken for inference (columns 5-7) includes parsing, preprocessing, modular type inference and specialization. For comparison, we present the time taken for checking pre-annotated programs (column 4), composed from parsing and dependent type checking. The size of the method constraints (preconditions, postconditions and recursive invariants) is on average around 15% of the size of the source program. Thus, our inference eliminates the effort to annotate methods required of programmers with access to only a dependent type checker.

Due to the precision of our inference system, we were able to eliminate 100% of array checks for all the programs we tested, except for `sentinel` and `foo` (column 8). The `sentinel` example illustrates a pattern where some checks cannot be eliminated by our method, since it makes use of a sentinel/guard against falling off one end of the array. Like [149, 147], we were unable to capture the existential property that is required for check elimination. For the `foo` example, strong prederivation and selective prederivation eliminate 50% and 75%, respectively, of the static checks.

We can compare our experimental results to other analyses that are based on disjunctive domains similar to ours, but employ only forward derivation [138, 123]. For the benchmark set used in our previous work [123], a forward derivation and a fixed-point analysis with Hausdorff affinity akin to [138] led to 76% check elimination, while a forward analysis using planar affinity introduced in [123] was able to eliminate 84% of the checks. Compared to these two previous analyses, our current techniques achieve 100% check elimination. We can attribute this improvement to the combination of the forward derivation of postconditions with the backward derivation of preconditions. Another reason for our improved results was the handling of array indirections present in the `sparse multiply` and `Linpack` benchmarks.

In almost all cases, strong prederivation takes less time than selective prederivation, followed by weak prederivation. As an exception, the increased precision of weak prederivation allows a faster analysis of `mergesort`, since some bound checks are proved redundant at an earlier point than the other two prederivation methods. On the other hand, for those larger programs we found it crucial to use either selective or strong prederivation; weak prederivation does not scale up as inference fails to complete in reasonable time (cases denoted by * signify over an hour inference time).

To summarize our experiences, we observe that our initial goal was to build a precise inference system and make it practical by employing a modular analysis that computes method summaries. However, the small number of size variables at each method boundary was not enough to ensure the efficiency of our system. The backward component of our system proved to be expensive mostly due to two reasons. Firstly, precondition derivation was done via negation of a (possibly very large) program state formula. Secondly, array bound checks were specialized by deriving individual preconditions, one for each check. This was our intention in order to enable aggressive program optimization. Note that proving program safety does not necessarily require individual precondition derivation (and, in our setting, can be less expensive). To cope with these additional difficulties, we employed additional approximations to reduce the size of method summaries: weakening of postconditions via selective hulling and strengthening of preconditions via gisting. With these techniques, both the inference and the specializer were integrated into a system that was shown to be practical and precise enough for our purposes.

## 4.9 Correctness

The soundness of our type inference is defined with respect to a type checking system and a specialization process. After type inference (that includes fixed-point analysis), the inferred program must be *specialized* to include the runtime tests discovered during inference, before it becomes well-typed. We state the soundness of our system below and refer the reader to the technical report [126] for details on the proof.

**Theorem 4.1 (Soundness).** *Let $P$ be a program and a type inference judgement such that $(P_m \vdash_I \ P \rightsquigarrow P_I)$. Let $(\triangleright_{flex}\ P_I \rightharpoonup P_T)$ be the specialization of $P_I$ to $P_T$ guided by the inferred runtime tests. Then $P_T$ is well-typed.*

As a special case, if no unsafe check is discovered during inference then $P_I$ is well-typed. However, if unsafe checks are discovered, the use of label sequences (*eg.*, $\ell_6.\ell_1.\mathtt{H}$) to identify array checks also enables *debugging feedback*. Specifically, our analysis can pin-point the exact location of each unsafe check based on the calling hierarchy up until an unsatisfied precondition.

## 4.10 Related Work

Traditionally, data-flow analysis techniques have been employed to gather information for the purpose of identifying redundant array checks [71]. Within the scope of intra-procedural analysis, these techniques are also used to gather anticipatable information for the purpose of hoisting partially-redundant checks to more profitable locations. The techniques have gradually evolved in sophistication, from the use of family of checks in [90], to the use of difference constraints in [13].

To identify redundant checks more accurately, verification-based methods have been used by Suzuki and Ishihata [144], Necula and Lee [113] and Xu *et al* [149]. Xi and Pfenning have advocated the use of dependent types for array bound check elimination [147]. Their approach is limited to totally redundant checks. Moreover, the onus for supplying suitable dependent types rests squarely on the programmers, as only a type checker is available.

Precondition derivation with respect to a postcondition (or check) has been formulated via generating its *Verification Condition* (VC) by Flanagan *et al* [56, 57]. Their focus was to obtain compact VCs whose size is worst-case quadratic to the size of the

source. However, they do not attempt to make preconditions and postconditions any *smaller* through strengthening and weakening, respectively. Furthermore, these VCs are for totally-redundant checks. In contrast, our technique stresses on modularity and deals with inter-procedural analysis over recursive methods, whereas they focus on intra-procedural analysis and loops. Recently, Flanagan [54] introduced the idea of inserting assertions that cannot be proven during type checking as run-time checks. Our use of a flexivariant specializer to insert runtime checks (after inference) shares a similar flavour. However, our proposal is based on inference, while his is formalised for a type-checker.

Identifying redundant array bound checks can also be done using abstract interpretation techniques over numerical domains. In a seminal paper, Cousot and Halbwachs [43] introduced the polyhedra abstract domain and defined convex-hull and widening operators for this domain. Subsequently, various other abstract domains have been proposed, varying from conjunctive weakly-relational domains like octagons [104], pentagons [100] or template constraint matrices [139, Chapter 4] to disjunctive domains [138, 123]. In fact, safety analyzers that scale to large critical programs like Astrée [12] or C Global Surveyor [145] use elaborate combinations of abstract domains to achieve maximum efficiency. For example, the static analyzer that has been described by Cousot *et al* [12, 42] succeeds in analyzing a program of 75 kloc with no false alarm. It achieves this by varying the precision of arithmetic abstract domains from interval domain to ellipsoid domain. It also uses a decision tree abstract domain and trace partitioning for path-sensitivity. These relational domains operate on packs of variables for efficiency reasons. However, our analysis maintains path-sensitivity and the same level of precision over the entire program by exploiting modularity. Being a summary-based approach, we have a bounded number of variables at method boundary and we further ensure that preconditions are kept small via suitable prederivation. Modularity has also been recognized as an important step for static program analyses to scale up to precise analysis of large programs [40] and our proposal is a solution in this direction.

To avoid fix-point iteration, Rugina and Rinard [137] proposed an analysis method (using linear programming) to synthesize polynomial symbolic bounds. While efficient, fixing a target form (without disjunction) for the symbolic bound may result in loss of precision. Dor *et al* advocated for linear constraints, expressed using pre/post conditions,

to help determine the safety of C pointers to string buffers [52]. For their experiments, the inference result is, however, less precise than user-supplied annotations. This is likely due to the absence of disjunction and path-sensitivity during inference.

The idea of deriving preconditions for partially redundant checks was first proposed in [28] to complement postcondition inference on sized types [27] for a first-order functional language. However, this early work was mostly informal and had no implementation. We formalize this early idea by inferring a sound dependent-type annotation for an imperative language, and integrating its results with a program specializer. Moreover, we now have a practical and precise implementation.

Unlike the work in [24] which uses a separate set-based analysis for properties of elements in a collection, our current proposal uses arithmetic constraints to represent such properties directly for indirection arrays. This decision reduces the burden of using two different analyses. On the other hand, the set-based analysis approach [24] may give more precise results via universal and existential properties, and deal with elements which may not be integers.

Flexivariant specialization scheme enables a trade-off to be made, that can give up some array check optimization for a reduction in code size. Such trade-off can be guided with the help of suitable path-profiling techniques[146]. Such a compromise was originally pioneered in a technique, called *selective specialization* [47], to convert expensive dynamic method dispatches for object-oriented programs into static counterparts, where possible. Our flexivariant scheme supports the proposed inference with a family of specializers, with selective specialization as a possible option.

## 4.11 Summary

We have proposed a new inference mechanism for a dependent type system with size relations. Our approach captures postcondition in the presence of imperative updates, and derives safety preconditions for each check encountered. Both the postcondition and safety precondition are propagated interprocedurally, though in opposite directions. Recursive methods are also handled through a fix-point analysis on constraint abstraction derived via inference. The resulting analysis is not only flow and context-sensitive, but is also path-sensitive. It can capture symbolic program states between local variables,

inputs and outputs. Initial experiences with a prototype implementation suggest that such an advanced form of type inference is both precise and efficient. Just as the present analysis is empowered by the use of Presburger arithmetic, it is inevitably limited by the linearity of expressible constraints. However, by first subjecting the original program to pre-processing such as partial evaluation (using constant propagation and loop unrolling), our analysis can discover more linear constraints, and thus further improve its effectiveness.

# CHAPTER V

# DUAL STATIC ANALYSIS

## 5.1 Introduction

Program bugs remain a major challenge for software developers and various tools have been proposed to help with their localization and elimination. Traditionally, program testing and model checking [59, 62, 45] have been applied to detect the presence of real bugs. However, one shortcoming of the testing process is that it is unable to prove the absence of bugs, compromising on program safety. In contrast, static analysis which uses abstraction on program states can be used to prove program safety [43, 138]. It achieves this by showing that bad error states are not reachable via an exhaustive interpretation in the abstract domain. Due to approximation, static analysis may report false positives that are possible bugs that do not exist in practice. High incidents of false positives can make static analysis tools impractical to use for finding and eliminating bugs. This problem is serious enough that Jung et al [87] have resorted to machine learning (that are neither sound nor complete) to heuristically cut down on the numerous false-positives that were reported by their static analyzer. Furthermore, as reported in the Astrée project [12, 136], manual inspection of alarms (possible bugs) can be a very time- consuming process and may take several days even for simple alarms.

Recently, there have been some proposals [120, 150, 67] that advocate for over-approximation techniques (based on static analysis) to be synergistically combined with under-approximation techniques (based on concrete execution or program testing). One main goal of this combination, as advocated in [67], is to leverage on the strengths of the two techniques so that program bugs or their absence can be discovered more accurately and effectively. While such a proposal can exploit the complementary strengths of its constituent techniques, it is also more complex to construct due to the need to combine quite different techniques and to consider potential interplays between them. Furthermore, it is often useful to explore what can be achieved within a single methodology before considering synergistic combinations of different techniques, to allow the

strengths of each technique to be more fully exploited.

In this chapter, we shall propose a dual static analysis that is different from past approaches as both its components are based *primarily on over-approximation*. Our approach is also modular and computes (on a per method basis) trigger conditions for each bug expressed symbolically in terms of the method's parameters. Specifically, we support the *concurrent discovery* of three conditions for bugs, called *must-bug*, *may-bug* and *never-bug*, respectively. To illustrate the three different kinds of bugs, consider a simple example :

```
int foo(int x, int y)
{ if (x≤y) then { if (x>10) then ℓ1:error else 1 }
  else { if complexTest(x,y) then ℓ2:error
         else { if x≥y then 2 else ℓ3:error }
       }
}
```

The bugs in our programs shall be flagged using a special `error` construct. This approach is simple but general as we can translate the more conventional (`assert c`) command for bug detection, directly to (`if c then skip else error`). The method `complexTest` denotes a predicate whose outcome cannot be modelled by the underlying static analyser, for example modelling the predicate $x^3+y^3\geq 0$ is beyond the capability of linear arithmetic solvers. According to our analysis, the error at location $\ell_1$ is a *must-bug* as we can determine an input trigger condition $x\leq y \wedge x>10$ that *must* lead to the error. In contrast, the error at location $\ell_2$ is a *may-bug* as our analysis can only determine a trigger condition $x>y$ that *may* lead to this error, since its occurrence is still dependent on the second conditional with a statically unknown test. Lastly, the error at location $\ell_3$ is a *never-bug* as our analysis can determine a trigger condition $x>y \wedge x<y$ that can never happen, namely `false`.

Our classification of bugs is dependent on the precision of the underlying static analyser. A more precise analyser can classify more of the may-bugs as either must-bugs or never-bugs. In our approach, each must (or may) bug is guarded by a trigger condition that specifies a condition on the parameters of a method that will (or could) lead to the bug. These trigger conditions are useful for interprocedural analysis as they

allow their associated bugs at each call site to be either propagated to caller, downgraded to a may-bug or proven to be safe (as a never-bug). For example, a trigger condition for a bug in the callee may be propagated as a bug for the caller with a new trigger condition based on the caller's inputs. A trigger condition may also be shown to be `false` in some callers' contexts which renders its associated bug unreachable and hence safe. Alternatively, the trigger condition of a must-bug might be shown to be always `true`. This means such a must-bug is always triggered whenever the method is invoked. In this scenario, a definite (or real) error can be reported for the method. More specifically, this chapter makes the following contributions :

- We propose a new *dual static analysis* to support either bug finding or a proof of the absence of bugs, where possible. This integrated analysis is based only on over-approximation and tracks concurrently both *success* and *failure outcomes*. Though this idea is simple (and may appear obvious on hindsight), it has never been used in mainstream work on static analyses.

- Our analysis has adjustable precision based on disjunctive formulae. The overlap between the two outcomes signifies imprecision of the analysis and can be used to guide the *precision refinement*.

- We propose a new technique to classify a sub-class of *definite non-terminations* as bugs. Our technique catches this class of non-termination bugs by explicitly identifying unreachable states during fixed point analysis. We can achieve this despite its converse problem (termination) being a liveness property, that is not usually addressed by safety-based static analyses.

- We formalise a correctness proof for our technique, and conduct a set of experiments to validate our proposal.

## 5.2 Our Approach

We propose a new approach to static analysis that can both detect real bugs and also be sound with respect to some stated safety property by reporting all of its possible bugs. Each concrete execution may result in one of three possible outcomes : (i) `ok` for a successful execution, (ii) `err` for a failed execution and (iii) `loop` for an execution that

does not terminate. Our strategy is to track two over-approximations, denoted by OK and ERR, to capture *all* executions that lead to (i) ok outcomes, and (ii) err outcomes, respectively. Formally:

**Definition 5.1. (Entire Success Outcomes)**

*Given a method, we can capture a condition on the method's inputs that leads to* all *possible* ok *outcomes. This condition is named* OK *and is an over-approximation that may include* some err *and* loop *outcomes. It represents a condition necessary for safety.*

**Definition 5.2. (Entire Failure Outcomes)**

*Given a method, we can capture a condition on the method's inputs that leads to* all *possible* err *outcomes. This condition is named* ERR *and is an over-approximation that may include* some ok *and* loop *outcomes. It represents a condition necessary for error.*

For example, consider the earlier foo example with two input parameters, x and y. Using our static analysis, we may compute a condition that covers all its ok outcomes :

$\texttt{OK} = (\texttt{x} \leq \texttt{y} \wedge \texttt{x} \leq \texttt{10}) \vee \texttt{x} > \texttt{y}$.

Correspondingly, a condition that covers all its err outcomes can be computed to be:

$\texttt{ERR} = (\texttt{x} \leq \texttt{y} \wedge \texttt{x} > \texttt{10}) \vee \texttt{x} > \texttt{y}$.

Our analysis may also identify individual errors separately by attaching each ERR outcome with a distinct program label, but we defer the presentation of this technique for Section 5.4.1.

Based on the two over-approximation results OK and ERR, we can determine the conditions for must-bug, may-bug and never-bug for each given method, as follows :

**Definition 5.3. (Never-Bug Condition)**

*A condition* c *(on the inputs) of a method is a never-bug condition if each of its inputs leads to either the* ok *or* loop *outcomes, but never the* err *outcome. This condition* c *can be computed using* $\texttt{OK} \wedge \neg(\texttt{ERR})$ *where* $\neg(\texttt{ERR})$ *ensures that none of the* err *outcomes are possible. It represents a condition sufficient for safety.* [1]

**Definition 5.4. (Must-Bug Condition)**

*A condition* c *(on the inputs) of a method is a must-bug condition if each of its inputs*

[1] For proving program safety, our classification is closer to partial correctness terminology, where the safety proof does not ensure program termination.

*leads to either the* `err` *or* `loop` *outcomes, but never the* `ok` *outcome. This condition* `c` *can be computed using* $\texttt{ERR} \wedge \neg(\texttt{OK})$ *where* $\neg(\texttt{OK})$ *ensures that none of the* `ok` *outcomes are possible. It represents a condition sufficient for error (where non-termination can also be considered a kind of bug).*

**Definition 5.5. (May-Bug Condition)**

*A condition* `c` *(on the inputs) of a method is a may-bug condition if each of its inputs leads to either* `ok`, `err` *or* `loop` *outcomes. This condition* `c` *arises from imprecise analysis and can be computed using* $\texttt{OK} \wedge \texttt{ERR}$ *which covers an overlap where all three outcomes are possible.*

A graphical illustration of these three categories of bug conditions is shown in Figure 5.1. The two circles denote the conditions for `OK` and `ERR`, while the three areas being partitioned by the two circles are the conditions for *never-bug*, *may-bug* and *must-bug*. The goal of our analysis is to minimise the overlap between `OK` and `ERR` outcomes, so that fewest possible inputs are classified under the may-bug category. This can be achieved by using a more precise analysis on the two over-approximated outcomes. Note that `ok` outcomes may only appear inside `OK`, while `err` outcomes may only appear inside `ERR`.

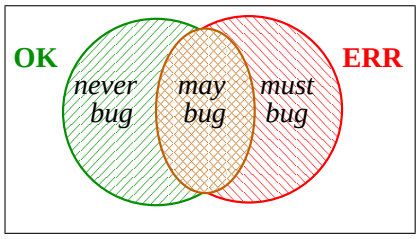



**Figure 5.1:** Classifying bugs

Due to our use of over-approximation analysis, our analysis can only guarantee that a bug will occur, assuming the absence of non-termination outcome. In other words, we may sometimes report a must-bug when the outcome is actually a `loop`. Nevertheless, we will discuss how to discover some non-termination outcomes in Section 5.4.2. Even though `loop` outcomes may appear everywhere, the condition $\neg(\texttt{OK} \vee \texttt{ERR})$ contains exclusively `loop` outcomes.

Going back to the `foo` example, we may now compute this method's conditions for must-bug, may-bug and never-bug :

$$\begin{aligned}\texttt{MUST\_BUG} &= \neg(\texttt{OK}) \wedge \texttt{ERR} \\ &= \neg(\texttt{x}{\leq}\texttt{y} \wedge \texttt{x}{\leq}\texttt{10} \vee \texttt{x}{>}\texttt{y}) \ \wedge \ (\texttt{x}{\leq}\texttt{y} \wedge \texttt{x}{>}\texttt{10} \vee \texttt{x}{>}\texttt{y}) \\ &= \texttt{x}{\leq}\texttt{y} \wedge \texttt{x}{>}\texttt{10}\end{aligned}$$

$$
\begin{aligned}
\texttt{MAY\_BUG} &= \texttt{OK} \wedge \texttt{ERR} \\
&= (\texttt{x}{\leq}\texttt{y} \wedge \texttt{x}{\leq}\texttt{10} \vee \texttt{x}{>}\texttt{y}) \wedge (\texttt{x}{\leq}\texttt{y} \wedge \texttt{x}{>}\texttt{10} \vee \texttt{x}{>}\texttt{y}) \\
&= \texttt{x}{>}\texttt{y} \\
\texttt{NEVER\_BUG} &= \texttt{OK} \wedge \neg(\texttt{ERR}) \\
&= (\texttt{x}{\leq}\texttt{y} \wedge \texttt{x}{\leq}\texttt{10} \vee \texttt{x}{>}\texttt{y}) \wedge \neg(\texttt{x}{\leq}\texttt{y} \wedge \texttt{x}{>}\texttt{10} \vee \texttt{x}{>}\texttt{y}) \\
&= \texttt{x}{\leq}\texttt{y} \wedge \texttt{x}{\leq}\texttt{10}
\end{aligned}
$$

Thus, to analyse for must, may and never-bugs, we only need to determine the condition for possible successful execution OK and the condition for possible program errors ERR. Analysing both these outcomes *concurrently* is the main novelty behind our approach for capturing both must and may analyses under a dual static analysis. As we shall see later, our approach supports adjustable precision with the help of disjunctive formulae to accurately report must-bugs or prove the absence of bugs, where possible. When none of these scenarios is possible, we can report may-bugs that could be either real bugs or false positives. May-bugs usually occur when the abstraction domain (used by our static analysis) is not precise enough.

## 5.3 Summarizing Dual Over-approximations

The results of our analysis are encoded as summaries $\Phi$ computed individually for each method (excepting mutually recursive methods that are analyzed simultaneously). Summaries shall be inferred bottom-up, starting with the methods lowest in the calling hierarchy. A method summary $\Phi = \{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}$ combines all the traces leading to success outcomes under the OK label as ($\phi_1$). The traces leading to failure outcomes are combined under the ERR label as ($\phi_2$). The form of the $\phi$ formula depends on the constraint solver used. Our analysis makes use of a linear arithmetic domain. Due to its forward nature, our analysis can capture each successful OK outcome with a postcondition that tracks the relation between inputs and output. The output of a method (or expression) is identified by a special **res** variable.

Our analysis summarizes entire method bodies, but also parts of the method body (individual expressions). It models state changes in a symbolic manner through *transition formulae* $\phi$. This is done using two symbolic values per program variable for capturing new and old variable values, respectively. Given a program variable v, the

prime notation $\mathtt{v}'$ denotes the new value, while $\mathtt{v}$ itself denotes the old value of the program variable. Two transition-based formulae may be composed in a natural way using an operator *compose with update* effect on a set of variables $\mathtt{W}$.

**Definition 5.6. (Compose with Update)**

*Given transition formulae $\phi_1, \phi_2$, and a set of variables to be updated $W=\{w_1, \ldots, w_n\}$, the operator $\circ_W$ is defined as:*

$$\phi_1 \circ_W \phi_2 =_{df} \exists\ r_1..r_n \cdot \rho_1\, \phi_1 \wedge \rho_2\, \phi_2$$

$$\text{where}\quad r_1, \ldots, r_n \text{ are fresh variables;}$$

$$\rho_1 = [w_i' \mapsto r_i]_{i=1}^n; \rho_2 = [w_i \mapsto r_i]_{i=1}^n$$

*Note that $\rho_1$ and $\rho_2$ are substitutions that link each latest value of $w_i'$ in $\phi_1$ with the corresponding initial value $w_i$ in $\phi_2$ via a fresh variable $r_i$. Unchanged variables in $\phi_2$ are used in primed form.*

Consider the sequence $\mathtt{x}{:=}(\mathtt{x}{+}\mathtt{y}){*}\mathtt{2};\ \mathtt{y}{:=}\mathtt{x}{+}\mathtt{y}$. Its effect can be captured by composing two transition formulae, each corresponding to one assignment :

$(\mathtt{x}'{=}\mathtt{2}{*}(\mathtt{x}{+}\mathtt{y}) \wedge \mathtt{y}'{=}\mathtt{y}) \circ_{\{\mathtt{x},\mathtt{y}\}} (\mathtt{y}'{=}\mathtt{x}{+}\mathtt{y} \wedge \mathtt{x}'{=}\mathtt{x})$

$\equiv \exists \mathtt{r_1}, \mathtt{r_2} \cdot (\mathtt{r_1}{=}\mathtt{2}{*}(\mathtt{x}{+}\mathtt{y}) \wedge \mathtt{r_2}{=}\mathtt{y}) \wedge (\mathtt{y}'{=}\mathtt{r_1}{+}\mathtt{r_2} \wedge \mathtt{x}'{=}\mathtt{r_1})$

$\equiv \mathtt{x}'{=}\mathtt{2}{*}(\mathtt{x}{+}\mathtt{y}) \wedge \mathtt{y}'{=}\mathtt{2}{*}\mathtt{x}{+}\mathtt{3}{*}\mathtt{y}$

The use of transition formulae in program analysis has been known since [38], and intensely studied in [30].

### 5.3.1 Forward Reasoning Rules

To compute method summaries, we shall now propose a set of forward rules that relies on transition formulae in Figure 5.2. These rules resemble those from weakest precondition/strongest postcondition calculi with two important distinctions. Firstly, our integrated approach is entirely forward and does not derive backwards weakest precondition. Secondly, we use a set of outcomes to compute simultaneously two over-approximations. Deriving both sound bugs and proving safety is made possible by this combination.

The rules are written in Hoare-style form using the judgement $\vdash \{\Phi_1\}\, e\, \{\Phi_2\}$. Given the $\mathtt{OK}$ outcome from $\Phi_1$ (a transition from the beginning of the current method to the prestate before $e$'s evaluation), the judgement derives $\Phi_2$: firstly, a transition from the beginning of the current method to the poststate after $e$'s evaluation; secondly, an $\mathtt{ERR}$ condition, in part from $\Phi_1$ and also from possible errors happening during $e$'s evaluation.

$$\frac{\Phi_1 = (\Phi \wedge \mathbf{res}{=}k)}{\vdash \{\Phi\}\, k\, \{\Phi_1\}}\ [\textsc{const}] \qquad \frac{\vdash \{\Phi \wedge \mathit{default}(t, v')\}\, e\, \{\Phi_1\}}{\vdash \{\Phi\}\, t\ v;\ e\, \{\exists v' \cdot \Phi_1\}}\ [\textsc{blk}]$$

$$\frac{\vdash \{\Phi\}\, e_1\, \{\Phi_1\} \quad \vdash \{\exists \mathbf{res} \cdot \Phi_1\}\, e_2\, \{\Phi_2\}}{\vdash \{\Phi\}\, e_1; e_2\, \{\Phi_2\}}\ [\textsc{seq}] \qquad \frac{\vdash \{\Phi\}\, e\, \{\Phi_1\} \quad \Phi_2 = \exists \mathbf{res} \cdot (\Phi_1 \circ_{\{v\}} v'{=}\mathbf{res})}{\vdash \{\Phi\}\, v{:=}e\, \{\Phi_2\}}\ [\textsc{assign}]$$

$$\frac{\Phi_1 = (\Phi \wedge \mathbf{res}{=}v')}{\vdash \{\Phi\}\, v\, \{\Phi_1\}}\ [\textsc{var}] \qquad \frac{V{=}\{v_i\}_{i=1}^{m-1} \quad \mathit{distinct}(V) \quad t_0\ mn((\texttt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \texttt{where}\ \Phi_{mn}\ \{...\}}{\vdash \{\Phi\}\, \ell : mn(v_1..v_n)\, \{\Phi \circ_V \Phi_{mn}\}}\ [\textsc{call}]$$

$$\frac{\vdash \{\Phi \wedge v'{=}1\}\, e_1\, \{\Phi_1\} \quad \vdash \{\Phi \wedge v'{=}0\}\, e_2\, \{\Phi_2\}}{\vdash \{\Phi\}\, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\, \{\Phi_1 \vee \Phi_2\}}\ [\textsc{if}] \qquad \frac{\Phi_1 = \Phi \circ_{\emptyset} \quad \{\texttt{OK} : \texttt{false},\ \texttt{ERR} : \texttt{true}\}}{\vdash \{\Phi\}\, \ell : \texttt{error}\, \{\Phi_1\}}\ [\textsc{error}]$$

$$\frac{X{=}\{v_1, .., v_n, \mathbf{res}, v'_1, .., v'_{m-1}\} \quad V{=}\{v'_i\}_{i=m}^{n} \quad R{=}\{\mathbf{res}, v'_1, .., v'_n\} \quad W{=}\{v_i\}_{i=1}^{n} \quad \vdash \{\{\texttt{OK} : \mathit{nochange}(W)\}\}\, e\, \{\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}\} \quad Q{=}\{\texttt{mnOK}(X){\equiv}\exists V \cdot \phi_1, \texttt{mnERR}(W){\equiv}\exists R \cdot \phi_2\} \quad \Phi'_{mn} = \mathit{fix}(Q)}{\vdash t_0\ mn((\texttt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \texttt{where}\ \Phi_{mn}\{e\} \Rightarrow \Phi'_{mn}}\ [\textsc{meth}]$$

**Figure 5.2:** Forward reasoning rules

Our rules use logical operators with set of outcomes as arguments: $\exists V \cdot \Phi$, $\Phi \vee \Phi$ and $\Phi \circ \Phi$. These logical operators are distributed to the components of $\Phi$ as follows:

$$\exists V \cdot \{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\} \equiv \{\texttt{OK} : \exists V \cdot \phi_1, \texttt{ERR} : \exists V \cdot \phi_2\}$$

$$\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\} \vee \{\texttt{OK} : \phi_A, \texttt{ERR} : \phi_B\}$$
$$\equiv \{\texttt{OK} : \phi_1 \vee \phi_A, \texttt{ERR} : \phi_2 \vee \phi_B\}$$

$$\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\} \circ_W \{\texttt{OK} : \phi_A, \texttt{ERR} : \phi_B\}$$
$$\equiv \{\texttt{OK} : \phi_1 \circ_W \phi_A, \texttt{ERR} : \phi_2 \vee (\phi_1 \circ_W \phi_B)\}$$

The rule that involves $\circ$ is more complex. The ERR outcome of the result (condition: $\phi_2 \vee (\phi_1 \circ \phi_B)$) indicates either failure from the first argument (condition: $\phi_2$), or from the success of the first argument followed by the failure of the second argument (condition: $\phi_1 \circ \phi_B$). The $\circ$ operator is not commutative as there is an implied order in the execution of the two summary outcomes. For brevity, a singleton set can also be expressed as:

$\phi \equiv \{\texttt{OK} : \phi\}$. In that case, operators like $\Phi \wedge \phi$ and $\Phi \circ_W \phi$ are shorthands for :

$$\begin{aligned} \Phi \wedge \phi &\equiv \Phi \circ_\emptyset \{\texttt{OK} : \phi,\ \texttt{ERR} : \texttt{false}\} \\ \Phi \circ_W \phi &\equiv \Phi \circ_W \{\texttt{OK} : \phi,\ \texttt{ERR} : \texttt{false}\} \end{aligned}$$

For the [**BLK**] rule, we provided some constraints as default values depending on the respective types. According to the language semantics, a possible set of defaults could be *default*(int, v)≡v=0, *default*(int[], v)≡v=null and *default*(void, v)≡true. Note that true may be used if there are no defaults. For the [**CALL**] rule, we used *distinct*(V) to ensure that the list of variables in V are different from each other. This is used to avoid aliases for pass-by-reference parameters. In the case of boolean values, we encode them in the integer domain by using 0 for false, and 1 for true, as can be seen in the [**IF**] rule. For the [**METH**] rule, we use $nochange(\{v_1..v_n\}) \equiv \bigwedge_{i=1}^{n} v_i{=}v_i'$ to compute the initial prestate for the method's body.

For a recursive method, the rules will derive a set of recursive constraints, one for each outcome of the method. The recursive constraints can be viewed as an intermediate form and are represented as constraint abstraction functions [73]. During the inference of a recursive method $mn$, the rule [**CALL**] will use the following placeholder for its summary: $\Phi_{mn}{=}\{\texttt{OK} : mn\texttt{OK}(X), \texttt{ERR} : mn\texttt{ERR}(W)\}$. The rule [**METH**] collects in $Q$ the constraint abstractions and then invokes an iterative fixed point analysis to compute the summary $\Phi'_{mn} = fix(Q)$. This fixed point analysis will be described in more detail in Sec 5.3.2.

While the summary of each user-defined method can be inferred, some methods are primitives in that they lack a method body and are provided instead with a summary formula. As an example, consider the following two primitives that may incur divide-by-zero and some array-related errors, respectively.

$$\begin{array}{l} \texttt{int div(int x, int y) where}\{\texttt{OK}:\texttt{y}{\neq}\texttt{0}{\wedge}\textbf{res}{=}\texttt{x/y},\ \texttt{ERR}:\texttt{y}{=}\texttt{0}\} \\ \texttt{void assign(int[] a, int i, int v) where} \\ \quad \{\texttt{OK}:\texttt{a}{\neq}\texttt{null}{\wedge}\texttt{0}{\leq}\texttt{i}{<}\texttt{a.len},\ \texttt{ERR}:\texttt{a}{=}\texttt{null}{\vee}\texttt{i}{<}\texttt{0}{\vee}\texttt{i}{\geq}\texttt{a.len}\} \end{array}$$

The div operation succeeds only when the argument y is non-zero, while the array assign method succeeds only when index i is within the bounds of a non-null array. For the assign method, we may split its ERR outcome to {ERR.null : a=null, ERR.low : i<0,

$\texttt{ERR.high} : \texttt{i} \geq \texttt{a.len}\}$, so as to capture the null dereferencing, low bound and high bound errors individually. In general, we expect the outcomes of a primitive to represent non-overlapping conditions that completely characterize the inputs of the primitive.

Note that `null` may be modelled by the value `0`, while `nonnull` may be modelled by a value $\geq$`1`. In the implementation, we rely exclusively on an arithmetic constraint form as our abstraction domain and solver. Due to the use of the integer domain to encode array lengths ($\texttt{a.len} > \texttt{0}$), boolean values ($\texttt{false} \equiv \texttt{0}, \texttt{true} \equiv \texttt{1}$) or nullness ($\texttt{null} \equiv \texttt{0}, \texttt{nonnull} \equiv \geq\texttt{1}$), we ensure that derived formulae always satisfy a type-invariant. This is not required for correctness; it merely filters out the unnecessary part of the integer domain. For example, given a variable of boolean sort `b`, after filtering we obtain: $\neg(\texttt{b}{=}\texttt{0} \vee \texttt{b}{=}\texttt{1}) \equiv \texttt{false}$.

**Example:** Let us illustrate the forward reasoning process using a simple example, a method that assigns 0 to those elements in the array $a$ from the range $i$ to 1.

```
void g(int[] a, ref int i)
{ if (i≤0) then ()
    else { assign(a, i, 0); i:=i−1; g(a, i) } }
```

We will use the forward rules to derive formulae at intermediate points from the method $g$. To improve readability, formulae are simplified and we omit the tracking on nullness of array variables:

For $\texttt{assign}(\texttt{a}, \texttt{i}, \texttt{0})$, [**CALL**] rule is applied:

$\Phi_1 \equiv \{\texttt{OK} : \texttt{i}'{=}\texttt{i} \wedge \texttt{i}'{>}\texttt{0} \wedge \texttt{i}'{<}\texttt{a.len}, \texttt{ERR} : \texttt{i}{>}\texttt{0} \wedge \texttt{i}{\geq}\texttt{a.len}\}$

For $\texttt{i}{:=}\texttt{i}{-}\texttt{1}$, [**ASSIGN**] rule is applied:

$\Phi_2 \equiv \{\texttt{OK} : \texttt{i}'{=}\texttt{i}{-}\texttt{1} \wedge \texttt{i}{>}\texttt{0} \wedge \texttt{i}{<}\texttt{a.len}, \texttt{ERR} : \texttt{i}{>}\texttt{0} \wedge \texttt{i}{\geq}\texttt{a.len}\}$

For $\texttt{g}(\texttt{a}, \texttt{i})$, [**CALL**] rule is applied:

$\Phi_3 \equiv \Phi_2 \circ_{\{\texttt{i}\}} \{\texttt{OK} : \texttt{gOK}(\texttt{a}, \texttt{i}_1, \texttt{i}'), \texttt{ERR} : \texttt{gERR}(\texttt{a}, \texttt{i}_1)\}$

For the conditional expression, the [**IF**] rule is applied:

$\Phi_4 \equiv \{\texttt{OK} : \texttt{i}'{=}\texttt{i} \wedge \texttt{i}'{\leq}\texttt{0}, \texttt{ERR} : \texttt{false}\} \vee \Phi_3$

For the method's body, the [**METH**] rule is applied:

$$\texttt{Q} = \{\texttt{gOK}(\texttt{a}, \texttt{i}, \texttt{i}') \equiv \phi_{\texttt{OK4}}, \texttt{gERR}(\texttt{a}, \texttt{i}) \equiv \exists \texttt{i}' \cdot \phi_{\texttt{ERR4}}\}$$

After simplifications, Q reduces to two independent constraint abstractions, one for the OK outcome, the other for the ERR outcome:

$$\texttt{gOK}(\texttt{a},\texttt{i},\texttt{i}')\equiv(\texttt{i}\leq 0 \wedge \texttt{i}'=\texttt{i}) \vee (\texttt{i}>0\wedge 0\leq\texttt{i}<\texttt{a.len}\wedge \exists \texttt{i}_1\cdot \texttt{i}_1=\texttt{i}-1 \wedge \texttt{gOK}(\texttt{a},\texttt{i}_1,\texttt{i}'))$$

$$\texttt{gERR}(\texttt{a},\texttt{i})\equiv\texttt{i}>0\wedge((\texttt{i}<0 \vee \texttt{i}\geq\texttt{a.len}) \vee 0\leq\texttt{i}<\texttt{a.len} \wedge \exists \texttt{i}_1\cdot \texttt{i}_1=\texttt{i}-1 \wedge \texttt{gERR}(\texttt{a},\texttt{i}_1))$$

### 5.3.2 Fixed-Point Analysis

Our approach to analysing recursive methods is to first build two constraint abstractions for OK and ERR outcomes. Once built, we can apply traditional fixed point analysis (for example, with hulling and widening approximations [43]) to derive a closed-form formula for each recursive constraint abstraction. The constraint abstractions are monotone functions and can be interpreted over various abstract domains. We will fix the abstract domain to the disjunctive polyhedral analysis as recently proposed in [138, 123]. This abstract domain is essentially based on the seminal work of Cousot and Halbwachs [43], but is more fine-grained by allowing disjunctions of linear inequalities to be captured. Though the abstract domain expresses disjunctive invariants, it still uses a (selective) hull operator reminiscent of the convex-hull operator originally proposed in polyhedral analysis.

**Example:** We apply fixed point analysis to gOK obtained previously. The iteration starts with the least element of the abstract domain represented by $\texttt{gOK}_0(\texttt{a},\texttt{i},\texttt{i}')\equiv\texttt{false}$. After few iterations, we can obtain a post fixed point $\texttt{gOK}_4(\texttt{a},\texttt{i},\texttt{i}')$ as follows:

$$\begin{aligned}
\texttt{gOK}_1(\texttt{a},\texttt{i},\texttt{i}') &\equiv (\texttt{i}\leq 0 \wedge \texttt{i}'=\texttt{i}) \vee (\texttt{i}>0\wedge 0\leq\texttt{i}<\texttt{a.len}\wedge \exists \texttt{i}_1\cdot \texttt{i}_1=\texttt{i}-1 \wedge \texttt{gOK}_0(\texttt{a},\texttt{i}_1,\texttt{i}'))\\
&\equiv \texttt{i}\leq 0 \wedge \texttt{i}'=\texttt{i}\\
\texttt{gOK}_2(\texttt{a},\texttt{i},\texttt{i}') &\equiv (\texttt{i}\leq 0\wedge\texttt{i}'=\texttt{i}) \vee (\texttt{i}=1\wedge\texttt{i}'=0\wedge 2\leq\texttt{a.len})\\
\texttt{gOK}_3(\texttt{a},\texttt{i},\texttt{i}') &\equiv (\texttt{i}\leq 0\wedge\texttt{i}'=\texttt{i}) \vee (1\leq\texttt{i}\leq\texttt{a.len}-1\wedge\texttt{i}\leq 2\wedge\texttt{i}'=0)\\
\texttt{gOK}_4(\texttt{a},\texttt{i},\texttt{i}') &\equiv_{\texttt{W}} (\texttt{i}\leq 0\wedge\texttt{i}'=\texttt{i}) \vee (1\leq\texttt{i}\leq\texttt{a.len}-1\wedge\texttt{i}'=0)
\end{aligned}$$

Note that $\equiv_{\texttt{W}}$ denotes a widening step. A similar analysis would derive the following closed-form formula for $\texttt{gERR}(\texttt{a},\texttt{i})$.

$$\texttt{gERR}(\texttt{a},\texttt{i}) \equiv \texttt{i}>0 \wedge \texttt{i}\geq\texttt{a.len}$$

The input conditions corresponding to the two over-approximations $\exists \mathtt{i}' \cdot \mathtt{gOK}(\mathtt{a}, \mathtt{i}, \mathtt{i}')$ and $\mathtt{gERR}(\mathtt{a}, \mathtt{i})$ do not overlap. Hence, we have a precise result that leads to either never-bug or must-bug.

$$\begin{aligned} \mathtt{NEVER\_BUG} &= \mathtt{i} \leq 0 \vee 1 \leq \mathtt{i} \leq \mathtt{a.len} - 1 \\ \mathtt{MUST\_BUG} &= \mathtt{i} > 0 \wedge \mathtt{i} \geq \mathtt{a.len} \\ \mathtt{MAY\_BUG} &= \mathtt{false} \end{aligned}$$

To compute the never-bug and the must-bug conditions, we make use of the negation operator $\neg$ which can be provided precisely for some abstract domains. This is the case for our disjunctive polyhedron abstract domain. Disjunctions are crucial: for the (conjunctive) polyhedron abstract domain a sound under-approximating version of the $\neg$ operator has to be used (which may lose a lot of precision). A simple example of a disjunctive formula:

$$\neg(x \geq 0 \wedge y \geq 0) = (x \leq -1) \vee (y \leq -1 \wedge x \geq 0)$$

This formula could be approximated in the polyhedron domain by dropping one of its disjuncts.

Our technique for fixed-point analysis can also be applied to imperative loops. It is folklore in the functional community that loops are but tail-recursive functions. This same idea can also be used for imperative languages, except that pass-by-reference parameters are critical for modelling variables that may be updated across method invocations (or loop iterations). For example, consider the following loop where variables $\{\mathtt{r}, \mathtt{i}\}$ are updated :

$$\mathtt{while}\ (\mathtt{i} < \mathtt{n})\ \mathtt{do}\ \{\ \mathtt{r} := \mathtt{r} + 2;\ \mathtt{i} := \mathtt{i} + 1\ \}$$

To model the effect of this loop, our system transforms it automatically to the following tail-recursive method :

```
void tail(ref int r, ref int i; int n) {
  if (i<n) then { r := r + 2; i := i + 1; tail(r, i, n) }
  else () }
```

The following summary is computed for the above loop:

$\{\mathtt{OK} : (\mathtt{i} \geq \mathtt{n} \wedge \mathtt{r}' = \mathtt{r} \wedge \mathtt{i}' = \mathtt{i}) \vee (\mathtt{i} < \mathtt{n} \wedge \mathtt{i}' = \mathtt{n} \wedge \mathtt{r}' = \mathtt{r} + 2(\mathtt{n} - \mathtt{i}))\}$

## 5.4 Further Improvements

There are at least three avenues to further improve on our static analysis. First, we can provide a more precise way to capture the origin of each detected error. Second, we can capture a sub-class of definite non-termination as bugs. Lastly, we shall look at an alias analysis to support the handling of heap-allocated objects.

### 5.4.1 Precise Error Tracing

Our analysis may pin-point the precise location of a discovered error by the notation $\{\texttt{ERR}.\ell : \texttt{e}\}$ where $\ell$ is a sequence of program locations that corresponds to the method call chain leading to the specified error. As an example, consider the following :

| | |
|---|---|
| `void foo3(int x) {` | `void foo4(int x, int y) {` |
| $\ell_1$ : `foo4(x, x + 1);` | `if x=y then` $\ell_3$ : `error` |
| $\ell_2$ : `foo4(x, 3);` } | `else ()` } |

The error in `foo4` will only be flagged if `x=y`. We may therefore capture its summary outcome as $\{\texttt{OK} : \texttt{x}{\neq}\texttt{y},\ \texttt{ERR}.\ell_3 : \texttt{x=y}\}$. This $\texttt{ERR}.\ell_3$ error is impossible when invoked from $\ell_1$:`foo4(x, x+1)`, but can occur when it is invoked from the context of $\ell_2$:`foo4(x, 3)`. The summary outcome for the `foo3` method is therefore inferred as: $\{\texttt{OK} : \texttt{x}{\neq}\texttt{3},\ \texttt{ERR}.\ell_1.\ell_3 : \texttt{false},\ \texttt{ERR}.\ell_2.\ell_3 : \texttt{x=3}\}$. The formula `false` at $\texttt{ERR}.\ell_1.\ell_3$ indicates that the bug at $\ell_1$ can never occur. We can omit this never-bug from the summary outcome of `foo3` which would simplify to $\{\texttt{OK} : \texttt{x}{\neq}\texttt{3},\ \texttt{ERR}.\ell_2.\ell_3 : \texttt{x=3}\}$. In contrast, the bug at call $\ell_2$ can occur under the input condition `x=3`. The label $\ell_2.\ell_3$ is used to indicate the call chain leading to the bug at final destination $\ell_3$. Each label and trigger condition essentially captures a potential *counter-example* (in the form of a must or may bug) that violates safety.

To provide precise reporting of errors, we only need to change two rules where label $\ell$ is added to trace the calling hierarchy of each error:

$$[\textbf{ERROR}]$$

$$\frac{\Phi_1{=}\Phi \circ_{\emptyset} \{\texttt{OK} : \texttt{false},\ \texttt{ERR}.\ell : \texttt{true}\}}{\vdash \{\Phi\}\, \ell : \texttt{error}\, \{\Phi_1\}}$$

$$[\underline{\textbf{CALL}}]$$

$$V{=}\{v_i\}_{i=1}^{m-1} \quad \mathit{distinct}(V)$$

$$\frac{t_0\ mn((\texttt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \texttt{where}\ \Phi_{mn}\ \{...\}}{\vdash \{\Phi\}\,\ell : mn(v_1..v_n)\,\{\Phi \circ_V \mathit{add}(\Phi_{mn}, \ell)\}}$$

The *add* command is defined as follows:

$$\mathit{add}(\{\texttt{OK}{:}\ \phi,\ (\texttt{ERR}.\ell_\texttt{i} : \phi_\texttt{i})^*\}, \ell) = \{\texttt{OK} : \phi,\ (\texttt{ERR}.\ell.\ell_\texttt{i} : \phi_\texttt{i})^*\}$$

In the case of recursive methods, all bugs that originate from the same location in the recursive method are grouped under the same error outcome. This ensures that the label sequences are always finite, and shall be bounded by the static height of the method call hierarchy. Note that all elements in a set of mutual-recursive methods have the same height in the call hierarchy.

We can classify each individual bug, as follows:

**Definition 5.7. (Individual Bug Classification)**

*Consider a method with the following summary outcomes :*

$$\{\texttt{OK} : \phi_0,\ \texttt{ERR}.\ell_1 : \phi_1, .., \ \texttt{ERR}.\ell_\texttt{n} : \phi_\texttt{n}\}$$

*A bug* $\texttt{ERR}.\ell_\texttt{i}$ *is said to be a never-bug if* $\phi_\texttt{i} = \texttt{false}$.

*A bug* $\texttt{ERR}.\ell_\texttt{i}$ *is said to be a must-bug if* $(\phi_\texttt{i} \wedge \phi_0 = \texttt{false})$ *and* $\phi_\texttt{i} \wedge (\bigvee_{\texttt{j} \in \{1..\texttt{n}\} - \{\texttt{i}\}}\ \phi_\texttt{j})) = \texttt{false}$.

*A bug* $\texttt{ERR}.\ell_\texttt{i}$ *is said to be a may-bug otherwise.*

Two bugs $\texttt{ERR}.\ell_\texttt{a}$ and $\texttt{ERR}.\ell_\texttt{b}$ are said to be *closely-related* if either $\phi_\texttt{a} \Longrightarrow \phi_\texttt{b}$ or $\phi_\texttt{b} \Longrightarrow \phi_\texttt{a}$. As closely-related bugs may be indistinguishable from each other, we shall group them together in the must-bug category, if the condition $(\phi_\texttt{a} \vee \phi_\texttt{b}) \wedge \phi_0$ is unsatisfiable. This amalgamation of closely-related must-bugs allows us to report that a bug from the amalgamated set will be definitely triggered as a must-bug, except that we are unable to pin-point the exact bug from this set.

### 5.4.2 Non-Termination as Bugs

Non-termination can be considered another source of bugs that is difficult to detect, since static analyses are typically formalised for safety property rather than liveness property. In general, static analyses may be used to partition the input domain of a given program.

Our analysis computes over-approximations for inputs that lead to all successful executions (all ok executions approximated by OK), and all failed executions (all err executions approximated by ERR). Similarly, we can specify an over-approximation for inputs that lead to all non-terminating executions (all loop executions approximated by LOOP). The resulting input partitioning is shown in Figure 5.3, where each circle corresponds to one over-approximation. [2] While computing a reasonably precise LOOP condition is still an open problem [33], our computation of both OK and ERR outcomes has the nice side-effect of being able to detect a sub-class of non-termination bugs. This class of non-termination bugs are due to recursive methods and may be discovered by fixed point analysis: with both OK and ERR outcomes soundly covered, any state left unreachable after analysis would have to belong to the non-termination outcome.

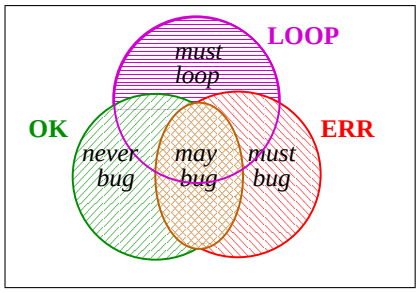



**Figure 5.3:** Adding LOOP to the bug classification

For example, consider a recursive method whose summary has been inferred to be $\{\texttt{OK}:\phi_1,\ \texttt{ERR}:\phi_2\}$. As these two outcomes cover all executions that either succeed or fail, whatever is left in the complement $\neg(\exists \texttt{R}\cdot\phi_1 \vee \phi_2)$ can only be executions leading to non-terminating loop, where R denotes the set of output variables including **res** from $\phi_1$. We will use MUST_LOOP to denote this precondition sufficient for non-termination. This is on the assumption that all errors have been modelled and captured under the ERR outcome. For a precise classification of this class of non-termination bugs, we can use $\{\texttt{OK}:\phi_1,\ \texttt{ERR}:\phi_2, \texttt{ERR.fn.MUST\_LOOP}:\neg(\exists \texttt{R}\cdot\phi_1 \vee \phi_2)\}$. In this case, fn denotes the name of the recursive method that is causing the non-termination bug.

To illustrate how non-termination bugs can be captured, consider the following recursive method:

```
int foo5(int i) { if i=10 then 1 else 2 + foo5(i+1) }
```

From fixed point analysis, we can obtain:

[2] This input partitioning is inspired by the seminal work of Dijkstra on semantic characterization and weakest-precondition calculus [49, pages 20-23]. While our classification is concerned with program safety and bug detection, Dijkstra's classification is more concerned with characterizing termination and whether the final state satisfies a given postcondition.

$\{\texttt{OK} : \texttt{i}{\leq}\texttt{10} \wedge \textbf{res}{=}\texttt{2(10}{-}\texttt{i)}{+}\texttt{1}\}\}$.

Since the ERR condition is false, we can determine $\neg(\exists \textbf{res} \cdot \texttt{i}{\leq}\texttt{10} \wedge \textbf{res}{=}\texttt{2(10}{-}\texttt{i)}{+}\texttt{1})$ which simplifies to $(\texttt{i}{>}\texttt{10})$ that is clearly a non-termination must-bug. Our summary can now be modified to the following :

$\{\texttt{OK} : \texttt{i}{\leq}\texttt{10} \wedge \textbf{res}{=}\texttt{2(10}{-}\texttt{i)}{+}\texttt{1},\ \texttt{ERR.foo5.MUST\_LOOP} : \texttt{i}{>}\texttt{10}\}\}$

Once a non-termination bug has been detected for a given recursive method, it can be treated like any other bug where it could be propagated, downgraded to a may bug or proven safe, depending on the context of its callers. Lastly, we emphasize that we can only catch a subset of the non-termination bugs and *cannot* guarantee that all non-termination bugs are captured.

### 5.4.3 Alias Analysis for Heap-Allocated Objects

Heap-allocated objects pose an extra challenge to program analysis as we are required to undertake a separate alias analysis first. As the problem of alias analysis is orthogonal to our analysis for bug discovery, we discuss a generic solution for aliasing here. Our proposal incorporates the major trends in alias analyses, which could be divided into three broad categories, as follows:

**Definition 5.8. (Must-Aliases)**

*A set of references* $\{\texttt{x}_1, .., \texttt{x}_\texttt{m}\}$ *is said to be* must-aliases *if they definitely refer to the same object. We may represent each must-aliased object by the following notation* $\texttt{x}{::}\ \texttt{Obj}\langle \texttt{f}_1, .., \texttt{f}_\texttt{n}\rangle \wedge \bigwedge_{\texttt{i}\in 1..\texttt{m}} \texttt{x}{=}\texttt{x}_\texttt{i}$ *where* $\texttt{f}_1, .., \texttt{f}_\texttt{n}$ *denote its fields that can be subjected to strong updates.*

**Definition 5.9. (May-Aliases)**

*A set of references* $\{\texttt{x}_1, .., \texttt{x}_\texttt{m}\}$ *is said to be* may-aliases *if they are possibly aliased with each other but may refer to zero or more objects. We shall represent each may-aliased object by the notation* $\texttt{x}{::}\texttt{Obj}\langle \texttt{f}_1, .., \texttt{f}_\texttt{n}\rangle @\texttt{M} \wedge \texttt{x} \in \{\texttt{x}_1, .., \texttt{x}_\texttt{m}\}$ *where* $\texttt{f}_1, .., \texttt{f}_\texttt{n}$ *denote its fields which can be subjected to weak-updates.*

Strong update allows the states of its fields to be directly changed, while weak update can only support either unchanged or possibly changed weakening of its fields' states.

**Definition 5.10. (Arbitrary-Aliases)**

*A reference is said to be* arbitrary-aliased *if it points to an object where only the immutable fields are tracked. We may represent each arbitrary-aliased object by the following notation* $\texttt{x::Obj}\langle \texttt{f}_1, .., \texttt{f}_n \rangle \texttt{@I}$. *A field* $\texttt{f}_i$ *is said to be immutable if its value is never changed after the object is constructed. For example, the length field of an array is immutable and can be tracked using just arbitrary-aliases.*

Arbitrary-aliases are the least precise but simplest to analyse. May-aliases are typically obtained with the help of points-to graph, while must-aliases require also linearity analysis. Our current prototype system only supports arbitrary-aliases and may-aliases but not must-aliases. Nevertheless, we design alias analysis as an orthogonal component to dual analysis, as it can help improve the precision of dual analyzer without affecting its correctness.

## 5.5 Experimental Results

We have implemented the proposed inference mechanisms in a tool named Dualyzer (from Dual analyzer). The goal of Dualyzer is to analyze imperative programs for proving safety or discovering bugs. We have also implemented the enhancements (except must-aliasing) described in the preceding Section 5.4. The prototype system was built using the Haskell language and the Glasgow Haskell compiler [121]. We use the Omega library [129] to simplify and check for satisfiability of Presburger formulae. Our test platform was a Pentium 3.0 GHz system with 2GBytes main memory, running Fedora 4. A web-demo of our system can be found at `http://loris-7.ddns.comp.nus.edu.sg/~popeeaco/bugs/`.

One objective of Dualyzer is to prove the absence of bugs, whenever possible. For this purpose, we tested our system on a set of small programs with challenging recursion and some programs from two benchmark suites: SciMark (Fast Fourier Transform, LU decomposition, Successive Over-Relaxation) [112] and Linpack [51]. These programs are free from array bound check errors. We capture array accesses in primitive methods with an `OK` outcome (when the index is within the bounds of the array) and an `ERR` outcome (when the index is out-of-bounds). In our experiments there was no programmer effort required to place error/assert constructs, since for array bound checks such assertions can be generated automatically.

| Benchmark Programs | Source (lines) | Rec. constr. | All checks | Dualyzer | |
|---|---|---|---|---|---|
| | | | | May | (secs) |
| binary search | 31 | 1 | 2 | 0 | 3.16 |
| bubble sort | 39 | 2 | 12 | 0 | 0.82 |
| init array | 11 | 1 | 2 | 0 | 0.26 |
| merge sort | 58 | 3 | 16 | 0 | 4.63 |
| queens | 39 | 2 | 8 | 0 | 1.47 |
| quick sort | 43 | 2 | 12 | 0 | 1.50 |
| sentinel | 17 | 1 | 4 | 1 | 0.12 |
| FFT | 336 | 9 | 62 | 0 | 13.50 |
| LU Decomp. | 191 | 10 | 82 | 0 | 14.34 |
| SOR | 84 | 5 | 32 | 0 | 3.50 |
| Linpack | 903 | 25 | 166 | 0 | 38.91 |

**Figure 5.4:** Statistics for a set of array-based programs without bugs

Figure 5.4 summarizes the statistics obtained for each program. To quantify the analysis complexity of the benchmark programs, we counted the number of lines of C code (column 2) and also the number of recursive methods and loops present in each program (column 3). Column 4 presents the total number of array accesses (counted statically) from the original programs. The number of array accesses that cannot be proven safe (may-bugs) is shown in column 5, while the analysis time is given in column 6. Our analysis determines automatically the number of disjuncts for fixed point analysis of each method. Consequently, by using at most three disjuncts (`m=3`) during fixed point analysis, we can prove all array accesses safe, except for sentinel program with a may-bug. This program uses a guard against falling off one end of the array. To eliminate this may-bug, we require an existential property on the collection of array elements which is beyond the capability of our current system.

To evaluate the bug finding credentials of Dualyzer, we used a simple procedure to seed bugs into a subset of the correct programs from Figure 5.4. Arbitrary array accesses were changed using an offset by a random value constrained to exhibit true errors. The results of this experiment are shown in Figure 5.5 where we are able to report some of the new errors as must-bugs. Note that we were using precise error tracing to capture the exact location and

| Faulty Programs | Dualyzer | | |
|---|---|---|---|
| | May | Must | (secs) |
| bsort | 0 | 1 | 2.44 |
| initarr | 0 | 1 | 1.12 |
| qsort | 0 | 1 | 2.24 |
| sentinel | 0 | 1 | 1.31 |

**Figure 5.5:** Buggy codes

trigger condition of each individual bug. While the programs we used to validate Dualyzer are small in size, we believe they show the versatility of dual static analysis in both proving program safety and finding true bugs.

### 5.5.1 Comparison with BLAST

| Benchmark Programs | BLAST | | Dualyzer | |
|---|---|---|---|---|
| | Result | (secs) | Result | (secs) |
| binary search | * | 0.06 | √ | 3.16 |
| bubble sort | * | 0.10 | √ | 0.82 |
| init array | √ | 1.15 | √ | 0.26 |
| merge sort | * | 0.12 | √ | 4.63 |
| queens | √ | 3.45 | √ | 1.47 |
| quick sort | √ | 28.82 | √ | 1.50 |
| sentinel | * | 1.31 | * | 0.12 |
| FFT | * | 0.57 | √ | 13.50 |
| LU | √ | 7.26 | √ | 14.34 |
| SOR | √ | 2.14 | √ | 3.50 |
| Linpack | *(exc) | 408.1 | √ | 38.91 |

**Figure 5.6:** Comparison with Blast

We used the same set of programs (shown in Figure 5.4) to make a comparison with the Blast software verification system [80].[3] With a similar goal to Dualyzer, the Blast software verification system aims at statically proving safety or finding true bugs otherwise. We present the results in Figure 5.6. Compared to our prototype, Blast performed as well in proving safety for `init array`, `queens`, `quicksort`, `LU and SOR`. However, Blast was not able to prove the safety of `binary search`, `merge sort` and `FFT` for which it reported (false) bugs due to division being treated as an uninterpreted function[10]. Blast also reported a false bug for `bubble sort`; for `Linpack` the analysis ended prematurely with an exception (raised in the Simplify prover).

Though an intended goal of BLAST is to report true bugs where possible, the representation of program states by symbolic constraints ultimately leads to some approximation (for e.g. via uninterpreted functions) that could lead unwittingly to false alarms. This scenario does not occur for Dualyzer since we rely on dual analysis to help distinguish program safety from must-bug, but can revert to may-bug reporting whenever

[3] We used the latest version of Blast 2.4, available from http://mtc.epfl.ch/software-tools/blast/. The running times reported for Blast correspond to several runs of abstraction refinement as we invoked Blast with the default set of arguments.

| Benchmark Programs | SYNERGY | DUALYZER | |
|---|---|---|---|
| | | May | Must |
| ex_fig1 | BUG | 0 | 1 |
| ex_fig3 | SAFE | 0 | 0 |
| ex_fig4 | BUG | 0 | 1 |
| ex_fig6 | SAFE | 0 | 0 |
| ex_fig7 | BUG | 0 | 1 |
| ex_fig8 | SAFE | 0 | 0 |
| ex_fig9 | ABORT | 0 | 1(`MUST_LOOP`) |

**Figure 5.7:** Examples from the SYNERGY paper [67]

there is uncertainty.

### 5.5.2 Comparison with SYNERGY

SYNERGY [67] is a recently proposed system that complements the capabilities of predicate abstraction refinement (as in BLAST) with DART-style testing [62] to prove safety and also find true bugs. To test the Dualyzer capability, we shall use a set of illustrative programs that were highlighted as figures in [67] with at most one bug per program. Column 2 from Figure 5.7 shows on what examples SYNERGY would discover a true BUG, prove that the program is SAFE or time-out during the refinement process (ABORT). Our analysis took less than a second on each of these programs. Compared to SYNERGY, we performed equally well in finding real bugs in `ex_fig1`, `ex_fig4`, `ex_fig7`, and also proving safety for `ex_fig3`, `ex_fig6`, `ex_fig8`. We highlight three examples. The first example is reproduced below:

```
void ex_fig1 (int a) {
    int i, c; i := 0; c := 0;
 ℓ1 : while (i<1000) { c := c + 1; i := i + 1; }
    if (a≤0) then {ℓ2 : error;} else {}
}
```

This example is difficult for tools like SLAM [5] and BLAST [80], that have to discover 1000 predicates, before finding a feasible path to the error. In contrast, SYNERGY finds the error quickly using DART-style testing by generating input constraints for `a`$>0$, and then `a`$\leq 0$. Our solution is also fast but relies on static analysis. It first discovers a postcondition ($c' \geq c \wedge i' \geq i \wedge i' \geq 1000$) for the loop at $\ell_1$. Subsequently, it reports a

precise must-bug condition at $\ell_2$ from the outcome {OK : (a>0), ERR.$\ell_2$ : (a≤0)} of method ex_fig1.

The code ex_fig3 is an example where both Synergy and Slam-like tools make use of path-sensitivity to prove correctness. By using intraprocedural path-sensitivity, we discover a precise set of outcomes and characterize the error at $\ell_4$ as unreachable. Specifically, the outcome for the loop at $\ell_3$ is computed as {OK : (lock′=1∧x′=y′∧y′≥y)} which can subsequently confirm that $\ell_4$ is unreachable.

```
void ex_fig3 (int y) {
    int x := randInt();
    int lock := 0;   //0 means Unlocked
 ℓ3 : do {
      lock := 1;       //1 means Locked
      x := y;
      if (randBool()) then {
        lock := 0;
        y := y + 1;
      } else ()
    } while (x != y);
    if (lock != 1) then {ℓ4 : error;} else{}
}
```

While able to prove safety and also find bugs, the Synergy system may fail to terminate due to abstraction refinement. The last example from [67] illustrates a case when Synergy fails to terminate as it generates longer and longer test sequences starting with the predicates (y<0), (y+x<0), (y+2x<0), and so on. The code ex_fig9 is reproduced below:

```
void ex_fig9() { int x, y; x := 0; y := 0;
                 ℓ5 : while (y≥0) { y := y + x; }
                 ℓ6 : error; }
```

Our system is able to initially confirm a must-bug at $\ell_6$ with conjunctive fixed-point

(m=1). The loop outcomes are computed as follows:

$$\texttt{loop}_{(\texttt{m=1})} = \{\texttt{OK} : (\texttt{x}'{=}\texttt{x} \wedge \texttt{y}'{\leq}\texttt{y} \wedge \texttt{y}'{<}0)\}$$

This conjunctive formula is unable to capture the non-termination of the loop. Combined with the information prior to the loop ($x{=}0 \wedge y{=}0$), the outcome for the entire method can only confirm the presence of a must bug at $\ell_6$ (if the program terminates):

$$\texttt{ex\_fig9}_{(\texttt{m=1})} = \{\texttt{OK} : \texttt{false},\ \texttt{ERR}.\ell_6 : \texttt{true}\}$$

However, using disjunctive fixed point analysis (m=2), we can capture non-termination in the outcome of the `while` loop and prove that the error at $\ell_6$ is unreachable:

$$\begin{aligned}\texttt{loop}_{(\texttt{m=2})} &= \{\texttt{OK} : (x'{=}x \wedge y'{=}y \wedge y'{<}0)\\ &\qquad \vee (x'{=}x \wedge x{\leq}y'{\leq}x{+}y \wedge y'{<}0),\\ &\qquad \texttt{ERR.MUST\_LOOP} : (x{\geq}0 \wedge y{\geq}0)\}\\ \texttt{ex\_fig9}_{(\texttt{m=2})} &= \{\texttt{OK} : \texttt{false},\ \texttt{ERR}.\ell_5.\texttt{MUST\_LOOP} : \texttt{true}\}\end{aligned}$$

Thus, with increased precision, our analysis is able to re-classify a must-bug more accurately and indicate the source of non-termination, where possible.

### 5.5.3 Examples from Verisec Benchmark

We have also analyzed several buffer overflow vulnerabilities from the CVE database as grouped in the Verisec benchmark suite [91]. This suite contains testcases with the actual vulnerabilities as well as corrected versions of these testcases. We were surprised that DUALYZER found two must-bugs in the corrected versions of the testcases, bugs that were later confirmed by the authors of the Verisec benchmark. The first must-bug is from the `Samba` implementation of the SMB networking protocol (CVE-2007-0453). It corresponds to a buffer access with an off-by-one error in the `r_strncpy` function. The second example is from the `SpamAssassin` open-source email filter and corresponds to a non-termination must-bug. We show the relevant C code below, originally split in two

different files:

```
#define BASE_SZ 2
#define BUFSZ BASE_SZ+2
void message_write (char * msg, int len) {
  char buffer[BUFSZ];
  int limit = BUFSZ - 4;
  for (int i=0; i<len; ){
    for (int j=0; i<len && j<limit; ){
      ...
      buffer[j] = msg[i];
      j++;
      ...
}}}
```

Since both the local variables `limit` and `j` are initialized to `0` and the value of `j` is increased through the inner-loop, the loop condition (`j<limit`) cannot be satisfied causing a non-terminating execution.

### 5.5.4 Beyond Safety to Memory Bounds Inference

Though Dualyzer has been originally formulated for finding bugs or proving safety, its ability to determine numeric trigger conditions can be put to other uses. In this experiment, we formulated the memory usage needed by each program into a safety problem, and raised a *memory adequacy* error whenever memory use exceeded a given initial memory bound. We achieved this by tracking two values : *memory usage* and *memory bound*. The former symbolically tracks the current memory that is in use, while the latter denotes the memory upper bound (high watermark). This tracking can be automatically instrumented for each given program. More details on the formalization of this analysis can be found in [26].

Dualyzer is able to infer statically both a lower-bound and an upper-bound for the memory needed. The more useful is the upper-bound: our analysis can guarantee that, if given at least this amount of memory, the program execution will be free of memory adequacy errors (never-bug condition). Secondly, the analysis guarantees that a memory adequacy error will definitely happen when the program execution is given

| Benchmark Programs | Source (lines) | Bounds Inferred | Stack Inf. (secs) | Heap Inf. (secs) |
|---|---|---|---|---|
| Ackermann | 16 | * | 2.24 | 1.52 |
| binary search | 31 | √ | 1.49 | 0.97 |
| bubble sort | 39 | √ | 1.55 | 0.82 |
| init array | 11 | √ | 0.45 | 0.28 |
| queens | 39 | √ | 2.39 | 1.41 |
| quick sort | 43 | √ | 4.53 | 2.34 |
| FFT | 336 | √ | 26.12 | 17.54 |
| LU Decomp. | 191 | √ | 32.71 | 18.73 |
| SOR | 84 | √ | 7.34 | 4.57 |
| sha | 211 | √ | 13.92 | 13.25 |
| susan | 2123 | √ | 57.1 | 98.3 |

**Figure 5.8:** Memory bounds estimation

less memory than the inferred lower bound (must-bug condition).

We have carried out experiments to infer stack/heap bounds for a set of small programs with challenging recursion and for some programs from two benchmark suites: SciMark [112] and MiBench (sha, susan) [74]. Figure 5.8 shows the statistics obtained for each program that we inferred. Column 3 captures time taken for stack-bounds inference, while Column 4 is for heap-bounds inference.

The time for inference roughly correlates with the program size and with the complexity of the relations between program variables. Specifically, the time taken for stack inference was more significant due to the intensive use of the stack by all of the programs. All stack usage bounds were successfully captured, except for the Ackermann function which requires a stack space that is exponential to its parameters' sizes. This stack bound is beyond the linear arithmetic form used in our current system. The time taken for heap inference was less substantial, due to the nature of our programs. Most of the benchmarks used few heap objects, with the exception of the susan benchmark. Susan is an image processing package that uses more heap-allocated arrays to represent patterns for image recognition.

## 5.6 Correctness of the Dual Static Analysis

In this section, we shall prove that our forward reasoning rules are correct in the following ways:

- the inferred summary outcomes for both *success* and *failure* are safe approximations of the expected final program state.
- each program never fails from an input of *never-bug* condition, never succeeds from an input of *must-bug* condition, and diverges from an input of *loop* condition.
- the forward analysis algorithm terminates.

### 5.6.1 Consistency between Static and Dynamic Semantics

The dynamic semantics for our core imperative language was defined previously in a small-step operational style (see Section 2.2). Here, we extend the source language with a new construct representing the intermediate result of a method call: the evaluation of the expression $\texttt{ret}(v^*, e)$ proceeds first with the method's body $e$ (rule [**D–RET–2**]) and, after its reduction to a value, the parameters passed by value $v^*$ are removed from the current stack (rule [**D–RET–1**]). If the evaluation of the body reaches an error, then the rule [**D–RET–3**] will throw the error back to the caller. A forward rule will be used in the static semantics for the `ret` construct, as follows:

$$\frac{[\underline{\textbf{RET}}] \quad \vdash \{\Phi\}\, e\, \{\Phi_1\}}{\vdash \{\Phi\}\, \texttt{ret}(v^*, e)\, \{\exists (v, v')^* \cdot \Phi_1\}}$$

The correctness proof requires analogous rules in static and dynamic (concrete) semantics. The static semantics rules for local variable declaration and method call are modified to be closer to their dynamic counterparts:

$$\frac{[\underline{\textbf{BLK}}] \quad \mathit{fresh}\ x \quad \rho = [v \mapsto x] \quad \vdash \{\Phi \wedge \mathit{default}(t, x')\}\, \rho e\, \{\Phi_1\}}{\vdash \{\Phi\}\, t\ v;\ e\, \{\exists x' \cdot \Phi_1\}}$$

$$\frac{[\underline{\textbf{CALL}}] \quad t_0\ mn((\texttt{ref}\ t_i\ w_i)_{i=1}^{m-1}, (t_i\ w_i)_{i=m}^{n})\ \texttt{where}\ \Phi_{mn}\ \{...\} \quad W = \{w_i\}_{i=1}^{m-1} \quad \mathit{distinct}(W) \quad \rho = [w_i \mapsto v_i]_{i=1}^{n} + [w'_i \mapsto v'_i]_{i=1}^{m-1}}{\vdash \{\Phi\}\, \ell : mn(v_1..v_n)\, \{\Phi \circ_{\rho W} \rho\Phi_{mn}\}}$$

While the concrete state is captured by the stack $s \in S$, the abstract state that we infer is captured by a relational constraint $\phi \in D$ between the program variables. When used with both unprimed and primed variables, $\phi \in \mathcal{D} \times \mathcal{D}$ actually denotes a transition

between abstract states. In this situation, we shall use two operators to distinguish between the original and final abstract states, as follows:

**Definition 5.11 (Prestate and Poststate).** *Given an abstract state transition $\phi \in \mathcal{D} \times \mathcal{D}$, its prestate PreSt($\phi$) captures the relation between unprimed variables of $\phi$. Correspondingly, its poststate PostSt($\phi$), captures the relation between primed variables of $\phi$.*

$$\mathit{PreSt}(\phi) = \exists X \cdot \phi, \quad \textit{where } X = \mathcal{V}'(\phi) \cup \{\mathit{res}\}$$
$$\mathit{PostSt}(\phi) = \rho(\exists X \cdot \phi), \quad \textit{where } X = \mathcal{V}(\phi) \textit{ and } \rho = [x' \mapsto x \mid x \in \mathcal{V}(\phi)]$$

To formalise the relation between the concrete and abstract domains, we introduce an abstraction operator: $\alpha(s) = \bigwedge\{v = \delta \mid [v \mapsto \delta] \in s\}$. If the stack contains two or more variables with the same name, only the leftmost variable is considered. For example, $\alpha([x \mapsto 1, y \mapsto 1, x \mapsto 0]) = (x{=}1 \wedge y{=}1)$.

Two consistency relations between the concrete and abstract domains are defined to either agree in the current state, or in a pair of both pre- and post-state. These consistency relations rely on an implication operator for the constraint language:

$$\frac{\alpha(s) \Rightarrow \phi}{s \models \phi} \qquad \frac{\alpha(s_1) \wedge \rho\alpha(s_2) \Rightarrow \phi \qquad \rho = [x \mapsto x' \mid x \in \mathcal{V}(\alpha(s_2))]}{(s_1, s_2) \models \phi}$$

In general, $(s_1, s_2) \models \phi$ implies $s_1 \models \mathit{PreSt}(\phi) \wedge s_2 \models \mathit{PostSt}(\phi)$, but the implication does not hold in the other direction (for constraints relating both primed and unprimed variables).

### 5.6.2 Proof Methodology

In general, we want to prove that the results obtained by the static semantics correctly reflect what happens during execution, as predicted by the dynamic semantics. The dynamic semantics is formulated in small-step style and the proof proceeds by showing that some property is preserved by each step of (dynamic) evaluation:

$$\begin{array}{ccccccc}
\langle s_1, e_1 \rangle & \hookrightarrow & \langle s_2, e_2 \rangle & \hookrightarrow & \ldots & \hookrightarrow & \langle s_n, e_n \rangle \\
\updownarrow & & \updownarrow & & & & \updownarrow \\
\vdash \{P_1\}\, e_1\, \{O_1, E_1\} & & \vdash \{P_2\}\, e_2\, \{O_2, E_2\} & & & & \vdash \{P_n\}\, e_n\, \{O_n, E_n\}
\end{array}$$

Thus, we will proceed by induction on the length of the (dynamic semantics) reduction sequence:

- base case: prove the property for $\langle s_1, e_1 \rangle$.
- induction step: assume the property holds for $\langle s_i, e_i \rangle$ and prove it for $\langle s_{i+1}, e_{i+1} \rangle$.

To prove the property for a configuration $\langle s_i, e_i \rangle$, the proof proceeds by induction on the height of the (dynamic semantics) reduction tree:

- base case: prove the property for those reduction rules without premises.
- induction step: assume the property holds for the premises and prove it for the concluding reduction rule.

The proof is completed when all the reduction rules are shown to preserve the required property. More details on the induction principle and proof examples for various program analyses can be found in the "Principles of Program Analysis" book [117, Sec 2.2, Sec 3.2, Sec 4.5.2, Sec 5.2, Appendix B].

**Method Summaries: Checking and Inference**

We formalize the notion of a sound method summary meaning that the summary is an over-approximation of the outcomes collected from the method's body. A summary is checked to be sound using an alternative static rule for a method declaration:

$$\frac{\begin{array}{c}[\textbf{CHECK–METH}] \\ W{=}\{v_i\}_{i=1}^{n} \quad V{=}\{v'_i\}_{i=m}^{n} \quad R{=}\{\textbf{res}, v'_1, .., v'_n\} \\ \vdash \{nochange(W)\}\, e\, \{\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}\} \\ \exists V{\cdot}\phi_1 \Rightarrow O_{mn} \qquad \exists R{\cdot}\phi_2 \Rightarrow E_{mn}\end{array}}{\vdash t_0\ mn((\texttt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \texttt{where}\ \{\texttt{OK} : O_{mn}, \texttt{ERR} : E_{mn}\}\{e\}}$$

Using this soundness notion for method summaries, we can split the main proof in two parts. For the first part, the proof is done assuming a program where all methods are given sound summaries. For the second part, we show that our fixed-point analysis always infers sound method summaries.

### 5.6.3 Main Theorem and Its Proof

The summary outcomes that we infer are a conservative approximation of the program state that we expect for our program. We can prove this by the following soundness theorem:

**Theorem 5.1 (Soundness of Summary Outcomes).** *Given an arbitrary expression $e_1$, an initial state $s_1$ and a transition formula $P_1$ consistent with $s_1$ such that $\alpha(s_1) \wedge nochange(\mathcal{V}(s_1)) \Rightarrow P_1$, where $\mathcal{V}(s_1)$ returns the variables defined by the state $s_1$.*

*Using $P_1$, we may obtain the following judgement $\vdash \{P_1\}\, e_1\, \{\{\texttt{OK}\colon O_1,\ \texttt{ERR}\colon E_1\}\}$. The success outcome $O_1$ is sound as we can show that the following holds, namely: if $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \delta\rangle$, then it must be the case that $(s_1, [\mathbf{res}\mapsto\delta]{+}s_n) \models O_1$. The failure outcome $E_1$ is also sound as we can show that the following holds: if $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \bot\rangle$, then it must be the case that $s_1 \models E_1$.*

**Proof:**

This result can be shown using induction on the length of the reduction sequence. We consider an arbitrary reduction step $\langle s_i, e_i\rangle \hookrightarrow \langle s_{i+1}, e_{i+1}\rangle$ and inference judgements such that the prestate is consistent with dynamic state: $P_i = P_1 \circ \rho\alpha(s_i)$.

**Success outcome is sound:** $(s_1, [\mathbf{res}\mapsto\delta]{+}s_n) \models O_1$

The base case proves the property for the last expression in a successful reduction sequence. When $e_n$ is a constant expression $\delta$, we can infer using the rule [**CONST**]: $\vdash \{P_n\}\, \delta\, \{\{\texttt{OK}\colon O_n,\ \texttt{ERR}\colon E_n\}\}$ such that $O_n = P_n \wedge (res{=}\delta) = (P_1 \circ \rho\alpha(s_n)) \wedge (res{=}\delta)$. As a consequence, the following consistency relation holds: $(s_1, [res\mapsto\delta]{+}s_n) \models O_n$.

The main part of the proof is based on a subject reduction lemma. This lemma proves the induction step corresponding to an arbitrary reduction step: $\langle s_i, e_i\rangle \hookrightarrow \langle s_{i+1}, e_{i+1}\rangle$. In particular, the lemma proves that the success outcome for $e_{i+1}$ is more precise than the one inferred from $e_i$: $O_{i+1} \Rightarrow O_i$. By repeated applications, we can conclude that the success outcome obtained from the inference of the original expression $e_1$ is sound: $(s_1, [res\mapsto\delta]{+}s_n) \models O_1$. □

**Failure outcome is sound:** $s_1 \models E_1$

The base case proves the property for the last expression in a failed reduction sequence. When $e_n$ is an `error` expression then from the rule [**ERROR**] we can deduce $\vdash \{\{\texttt{OK} : P_n, \texttt{ERR} : \texttt{false}\}\}\, l : \texttt{error}\, \{\{\texttt{OK} : O_n, \texttt{ERR} : E_n\}\}$ such that $E_n = P_n \wedge \{\texttt{OK} : \texttt{false}, \texttt{ERR} : \texttt{true}\} = P_n$. From the definition of $P_n$ we can conclude that $s_1 \models E_n$.

The subject reduction lemma is used to prove the induction step. As a direct consequence of the lemma, the failure outcome for $e_{i+1}$ is more precise than the one inferred from $e_i$: $E_{i+1} \Rightarrow E_i$. From the base case and the induction step, we can conclude that

the failure outcome is sound: $s_1 \models E_1$. □

The proof of Theorem 5.1 is based on induction over the length of reduction sequence and uses a subject reduction lemma as induction step. This lemma states the properties that are satisfied by an arbitrary reduction step.

**Lemma 5.2 (Subject reduction).** *Consider an execution started from the initial state $s_1$ and some arbitrary reduction step: $\langle s_1, e_1\rangle \hookrightarrow \ldots \hookrightarrow \langle s_i, e_i\rangle \hookrightarrow \langle s_{i+1}, e_{i+1}\rangle \hookrightarrow \ldots$. Further, consider a transition formula $P_i$ consistent with the execution states: $(s_1, s_i) \models P_i$ and an inference $\vdash \{P_i\}\, e_i\, \{\{\texttt{OK} : O_i, \texttt{ERR} : E_i\}\}$.*

*Then there exists $P_{i+1}$ consistent with the execution states $(s_1, s_{i+1}) \models P_{i+1}$ and the results of the inference $\vdash \{P_{i+1}\}\, e_{i+1}\, \{\{\texttt{OK} : O_{i+1}, \texttt{ERR} : E_{i+1}\}\}$ satisfy the following relations:*

- *$O_{i+1} \Rightarrow O_i$. We also have $PreSt(O_{i+1}) \Rightarrow PreSt(O_i)$ and $PostSt(O_{i+1}) \Rightarrow PostSt(O_i)$.*
- *$E_{i+1} \Rightarrow E_i$. We also have $E_i \equiv PreSt(E_i)$.*

**Proof:** We will prove that there is a relation between the inference result for $e_i$ and the inference result for $e_{i+1}$ by induction on the height of the (dynamic semantics) reduction tree. The induction hypothesis assumes that this relation holds for the reduction steps for the subexpressions of $e_i$. Various cases are denoted by the name of the evaluation rule that applies in the conclusion.

- Case [**D–VAR**]: With $s_{i+1} = s_i$, $s_i(v) = \delta$, we have the following reduction step:

$$\begin{array}{ccc} \langle s_i, v\rangle & \hookrightarrow & \langle s_{i+1}, s_i(v)\rangle \\ \updownarrow & & \updownarrow \\ \vdash \{P_i\}\, v\, \{P_i \wedge res = v\} & & \vdash \{P_{i+1}\}\, \delta\, \{P_{i+1} \wedge res = \delta\} \end{array}$$

  Let us choose $P_{i+1} = P_i$.

  - We can prove $P_{i+1}$ is consistent with the execution states $(s_1, s_{i+1})$ since $P_i$ is consistent with $(s_1, s_i)$ and $s_{i+1} = s_i$.
  - $O_{i+1} \Rightarrow O_i$ since $O_{i+1} = (P_{i+1} \wedge res{=}\delta)$ and $O_i = (P_i \wedge res{=}v)$.
  - $E_{i+1} \Rightarrow E_i$ since $E_{i+1} = E_i = \texttt{false}$.

- Case [**D−ASSIGN−1**]:

$$\begin{array}{ccc} \langle s_i, v := \delta \rangle & \hookrightarrow & \langle s_i[v \mapsto \delta], () \rangle \\ \updownarrow & & \updownarrow \\ \vdash \{P_i\}\, v := \delta\, \{P_i \circ_{\{v\}} v'{=}\delta\} & & \vdash \{P_{i+1}\}\, ()\, \{P_{i+1}\} \end{array}$$

  Let us choose $P_{i+1} = P_i \circ_{\{v\}} v'{=}\delta$.

  - By definition of the consistency relation, $(s_1, s_i[v{\mapsto}\delta]) \models P_{i+1}$ reduces to $\alpha(s_1) \wedge \rho\alpha(s_i[v{\mapsto}\delta]) \Rightarrow P_i \circ_{\{v\}} v'{=}\delta$. To prove this implication, we rely on the hypothesis that $P_i$ is consistent: $\alpha(s_1) \wedge \rho\alpha(s_i) \Rightarrow P_i$.
  - $O_{i+1} \Rightarrow O_i$ since $O_{i+1} = O_i$.
  - $E_{i+1} \Rightarrow E_i$ since $E_{i+1} = E_i = \texttt{false}$.

- Case [**D−ASSIGN−2**]:

$$\begin{array}{ccc} \langle s_i, v := e \rangle & \hookrightarrow & \langle s_{i+1}, v := e' \rangle \\ \updownarrow & & \updownarrow \\ \dfrac{\vdash \{P_i\}\, e\, \{\Phi\}}{\vdash \{P_i\}\, v := e\, \{\Phi \circ_{\{v\}} v'{=}res\}} & & \dfrac{\vdash \{P'_i\}\, e'\, \{\Phi'\}}{\vdash \{P_{i+1}\}\, v := e'\, \{\Phi' \circ_{\{v\}} v'{=}res\}} \end{array}$$

  We use the induction hypotheses corresponding to the following reduction step: $\langle s_i, e\rangle{\hookrightarrow}\langle s_{i+1}, e'\rangle$. By these hypotheses, there exists $P'_i$ that satisfies the consistency relation: $(s_1, s_{i+1}) \models P'_i$. Also, the results of the inference judgements $\vdash \{P_i\}\, e\, \{\Phi\}$ and $\vdash \{P'_i\}\, e'\, \{\Phi'\}$ satisfy the relation $\Phi' \Rightarrow \Phi$.

  Let us choose $P_{i+1}{=}P'_i$, where $P'_i$ is the prestate constructed using the induction hypothesis.

  - $(s_1, s_{i+1}) \models P_{i+1}$ from the induction hypothesis $(s_1, s_{i+1}) \models P'_i$
  - From the induction hypothesis $\Phi' \Rightarrow \Phi$, we can deduce that $(\Phi' \circ_{\{v\}} v'{=}res) \Rightarrow (\Phi \circ_{\{v\}} v'{=}res)$. This fact implies that $O_{i+1} \Rightarrow O_i$ .
  - From the induction hypothesis $\Phi' \Rightarrow \Phi$, we can deduce that $(\Phi' \circ_{\{v\}} v'{=}res) \Rightarrow (\Phi \circ_{\{v\}} v'{=}res)$. This fact implies that $E_{i+1} \Rightarrow E_i$.

- Case [**D–SEQ–1**]: We have $s_{i+1} = s_i$:

$$\begin{array}{ccc} \langle s_i, \delta; e_2\rangle & \hookrightarrow & \langle s_{i+1}, e_2\rangle \\ \updownarrow & & \updownarrow \\ \vdash \{\exists \mathbf{res} \cdot (P_i \wedge \mathbf{res}{=}\delta)\}\, e_2\, \{\Phi\} & & \\ \hline \vdash \{P_i\}\, \delta; e_2\, \{\Phi\} & & \vdash \{P_{i+1}\}\, e_2\, \{\Phi'\} \end{array}$$

  Let us choose $P_{i+1} = P_i$.

  - We can prove $P_{i+1}$ is consistent with the execution states $(s_1, s_{i+1})$ since $P_i$ is consistent with $(s_1, s_i)$ and $s_{i+1} = s_i$.
  - From the construction, $P_i$ does not refer to the variable **res**. Consequently, $P_{i+1} = P_i = \exists \mathbf{res} \cdot (P_i \wedge \mathbf{res}{=}\delta)$. Two inference judgements starting with equivalent prestates will have equivalent summary outcomes: $\Phi' = \Phi$. This implies that $O_{i+1} \Rightarrow O_i$.
  - $\Phi' = \Phi$ implies that $E_{i+1} \Rightarrow E_i$.

- Case [**D–PRIM**]: Since the code for primitive methods is not available for analysis, we assume that the summaries of primitives methods are sound with respect to the operational semantics of the primitive's implementation.

$$\begin{array}{ccc} \langle s_i, mn(v_1, .., v_n)\rangle & \hookrightarrow & \langle s_{i+1}, \delta\rangle \\ \updownarrow & & \updownarrow \\ \vdash \{P_i\}\, mn(v_1, .., v_n)\, \{P_i \circ_{\mathcal{V}(\Phi_{mn})} \Phi_{mn}\} & & \vdash \{P_{i+1}\}\, \delta\, \{P_{i+1} \wedge \mathtt{res}{=}\delta\} \end{array}$$

  The soundness of the primitive summaries can be formalized with the following condition: $(s_i, [\mathtt{res}{\mapsto}\delta]{+}s_{i+1}) \models \Phi_{mn} \wedge \mathit{nochange}(X)$, where $X = \mathcal{V}(s_i){-}\mathcal{V}(\Phi_{mn})$.

  Let us choose $P_{i+1} = P_i \circ_{\mathcal{V}(\Phi_{mn})} \Phi_{mn}$.

  - From $(s_1, s_i) \models P_i$ and $(s_i, s_{i+1}) \models \Phi_{mn} \wedge \mathit{nochange}(X)$ we should be able to prove that $(s_1, s_{i+1}) \models P_i \circ_{\mathcal{V}(\Phi_{mn})} \Phi_{mn}$.
  - Using the chosen $P_{i+1}$, we can derive trivially $O_{i+1} \Rightarrow O_i$.
  - Using the chosen $P_{i+1}$, we can derive trivially $E_{i+1} \Rightarrow E_i$.

- Case [**D–CALL**]: For this reduction step, we assume that each method is annotated with a sound summary. Using the soundness of the method summary, we can

successfully apply the rule [**CHECK–METH**] to the method declaration $mn$: $t_0\ mn((\texttt{ref}\ t_i\ w_i)_{i=1}^{m-1}, (t_i\ w_i)_{i=m}^{n})\ \texttt{where}\ \Phi_{mn}\{e\}$. Consequently, the result of the judgement $\vdash \{nochange(W)\}\, e\, \{\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}\}$ is more precise than $\Phi_{mn}$ as follows: $\exists V \cdot \phi_1 \Rightarrow O_{mn}$ and $\exists R \cdot \phi_2 \Rightarrow E_{mn}$.

The reduction step corresponding to a method call follows:

$$\frac{\begin{array}{ccc}\langle s_i, mn(v_1,..,v_n)\rangle & \hookrightarrow & \langle [w_i{\mapsto}s_i(v_i)]_{i=m}^{n}{+}s_i, \texttt{ret}(\{w_i\}_{i=m}^{n}, e'\rangle \\ \updownarrow & & \updownarrow \\ & & \vdash \{P_{i+1}\}\, e'\, \{\Phi'\}\end{array}}{\vdash \{P_i\}\, mn(v_1,..,v_n)\, \{P_i \circ_{\rho W} \rho\Phi_{mn}\} \qquad \vdash \{P_{i+1}\}\, \texttt{ret}(\{w_i\}_{i=m}^{n}, e')\, \{\exists (w_i, w_i')^* \cdot \Phi'\}}$$

where $e' = [v_i/w_i]_{i=1}^{m-1}e$, $W = \{v_i\}_{i=1}^{n}$ and $\rho = [v_i/w_i]_{i=1}^{n} + [v_i'/w_i']_{i=1}^{m-1}$.

Let us choose $P_{i+1} = P_i \wedge \bigwedge_{i=m}^{n}(w_i{=}v_i)$.

- From the induction hypothesis $(s_1, s_i) \models P_i$, we can prove that the following holds: $(s_1, [w_i{\mapsto}s_i(v_i)]_{i=m}^{n}{+}s_i) \models P_i \wedge \bigwedge_{i=m}^{n}(w_i{=}v_i)$.
- We can prove a pre-transition lemma that allows us to use a given judgement $\vdash \{nochange(W)\}\, e\, \{\Phi\}$ to deduce a related judgement where a transition formula $\Phi_{pre}$ is used to translate both the prestate and the poststate: $\vdash \{\Phi_{pre} \circ_W nochange(W)\}\, e\, \{\Phi_{pre} \circ_W \Phi\}$. Note that the formula $(\Phi_{pre} \circ_W nochange(W))$ can be simplified to $\Phi_{pre}$.

  Using this lemma with $\Phi_{pre}{=}\rho^{-1}P_i$, the judgement $\vdash \{nochange(W)\}\, e\, \{\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}\}$ can be transformed to $\vdash \{\rho^{-1}P_i\}\, e\, \{\rho^{-1}P_i \circ_W \{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}\}$. After proper renaming, this last judgement is equivalent with the following judgement $\vdash \{P_i\}\, \rho e\, \{P_i \circ_{\rho W} \rho\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}\}$, which we denote as (JUDG-1).

  The judgement from the induction hypothesis $\vdash \{P_{i+1}\}\, e'\, \{\Phi'\}$ can be used together with (JUDG-1) to conclude that $\Phi' \equiv P_i \circ_{\rho W} \rho\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}$. Since we know that $\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\} \Rightarrow \Phi_{mn}$, we can finally conclude that $\Phi' \Rightarrow P_i \circ_{\rho W} \rho\Phi_{mn}$. From this implication, we can directly derive that $O_{i+1} \Rightarrow O_i$.
- From the above proof, we can also conclude that $E_{i+1} \Rightarrow E_i$.

- Case [**D–BLK**]:

$$\frac{\langle s_i,\ t\ v; e\rangle \quad\updownarrow\quad \vdash \{P_i \wedge \mathit{default}(t, x')\}\ \rho e\ \{\Phi\}}{\vdash \{P_i\}\ \ t\ v; e\ \{\exists x' \cdot \Phi\}} \qquad \frac{\langle [x{\mapsto}\delta]{+}s, \mathtt{ret}(x, \rho e)\rangle \quad\updownarrow\quad \vdash \{P_{i+1}\}\ \rho e\ \{\Phi'\}}{\vdash \{P_{i+1}\}\ \mathtt{ret}(x, \rho e)\ \{\exists (x, x') \cdot \Phi'\}}$$

  Let us choose $P_{i+1} = P_i \wedge \mathit{default}(t, x')$.

  - From the induction hypothesis $(s_1, s_i) \models P_i$ and $\delta{=}\mathit{default}(t)$, we can prove that $(s_1, [x{\mapsto}\delta] + s_1) \models (P_i \wedge \mathit{default}(t, x'))$.
  - From the induction hypothesis $\Phi' \Rightarrow \Phi$ and the fact that $x$ does not appear in the formulae $\Phi$ and $\Phi'$, we can conclude that $\exists (x, x') \cdot \Phi' \Rightarrow \exists x' \cdot \Phi$. Consequently, we have $O_{i+1} \Rightarrow O_i$.
  - By a similar reasoning as above, we can conclude that $E_{i+1} \Rightarrow E_i$.

- Case [**D–RET–1**]:

$$\frac{\langle s_i, \mathtt{ret}(v^*, \delta)\rangle \quad\updownarrow\quad \vdash \{P_i\}\ \delta\ \{P_i \wedge \mathtt{res}{=}\delta\}}{\vdash \{P_i\}\ \mathtt{ret}(v^*, \delta)\ \{\exists (v, v')^* \cdot (P_i \wedge \mathtt{res}{=}\delta)\}} \qquad \frac{\langle s_i{-}\{v^*\}, \delta\rangle \quad\updownarrow\quad}{\vdash \{P_{i+1}\}\ \delta\ \{P_{i+1} \wedge \mathtt{res}{=}\delta\}}$$

  Let us choose $P_{i+1} = \exists (v, v')^* \cdot P_i$.

  - From the induction hypothesis $(s_1, s_i) \models P_i$, we can prove that the following holds: $(s_1, s_i{-}\{v^*\}) \models \exists (v, v')^* \cdot P_i$.
  - We have $O_i = \exists (v, v')^* \cdot (P_i \wedge \mathtt{res}{=}\delta)$ and $O_{i+1} = (\exists (v, v')^* \cdot P_i) \wedge \mathtt{res}{=}\delta$, where $v^*$ are either local variables or parameters passed by value. We can conclude that $O_{i+1} \Rightarrow O_i$.
  - We can conclude that $E_{i+1} \Rightarrow E_i$ from $E_{i+1} = E_i = \mathtt{false}$.

- Case [**D–ERROR**]: We have $s_{i+1} = s_i$:

$$\frac{\langle s_i, l : \mathtt{error}\rangle \quad\updownarrow\quad \vdash \{P_i\}\ l{:}\mathtt{error}\ \{\{\mathtt{OK} : \mathtt{false}, \mathtt{ERR} : \mathtt{true}\}\}}{\qquad\qquad} \qquad \frac{\langle s_i, \bot\rangle \quad\updownarrow\quad \vdash \{P_i\}\ \bot\ \{\{\mathtt{OK} : \mathtt{false}, \mathtt{ERR} : \mathtt{true}\}\}}{\qquad\qquad}$$

  Let us choose $P_{i+1} = P_i$.

- $(s_1, s_{i+1}) \models P_{i+1}$ since $(s_1, s_i) \models P_i$.
- We have $O_{i+1} = O_i = \texttt{false}$.
- We have $E_{i+1} = E_i = \texttt{true}$.

- Case [**D–IF–1**]: We have $s_{i+1} = s_i$:

$$\begin{array}{ccc} \langle s_i, \texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2\rangle & \hookrightarrow & \langle s_{i+1}, e_1\rangle \\ \updownarrow & & \updownarrow \\ \dfrac{\vdash \{P_i \wedge v'{=}1\}\, e_1\, \{\Phi\} \quad \vdash \{\texttt{false}\}\, e_2\, \{\texttt{false}\}}{\vdash \{P_i\}\, \texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2\, \{\Phi \vee \texttt{false}\}} & & \vdash \{P_{i+1}\}\, e_1\, \{\Phi'\} \end{array}$$

  Let us choose $P_{i+1} = P_i$.

  - We can prove $P_{i+1}$ is consistent with the execution states $(s_1, s_{i+1})$ since $P_i$ is consistent with $(s_1, s_i)$ and $s_{i+1} = s_i$.
  - The reduction step assumes that $s_i(v) = \texttt{true}$. Since $P_i$ is consistent with the execution state $s_i$, we can conclude that the prestate $P_i \wedge v'{=}0$ simplifies to the `false` formula. Consequently, two inference judgements for $e_2$ with equivalent prestates $P_i$ and $P_{i+1}$ will have equivalent poststates: $\Phi$ and $\Phi'$. This implies that $O_{i+1} \Rightarrow O_i$.
  - $\Phi' = \Phi$ implies that $E_{i+1} \Rightarrow E_i$

- Case [**D–IF–2**]: We have $s_{i+1} = s_i$:

$$\begin{array}{ccc} \langle s_i, \texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2\rangle & \hookrightarrow & \langle s_{i+1}, e_2\rangle \\ \updownarrow & & \updownarrow \\ \dfrac{\vdash \{\texttt{false}\}\, e_1\, \{\texttt{false}\} \quad \vdash \{P_i \wedge v'{=}0\}\, e_2\, \{\Phi\}}{\vdash \{P_i\}\, \texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2\, \{\Phi \vee \texttt{false}\}} & & \vdash \{P_{i+1}\}\, e_2\, \{\Phi'\} \end{array}$$

  Let us choose $P_{i+1} = P_i$.

  - We can prove $P_{i+1}$ is consistent with the execution states $(s_1, s_{i+1})$ since $P_i$ is consistent with $(s_1, s_i)$ and $s_{i+1} = s_i$.
  - The reduction step assumes that $s_i(v) = \texttt{false}$. Since $P_i$ is consistent with the execution state $s_i$, we can conclude that the prestate $P_i \wedge v'{=}1$ simplifies to the `false` formula. Consequently, two inference judgements for $e_1$ with

equivalent prestates $P_i$ and $P_{i+1}$ will have equivalent poststates: $\Phi$ and $\Phi'$. This implies that $O_{i+1} \Rightarrow O_i$.

– $\Phi' = \Phi$ implies that $E_{i+1} \Rightarrow E_i$

- Case [**D–SEQ–2**]: by induction hypothesis (similar to [**D–ASSIGN–2**]).
- Case [**D–RET–2**]: by induction hypothesis (similar to [**D–ASSIGN–2**]).

### 5.6.4 Soundness of the Fixed-Point Analysis

To complement the proof of the main theorem 5.1, we will show that our fixed-point analysis always infers sound method summaries.

**Theorem 5.3 (Soundness of the Fixed-Point Analysis).** *Given a method declaration, we can show that the fixed-point inference applied to the constraint abstraction (obtained via the forward reasoning rules) would result in a sound summary* $\Phi_{mn}$.

**Proof:** The proof is done using induction on the height of the call graph dominated by the method $mn$. The base case where $mn$ does not call methods other than itself, can be proven as a special case of the induction step. For the induction step, the induction hypothesis assumes that the inference of methods called by $mn$ has computed sound summaries. Using this hypothesis, we aim to prove that the fixed-point analysis computes a sound summary for the method $mn$.

The first step in the inference process is to derive constraint abstractions from a given method declaration using the forward reasoning rules. These rules are applied recursively on sub-expressions of the method body and, with one exception, derive constraints equivalent to the respective sub-expression. The exception is the rule [**CALL**], where, rather than equivalent, the constraint that is derived is an over-approximation of the method call since the callee has a sound summary (from the induction hypothesis). Consequently, we can show that the constraint abstractions $mnOK$ and $mnERR$ are consistent with the method declaration from which they are derived using the forward reasoning rules. Furthermore the constraint abstractions are monotonic functions defined on the abstract domain (e.g. disjunctive polyhedron domain) with values in the same domain. The domain contains elements that are formulae over a fixed set of variables $\{v_1, .., v_n, v'_1, .., v'_{m-1}\}$, where $v_1, .., v_{m-1}$ are parameters passed by reference and $v_m, .., v_n$ are parameters passed by value.

The fixed-point analysis computes iteratively a sequence starting with the least element of the domain (the formula `false`). Being applied to a monotonic function, this computation will result in an ascending sequence. To ensure convergence of this sequence, a widening operator is used. The result is then guaranteed to be an upper approximation of the least fixed point for the constraint abstractions $mnOK/mnERR$.

Given that the results of the fixed-point analysis $O_{mn}/E_{mn}$ are over-approximations of the least fixed points (lfp) of $mnOK/mnERR$ abstractions, we can apply the judgement for the method body $e$ and prove that the implications required by the checking rule [**CHECK–METH**] hold as follows:

$$\frac{\vdash \{nochange(W)\}\, e\, \{\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}\} \quad \exists V \cdot \phi_1 \Rightarrow O_{mn} \qquad \exists R \cdot \phi_2 \Rightarrow E_{mn}}{\vdash t_0\ mn((\texttt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \texttt{where}\ \{\texttt{OK} : O_{mn}, \texttt{ERR} : E_{mn}\}\{e\}}$$

The formulae $\phi_1$ and $\phi_2$ correspond to the constraint abstraction functions $mnOK$ and $mnERR$ where the recursive calls are replaced by $O_{mn}$ and $E_{mn}$. Since $O_{mn}$ and $E_{mn}$ are over approximations of the lfp, they are reductive points of these functions [117, Sec 4.2, Appendix A.4]. As a consequence, the results from the judgement are more precise formulae and the following implications hold: $\exists V \cdot \phi_1 \Rightarrow O_{mn}$ and $\exists R \cdot \phi_2 \Rightarrow E_{mn}$. Thus the premises of the checking rule are satisfied and $\Phi_{mn}$ is shown to be a sound summary for its corresponding method declaration.

The current proof can be extended to handle mutual recursive functions. In this case, fixed points are computed simultaneously for all the mutually recursive constraint abstraction functions. □

A further lemma shows that our fixed-point analysis always terminates:

**Lemma 5.4 (Termination of Forward Analysis).** *Forward analysis comprises of two main parts (i) to build two constraint abstractions per method, (ii) fixed point analysis for each recursive abstraction. Both parts terminate.*

**Proof:** The forward reasoning traverses each program via a well-founded recursion over the expression and is therefore guaranteed to terminate for programs of finite code size. The termination property of fixed point analysis is dependent on the abstraction domain and techniques used for approximation and widening. For linear arithmetic domain, we can use the result of [37] whereby hulling and widening are used to ensure that

constraints encountered during conjunctive fixed-point have at most finite variations. This result extends also to k-bounded disjunctive formulae [138].

### 5.6.5 Corollaries of the Main Theorem

We shall now show four results that are corollaries of the Theorem on Soundness of Summary Outcomes. The first corollary confirms that we have a true error from the must-bug condition; secondly, we can guarantee a safe execution from the never-bug condition. Thirdly, we have a diverging execution from the loop condition. The fourth corollary applies when neither of the previous three cases holds: for inputs that satisfy the may-bug condition, it is possible to have either a safe execution, a true error or a diverging execution.

**Corollary 5.5 (Definite Error from Must-Bug).** *Given an arbitrary expression $e_1$, consider an inference judgement as follows: $\vdash \{P_1\}\, e_1\, \{\{\texttt{OK}{:}O_1,\ \texttt{ERR}{:}E_1\}\}$. For each state $s_1$ such that $s_1 \not\models \mathit{PreSt}(O_1)$, it is never the case that $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \delta\rangle$. This means that either $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \bot\rangle$ or $\langle s_1, e_1\rangle \not\hookrightarrow^*$ does not terminate.*

**Proof:** Theorem 5.1 confirms that if the execution is successful then the consistency property $s_1 \models \mathit{PreSt}(O_1)$ holds. Consequently, if the consistency property does not hold $s_1 \not\models \mathit{PreSt}(O_1)$, then the execution cannot be successful; it either fails or diverges. □

**Corollary 5.6 (Definite Safety from Never-Bug).** *Given an arbitrary expression $e_1$, consider an inference judgement as follows: $\vdash \{P_1\}\, e_1\, \{\{\texttt{OK}{:}O_1,\ \texttt{ERR}{:}E_1\}\}$. For each state $s_1$ such that $s_1 \not\models \mathit{PreSt}(E_1)$, it is never the case that $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \bot\rangle$. This means that either $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \delta\rangle$ or $\langle s_1, e_1\rangle \not\hookrightarrow^*$ does not terminate.*

**Proof:** Theorem 5.1 confirms that if the execution fails then the consistency property $s_1 \models \mathit{PreSt}(E_1)$ holds. Consequently, if the consistency property does not hold $s_1 \not\models \mathit{PreSt}(E_1)$, then the execution cannot fail; it is either successful or it diverges. □

**Corollary 5.7 (Definite Non-termination from Loop Condition).** *Given an arbitrary expression $e_1$, consider a judgement as follows: $\vdash \{P_1\}\, e_1\, \{\{\texttt{OK}{:}O_1,\ \texttt{ERR}{:}E_1\}\}$. For each state $s_1$ such that $s_1 \not\models \mathit{PreSt}(O_1)$ and $s_1 \not\models \mathit{PreSt}(E_1)$, it is neither the case that $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \bot\rangle$, nor that $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \delta\rangle$. This means that $\langle s_1, e_1\rangle \not\hookrightarrow^*$ does not terminate.*

**Proof:** Using the previous two corollaries, we denote by $s_1$ a state that does not satisfy neither of the following consistency properties $s_1 \not\models PreSt(O_1)$ and $s_1 \not\models PreSt(E_1)$. Then the execution is neither successful nor failed. The only possible alternative is that $\langle s_1, e_1\rangle \not\rightarrow^*$ does not terminate. □

**Corollary 5.8 (Indefinite kind of execution from May-Bug).** *Given an arbitrary expression $e_1$, consider an inference judgement as follows: $\vdash \{P_1\}\, e_1\, \{\{\texttt{OK}{:}O_1,\ \texttt{ERR}{:}E_1\}\}$. For each state $s_1$ such that $s_1 \models PreSt(O_1)$ and $s_1 \models PreSt(E_1)$, all the following three alternatives are possible: $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \delta\rangle$, $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \bot\rangle$ or $\langle s_1, e_1\rangle \not\rightarrow^*$ does not terminate.*

**Proof:** This corollary is vacuously true. □

## 5.7 Related Work

Most program analyses working towards the goal of bug-free programs can be divided broadly depending on their goal: proving safety of programs, finding bugs or synergistic approaches trying to prove safety and, at the same time, find bugs where possible. We summarize the main approaches from these three categories in Figure 5.9. The second column from the figure lists the direction in which the program is traversed, either forward (FW), backward (BW) or a combination of the two. The presented approaches can also be classified depending on the approximation done: either over-approximating, under-approximating, exact symbolic execution or a combination. Note that the effectiveness of the symbolic execution is inherently limited by the constraint solver or the theorem prover that is used. We also list if the analysis is designed to be modular, where each method is analysed in isolation to derive a summary. The method summary would further be used instead of reanalyzing the method at each of its call sites. Column 5 shows if the respective approaches are either meant to terminate or run until a time limit is reached. In general, it is difficult to have a converging answer from approaches based on abstraction refinement or by testing an unbounded number of execution paths, so in practice a time limit is imposed. Finally, column 6 shows what is the general goal of the analysis: safety, bugs or a combination of the two. Additionally, the alarms reported by the analysis can be classified as *bugs(1)* (an alarm that is either a true bug or a false positive), *bugs(2)* (an alarm that is a true bug, if the program

| Analysis | Direction | Approximation | Mod | Term | Goal |
|---|---|---|---|---|---|
| Suzuki et al.[144] | BW | under | NO | YES | safety |
| Astrée[12] | FW | over | NO | YES | safety |
| VeriSoft [59], CMC [111] | FW | under | NO | NO | bugs |
| DART [62], EXE [17] | FW | under+sym | NO | NO | bugs |
| SMART [61] | FW | under+sym | YES | NO | bugs |
| Saturn [148] | FW | over+under | YES | YES | bugs(1) |
| Slam[5], BLAST [80] | FW+BW | over+sym | NO | NO | safety+bugs(2) |
| Pasareanu et al. [120] | FW | under+over | NO | NO | safety+bugs |
| Synergy [67] | FW+BW | over+under+sym | NO | NO | safety+bugs |
| Syntox [15] | FW+BW | over+under | NO | YES | safety+bugs(2) |
| Rival [136] | FW+BW | over+under | NO | YES | safety+bugs(2) |
| Dualyzer | FW | over | YES | YES | safety+bugs(2) |

**Figure 5.9:** Classification of various analyses for proving safety and finding bugs

terminates), or *bugs* (an alarm that is unconditionally a true bug).

**Proving safety:** The first camp is concerned with proving safety of programs and its proponents are shown in the top lines of the Figure 5.9. It needs to find a way to abstract all the possible concrete executions into a statically computable form. The abstraction may represent an over-approximation of the state at some program point computed using a forward traversal of the program as in the seminal paper of Cousot and Halbwachs [43]. Conversely, the statically computed abstraction may represent an under-approximation of the state leading to a program error derived using a backward traversal of the program. This second approach computes loop invariants using the induction-iteration method pioneered by Suzuki and Ishihata [144] and later enhanced by Xu et al. [149]. Various trade-offs between precision of the underlying abstraction and efficiency of the safety analysis have been explored: the interval domain, the polyhedron domain [43] and the octagon domain [104] are just a few of the proposed abstractions. In fact, safety analyzers that scale to large critical programs like Astrée[12] or C Global Surveyor [145] use elaborate combinations of abstract domains to achieve maximum efficiency. Two other approaches towards proving safety are extended static checking [55] and program verification. In these cases, pre/post annotations have to be designed by programmers to provide further guidance on over-approximation, especially for recursive methods. As a summary for all these analyses, when they cannot prove safety, alarms that may include false positives will be signaled. The user of the analyzer is left with the job of manually distinguishing false alarms from real bugs.

Our modelling of error outcomes is related to the specification of exceptional conditions that has been pioneered for SPEC# [97]. However, this work focuses on safety guarantee for exception-based programs, and is not designed for bug finding purposes. Moreover, it currently requires users to specify each exceptional behaviour, while our approach automatically infers both success and failure outcomes.

**Finding bugs:** The second camp is primarily concerned with finding bugs in software, so that faulty programs could be quickly patched. Traditionally, program testing has been used for detecting incorrect programs. More recently, systematic testing or concrete state space exploration has been implemented in model checkers like VeriSoft [59] or CMC [111]. It attempts to search through all the feasible paths of the program, uncovering real bugs (no false positives). Systematic testing cannot achieve full path coverage, so its results represent an under-approximation of all the concrete executions of the program. As search may not terminate in a reasonable amount of time, a limit is set in practice on the number of paths that are covered.

A recent project, called DART [62], attempts to find more errors in a systematic fashion by keeping a stack of conditional tests encountered during execution. The gathered conditionals are used to generate new test cases that would allow deeper branches to be explored, so as to find real bugs, where possible. It also combines concrete with symbolic execution in order to alleviate the limitations of the constraint solver used in symbolic execution. Whenever the constraint solver does not know how to resolve a conditional test, DART simplifies this constraint using the concrete values of the inputs involved in the test. EXE [17] even used symbolic execution to explore conditional branches exhaustively. One problem with exact symbolic execution is that it requires heuristic to analyse each loop/recursion that is not bounded by a constant, and would fail for infinite loop. Another issue is that symbolic solvers are typically restricted in scope and may be inefficient, but [17] provided some solutions to this difficulty. To reduce the complexity of the automated testing approach, an extension of DART named SMART [61] generates the tests compositionally on a per method basis. On the whole, the bugs that are discovered are sound, but some bugs may remain undiscovered.

Closer in spirit to over-approximating static analyses, bug-finding tools like xgcc [78],

FindBugs [83], Saturn [148] or FastCheck [22] take some unsound (under-approximating) decisions in order to minimize the number of false positives. For example, Saturn only considers some aliasing possibilities between the parameters of a method and only analyzes a bounded number of iterations through a loop. Due to this combination of over and under approximation, these tools aim to report few false positives, but neither guarantee program safety, nor report only true bugs.

**Proving safety + finding bugs:** Synergistic approaches for both proving safety and finding bugs usually rely on a combination of over and under approximation. In contrast, our proposal is more integrated as it computes forward only over-approximations.

Model checking based on abstraction refinement is often referred as CEGAR (counter-example guided automated refinement) and tools like Slam [5] or BLAST [80, 81, 11] are based on this paradigm. In a first step, Slam and BLAST perform a forward-directed over-approximating search for possible bugs. If no bugs are found, then the safety of the program has been proven. Otherwise, starting with a possible bug, a counter-example trace is analyzed backward via symbolic reasoning in order to derive its weakest liberal precondition. If the counter-example is shown to be feasible, then a true bug is reported. If the counter-example is shown to be infeasible, the abstraction is refined and the search process is iterated. More recently, a model checking algorithm has been devised around refinement of under-approximations to better preserve the bug detection ability of the checker [120]. Yet another abstraction refinement is based on mixed transition systems that represent both must and may transitions [72]. This approach can reason about both safety and true bugs, but is (as yet) restricted to non-recursive programs. Similarly, as elaborated earlier, Synergy [67] extends predicate abstraction and refinement mechanism with DART-style program testing.

Syntox [15] is a system for abstract debugging of imperative Pascal programs. It can prove safety and find bugs by using a combination of forward over-approximating analysis and backward under-approximating analysis. We highlight the main differences compared to our approach. Firstly, our analysis does not require a separate backward phase, since it is based on a relational semantics and can derive input conditions directly from the forward phase. Secondly, Syntox uses a less precise interval abstract domain,

where in general the complement of an interval cannot be represented as an interval. Thus a separate greatest fixed point computation is used to determine must-bug conditions. Finally, the correctness conditions determined by Syntox are necessary, but not sufficient like the never-bug condition derived by Dualyzer.

In order to investigate the origin of the alarms raised by the static analyzer Astrée[12], Rival used iterated forward-backward over-approximating analysis to prove safety of assertions [136]. Despite elaborate combination of abstractions, some alarms cannot be resolved by over-approximation alone. Understanding if an alarm is a true bug is facilitated by under-approximating techniques such as input selection or restriction to an execution pattern. The input selection process is not currently automated, but made easier by semantic slicing techniques. The process of restriction to an execution pattern and guiding the analysis towards true bugs is in general incomplete and may not converge. However, [136] reports that in practice all considered alarms from their set of benchmarks could be classified by the above-mentioned techniques. The classification of alarms is similar to ours in that an alarm indicates either a true bug or a non-terminating program.

In the absence of runtime errors, over-approximating static analysis may conclude that, if a program location is unreachable, then the program exhibits non-terminating behaviour. In general, in the presence of runtime errors, an unreachable location may indicate either non-termination or a runtime error [15, 41]. In our approach, we can prove non-termination by tracking reachability of *both* successful and failed executions. A different approach to proving non-termination [70] proceeds in two steps: first it dynamically enumerates possible non-terminating program paths and then statically proves their feasibility by inferring a recurrent set of states.

In a recent position paper [60], Godefroid likens may- and must- analyses to the Yin and Yang of program analysis. He advocated for greater efforts to be devoted to must-analysis, especially because practitioners found it more useful, while researchers (on program analysis and verification) have focused mostly on may-analysis instead. Our proposal can be viewed as heeding this call, as we attempt to achieve a balance between the Yin and Yang of program analyses, but using a new framework based on dual static analysis.

## 5.8 Summary

Our approach is based on a modular static analysis and is aimed at proving safety or discovering true bugs. To achieve both goals, our key innovation is the simultaneous capture of *error outcomes* and *successful outcomes*. Moreover, we have also shown that our static analysis technique is able to detect a subclass of *definite non-termination* when identifying unreachable states of recursive methods. Our experiments have shown that this approach can also use may-bug conditions to guide precision improvement based on disjunctive abstract domain. While we have focused our efforts on bugs discovery, our use of dual static analysis can be viewed as an instance of a general framework that simultaneously infers trigger conditions for an arbitrary property `P` and its complement $\overline{\texttt{P}}$. We believe that this exploration might open up a fertile ground whereby better may- and must-analyses can be more effectively developed.

# CHAPTER VI

# CONCLUSIONS

The focus of this thesis was to investigate modular static analysis with the goal of proving program safety and detecting program errors. This chapter reviews the main results of the current work and then makes suggestions for future research and possible applications.

## 6.1 Main Results

In this thesis, we described three main results. Firstly, we presented a new disjunctive abstract domain meant to enhance analysis precision at a reasonable cost. Secondly, we proposed a modular technique for deriving preconditions sufficient to guarantee program safety. With these two techniques, we were able to derive both postconditions and preconditions and realized a completely modular analyzer for proving program safety. Since an analyzer that aims to prove program safety may report alarms that correspond in part to false positives, there is a need to (manually) classify the feasibility of alarms. Our third proposal was a dual static analysis that can identify (automatically) a part of the alarms as being true errors. More specifically, our dual static analysis was able to identify both a never-bug condition that implies program safety and a must-bug condition that leads to true errors (modulo program termination).

## 6.2 Future Work

In this section, we make some suggestions on how to improve the research presented in this thesis and increase its practical impact.

**CIL front-end :** Being based on the core IMP language, our prototype implementation was restricted to smaller-sized programs. In our experiments, we circumvented this restriction by building a code pre-processor. We plan to extend it to the point that it can accept CIL programs as input. The CIL framework [114] is a widely used tool for analysis of C programs and should help in making our analyses more widely available.

Indirectly, by this extension we hope to successfully analyze larger programs.

**Disjunctive weakly relational abstract domains :** Another opportunity for improving the efficiency of our analyses is to adapt the proposal from Chapter 3 to a more efficient but less precise base domain. There we used the affinity function for lifting a base (conjunctive) abstract domain to its (disjunctive) powerset extension, and we demonstrated this idea in the context of the base polyhedron domain. It seems promising to extend this proposal to weakly relational abstract domains like the octagons [104] or the template constraint matrices [139].

**Fixed-point analyzer :** With the general applicability of fixed-point computations to static analyses, we believe in the utility of making Disj-Fix as a separate application package. To increase its applicability, we should make its implementation parametric in terms of abstract domain and disjunctive heuristic function.

**Abstraction refinement :** Our dual static analysis generates method summaries that are useful for abstraction refinement. Computing both an input condition that leads to safety (never-bug) and an input condition that leads to errors (must-bug) allows an abstraction refinement procedure to concentrate on those inputs that do not satisfy either of these conditions. We used this observation to selectively increase the precision of the abstract domain. A natural extension is to provide more strategies for abstraction refinement, including the iteration of the forward analysis assuming as precondition the may-bug determined previously. Inspired by the refinement techniques proposed in [68, 66], we speculate that it should also be useful to refine abstract operations like selective hulling or powerset widening when they are performed with a value less than 100% affinity.

**Non-termination and termination analyses :** With the dual static analysis, we proposed to classify inputs based on whether they are sufficient/necessary to guarantee safety or the encounter of an error. Our classification is related to the classification given in the weakest precondition calculus of Dijkstra [49, Chapter 3]. While our classification is concerned with program safety and bug detection, Dijkstra's classification is more

concerned with characterizing termination and whether the final state satisfies a given postcondition. In addition to this semantic classification of inputs, our static analysis also provided algorithms that compute a precise input partitioning. A further challenging task is to provide algorithms that classify inputs with regard to program termination. There are some promising forays in finding inputs that lead to non-termination [70] and inputs that lead to conditional termination [33].

**Parallelization of analysis :** For the analysis of larger programs, we should consider an unavoidable increase in the analysis time. To counter this increase, we could efficiently parallelize our analyses due to their modular characteristic. The parallelization algorithm would naturally follow the method boundaries for dividing the analysis tasks, as proposed and implemented in other summary-based analyses [20, 50].

This thesis has focused on investigating modular static analysis in the context of numerical abstract domains. To this goal, we derived efficient method summaries and obtained precise method abstractions by using disjunctive invariants. We hope that the techniques developed here will be helpful in devising more general modular static analyses, increase their applicability and finally lead to more dependable software.

# BIBLIOGRAPHY


[1] Apt, K. R., "Ten years of Hoare's Logic: A survey - Part 1," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 3, no. 4, pp. 431–483, 1981.

[2] Bagnara, R., Hill, P., and Zaffanella, E., "Widening operators for powerset domains.," in *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pp. 135–148, 2004.

[3] Bagnara, R., Hill, P. M., and Zaffanella, E., "Not necessarily closed convex polyhedra and the double description method," *Formal Aspects of Computing*, vol. 17, no. 2, pp. 222–257, 2005.

[4] Bagnara, R., Hill, P. M., and Zaffanella, E., "Widening operators for powerset domains," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 4-5, pp. 449–466, 2006.

[5] Ball, T. and Rajamani, S., "Automatically validating temporal safety properties of interfaces.," in *SPIN Workshop*, pp. 103–122, 2001.

[6] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A., "Thorough static analysis of device drivers," in *EuroSys*, pp. 73–85, 2006.

[7] Ball, T., Millstein, T. D., and Rajamani, S. K., "Polymorphic predicate abstraction," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 2, pp. 314–343, 2005.

[8] Bell Labs, "Ckit: A front end for C in SML." Available from: http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ckit/overview.html.

[9] Berdine, J., Cook, B., Distefano, D., and O'Hearn, P. W., "Automatic termination proofs for programs with shape-shifting heaps," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pp. 386–400, 2006.

[10] Beyer, D., Henzinger, T., Jhala, R., and Majumdar, R., "The software model checker Blast: Applications to software engineering," *International Journal on Software Tools for Technology Transfer*, 2007.

[11] Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R., "Checking memory safety with Blast," in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, pp. 2–18, 2005.

[12] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X., "A static analyzer for large safety-critical software," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 196–207, 2003.

[13] Bodik, R., Gupta, R., and Sarkar, V., "ABCD: Eliminating array bounds checks on demand," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 321–333, 2000.

[14] Bodik, R., Gupta, R., and Soffa, M., "Complete removal of redundant expressions," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 1–14, June 1998.

[15] Bourdoncle, F., "Abstract debugging of higher-order imperative languages," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 46–55, 1993.

[16] Bradley, A., Manna, Z., and Sipma, H., "What's decidable about arrays?," in *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pp. 427–442, 2006.

[17] Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., and Engler, D., "EXE: Automatically generating inputs of death," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2006.

[18] Chang, B. Y. E. and Rival, X., "Relational inductive shape analysis," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 247–260, 2008.

[19] Chatterjee, R., Ryder, B., and Landi, W., "Relevant context inference.," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1999.

[20] Chatterjee, R., *Modular Data-flow Analysis of Statically Typed Object-oriented Programming Languages*. PhD thesis, Rutgers, The State University of New Jersey, 1999.

[21] Cheng, B.-C. and mei W. Hwu, W., "Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 57–69, 2000.

[22] Cherem, S., Princehouse, L., and Rugina, R., "Practical memory leak detection using guarded value-flow analysis," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[23] Chin, W. N. and Khoo, S. C., "Calculating sized types," *Higher-Order and Symbolic Computation*, vol. 14, no. 2-3, pp. 261–300, 2001.

[24] Chin, W. N., Khoo, S. C., and Xu, D. N., "Extending sized type with collection analysis," in *Proceedings of the ACM Symposium on Partial Evaluation and Program Manipulation (PEPM)*, pp. 75–84, ACM Press, 2003.

[25] Chin, W.-N., Khoo, S.-C., Qin, S., Popeea, C., and Nguyen, H. H., "Verifying Safety Policies with Size Properties and Alias Controls," in *Proceedings of the International Conference on Software Engineering (ICSE)*, (St. Louis, Missouri), May 2005.

[26] Chin, W.-N., Nguyen, H. H., Popeea, C., and Qin, S., "Analysing memory resource bounds for low-level programs," in *Proceedings of the International Symposium on Memory Management (ISMM)*, ACM, 2008.

[27] Chin, W. and Khoo, S., "Calculating sized types," in *Proceedings of the ACM Symposium on Partial Evaluation and Program Manipulation (PEPM)*, (Boston, Massachusetts), pp. 62–72, Jan. 2000.

[28] Chin, W., Khoo, S., and Xu, D. N., "Deriving pre-conditions for array bound check elimination," in *2nd Symp. on Programs as Data Objects*, (Aarhus, Denmark), pp. 2–24, Springer Verlag, May 2001.

[29] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H., "Counterexample-guided abstraction refinement.," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pp. 154–169, 2000.

[30] Colby, C. and Lee, P., "Trace-based program analysis.," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 195–207, 1996.

[31] Condit, J., Harren, M., Anderson, Z., Gay, D., and Necula, G. C., "Dependent types for low-level programming," in *Proceedings of the European Symposium on Programming (ESOP)*, pp. 520–535, 2007.

[32] Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., and Vardi, M. Y., "Proving that programs eventually do something good," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 265–276, 2007.

[33] Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., and Sagiv, M., "Proving conditional termination." Unpublished, 2008.

[34] Cook, B., Podelski, A., and Rybalchenko, A., "Termination proofs for systems code," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 415–426, 2006.

[35] Cousot, P., "Constructive design of a hierarchy of semantics of a transition system by abstract interpretation," *Theoretical Computer Science*, vol. 277, no. 1-2, pp. 47–103, 2002.

[36] Cousot, P. and Cousot, R., "Static determination of dynamic properties of programs," in *Proceedings of the Second International Symposium on Programming*, pp. 106–130, 1976.

[37] Cousot, P. and Cousot, R., "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1977.

[38] Cousot, P. and Cousot, R., "Static determination of dynamic properties of recursive procedures," in *IFIP Conference on Formal Description of Programming Concepts*, pp. 237–277, 1977.

[39] Cousot, P. and Cousot, R., "Systematic design of program analysis frameworks," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, (San Antonio, Texas), 1979.

[40] Cousot, P. and Cousot, R., "Modular static program analysis," in *Proceedings of the International Conference on Compiler Construction (CC)*, 2002.

[41] Cousot, P. and Cousot, R., "On abstraction in software verification," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pp. 37–56, Springer-Verlag London, UK, 2002.

[42] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X., "The ASTREÉ analyzer," in *Proceedings of the European Symposium on Programming (ESOP)*, pp. 21–30, 2005.

[43] Cousot, P. and Halbwachs, N., "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 84–96, 1978.

[44] Cousot, P., "Semantic foundations of program analysis," in *Program Flow Analysis: Theory and Applications* (Muchnick, S. and Jones, N., eds.), ch. 10, pp. 303–342, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[45] Darga, P. and Boyapati, C., "Efficient software model checking of data structure properties," in *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.

[46] Das, M., Lerner, S., and Seigle, M., "ESP: path-sensitive program verification in polynomial time," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 57–68, 2002.

[47] Dean, J., Chambers, C., and Grove, D., "Selective specialization for object-oriented languages," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 93–102, 1995.

[48] DeLine, R. and Fahndrich, M., "Enforcing high-level protocols in low-level software," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2001.

[49] Dijkstra, E. W., *A Discipline of Programming*. Prentice-Hall, 1976.

[50] Dillig, I., Dillig, T., and Aiken, A., "Static error detection using semantic inconsistency inference," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 435–445, 2007.

[51] Dongarra, J., Luszczek, P., and Petitet, A., "The LINPACK benchmark: Past, present, and future," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 1–18, 2003.

[52] Dor, N., Rodeh, M., and Sagiv, M., "CSSV: towards a realistic tool for statically detecting all buffer overflows in C," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 155–167, 2003.

[53] E.M.Clarke, Grumberg, O., and Peled, D., *Model checking*. The MIT Press, 2000.

[54] Flanagan, C., "Hybrid type checking.," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 245–256, 2006.

[55] Flanagan, C., Leino, K., Lillibridge, M., Nelson, G., Saxe, J., and Stata, R., "Extended Static Checking for Java," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2002.

[56] Flanagan, C. and Qadeer, S., "Predicate abstraction for software verification," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2002.

[57] Flanagan, C. and Saxe, J., "Avoiding exponential explosion: Generating compact verification conditions," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2001.

[58] Giacobazzi, R. and Ranzato, F., "Optimal domains for disjunctive abstract intepretation.," *Science of Computer Programming*, vol. 32, no. 1-3, pp. 177–210, 1998.

[59] Godefroid, P., "Model checking for programming languages using VeriSoft," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[60] Godefroid, P., "The soundness of bugs is what matters," in *Workshop on Eval. of Software Defect Detection Tools*, 2005.

[61] Godefroid, P., "Compositional dynamic test generation," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 47–54, 2007.

[62] Godefroid, P., Klarlund, N., and Sen, K., "DART: Directed Automated Random Testing.," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 213–223, 2005.

[63] Gopan, D. and Reps, T. W., "Low-level library analysis and summarization," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pp. 68–81, 2007.

[64] Granger, P., "Static analysis of arithmetical congruences," *International Journal of Computer Mathematics*, vol. 30, p. 165190, 1989.

[65] Granger, P., "Static analysis of linear congruence equalities among variables of a program," in *TAPSOFT, Vol.1*, pp. 169–192, 1991.

[66] Gulavani, B. S., Chakraborty, S., Nori, A. V., and Rajamani, S. K., "Refining abstract interpretations," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[67] Gulavani, B., Henzinger, T., Kannan, Y., Nori, A., and Rajamani, S., "SYNERGY: A new algorithm for property checking," in *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)*, 2006.

[68] Gulavani, B. and Rajamani, S., "Counterexample driven refinement for abstract interpretation.," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2006.

[69] Gulwani, S. and Tiwari, A., "Computing procedure summaries for interprocedural analysis," in *Proceedings of the European Symposium on Programming (ESOP)*, pp. 253–267, 2007.

[70] Gupta, A., Henzinger, T. A., Majumdar, R., Rybalchenko, A., and Xu, R.-G., "Proving non-termination," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 147–158, 2008.

[71] Gupta, R., "A fresh look at optimizing array bound checking," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, (New York), pp. 272–282, June 1990.

[72] Gurfinkel, A., Wei, O., and Chechik, M., "YASM: A software model-checker for verification and refutation," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pp. 170–174, 2006.

[73] Gustavsson, J. and Svenningsson, J., "Constraint abstractions," in *Programs as Data Objects II*, (Aarhus, Denmark), pp. 63–83, May 2001.

[74] Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., and Brown, T., "Mibench: A free, commercially representative embedded benchmark suite," in *4th IEEE International Workshop on Workload Characteristics*, 2001.

[75] Hackett, B. and Aiken, A., "How is aliasing used in systems software?," in *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)*, pp. 69–80, 2006.

[76] Hackett, B., Das, M., Wang, D., and Yang, Z., "Modular checking for buffer overflows in the large," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 232–241, 2006.

[77] Halbwachs, N., *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme*. Thèse de 3ème cycle d'informatique, Université scientifique et médicale de Grenoble, Grenoble, France, Mar. 1979.

[78] Hallem, S., Chelf, B., Xie, Y., and Engler, D., "A system and language for building system-specific, static analyses," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[79] Heintze, N. and Tardieu, O., "Ultra-fast aliasing analysis using CLA: A million lines of C code in a second.," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[80] Henzinger, T., Jhala, R., Majumdar, R., and Sutre, G., "Lazy abstraction," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2002.

[81] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G., "Software verification with BLAST," in *SPIN Workshop*, pp. 235–239, 2003.

[82] Hoare, C. A. R., "An axiomatic basis for computer programming," *Communications of the ACM*, 1969.

[83] Hovemeyer, D. and Pugh, W., "Finding bugs is easy.," *SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[84] Hughes, J., Pareto, L., and Sabry, A., "Proving the correctness of reactive systems using sized types," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 410–423, ACM Press, Jan. 1996.

[85] Jaffar, J., Santosa, A. E., and Voicu, R., "A CLP method for compositional and intermittent predicate abstraction," in *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pp. 17–32, 2006.

[86] Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y., "Cyclone: A safe dialect of C," in *USENIX Annual Technical Conference, General Track*, pp. 275–288, 2002.

[87] Jung, Y., Kim, J., Shin, J., and Yi, K., "Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis.," in *Proceedings of the International Static Analysis Symposium (SAS)*, pp. 203–217, 2005.

[88] Kelly, P., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., and Wonnacott, D., "The Omega Library Version 1.1.0 Interface Guide," tech. rep., University of Maryland, College Park, Nov. 1996. `http://www.cs.umd.edu/projects/omega`.

[89] King, J., "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[90] Kolte, P. and Wolfe, M., "Elimination of redundant array subscript range checks," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 270–278, ACM Press, June 1995.

[91] Ku, K., Hart, T. E., Chechik, M., and Lie, D., "A buffer overflow benchmark for software model checkers," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pp. 389–392, 2007.

[92] Lahiri, S. K. and Bryant, R. E., "Indexed predicate discovery for unbounded system verification," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2004.

[93] Lalire, G., Argoud, M., and Jeannet, B., "Interproc analyzer," 2008. http://bjeannet.gforge.inria.fr/interproc/.

[94] Lam, P., Kuncak, V., and Rinard, M., "Cross-cutting techniques in program specification and analysis," in *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, March 2005.

[95] Lassez, J.-L., Huynh, T., and McAloon, K., "Simplification and elimination of redundant linear arithmetic constraints," in *Workshop on Constraint Logic Programming*, pp. 73–87, 1991.

[96] Lassez, J.-L. and McAloon, K., "A canonical form for generalized linear constraints," *Journal of Symbolic Computation*, vol. 13, no. 1, pp. 1–24, 1992.

[97] Leino, K. R. M. and Schulte, W., "Exception Safety for C#.," in *Proceedings of the Conference on Software Engineering and Formal Methods (SEFM)*, pp. 218–227, 2004.

[98] Leino, K. R. M. and Logozzo, F., "Loop invariants on demand.," in *Proceedings of the ASIAN Symposium on Programming Languages and Systems (APLAS)*, 2005.

[99] Lions, J. L. and et al, "ARIANE 5 - Flight 501 failure - Report by the inquiry board," 1996. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

[100] Logozzo, F. and Fahndrich, M., "Pentagons: A weakly relational abstract domain for the efficient validation of array accesses," in *ACM Symposium on Applied Computing (SAC)*, 2008.

[101] Luján, M., Gurd, J. R., Freeman, T. L., and Miguel, J., "Elimination of Java array bounds checks in the presence of indirection," in *ACM Joint Java Grande-IScope Conf.*, pp. 76–85, 2002.

[102] Martel, M., "Propagation of roundoff errors in finite precision computations: A semantics approach," in *Proceedings of the European Symposium on Programming (ESOP)*, pp. 194–208, 2002.

[103] Mauborgne, L. and Rival, X., "Trace Partitioning in Abstract Interpretation Based Static Analyzers," in *Proceedings of the European Symposium on Programming (ESOP)*, 2005.

[104] Miné, A., "The octagon abstract domain," in *the Eighth Working Conference on Reverse Engineering*, 2001.

[105] Miné, A., "Relational abstract domains for the detection of floating-point runtime errors," in *Proceedings of the European Symposium on Programming (ESOP)*, pp. 3–17, 2004.

[106] Miné, A., *Weakly relational numerical abstract domains*. PhD thesis, École Normale Supérieure, 2004.

[107] Motzkin, T., Raiffa, H., Thompson, G., and Thrall, R., "The double description method," *Annals of Mathematics Studies*, pp. 51–73, 1953.

[108] Moy, Y., "Sufficient preconditions for modular assertion checking," in *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2008.

[109] Müller-Olm, M. and Seidl, H., "Precise interprocedural analysis through linear algebra," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 330–341, 2004.

[110] Müller-Olm, M. and Seidl, H., "A generic framework for interprocedural analysis of numerical properties," in *Proceedings of the International Static Analysis Symposium (SAS)*, pp. 235–250, 2005.

[111] Musuvathi, M., Park, D., Chou, A., Engler, D., and Dill, D., "CMC: a pragmatic approach to model checking real code," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[112] National Institute of Standards and Technology, "Java SciMark benchmark for scientific computing." http://math.nist.gov/scimark2.

[113] Necula, G. and Lee., P., "The design and implementation of a certifying compiler," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 333–344, 1998.

[114] Necula, G., McPeak, S., Rahul, S., and Weimer, W., "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proceedings of the International Conference on Compiler Construction (CC)*, 2002.

[115] Nguyen, H. H., David, C., Qin, S., and Chin, W. N., "Automated verification of shape and size properties via separation logic," in *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pp. 251–266, 2007.

[116] Nguyen, T. and Irigoin, F., "Efficient and effective array bound checking," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 527–570, 2005.

[117] Nielson, F., Nielson, H., and Hankin, C., *Principles of Program Analysis*. Berlin: Springer-Verlag, 1999.

[118] Nystrom, E. M., Kim, H.-S., and mei W. Hwu, W., "Bottom-up and top-down context-sensitive summary-based pointer analysis," in *Proceedings of the International Static Analysis Symposium (SAS)*, pp. 165–180, 2004.

[119] Orlovich, M. and Rugina, R., "Core expressions: An intermediate representation for expressions in C." Unpublished, 2006.

[120] Pasareanu, C., Pelánek, R., and Visser, W., "Concrete model checking with abstract matching and refinement.," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pp. 52–66, 2005.

[121] Peyton-Jones, S. and et al, "Glasgow Haskell Compiler." http://www.haskell.org/ghc.

[122] Popeea, C. and Chin, W.-N., "A type system for resource protocol verification and its correctness proof," in *Proceedings of the ACM Symposium on Partial Evaluation and Program Manipulation (PEPM)*, pp. 135–146, ACM, 2004.

[123] Popeea, C. and Chin, W.-N., "Inferring disjunctive postconditions," in *Asian Computing Science Conference (ASIAN)*, vol. 4435 of *Lecture Notes in Computer Science*, pp. 331–345, Springer, 2006.

[124] Popeea, C. and Chin, W.-N., "$\mu$CIL: A new core language for C." Unpublished, 2006.

[125] Popeea, C. and Chin, W.-N., "Dual analysis for proving safety and finding bugs," tech. rep., Dept of Computer Science, National Univ. of Singapore, 2008. `http://www.comp.nus.edu.sg/~corneliu/research/dual.tr.pdf`.

[126] Popeea, C., Xu, D. N., and Chin, W.-N., "A practical and precise inference and specializer for array bound checks elimination," tech. rep., Dept of Computer Science, National Univ. of Singapore, 2007. `http://www.comp.nus.edu.sg/~corneliu/research/array.tr.pdf`.

[127] Popeea, C., Xu, D. N., and Chin, W.-N., "A practical and precise inference and specializer for array bound checks elimination," in *Proceedings of the ACM Symposium on Partial Evaluation and Program Manipulation (PEPM)*, pp. 177–187, ACM, 2008.

[128] Presburger, M., "Uber die vollstandigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt," in *Congr. Math. des Pays Slaves*, 1929.

[129] Pugh, W., "The Omega Test: A fast practical integer programming algorithm for dependence analysis," *Communications of the ACM*, vol. 8, pp. 102–114, 1992.

[130] Pugh, W. and Wonnacott, D., "Eliminating false data dependences using the Omega Test," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 140–151, 1992.

[131] Pugh, W. and Wonnacott, D., "Constraint-based array dependence analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 3, pp. 635–678, 1998.

[132] Reps, T. W., Horwitz, S., and Sagiv, S., "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 49–61, 1995.

[133] Research Triangle Institute, *The Economic Impacts of Inadequate Infrastructure for Software Testing.* Planning Report 02-3, National Institute of Standards and Technology, May 2002.

[134] Rice, H. G., "Classes of recursively enumerable sets and their decision problems," *Trans. Amer. Math. Soc.*, vol. 74, p. 358366, 1953.

[135] Rival, X., *Traces Abstraction in Static Analysis and Program Transformation.* PhD thesis, École Normale Supérieure, 2005.

[136] Rival, X., "Understanding the origin of alarms in Astrée," in *Proceedings of the International Static Analysis Symposium (SAS)*, pp. 303–319, 2005.

[137] Rugina, R. and Rinard, M., "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 182–195, ACM Press, June 2000.

[138] Sankaranarayanan, S., Ivancic, F., Shlyakhter, I., and Gupta, A., "Static analysis in disjunctive numerical domains," in *Proceedings of the International Static Analysis Symposium (SAS)*, Springer LNCS, (Seoul, Korea), Aug. 2006.

[139] Sankaranarayanan, S., *Mathematical Analysis of Programs.* PhD thesis, Stanford University, 2005.

[140] Seidl, H., Flexeder, A., and Petter, M., "Interprocedurally analysing linear inequality relations," in *Proceedings of the European Symposium on Programming (ESOP)*, pp. 284–299, 2007.

[141] Sharir, M. and Pnueli, A., "Two approaches to interprocedural data flow analysis," in *Program Flow Analysis: Theory and Applications* (Muchnick, S. and Jones, N., eds.), ch. 7, pp. 189–234, Prentice-Hall, 1981.

[142] Sims, É.-J., "Extending separation logic with fixpoints and postponed substitution," *Theoretical Computer Science*, vol. 351, no. 2, pp. 258–275, 2006.

[143] Stephenson, A. G. and et al, "Mars Climate Orbiter - Mishap investigation board - Phase I report," 1999. ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf.

[144] Suzuki, N. and Ishihata, K., "Implementation of an array bound checker," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 132–143, 1977.

[145] Venet, A. and Brat, G., "Precise and efficient static array bound checking for large embedded C programs," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 231–242, 2004.

[146] Wu, Y. and Larus, J., "Static branch frequency and program profile analysis," in *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO)*, pp. 1–11, ACM Press, 1994.

[147] Xi, H. and Pfenning, F., "Eliminating array bound checking through dependent types," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, ACM Press, June 1998.

[148] Xie, Y. and Aiken, A., "Scalable error detection using boolean satisfiability," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 351–363, 2005.

[149] Xu, Z., Miller, B., and Reps, T., "Safety checking of machine code," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 70–82, ACM Press, June 2000.

[150] Yorsh, G., Ball, T., and Sagiv, M., "Testing, abstraction, theorem proving: Better together!," in *Proceedings of the International Symposium on Software Testing and Analysis(ISSTA)*, 2006.

[151] Yorsh, G. and Dor, N., "The design of CoreC," tech. rep., Tel-Aviv University, April 2003.

[152] Yorsh, G., Yahav, E., and Chandra, S., "Generating precise and concise procedure summaries," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 221–234, 2008.

# INDEX

affinity measure
  for hulling, 38
  for mixed constraints, 48
  for widening, 42
ascending chain, 27

composition operator, 34, 97, 98
constraint abstraction, 25

disjunctive abstract domain, 37

failure outcome (ERR), 94

gist operator, 65, 79
greatest element, 26, 37
greatest lower bound, 26

Hausdorff distance, 40
hull operator
  convex ($\oplus$), 26
  selective ($\oplus_m$), 36

lattice, 26
least element, 26, 37
least fixed point, 27
least upper bound, 26
linear inequality, 23

may-bug condition, 95
method summary, 10
modular analysis, 10
modularity principle, 3
monotone function, 26
must-bug condition, 94

never-bug condition, 94
nochange operator, 33

partial order, 26, 37
polyhedron abstract domain, 26
post fixed point, 27
postcondition, 25, 67, 96
poststate, 24, 33, 56, 116
precondition
  necessary for error, 29, 94
  necessary for safety, 28, 47, 94
  sufficient for error, 94
  sufficient for non-termination, 29, 105
  sufficient for safety, 28, 63, 67, 94
prestate, 24, 33, 56, 116
primitive method, 18, 68, 99

recursive invariant, 76

state-based analysis, 33
static judgement, 23, 45, 70
success outcome (OK), 94
summary-based analysis, 10

theory of linear arithmetic, 23
trace-based analysis, 33
transition formula, 24, 33

widening operator
  conjunctive ($\nabla$), 27
  powerset ($\nabla_m$), 42