Techniques for Precise and Scalable Verification of Array Programs

Thesis

Submitted in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

by

Unadkat Divyesh Pankaj
Roll No. 144057002

Under the guidance of
Prof. Supratik Chakraborty
and
Prof. Ashutosh Kumar Gupta

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
2022

© 2022 Unadkat Divyesh Pankaj
All Rights Reserved.

This work is licensed under the terms of the Creative Commons Attribution 4.0 International License. https://creativecommons.org/licenses/by/4.0/

To my loving parents

Gratitude for infrastructure and funding

tcs Research

Thesis Approval

The thesis entitled

Techniques for Precise and Scalable Verification of Array Programs

by

Unadkat Divyesh Pankaj
Roll No. 144057002

is approved for the degree of

Doctor of Philosophy

Examiner	Examiner
Guide	Co Guide

Chairman

Date: ______________

Place: ______________

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY, INDIA

CERTIFICATE OF COURSE WORK

This is to certify that **Unadkat Divyesh Pankaj** (Roll No. 144057002) was admitted to the candidacy of Ph.D. degree on 31 December 2014, after successfully completing all the courses required for the Ph.D. programme. The details of the course work done are given below.

S.No	Course Code	Course Name	Credits
1	CS 602	Applied Algorithms	6
2	CS 615	Formal Specification and Verification of Programs	6
3	CS 618	Program Analysis	6
4	CS 691	R & D Project	6
5	CS 713	Special Topics in Automata and Logics	6
6	CS 721	Introduction to Computational Complexity	6
7	CS 735	Formal Models for Concurrent and Asynchronous Systems	6
8	CSS 801	Seminar	4
9	HS 791	Communication Skills - I	PP
10	CS 702	Communication Skills - II	PP
		Total Credits	**46**

IIT Bombay

Date: Dy. Registrar (Academic)

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY, INDIA

A Declaration of Academic Honesty and Integrity

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Date:

Unadkat Divyesh Pankaj

Roll No. 144057002

Abstract

The use of software controlled systems in industrial and household appliances, where safety is of paramount concern, is increasing at an unprecedented rate. Programs with loops that manipulate arrays are quite common in such software and ensuring their correctness is extremely important. State-of-the-art tools use a combination of verification techniques that work well for certain classes of programs and assertions, and yield conservative results otherwise. In this thesis, we present three new compositional and property-driven verification techniques that use mathematical induction in novel ways to prove interesting properties of a class of array-manipulating programs with a symbolic parameter N. Specifically, if $\varphi(N)$ represents a pre-condition, $\psi(N)$ represents a post-condition and P_N represents a program, all parameterized by the same natural number N (can be extended to multiple parameters), the techniques presented in this thesis provide new ways of proving the parametric Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ for all $N > 0$.

Our first contribution is a property-driven verification method that infers array-access patterns in loops using simple heuristics and then uses this information to prove universally quantified assertions about arrays in a compositional manner. Specifically, we identify *tiles* of array-access patterns in a loop and use the tiling information to reduce the problem of checking a quantified assertion at the end of a loop to an inductive argument that checks only a *slice* of the assertion for a single iteration of the loop body. We show that this method extends naturally to programs with sequentially composed and nested loops. We have implemented the method in a tool called Tiler. We demonstrate that Tiler outperforms several state-of-the-art tools that verify array-manipulating programs on a large suite of interesting benchmarks.

The tiling technique requires invariants on slices of arrays for compositional reasoning. In general, such invariants are difficult to generate automatically. We overcome this limitation by presenting a second property-driven verification technique called '*full-*

program induction'. Instead of inducting over iterations of individual loops, this technique inducts over the parameter N considering the whole program, possibly containing multiple loops. It involves transforming the program P_N into a sequential composition of P_{N-1} and an appropriate '*difference*' program $\partial\mathsf{P}_N$. Using a standard induction argument, the proof of $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ is then reduced to proving a parametric Hoare triple over $\partial\mathsf{P}_N$, assuming $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$ holds. Since $\partial\mathsf{P}_N$ can be much simpler than P_N for certain classes of programs, this technique can result in significant simplification of our proof obligation. The novelty of this technique is that we can often bypass the generation and use of loop-specific invariants, even in programs with multiple loops, while such invariants are necessary in classical verification techniques. We have developed a prototype tool VAJRA to assess the efficacy of full-program induction. We demonstrate the performance of VAJRA vis-a-vis several state-of-the-art tools on a set of array-manipulating benchmarks from verification competitions and industry code.

As our third contribution, we adapt the full-program induction technique such that the computation of the difference program is significantly simplified. The resulting technique, called '*relational full-program induction*', involves transforming P_N into a sequential composition of two programs, Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$, such that $\mathsf{Peel}(\mathsf{P}_N)$ is provably simpler than P_N and is easily computed for a large class of interesting programs. The program Q_{N-1}, while being syntactically similar to P_{N-1}, may not be semantically equivalent to P_{N-1}. Therefore, the inductive hypothesis $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$, used in full-program induction, may not hold if P_{N-1} is replaced by Q_{N-1}. To rectify this situation, we compute simple relational invariants between corresponding variables at corresponding control points of P_{N-1} and Q_{N-1}. These relational invariants are then used to infer an appropriate post-condition $\psi'(N-1)$ such that $\{\varphi(N-1)\}\ \mathsf{Q}_{N-1}\ \{\psi'(N-1)\}$ holds whenever $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$ holds. This allows us to reduce proving $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ to proving an appropriate Hoare triple over $\mathsf{Peel}(\mathsf{P}_N)$. Significantly, relational full-program induction can be used to prove properties of programs beyond the reach of full-program induction, such as those with nested loops and branch conditions dependent on N. We describe a prototype tool called DIFFY that implements these ideas. We present results comparing the performance of DIFFY with that of various state-of-the-art tools. DIFFY significantly outperforms the winners of SV-COMP 2019, 2020 and 2021 in the ReachSafety-Arrays sub-category.

List of Publications

Journal Papers

1. Chakraborty, S., Gupta, A., Unadkat, D., 2022. Full-Program Induction: Verifying Array Programs sans Loop Invariants. International Journal on Software Tools for Technology Transfer, 24(5), (pp. 843–888).

Conference Papers

1. Chakraborty, S., Gupta, A., Unadkat, D., 2021, July. Diffy: Inductive Reasoning of Array Programs Using Difference Invariants. In International Conference on Computer Aided Verification (pp. 911–935).

2. Chakraborty, S., Gupta, A., Unadkat, D., 2020, April. Verifying Array Manipulating Programs with Full-Program Induction. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 22–39).

3. Chakraborty, S., Gupta, A. and Unadkat, D., 2017, August. Verifying Array Manipulating Programs by Tiling. In International Static Analysis Symposium (pp. 428–449).

List of Tools and Artifacts

Prototype Tools Designed and Implemented

During the course of this study, we have built verification tools and packaged them as software artifacts. These are well documented, consistent, complete, exercisable, fully-automatic and have been placed on publicly accessible archival repositories. We list them below:

1. DIFFY [CGU21a] implements the *relational full-program induction* technique described in our CAV 2021 paper [CGU21b]. It is endorsed by the Artifact Evaluation Committee of CAV 2021 and received the highest badge - functional, reuseable and available - during evaluation. The artifact zip file contains the source code and instructions to build and run DIFFY. It contains the benchmarks and scripts to reproduce the results reported in the paper. It also contains a docker script that can build a container based on Ubuntu 18.04 LTS operating system, install the required packages and run all the experiments described in the paper. The artifact is complete with respect to the claims made in [CGU21b]. The artifact is publicly available on *figshare* [CGU21a] and *github* [Unaa]. The size of the artifact is 31.67 MB.

2. VAJRA [CGU20b] implements the *full-program induction* technique described in our TACAS 2020 paper [CGU20a] and the STTT journal paper [CGU22]. It is endorsed by the Artifact Evaluation Committee of TACAS 2020. The artifact zip file contains the source code, benchmarks and instructions to build and run the tool on Ubuntu 18.04 LTS operating system. Alternately, the VM image from the TACAS 2020 artifact evaluation page `https://tacas.info/artifacts-20.php` can be used to run the tool. The artifact is complete with respect to the claims made in [CGU20a].

The artifact is publicly available on *figshare* [CGU20b] as well as *github* [Unac]. The size of the artifact is 20.88 MB.

3. TILER [Unab] implements the verification by tiling technique described in our SAS 2017 paper [CGU17]. The artifact is available as a VirtualBox image (of size 6.06 GB) based on Ubuntu 16.04 LTS operating system. The artifact contains the source code and instructions to build and run TILER, as well as the benchmarks to evaluate the tool and reproduce the results reported in the paper. The artifact can be downloaded from [Unab].

Contents

Abstract i

List of Publications iii

List of Tools and Artifacts v

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Problem Statement . . . 3
- 1.3 Contributions . . . 5
  - 1.3.1 Compositional Inductive Verification by Tiling . . . 7
  - 1.3.2 Verification by Full-Program Induction . . . 10
  - 1.3.3 Generalized Difference Computation . . . 11
  - 1.3.4 Relational Full-Program Induction . . . 13
- 1.4 Thesis Organization . . . 16

2 Related Work 17
- 2.1 Abstraction-based Techniques . . . 17
- 2.2 Invariant Generation . . . 19
- 2.3 Logic-based Reasoning . . . 23
- 2.4 Inductive Reasoning . . . 24
- 2.5 Differentiation and Integration . . . 25
- 2.6 Analysis Techniques in Compilers . . . 27

3 Preliminaries 31
- 3.1 Program Grammar . . . 31
- 3.2 Control Flow Graph . . . 34
- 3.3 Modeling and Proving Verification Conditions . . . 36
- 3.4 Hoare Triples . . . 38

4 Compositional Inductive Verification by Tiling 39
- 4.1 Introduction . . . 39
  - 4.1.1 Motivating Example . . . 40
  - 4.1.2 Overview of Verification by Tiling . . . 42
  - 4.1.3 Handling Nested and Sequence of Loops . . . 46

4.1.4 Prototyping and Evaluation . . . 47
4.1.5 Relation to Optimization Techniques in Compilers . . . 47
4.2 A Theory of Tiles . . . 49
4.2.1 Tiling in a Simple Setting . . . 49
4.2.2 Tiling in a General Setting . . . 54
4.3 Algorithms for Verification by Tiling . . . 55
4.4 Experimental Evaluation . . . 59
4.4.1 Implementation . . . 59
4.4.2 Benchmarks . . . 60
4.4.3 Experimental Setup . . . 62
4.4.4 Results . . . 62
4.4.5 Analysis . . . 65
4.4.6 Limitations . . . 66
4.5 Comparison with Related Techniques . . . 67
4.6 Conclusion . . . 69

5 Verification by Full-Program Induction **71**
5.1 Introduction . . . 71
5.1.1 Motivating Example . . . 72
5.1.2 Beyond Loop-Invariant based Proofs . . . 75
5.2 Overview of Full-Program Induction . . . 77
5.3 Input Programs, Pre-processing and Analysis . . . 83
5.3.1 Renaming Variables and Arrays . . . 86
5.3.2 Peeling the Loops . . . 94
5.3.3 Tracking Data Dependencies . . . 97
5.3.4 Identifying "Affected" Variables . . . 107
5.4 Computing the Difference Program $\partial \mathsf{P}_N$ . . . 115
5.4.1 Canonicalizing the Program . . . 116
5.4.2 Reordering the Peels . . . 121
5.4.3 De-canonicalizing the Reordered Program . . . 125
5.4.4 Peels of Loops as the Difference Program . . . 127
5.5 Computing the Difference Pre-condition $\partial\varphi(\mathbf{N})$ . . . 130
5.6 Verification using Full-Program Induction . . . 132
5.6.1 Generating the Formula $\mathsf{Pre}(\mathbf{N-1})$ . . . 132
5.6.2 The Full-program Induction Algorithm . . . 133
5.7 Conclusion . . . 136

6 Generalizing Difference Computation **137**
6.1 Introduction . . . 137
6.1.1 Motivating Examples . . . 138
6.1.2 Instantiation of Full-Program Induction in VAJRA . . . 143
6.2 Computing the Difference Program . . . 145
6.2.1 Generating $\partial \mathsf{P}_\mathbf{N}$ for Programs with Affected Variables . . . 148
6.2.2 Simplifying the Difference Program . . . 159
6.3 Extensions to Full-Program Induction . . . 162
6.3.1 The Recursive Full-program Induction Algorithm . . . 162
6.3.2 Full-Program Induction with Formula Decomposition . . . 167
6.4 Checking Progress . . . 169

6.5 Generalization to Different Problem Settings 173
6.5.1 Multiple Independent Program Parameters 174
6.5.2 Handling Loops with Increasing/Decreasing Counters 174
6.5.3 Limitations 175
6.5.4 Relation to Techniques in Compilers 177
6.6 Experimental Evaluation 180
6.6.1 Implementation 180
6.6.2 Benchmarks 181
6.6.3 Experimental Setup 182
6.6.4 Summary of the Results 183
6.6.5 Analysis on Safe Benchmarks 183
6.6.6 Analysis on Unsafe Benchmarks 185
6.6.7 Performance Comparison 185
6.7 Comparison with Related Techniques 186
6.8 Conclusion 190

7 Relational Full-Program Induction **193**
7.1 Introduction 194
7.1.1 Motivation and Examples 194
7.1.2 Instantiation of the Technique in DIFFY 197
7.2 Overview of Relational Full-Program Induction 198
7.2.1 Relation to Full-Program Induction 203
7.3 Inductive Verification using the Relational Full-Program Induction Technique 206
7.3.1 Generating Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$ 207
7.3.2 Generating $\varphi'(N-1)$ and $\Delta\varphi'(N)$ 222
7.3.3 Inferring Inductive Relational Invariants 223
7.3.4 The Relational Full-Program Induction Algorithm 225
7.4 Experimental Evaluation 229
7.4.1 Implementation 229
7.4.2 Experimental Setup 230
7.4.3 Benchmarks 230
7.4.4 Results and Analysis 232
7.4.5 Limitations 234
7.5 Verifying Existentially Quantified Properties 235
7.6 Comparison with Related Techniques 239
7.7 Conclusion 241

8 Conclusions & Prospects **243**
8.1 Future Prospects 245

List of Figures

1.1 Challenging Verification Problems with Branch Conditions and Nested Loops 4
1.2 An Example to Motivate Verification by Tiling 8
1.3 An Example to Motivate Verification by Full-Program Induction 10
1.4 An Example Highlighting the Need for Generalizing the Difference Computation 12
1.5 An Example Motivating the Need for Relational Full-Program Induction . 14

3.1 Program Grammar 31
3.2 Grammar for Loop-free Programs 34
3.3 A CFG 35

4.1 A Motivating Example - `BatteryController` 41
4.2 Instrumented Program for Verifying that Sliced Property Holds for the Tile 44
4.3 Instrumented Program for Checking Non-Interference Across Tiles 45
4.4 A Program with Interesting Tiling 50
4.5 Program with an Assertion 59
4.6 Transformed Program Input to Daikon 61
4.7 (a) `mclceu.c` and (b) `tcpy.c` 66
4.8 (a) `poly2.c` and (b) `poly.c` 67

5.1 Original Hoare Triple 73
5.2 Base-case Hoare Triple 74
5.3 Inductive Step Hoare Triple 74
5.4 Goal 77
5.5 Hypothesis 78
5.6 Transformations 78
5.7 Decomposition of P_N and Semantic Equivalence 79
5.8 Difference Pre-condition 80
5.9 Inductive Step 80
5.10 Strengthening Pre- and Post-conditions 81
5.11 Running Example 83
5.12 CFG of Fig. 5.11 84
5.13 Sequence of Steps for Computing the Difference Program $\partial\mathsf{P}_N$ 86
5.14 Renaming (a) Unoptimized and (b) Optimized 93
5.15 Program with Loops Peeled 97
5.16 CFG (solid edges) and DDG (dashed edges) of the Peeled Program in Fig. 5.15 105
5.17 A loop in (a) P_N, (b) $\mathsf{P}^{\star}_N$, and (c) P_{N-1} 110
5.18 Sequence of Program Transformations to Decompose P^p_N into P_{N-1}; $\mathsf{Peel}(\mathsf{P}^p_N)$ 116

6.1 Hoare Triple using an Affected Variable 139
6.2 (a) Base-case Hoare Triple and (b) Inductive-step Hoare Triple 140
6.3 (a) Running Example and (b) Its Renamed Version 141
6.4 (a) Induction Hypothesis Hoare Triple on P_{N-1} and (b) Inductive Step Hoare Triple on $\partial\mathsf{P}_N$ after Simplification and Strengthening 143
6.5 Sequence of Steps to Compute the Simplified Difference Program $\partial\mathsf{P}_N$ . . . 145
6.6 Sequence of Program Transformations to Decompose P^p_N into P_{N-1}; $\partial\mathsf{P}_N$. 147
6.7 Example Program (a) P_N and (b) P_{N-1} 156
6.8 (a) Peeled Program and (b) Difference Program 158
6.9 Simplified Difference Program 162
6.10 (a) Base-case (b) Inductive Step with Strengthening of Pre- and Post-conditions 166
6.11 Challenge Example 176
6.12 An Example with a Nested Loop 178
6.13 An Interesting Example 179
6.14 Summary of the Experimental Results 182
6.15 Quantile Plot Showing the Performance of the Tools on *Safe* Benchmarks . 183
6.16 Quantile Plot Showing the Performance of the Tools on *Unsafe* Benchmarks 183
6.17 Template-wise Analysis of the Results for the 'zerosum' Benchmarks 185

7.1 Motivating Example - I 195
7.2 Motivating Example - II 196
7.3 Pictorial Depiction of our Program Transformations 199
7.4 Decomposition of P_N and Semantic Equivalence 200
7.5 Difference Pre-condition 201
7.6 Inductive Step 202
7.7 Strengthening Pre- and Post-conditions 204
7.8 (a) P_N, (b) Program with Outer-most Loops Peeled, (c) P^p_N, and (d) Q_{N-1}; $\mathsf{Peel}(\mathsf{P}_N)$ 209
7.9 A Generic Nested Loop 211
7.10 Peel of the Nested Loop 211
7.11 Peel of L_1(N, L_2(N)) 212
7.12 Peel of L_1(N, L_2(N, L_3(N))) 213
7.13 Cactus Plots (a) All Safe Benchmarks (b) All Unsafe Benchmarks 231
7.14 Cactus Plots (a) Safe C1 Benchmarks (b) Unsafe C1 Benchmarks 231
7.15 Cactus Plots (a) Safe C2 & C3 Benchmarks (b) Unsafe C2 & C3 Benchmarks 233
7.16 Cactus Plots. TO=100s. (a) Safe Benchmarks (b) Unsafe Benchmarks . . . 234
7.17 (a) Hoare Triple with Existentially Quantified Pre- and Post-conditions, (b) Base-case, (c) Hoare Triple on Q_{N-1} and (d) Inductive Step 236
7.18 (a) Hoare Triple with Universally Quantified $\varphi(N)$ and Existentially Quantified $\psi(N)$, (b) Base-case, (c) Hoare Triple on Q_{N-1} and (d) Inductive Step after Strengthening 238

List of Tables

4.1 Results on Benchmarks from Booster & Vaphor Test-suites. 63
4.2 Results on Benchmarks from Code of an Industrial Battery Controller. . . 64

5.1 (a) Program P^p_N, (b) Transformed Program T^p_N, (c) Transformed & Re-ordered Program T^O_N and (d) Program P^o_N / Program $\mathsf{P}^\star_N$; $\mathsf{Peel}(\mathsf{P}^p_N)$ . . . 122

7.1 Summary of the Experimental Results. S is Successful Result. U is Inconclusive Result. TO is Timeout. 230

Chapter 1

Introduction

Software systems touch almost every aspect of human life, making software immensely pervasive. Software that controls modern-day industrial and household appliances is becoming increasingly complex. The code to program such software's functionalities extensively uses loops and conditional statements, among other programming language constructs. Different data structures such as arrays, vectors, maps, and lists are used to store and process data during the execution of modern software. Programs with loops that manipulate arrays are quite common in many such software. The functionalities of these applications are notoriously difficult to program correctly. Testing the software using budgeted resources on all inputs and relevant scenarios is time-consuming and practically impossible. As a result, bugs often go undetected, even when these software systems are deployed in the field. Any malfunction in such software can lead to catastrophic outcomes, including serious financial losses as well as loss of human lives (depending on the application). Ensuring correctness of such software before its deployment is thus of paramount importance.

One of the grand challenges in computer science is to automatically prove the correctness of industry scale software. Despite the undecidability of the problem in its full generality, the ability to prove special classes of properties for special classes of programs has been crucial for the successful deployment of high quality software in applications where the cost of bugs is very high. Indeed, over the last five decades, significant advances have been made in the science and practice of program verification. Yet we are far away from having satisfactorily solved the software verification problem. State-of-the-art verification tools and techniques often generate imprecise results or do not scale to real-life

software systems. This thesis contributes to the existing body of knowledge by introducing three new techniques for program verification that are particularly suited for proving properties of programs with loops that manipulate arrays of parametric size. Specifically, we adapt the principle of mathematical induction in different ways to design precise and scalable verification techniques. We exploit source-level syntactic and semantic features of programs that manipulate arrays to prove user-specified properties in a compositional and property-driven way.

1.1 Motivation

The techniques designed in this thesis are a consequence of our attempts at certifying correctness of programs, initially from an industrial code-base in the automotive domain, and later from a software verification competition as well as the test-suites of relevant state-of-the-art tools. The code extensively uses arrays of parametric size to program various operations, which in case of industrial code are of safety-critical in nature.

In the literature, techniques based on different paradigms have been proposed to tackle the problem of verifying safety properties of array-manipulating programs. These techniques include inference of (quantified) inductive invariants, abstraction-based methods, and logic-based reasoning techniques, among others. However, these techniques face several challenges while attempting to solve the verification problem. Automatically inferring precise invariants is often challenging, predominantly when programs have complex control flows and sequentially composed and nested loops. Many techniques are unable to produce the right level of abstraction to prove the specified property or are unable to refine the proposed abstractions. Relying solely on dedicated back-end solvers is not a magic bullet for the verification problems encountered in practice. Moreover, the existing techniques may not always scale to the level of complexity the problem offers and often exceed the specified time and memory limits for proving correctness. This highlights the need for automated techniques for verifying safety properties of programs that manipulate arrays of parametric size that can scale without compromising on precision.

The reasons mentioned above have spurred interest in devising techniques based on the principle of mathematical induction for verifying correctness of programs [SSS00], especially for proving properties of arrays [SB12]. Induction provides scalability while

retaining precision of the analysis. This makes induction-based techniques quite powerful and efficient. However, their Achilles heel is the automation of the inductive argument. Automating the induction step and expanding the class of array-manipulating programs to which induction-based techniques can be applied forms the primary motivation for the set of techniques proposed in this thesis. In the following section, we meticulously state the verification problem and present a couple of array-manipulating programs that the state-of-the-art techniques are often unable to verify.

1.2 Problem Statement

We consider programs that manipulate arrays and scalar variables within (possibly nested) loops and conditional branch statements. A special symbolic integer parameter N (> 0) parameterizes the programs. We use P_N to denote such programs, where the subscript indicates the special parameter. These programs do not modify the value of the symbolic parameter N but can freely use the parameter in expressions, including loop termination conditions and conditional branches. The size of arrays in P_N is a linear expression involving the parameter N. We consider pre-condition formulas, denoted as $\varphi(N)$, that constrain the range of values of inputs to P_N and post-condition formulas, denoted as $\psi(N)$, that constrain the values computed after executing P_N. $\varphi(N)$ and $\psi(N)$ are from a sub-class [1] of quantified and quantifier-free formulas. We restrict our attention to formulas where the quantification is on array indices and not on the arrays themselves. We aim to design techniques that can automatically verify that $\psi(N)$ indeed holds after executing P_N given that $\varphi(N)$ holds on the input values. We view the verification problem as proving the validity of the Hoare triple [Hoa69] $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ for all values of N (> 0). We defer a detailed formalization of such parametric Hoare triples to Section 3.4. We now present two illustrative parametric Hoare triples in Fig. 1.1.

The program in Fig. 1.1(a) has three sequentially composed loops that update an array `A`. The first loop increments each cell of `A` by the value of the program parameter N. In the body of the second loop, there is a branch condition that evaluates to true only if the program parameter N takes an even-numbered value. Each element of `A` is incremented by 2 along the then branch and by 1 along the else branch. The third loop

[1] Refer Section 3.4 for a detailed description of the class of formulas supported by our techniques.

```
// assume(∀x∈[0,N) A[x] = N)

1.  void branch(int A[], int N)
2.  {

3.    for (int i=0; i<N; i++)
4.      A[i] = A[i] + N;

5.    for (int j=0; j<N; j++)
6.      if(N%2 == 0)
7.        A[j] = A[j] + 2;
8.      else
9.        A[j] = A[j] + 1;

10.   for (int i=0; i<N; i++)
11.     A[i] = A[i] + 2;

12. }

// assert(∀x∈[0,N) A[x]%2 = N%2)
```

(a)

```
// assume(∀x∈[0,N) A[x] = N)

1. void nested(int A[], int N)
2. {

3.  int S = 0;

4.  for(int i=0; i<N; i++)
5.    for(int j=0; j<N; j++)
6.      if(i+j < N)
7.        S = S + A[i+j];

8.  for(int k=0; k<N; k++)
9.    for(int l=0; l<N; l++)
10.     A[l] = A[l] + 1;
11.   A[k] = A[k] + S;

12. }

// assert(∀x∈[0,N) A[x] = N×(N+5)/2)
```

(b)

Figure 1.1: Challenging Verification Problems with Branch Conditions and Nested Loops

increments each element of A by 2. The pre-condition of this program states that each element of A has the same value as the program parameter N and the post-condition asserts that, after the execution of the program, the elements of array A and the program parameter N have the same parity.

The program in Fig. 1.1(b) has a couple of sequentially composed nested loops that update arrays and scalars. The scalar variable S is initialized to 0 before the first nested loop starts iterating. The first nested loop computes a recurrence in variable S. If the sum of indices of the outer and the inner loop is less than N, then the element of array A at the index given by this sum is added to S. The inner loop of the second nested loop increments

each cell of `A` by 1 and the outer loop increments each cell by `S`. The pre-condition of this program states that each element of `A` has the same value as the program parameter N and the post-condition asserts that, after the execution of the program, each element of `A` has the value given by the non-linear expression $N \times (N + 5)/2$.

Verifying the parametric Hoare triples shown in Fig. 1.1 is challenging given the nature of the program and the specified properties. Existing state-of-the-art tools and techniques are often unable to verify the given post-conditions in such Hoare triples. In the subsequent section, we give a glimpse of our techniques for verifying such Hoare triples, highlight the obtained results, and summarize the core contributions made in this thesis.

1.3 Contributions

The main contribution of the thesis are three compositional and property-driven inductive verification techniques. These techniques exploit syntactic and semantic features of programs that manipulate arrays to enable inductive reasoning. Each technique uses different features from program constructs at different granularity levels and is extremely well suited for a specific class of programs and properties. While each devised technique has its own limitations as well, these limitations present challenges that act as a motivation for research and design of the subsequent ameliorated technique.

In the first part of our work, we focus on programs that access and update arrays using complex index expressions within loops but do not accumulate array content in scalar variables. We restrict our attention to verifying a sub-class of universally quantified properties of such programs, which is a challenging task in general. We begin the pursuit of our thesis by exploring syntactic and semantic features of the loop under analysis for automating the inductive argument. We ask the following question:

> *Can we identify ranges of indices an array where modifications by a single generic loop iteration are confined and use this information to inductively prove universally quantified properties?*

We answer the above question in the affirmative by capturing array-access patterns in a loop using predicates that we refer to as *tiles* and using them in our technique that inducts on individual iterations of a loop. Applying the technique to nested and

sequentially composed loops requires inference of (possibly quantified) invariants between such loops and separate inductive reasoning for each loop. Inability to tile arrays in certain programs, necessity of generating invariants and separate induction for each loop may act as bottlenecks for this approach.

Next, we focus on the dual objectives of alleviating the problem of automatically inferring invariants between loops as well as relaxing the class of programs and properties amenable to inductive reasoning. In the second part of our work, the programs access and update arrays and may accumulate array content in scalar variables. Further, the post-conditions are from a sub-class of quantified as well as quantifier-free formulas. In accordance with the objectives stated above, we ask the question:

> *For programs that process arrays of parametric sizes, can we induct on the program parameter while treating the program as a whole, instead of inducting on iterations of individual loops in the program?*

We explore and use syntactic features of such programs to design a novel technique, called *full-program induction*, that inducts on the entire program via a program parameter N (typically the array size). Specifically, we compute the *difference* of programs with different values of parameter N during the inductive step. Interestingly, in several cases, this difference can be derived from the peeled iterations of loops. The technique essentially reduces the reasoning about a class of programs with multiple sequentially composed loops to reasoning about a loop-free program containing only the peeled iterations of loops. Significantly, it does not require generation and use of loop-specific invariants. The peeled iterations of loops as the difference may not suffice to verify a more general class of programs. Hence, we next ask the following question:

> *Is it possible to generalize the difference computation required for inductive reasoning on the entire program after lifting all the restrictions imposed on the dependencies?*

We devise algorithms to automatically compute the dependence of variables and arrays in the program on the program parameter N. We perform non-trivial transformations on the given program and the pre-condition to generate their difference while taking into account such dependencies on N. The inductive step of full-program induction then uses the generalized difference computation algorithms. However, for certain

classes of programs, computing the difference program can at times be quite challenging. This prompts us to ask the following question:

> *Can the technique of full-program induction be adapted to make use of additional invariant information that may be easily obtained from an analysis of the program?*

In the third part of our work, we explore semantic features of array-manipulating programs to design a technique called *relational full-program induction*. In particular, we use specific kind of relations, called difference invariants, that relate the corresponding variables and arrays between a pair of closely related versions of the same program. This makes it possible to simplify the process of generating the difference programs. Specifically, the difference programs now consist of only the peeled iterations of loops. The technique can handle programs with nested loops and branch conditions dependent on the program parameter N. Most notably, the technique is relatively complete for a specific class of programs.

In the following subsections, we give further insights into our techniques aimed at affirmatively answering the questions stated above and achieving the associated objectives. We also highlight the contributions made through each inductive reasoning technique.

1.3.1 Compositional Inductive Verification by Tiling

Programs use complex index expressions to read and update array content in loops. Such programs often occur in safety-critical applications. Verifying universally quantified properties of such programs is important. However, the access patterns may not be easy to capture and reason with, making the verification task more challenging. This motivates the need for a technique that can incorporate array-access patterns in the analysis.

Consider the program shown in Fig. 1.2 that updates an array `A` of parametric size within a loop using different index expressions. We need to verify that the given quantified post-condition holds after the execution of the program.

As our first contribution, we have devised a novel verification technique based on *tiling* the loops in the program. We observed that in such programs a single iteration of a loop typically only ensures that the desired post-condition holds over a small region/slice of the array. To capture this region of the array at which the contribution of a generic

```
// assume(true)
1. for (l=0; l<N; l=l+1)
2.   if ((l == 0) || (l == N-1))
3.     A[l] = THRESH;
4.   else
5.     if (A[l] < THRESH) {
6.       A[l+1] = A[l] + 1;
7.       A[l] = A[l-1];
8.     }
// assert(∀i∈[0,N) A[i] ≥ THRESH)
```

Figure 1.2: An Example to Motivate Verification by Tiling

loop iteration occurs, we proposed the concept of *tiles*[2]. For the example in Fig. 1.2, the loop updates the indices of `A` in the range `[l, l+1]`. However, observe that it suffices to focus only on the updates at index `l`, making `[l]` the tile for array `A`.

We infer tiles from the array-access patterns in such loops using simple heuristics. We check that the composition of tiles for each loop iteration covers the entire range of array indices referred in the post-condition of the loop. We perform induction on the iterations of each individual loop in the program by focusing on a specific slice of the array updated in the iteration. During the inductive step, we consider the post-condition on the specific slice of the array as indicated by these tiles. We also verify that the post-condition on the slice of an array pertaining to each prior loop iteration continues to hold at the end of the generic iteration under analysis. Essentially, the technique reduces the problem of checking a quantified post-condition at the end of a loop to an inductive argument that checks only a slice of the post-condition for a single iteration of the loop body.

We extend our technique to programs with nested and sequentially composed loops in a compositional way. We automatically generate and prove loop-specific quantified

[2]Note that unlike the loop tiling optimization [Muc97] performed by compilers that transform array accesses and explicitly partition the computation within a loop, possibly by transforming a non-nested loop to a nested one, aimed at optimizing memory/cache performance, we are only concerned with the identification of a region in an array that pertains to a generic loop iteration and use it to simplify the verification problem. The memory/cache size and its performance do not have any influence on our technique.

invariants on arrays modified in each loop of the program. We use a guessing mechanism to infer such invariants. We prove invariants at the end of each loop using our technique prior to using them as pre-condition to the next sequentially composed loop.

We have implemented the compositional inductive verification technique based on tiling arrays in a prototype tool called TILER. We built the program transformations in our tool using the CLANG front-end bundled with the LLVM compiler framework. The tool uses Z3 SMT solver for checking the satisfiability of first-order logic formulae and CBMC for bounded model checking of the transformed programs. We present an experimental evaluation on a large suite of benchmarks from industrial code-base and academic test-suites. From the experimental results, we observed that TILER outperforms the state-of-the-art verification tools for array programs such as BOOSTER, VAPHOR, and the abstraction-based model checker SMACK+CORRAL.

The central technical contributions made through this technique are summarized as follows.

- A novel concept of *tiles* that formalizes the regions of an array capturing the contribution of a single loop iteration.
- A sound technique for verifying a sub-class of universally quantified post-conditions of array-manipulating programs using the inferred tiles.
- A prototype tool, TILER, that implements compositional verification by tiling and an experimental evaluation on an exciting suite of benchmarks from industry and academia demonstrating that TILER outperforms state-of-the-art tools that verify array-manipulating programs.

Our concept of *tiles* and the method for extracting and using the tile predicates to capture regions of arrays are different from the loop tiling compiler optimization [Muc97], and more importantly their goals are not aligned. The aim of our technique is to improve verification efficiency, while the latter is aimed at improving the runtime performance of the program. Our technique offers several advantages over the state-of-the-art techniques such as scalability, compositionality, automatability, efficiency, precision and it is property-driven. The identification of tiles and inference of loop-specific invariants for compositional reasoning may be quite challenging at times depending on the computation

in the program. We overcome these challenges using the technique presented in the next subsection.

1.3.2 Verification by Full-Program Induction

The technique based on the use of tiles of array-access patterns to reason about individual loops in the program is quite interesting. However, it faces several challenges while proving properties of array-manipulating programs.

```
// assume(∀i∈[0,N) A[i] = 0)
1. S=2;
2. for(t1=0; t1<N; t1++)
3.   A[t1] = A[t1] + S;
4. for(t2=0; t2<N; t2++)
5.   S = S + A[t2];
// assert(S = 2×(N + 1))
```

Figure 1.3: An Example to Motivate Verification by Full-Program Induction

Consider the Hoare triple in Fig. 1.3. The program in the Hoare triple has two sequentially composed loops. The first loop updates an array `A` and the second loop accumulates the content of `A` into the scalar variable `S`. The technique of verifying a program by tiling arrays faces the following challenges in proving the given post-condition. First, it is not always possible to capture the contribution of a generic iteration towards the given post-condition. For example, the second loop of Fig. 1.3 accumulates array content into a scalar variable, making it unamenable to tiling. Second, the technique is restricted to a sub-class of universally quantified post-conditions and it does not support verifying quantifier-free post-conditions on scalar variables, as shown in Fig. 1.3. Third, the inductive reasoning must be applied to each loop separately. Last and most important, the technique needs to generate invariants on arrays between loops in the program.

As our second contribution, we have devised a novel verification technique called *full-program induction* (FPI), that addresses the above mentioned challenges. The technique proves a sub-class of quantified as well as quantifier-free post-conditions of programs that manipulate arrays of parametric size N. Instead of inducting on individual loops,

the technique inducts over the entire program (possibly containing multiple sequentially composed non-nested loops) directly via the program parameter N. In the base case, the technique proves the parametric Hoare triple for a fixed constant value of $N(> 0)$, say $N = 1$. As the induction hypothesis, FPI assumes that the parametric Hoare triple $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$ holds. To enable the inductive step of the analysis, we have come up with the notions of *difference program* (that relates P_N and P_{N-1}) and *difference pre-condition* (that relates $\varphi(N)$ and $\varphi(N-1)$). When variables and arrays used in the program have dependencies that satisfy specific conditions, the difference program consists of just the peeled iterations of loops. As a consequence, for certain classes of programs, the computed difference program is loop-free, and hence, much simpler to analyze than the given program, resulting in a significant simplification of the proof obligation. For the Hoare triple in Fig. 1.3, just the peels of both the loops suffice as the difference program, which is much easier to reason with. During the inductive step, the technique infers predicates to strengthen the pre- and post-conditions when the proof goal is not immediately provable. Note that FPI does not require the generation or use of loop-specific invariants and the proof goals generated during FPI differ significantly from those generated while inducting over individual loops (as in verification by tiling).

We make the following vital technical contributions through the technique briefly described above.

- A novel *full-program induction* technique to verify a sub-class of quantified and quantifier-free properties of programs that manipulate arrays of parametric size without the need for generation and use of loop-specific invariants even when the program contains multiple sequentially composed loops.
- Notions of *difference program* and *difference pre-condition* that enable the inductive step of the analysis.
- Practical algorithms for performing full-program induction, their rigorous correctness proofs and detailed demonstration on examples.

1.3.3 Generalized Difference Computation

In general, the variables and arrays computed in a program may have dependencies that do not satisfy the conditions required to use a simple difference program that consists

of only the peeled iterations of loops. We overcome this limitation by identifying such dependencies and generalizing the difference computation.

```
// assume(∀i∈[0,N) A[i] = 1)
1. S=0;
2. for(t1=0; t1<N; t1++)
3.   S = S + A[t1];
4. for(t2=0; t2<N; t2++)
5.   A[t2] = A[t2] + S;
6. for(t3=0; t3<N; t3++)
7.   S = S + A[t3];
// assert(S = N×(N+2))
```

Figure 1.4: An Example Highlighting the Need for Generalizing the Difference Computation

Consider the Hoare triple in Fig. 1.4. The program in the Hoare triple has three sequentially composed loops. The first and third loops accumulate the content of array `A` into the scalar variable `S`, while the second loop updates `A` using the value in `S` computed at the end of the first loop. Notice that when the last iteration of each loop is peeled, the update to `A` in the second loop would have a dependency on the computation in the peel of the first loop. As a result, just peeled iterations of loops do not suffice as the difference program.

We generalize the computation of difference programs to address this challenge. We identify variables and arrays that have a dependence on values computed in peeled iterations or on the value of N. We automatically generate code to rectify the values of such variables and arrays during the inductive step of *full-program induction*. This is a non-trivial program transformation, enabling the technique to compute the difference program for a class of programs where the dependence on values in peels and N are not restricted. We also generalize the computation of the difference pre-conditions. Further, we recursively apply the generalized program transformations to reduce the complexity of the assertion checking problem. We devise a metric to indicate the need of applying inductive reasoning at each recursive iteration. This enables full-program induction to prove post-conditions for a larger class of programs and properties. With these generalizations, FPI successfully proves the Hoare triple in Fig. 1.4. The technique retains the

feature that it does not generate or use loop-specific invariants.

To assess the efficacy of the full-program induction technique, we have developed a prototype tool called Vajra. We demonstrate the performance of Vajra vis-a-vis several state-of-the-art tools on a large set of array-manipulating benchmarks from the international software verification competition (SV-COMP) and on several programs inspired by algebraic functions that perform polynomial computations.

We summarize the principal technical contributions made through the technique described above as follows.

- A generalized difference program computation algorithm and its integration with the *full-program induction* technique.
- Generalizations of the full-program induction technique to programs with multiple parameters and loops with increasing and/or decreasing loop counters.
- A metric to measure the progress of full-program induction and an algorithm to compute the same based on the characteristics of the difference program.
- A prototype tool, Vajra, that implements full-program induction in its generality and extensive experimental evaluation on a large suite of benchmarks that manipulate arrays, demonstrating that Vajra outperforms several state-of-the-art tools.

1.3.4 Relational Full-Program Induction

The difference program generated during full-program induction may not be simple enough to reduce the verification complexity. Further, computing the difference program can at times be quite challenging for certain classes of programs. We take these limitations as the primary motivation for designing the next inductive verification technique.

Consider the Hoare triple in Fig. 1.5 with two sequentially composed loops. The first loop in the program accumulates the content of `A` into the scalar variable `S` and the second loop updates an array `B` using the value of `S`. Both, variable `S` and array `B` have either a dependence on N or on a value computed in a peel. Hence, additional code needs to be generated to rectify their values. The difference program must consist of two loops, one loop each for rectifying the values of `S` and `B`. Automatically generating such difference programs with loops is challenging. Further, the effort required for verifying the given

```
// assume(∀i∈[0,N) A[i] = N)
1. S=0;
2. for(t1=0; t1<N; t1++)
3.   S = S + A[t1]*A[t1];
4. for(t2=0; t2<N; t2++)
5.   B[t2] = S + t2;
// assert(∀j∈[0,N) B[j] = N^3 + j)
```

Figure 1.5: An Example Motivating the Need for Relational Full-Program Induction

program and the difference program will be the same, since both the programs will have two loops.

We ameliorate the full-program induction technique in a way that we can use the program with just the peeled iterations of loops, as the difference program even for programs with nested loops and branch conditions dependent on the program parameter N. As our third contribution, we have devised a novel verification technique called *relational full-program induction*. The technique constructs two slightly different versions of the same program namely (i) P_{N-1} by substituting N with $N-1$ in P_N and (ii) Q_{N-1} by peeling each loop in the program and propagating each peel across subsequent code blocks. We infer relations between the corresponding variables at key control points of the joint control-flow graph of programs P_{N-1} and Q_{N-1}. The inferred relational invariants are typically much simpler than the inductive invariants required for proving the post-condition directly and are easy to synthesize. We use these relations, along with the post-condition $\psi(N-1)$, to infer a formula $\psi'(N-1)$ that holds as the post-condition of Q_{N-1}. The inferred post-condition is then used to formulate the inductive step of the analysis. We restrict the invariants to use only the differences between the values of corresponding variables/arrays in P_{N-1} and Q_{N-1}. We call such relations as *difference invariants* and study their effectiveness. We use Dijkstra's weakest pre-condition computation to infer predicates for simultaneously strengthening the pre- and post-conditions of the program. Relational full-program induction successfully proves the Hoare triple in Fig. 1.5.

We have implemented a prototype tool called DIFFY to demonstrate the efficacy of relational full-program induction. We have built DIFFY on top of the LLVM compiler framework and the Z3 SMT solver. We compare DIFFY vis-a-vis state-of-the-art tools

for verification of C programs that manipulate arrays on a large set of benchmarks. Our synergistic combination of inductive reasoning and finding simple difference invariants helps prove properties of programs that cannot be proved even by the winner of the Arrays sub-category from SV-COMP 2021.

The primary technical contributions made through this technique are summarized as follows.

- A novel technique called *relational full-program induction*, to prove a sub-class of quantified and quantifier-free properties of array-manipulating programs with nested loops and branch conditions dependent on N.
- The concept of *difference invariants* that relates the two versions of variables/arrays using only differences between values of variables/arrays.
- Algorithms to perform relational full-program induction, including novel program transformations and inference of difference invariants.
- A prototype tool Diffy that implements the relational full-program induction technique and experiments demonstrating that Diffy significantly outperforms the winners of SV-COMP 2019, 2020 and 2021 in the ReachSafety-Arrays sub-category.

While the proposed techniques have different strengths and limitations, they essentially boil down to the same core idea, that applies inductive reasoning to verify the given property, when the given program consists of only a single loop. Our techniques are orthogonal to different verification techniques proposed in the literature. Most such state-of-the-art techniques for checking correctness are aimed at proving special classes of programs and properties. Existing tools, therefore, use a combination of verification techniques that work well for certain classes of programs and assertions, and yield conservative results otherwise. Hence, rather than stand-alone use, we envisage the inductive reasoning techniques proposed in the thesis to be used as part of a portfolio of techniques in a modern software verification tool. We have designed and implemented our techniques in a way that they can be integrated seamlessly into other verification portfolios/frameworks. The extensive experimental evaluations demonstrate the efficacy of our tools and techniques vis-a-vis state-of-the-art tools used to verify array programs. We

show later that industrial verification frameworks use our tools and techniques to prove program correctness in practice and in verification competitions.

1.4 Thesis Organization

The rest of this thesis is organized as follows.

In Chapter 2, we present a high-level overview of the prior state-of-the-art techniques proposed in the literature that are related to our work, along with the advantages and drawbacks that these technique have to offer. Readers familiar with the prior literature in the area can skip Chapter 2.

Chapter 3 discusses the preliminaries, making it a prerequisite for the subsequent chapters in the thesis. The individual chapters refer to different parts of Chapter 3 where the necessary background and notational details are presented.

We expand the four subsections in Section 1.3 into Chapters 4, 5, 6 and 7. Each chapter presents in detail the problem under consideration, motivates the importance of the problem, describes the proposed solution, highlights the salient features of the technique, describes an implementation of the technique in a tool and gives a detailed experimental evaluation. We cite at the beginning of each chapter, the peer-reviewed conference and/or journal publication associated with the technique. Chapter 5, is a prerequisite for Chapter 6. Chapters 4, 5, and 7, however, have minimal dependence on one another and can be read linearly or in any order.

Chapter 8 concludes the thesis and presents several prospective directions for further research in designing inductive verification techniques to prove the correctness of parametric programs that manipulate arrays and other data-structures.

Chapter 2

Related Work

This chapter discusses different analysis and verification techniques proposed in the literature to prove the properties of programs that manipulate arrays. We compartmentalize the techniques and briefly describe how each technique approaches the problem.

2.1 Abstraction-based Techniques

Automated abstraction refinement techniques are amongst the most popular techniques for verifying programs that manipulate arrays.

Mann et al. [MIG+21] propose a counterexample-guided abstraction refinement scheme using auxiliary variables to verify properties of infinite-state systems with arrays. The technique uses two types of auxiliary variables. History variables preserve past values; prophecy variables refer to future values and are dual to history variables. During refinement, the method instantiates the violations of array axioms in the returned counterexample of size k. They use auxiliary variables to lift these instantiated axioms to the transition system, using the prophecy and history variables acting as universally and existentially quantified variables, respectively. Refinement ensures that the resultant abstraction has no counterexamples of size k. If the method terminates, it produces a proof or a valid counterexample. In general, k may be arbitrarily large, making it impossible to rule out all spurious counterexamples of any length. Thus, the method does not guarantee that it will eventually terminate. Identifying appropriate history and prophecy variables is also a challenge. Their method relies on heuristics to choose auxiliary variables. At times these heuristics are quite sensitive to the encoding syntax.

Bueno et al. [BCS20] propose an abstraction refinement algorithm for integrating array abstraction into incremental inductive model checking. They model arrays using uninterpreted functions. The technique utilizes interpolants for finding violated axioms during refinement. The method uses a lazy array axiom instantiation technique to refine the abstraction. It learns array lemmas only when they are absolutely required to (dis)prove the property. The method thus exercises greater control on the theory reasoning performed in the back-end SMT solver of the underlying model checker by postponing the costly array theory reasoning to the refinement step. A drawback of the method is that it cannot infer universally quantified invariants.

VeriAbs [ACC+20] deploys a sequence of strategies to verify universally quantified and quantifier-free properties in C programs. Each strategy consists of multiple techniques. Specifically for programs with arrays, it employs pruning and shrinking abstractions which identify loops that can be abstracted using a bounded number of iterations. The tool reports a program safe when a bounded model checker verifies the abstracted program. If the model checker reports a violation, then a subsequent abstraction technique in the sequence is employed to check the program's safety. As we show later, the techniques developed in this thesis are a part of the strategy in VeriAbs to verify array programs. As a last resort in the strategy, it tries to verify the original program using a bounded model checker with a fixed timeout value. The loop unwinding count is doubled after each timeout to overcome the unwinding assertion failures. Each abstraction-based technique in the strategy differs in the precision of abstraction and has limited ability to refine the abstraction automatically.

Vaphor [MG16] uses an approach that over-approximates array operations for proving universally quantified properties of programs. The approach tracks a bounded number of array cells by transforming the given program with arrays to an array-free system of Horn clauses. The generated system of Horn clauses may be non-linear, i.e., its unfolding will produce a tree. The method can also abstract programs with maps, sets, and multi-sets. Any Horn clause solver such as Z3 [MB08], Spacer [KGC14] and Eldarica [RHK13] can be plugged-in with their method to verify the generated array-free system of Horn clauses. One of the method's drawbacks is that it cannot automatically infer the number of array cells to track to prove the given property and relies on user input instead. For certain classes of programs, the abstraction may not be precise enough,

especially when the updates to the array happen at non-sequential indices, for example, when reversing the array content. Their method does not have a refinement loop that can automatically adjust the precision of the abstraction by targeting the relevant set of array indices.

Model-Checking Modulo Theories (MCMT) [GR10] extend model checking to array-manipulating programs. The method generates quantifier-free safety proofs in the first-order theory of arrays by eliminating a selected set of universally quantified terms. A procedure that combines SMT-based lazy abstraction with interpolation for arrays [BGR12, ABG+12a] is used for these purposes. The technique is implemented in SAFARI [ABG+12b]. They further improve the procedure by introducing acceleration for array programs [BIK10, JSS14] and is implemented in BOOSTER [AGS14]. For a class of programs, computing simple interpolants is challenging. The technique faces an uphill task while proving such programs.

Techniques in [MA15, JM07, DDA10, GRS05, HP08] propose partitioning the set of array indices to prove quantified properties. Abstractions in [MA15, JM07] partition the range of array indices to infer and prove facts on the partitioned array segments. Analyses proposed in [GRS05, HP08] partition the array into symbolic slices and abstracts each slice with a numeric scalar variable to prove facts about entire arrays. Fluid updates [DDA10] uses bracketing constraints, which are over- and under-approximations of indices, to specify the concrete elements updated in an array without explicit partitioning. While their abstraction is independent of the given property, the assumption that only a single index expression updates the array in each loop severely restricts the technique.

2.2 Invariant Generation

Invariants play a crucial role in formal verification. A significant amount of research work has thus gone into automated invariant generation methodologies. Numerous techniques work to produce quantified invariants for programs, and some of these techniques prove the properties of programs that manipulate arrays.

Abstract interpretation [CC77, Cou03] is a sound framework for program analysis that can compute the properties of programs. The abstract domains supported by the abstract interpreter determine the kind of invariants the method can compute. [CCL11]

presents a way of designing a parametric quantified abstract domain and its use for computing invariants over arrays. The abstract domain utilizes cell contents to split array cells into groups. It semantically divides arrays into consecutive non-overlapping segments and abstracts each segment during the analysis. The technique in [LR15] is especially useful when array cells with similar properties are non-contiguously present in the array. Templates are often employed within the abstract interpretation framework to focus the search. [GMT08] searches for inductive invariants, possibly with alternating quantifiers, by instantiating the parameters of a fixed set of user-supplied templates within the abstract interpretation framework. The method uses the abstract domain of uninterpreted functions that incur an extremely high cost of generating invariants. Extensive efforts are required to design and implement new abstract domains and for the the implementation of abstract transformers for each specialized domain. This makes such techniques less appealing in practice.

Predicate abstraction [GS97, FQ02] is a framework that builds abstractions with varying degrees of precision and complexity, automatically tuned to the program and property. Often heuristics are used to infer predicates in a counterexample guided abstraction refinement framework [CGJ+00]. [LB04] infers quantified invariants over arrays using predicate abstraction. Templates can be employed to focus the search for predicates [SG09]. The property directed reachability (PDR) technique pioneered in IC3 [Bra11, SB11, Bra12] and extended to quantified invariants in UPDR [KBI+17] and QUIC3 [GSV18], takes the post-condition into consideration while incrementally finding inductive invariants. It uses the verification goal to direct the search for predicates in a predicate abstraction-based framework. It employs heuristics to generalize the predicates to inductive invariants. UPDR [KBI+17] focuses on the effectively propositional fragment (EPR) of first-order logic with equality and uninterpreted functions in which the satisfiability of quantified formulas is decidable. QUIC3 [GSV18] extends the IC3 framework to a combination of SMT theories. It performs lazy quantifier instantiations and model based projections on programs specified as Constrained Horn Clauses (CHCs). [SGSV19] presents a property-directed invariant inference algorithm based on predicate abstraction that infers composition-invariant pairs for proving k-safety properties of programs with arrays and can establish that no such invariant exists when the property does not hold.

Many tools have used constraint solving for inferring invariants. The inference prob-

lem is reduced to solving a set of constraints that encode the program in a decidable fragment of first-order logic. Consequently, such techniques face a trade-off between the expressiveness and decidability of the underlying logic fragment. Further, the system of constraints generated from a program with linear arithmetic expressions is often non-linear. Hence, the form of invariants that can be inferred by relying on a constraint solving method is usually pre-defined. InvGen [GR09] infers Boolean combinations of linear arithmetic invariants by solving, possibly non-linear, constraints. It uses a combination of static analysis and dynamic execution to speed up and scale constraint solving. Inference of invariants with non-linear inequalities has been studied in [SSM04]. It relies on Gröbner bases to represent a set of non-linear constraints in the parametric linear form. The method can handle programs with branch conditions without the need to abstract the conditional expressions in the branch with non-deterministic choices. Logical fragments with quantification are considered less often as most of the time they turn out to be undecidable. The technique in [BHMR07] proposes a template-based constraint solving approach to infer invariants over arrays by expressing the program using the combined theory of linear arithmetic and uninterpreted function symbols. However, this technique is not automated for inferring quantified invariants for array-manipulating programs. Another exception that computes quantified post-conditions over arrays with respect to the given pre-condition using constraint solving is the technique in [BHI+09]. The method translates the pre- and post-conditions of programs into the Single Index Logic (SIL) [HIV08] constraints. These constraints along with the program loops, are compiled to counter automata and transducers for checking entailment. The inferred formulae are decompiled to SIL constraints via a counter automata that is over-approximated using flat automata with difference bound constraints. Unfortunately, this technique is not fully-automated, does not compute loop invariants but post-conditions, and does not support nested loops even in theory.

Theorem proving has been used to infer first-order invariants containing alternations of quantifiers for programs that manipulate arrays. [McM08] presents a complete method for generating inductive invariants from universally quantified interpolants obtained by refuting unwindings of loops in programs. The method increases the loop unwind count after each unsuccessful attempt. It restricts the types of clauses used for encoding the problem and uses a paramodulation-based saturation prover for proving first-order logic formulas

as well as for generating interpolants. The method in [KV09] extracts properties of loops in the given program and passes them on to a saturation-based theorem prover to simplify its task. It does not require guidance in the form of user provide templates, predicates, or post-conditions. The first-order theorem prover VAMPIRE [HKV11] implements this technique using the analysis and invariant generation tool ALIGATOR [HHKR10]. These techniques [McM08, KV09] are limited to programs with non-nested loops and linearly accessed array elements. [TW16] applies interpolation techniques in combination with theorem proving to programs. The technique supports invariant inference over expressive domains that can represent varied data structures. Their interpolation procedures are complete for theories of arrays and linked lists. However, the technique in [TW16] is not implemented for arrays and can only handle loops with simple linked list manipulations.

Dynamic analysis has been used for various purposes, including performance analysis, security analysis, testing, and invariant inference. [SGH+13] uses a data-driven approach to detect algebraic invariants in programs. DAIKON [EPG+07] pioneered the method for inference of likely invariants using (mostly linear) templates from actual program runs. The technique in [NKWF12] infers loop invariants with polynomial inequalities and arrays. One can also use dynamic analysis to generate function summaries. Dynamically generated function summaries provide for scalable test generation [YUA+13] and proving properties of programs in a feedback-driven way [YU13, ZYR+14].

A combination of multiple techniques for invariant inference has also been studied in the literature. DYNAMATE [GFM+15] combines mutation, test case generation, dynamic invariant detection, and static verification to prove the properties of array programs. However, it is quite restrictive in the kind of invariants it can infer due to the limitations of the underlying tools and techniques. [MGJ+19] verifies distributed protocols by combining model checking and incremental inference of predicates.

Constrained Horn Clause (CHC) solvers have gained traction recently for inferring universally quantified invariants for programs that manipulate arrays. This includes the PDR-based invariant generator QUIC3 [GSV18] described previously. The FREQHORN [FPMG19] CHC solver that infers universally quantified invariants from program syntax and behaviours within the syntax-guided synthesis (SyGuS) framework using the guess-and-check paradigm. The CHC solver in [BMR13] restricts the quantifier structure of invariants using templates and relies on quantifier instantiation to infer invariants ex-

pressed in the theory of arrays with equality. The method deploys instantiation heuristics such as E-matching and requires only quantifier-free reasoning to infer invariants.

For a detailed technical exposition of the concept of loop invariants, the classical techniques for invariant generation, and the use of invariants in program verification, we refer the interested reader to the survey [FMV14].

2.3 Logic-based Reasoning

[BEG+19, GGK20] propose *trace logic* as a framework to verify relational and safety properties of programs. Trace logic axiomatizes the semantics of each program statement using a first-order logic formula. The method precisely captures the semantics at arbitrary time points. The time points indicate the program locations and loop iterations. To aid the inductive reasoning in the method, they define the notion of *trace lemmas* that represent generic inductive properties over arbitrary quantification over the time points. The lemmas describe the progression of the values of the loop counter as well as the program variables and arrays in relation to the time point. It uses a theorem prover to introduce and prove trace lemmas, specified in trace logic, at arbitrary time points in the program, such that the lemmas are strong enough to establish the given safety property. Thus, the method can be thought of as implicitly capturing arbitrarily quantified inductive invariants. Rapid [GGK20] implements the described method and can prove properties of programs containing arrays of arbitrary length that may require invariants with alternating quantifiers. Interestingly, the method is sound and complete with respect to Hoare logic.

VIAP [RL18] encodes the given C program into a quantified first-order logic formula using the translation scheme proposed in [Lin16]. There is a clean separation between the translation strategy and the use of the translated formula for proving properties. Further, the translation is independent of the loop invariants. The loops in the program are translated to a set of recurrence relations. The technique then tries to simplify the generated first-order axioms (using dedicated solvers and external libraries) and compute the closed-form solutions of recurrences. The tool uses a portfolio of tactics to prove the simplified formula obtained after replacing the recurrences with the computed closed-form solutions. Solving the recurrences for some classes of programs is easy, while it

is extremely difficult for other classes. However, some tactics, such as induction, are useful even when the simplification step is unable to reduce verification complexity. The technique is useful in proving the safety of user-specified assertions; however, it is not useful for general purpose invariant generation.

2.4 Inductive Reasoning

Inductive reasoning is one of the fundamental mathematical concepts to approach program verification, and most first-order theorem provers do not have inbuilt induction ability. A plethora of techniques have been proposed in the literature that use induction [DM97, BC00, ES03, GLD09, Bra11, CJRS13, RK15, UTS17] and its pragmatically more useful version k-induction [SSS00, DMRS03, HT08, DKR10, KT11, DHKR11, BDW15, BJKS15, GIC17, KVGG19, ARG+21, YBH21]. These techniques implicitly/explicitly generate and use loop invariants, especially when aimed at verifying the safety properties of programs. While not all such techniques handle programs with arrays, several efforts have been made to adapt inductive reasoning to verify quantified properties of programs that manipulate arrays.

An induction-based approach to prove the properties of programs with arrays is proposed in [ISIRS20]. The method proves programs correct by induction on a rank, chosen as the size of program states, which is proportional to the length of the arrays in the program. It constructs a safety proof by automatically synthesizing a squeezing function that can map higher-ranked states to lower-ranked states while ensuring that original states are faithfully simulated by their squeezed counterparts. This strategy allows the method to shrink program traces of unbounded length, limiting the reasoning to only minimally-ranked states. A guess-and-check approach combined with heuristics for making educated guesses is employed for computing the squeezing function necessary to prove the post-condition in the given program. The successful synthesis of a squeezing function is equivalent to establishing the inductive step. The user can also supply these functions. They can be quite useful in practice, for example, to prove programs that may not have a first-order representable loop invariant. In general, they are not easy to synthesize, and automatically searching for such functions is non-trivial and exceedingly time-consuming. As a consequence, the technique is not fully-automated in an end-to-end

verifier. Further, the squeezing functions can only consist of commutative and invertible operations, restricting their applicability.

The technique in [SB12] inducts on the loop counter for verifying the given property. The method identifies the loop bodies such that one of the symbolic loop bound on the counter does not syntactically appear in the loop body. These are called *recurrent fragments*. The technique removes loops and quantified assertions in the program by transforming them to their pre- or post-recursive forms. In the pre-recursive representation, the loop is unrolled at the end such that the last iteration follows the loop wherein the upper bound is reduced by one. The last iteration becomes the object of analysis as the hypothesis generates a formula that summarizes the initial iterations. In the post-recursive representation, the loop is unrolled at the beginning such that the first iteration is followed by the loop wherein the lower bound of the loop counter is increased by one. For this form, the post condition is also weakened to an implication in which the antecedent assumes the correctness of the loop starting from the second iteration. This results in a post-condition that can possibly be simplified to one with fewer quantifiers. It then inducts over the bound variable appearing in the property. For the technique to be applicable, at least one of the terms that act as symbolic loop bounds must appear in the quantified property, and the same term must not occur in the loop body. Further, the method imposes severe restrictions on the input programs to move the peel of one loop across the next sequentially composed loop such that the program with the peeled loops composed with the program fragment consisting of only the peeled iterations is semantically equivalent to the input program. These restrictions on the input programs are described as *commutativity of statements*. In practice, such restrictive conditions and data dependencies are not satisfied by a large class of programs, making the technique of [SB12] applicable only to a small part of the program-assertion space.

2.5 Differentiation and Integration

Techniques such as formal differentiation [PK82], differential static analysis [LVH10] and program integration [HPR89] have been studied in the literature for various purposes.

[PK82] presents a program optimization method to compute finite differences of computable program expressions with an aim of improving the execution time of the optimized

program. The core idea is that if an expression was available at the entry of a block in the CFG of the program, then the method attempts to keep the expression available in the block by performing appropriate modifications to the expression depending on changes to its parameters such that the cost of computing the entire expression again is saved and redundant computations are replaced with the available, albeit modified, expression. The costly calculations performed due to repeated evaluation of the expression within other expressions are effectively replaced by inexpensive incremental counterparts. Essentially, the method generalizes the strength reduction optimization. Applications of differencing for incremental computation of expensive expressions [LST98], optimizing the execution time of programs that manipulate arrays [LSLR05], reducing the cost of regression testing [Bin92], checking data-structure invariants [SB07] as well as many others have been studied in the literature. It may be noted that a program optimized for execution time may not always reduce verification complexity and vice-versa.

SymDiff [LHKR12] is a tool, based on differential static analysis [LVH10], for displaying semantic differences between different versions of a program and checking their equivalence. The method unrolls loops, if any, up to a user-specified depth during the generation of SMT-based verification conditions. To achieve modularity in checking, it replaces procedure calls with uninterpreted function symbols or inlines procedure calls depending on the required precision. It requires as input a mapping between procedures, globals, and constants from the two program versions that are checked for equivalence. SymDiff generates a interprocedural counterexample trace in a procedure that is not equivalent to its previous version and highlights a path with semantically differing values. It supports the programming languages that can be translated to the intermediate (verification) language used by the tool. However, the method neither supports checking quantified post-conditions nor does it support loops and arrays of potentially unbounded size. Its scalability is limited by that of the back-end solver.

Different program versions get created during various stages of software development. Under varied scenarios, a new program version needs to be created that has the common base functionalities as well as the enhancements from the prior versions of the program that do not interfere with each other in a well-defined sense. This problem, called program integration, is formalized in [HPR89]. It aims to provide a semantics-based technique to automatically integrate the behavioral changes/enhancements in prior program versions

into an integrated version. The integrated version must satisfy the post-conditions of all prior versions when their pre-conditions are satisfied. Checking if a modification actually leads to a behavioral difference in the program is undecidable in general. Hence, the method approximately determines the changes by comparing a program slice with the corresponding slice from a prior version. It is further customized to semantics preserving transformations in [YHR92]. However, the integrated program needs to be examined by a human and both the techniques are restricted to programs that do not operate on arrays.

Unfortunately, the kinds of differences and integrations computed by the methods discussed above are not always directly well-suited for verifying properties of programs, especially for programs that manipulate arrays.

2.6 Analysis Techniques in Compilers

The polyhedral model of computation is a powerful mathematical framework that can concisely capture program executions. Polyhedral analysis techniques are extremely useful for parallelizing programs. The model has been studied extensively [KMW67, Lam74, Fea91, Pug91b, Len93, QRW00, Bas04, PBCV07, PBCC08, BBK+08, BPCB10, FL11, MR22, SSK22, TKA22, ZBY+22, ZLL+22] for performing various optimizations and transformations. The analysis is widely used for efficiently performing loop optimizations such as tiling, interchange, shifting, splitting, reversal, skewing, distribution, parallelization, and so on. The polyhedral model consists of the following three main parts: (i) *iteration domains*: to capture multiple execution instances of each statement, (ii) *space-time mapping functions*: to capture the order of execution among statements; also called scheduling functions, and (iii) *access functions*: to capture reads and writes to memory cells. Recent advances represent the inferred program properties as unions of general affine relations.

Over the years, various compilers, analyzers and libraries dedicated to polyhedral analysis and transformations have been built. The polyhedral model is used to optimize loops in many production level compilers and analyzers such as R-stream [LMS+04, SLLM06, MBS+22], PolyMage [MVB15], LLVM/Polly [GZA+11], GCC/Graphite [PCB+06, TCE+10], IBM XL [BGDR10], Chunky Loop Generator (CLooG) [Bas13], Chunky Analyzer for Dependences in Loops (CANDL) [BP12], Chunky Loop Alteration Wizardry (CLAY) [Bas12], Chunky Loop Analyzer (CLAN) [BCG+03], Polyhedral Parallel Code

Generation for CUDA (PPCG) [VCJC+13], Parallelization and Locality Optimization Tool (PLuTo) [BHRS08], Composing High-Level Loop Transformations (CHiLL) [CCH08], as well as many others. The libraries used for polyhedral analysis include the Polyhedral Library (Polylib) [CL98], Parametric Integer Linear programming solver (Piblib) [Fea88], Fourier-Motzkin library (PoCC/FM) [Pug91a, Pou10] and Integer Set Library (*isl*) [Ver10] among others.

The polyhedral analysis is extremely useful in aggressively optimizing computation, but has several limitations pertaining to its scope and the class of programs that it is applied to. The programs amenable to analysis and transformation using the polyhedral model must have the following characteristics: (i) each loop must have a unique loop counter, (ii) the instructions accessing memory are limited to scalar variables or elements of multi-dimensional arrays, (iii) expressions used in branch conditions, loop bounds and array indices are affine, and (iv) control flow must not be dependent on the data computed in the program. The expressions in branch conditions, loop bounds and array indices can have constants, invariant loop parameters and counters of enclosing loops. The presence of even a single non-affine entity in the program can limit some polyhedral optimizations in compilers. Further, the polyhedral analysis usually has a high worst-case complexity. The data dependence analysis performed during such optimizations may not scale to large programs. These analysis may not take into account the requirements regarding the granularity of the information required to perform optimizations or establish facts. Notably, the implementation of the analysis and transformations based on the polyhedral model are mostly seen in certain parallelizing and optimizing compilers.

Another analysis routinely performed by general-purpose compilers is called scalar evolution [Eng01]. This analysis can compute closed form expressions for programs that update variables of integer type. It tries to summarize statements that accumulate values over several loop iterations. The analysis represents the value of each variable as a chain of recurrence [BWZ94]. These recurrences are represented as a tuple consisting of the variable, its initial value, the arithmetic operator used to update its value and the value by which the variable is updated repeatedly (i.e. the step count). These chains of recurrences can be much more expressive than affine expressions. The chains of recurrences are parametric in the variables whose values are unknown at the compile time and the loop counter variables at which the recurrence should be evaluated. At a high-level, such an

analysis may also be used to infer quantified closed form expressions over arrays of parametric size. The scalar evolution analysis is also useful for identifying induction variables, such as loop counters. The LLVM compiler framework provides an implementation of this analysis as the SCEV pass. The technique implemented in the verifier VIAP [RL18] identifies recurrences in array-manipulating programs as a part of its methodology. The tool computes closed forms of the identified recurrences and uses them to simplify the verification task.

Chapter 3

Preliminaries

In this chapter, we present the notations and technical concepts used in the subsequent chapters of the thesis.

3.1 Program Grammar

We analyze programs generated by the grammar shown in Fig. 3.1. A program P_N that manipulates arrays of parametric size is a tuple $(\mathcal{V}, \mathcal{L}, \mathcal{A}, \mathsf{PB}, N)$, where $\mathcal{V}$ is a set of scalar variables, $\mathcal{L} \subseteq \mathcal{V}$ is a set of scalar loop counter variables, $\mathcal{A}$ is a set of array variables, PB is the program body, and N is a special symbol denoting a positive integer parameter.

$$
\begin{array}{rcl}
\mathsf{PB} & ::= & \mathsf{St} \\
\mathsf{St} & ::= & \mathsf{AssignSt} \ \mid \ \mathsf{St;St} \ \mid \ \textbf{if}(\mathsf{BoolE}) \ \textbf{then} \ \mathsf{St} \ \textbf{else} \ \mathsf{St} \ \mid \\
 & & \textbf{for} \ (\ell = 0; \ell < \mathsf{UB}; \ell = \ell{+}1) \ \ \{ \ \mathsf{St} \ \} \\
\mathsf{AssignSt} & ::= & v = \mathsf{E} \ \mid \ A[\mathsf{IndE}] = \mathsf{E} \\
\mathsf{E} & ::= & \mathsf{E} \ \ \mathsf{op} \ \ \mathsf{E} \ \mid \ A[\mathsf{IndE}] \ \mid \ v \ \mid \ \ell \ \mid \ \mathsf{c} \ \mid \ N \\
\mathsf{IndE} & ::= & \mathsf{IndE} \ \ \mathsf{op} \ \ \mathsf{IndE} \ \mid \ v \ \mid \ \ell \ \mid \ \mathsf{c} \ \mid \ N \\
\mathsf{UB} & ::= & \mathsf{UB} \ \mathsf{op} \ \mathsf{UB} \ \mid \ \ell \ \mid \ \mathsf{c} \ \mid \ N \\
\mathsf{op} & ::= & + \ \mid \ \text{-} \ \mid \ \times \ \mid \ \div \\
\mathsf{relop} & ::= & == \ \mid \ < \ \mid \ \leq \ \mid \ > \ \mid \ \geq \\
\mathsf{BoolE} & ::= & \mathsf{E} \ \mathsf{relop} \ \mathsf{E} \ \mid \ \mathsf{BoolE} \ \mathrm{AND} \ \mathsf{BoolE} \ \mid \ \mathrm{NOT} \ \mathsf{BoolE} \ \mid \ \mathsf{BoolE} \ \mathrm{OR} \ \mathsf{BoolE}
\end{array}
$$

Figure 3.1: Program Grammar

Here, we assume that $A \in \mathcal{A}$, $v \in \mathcal{V} \setminus \mathcal{L}$, $\ell \in \mathcal{L}$ and $\mathsf{c} \in \mathbb{Z}$. We also assume that "op" (resp. "relop") is one of a set of arithmetic (resp. relational) operators. For clarity of exposition, we abuse notation and use $\mathcal{V}$ and $\mathcal{A}$ to also denote a sequence of scalar and array variables, when there is no confusion. We wish to highlight the following features of programs generated by this grammar:

- Programs can have sequences of (possibly nested) loops, with non-looping program fragments between loops. Furthermore, the body of a loop and the corresponding *loop head*, i.e. the node where the loop is entered, are easily identifiable.

- Each loop has a unique scalar loop counter variable ℓ. In each loop L, the counter ℓ associated with it is initialized to 0 when the loop is entered, and incremented by 1 after every iteration of the loop.

- Each loop L has a termination condition $\ell <$ UB. We assume that UB is an expression in terms of N, constants c and variables in $\mathcal{L}$ representing loop counters of loops that nest L as shown in the grammar.

- Assignments in the body of L are assumed to not update ℓ. The only assignments to loop counters happen when a loop is entered for the first time and at the end of an iteration of the corresponding loop body. Other assignment statements in the program cannot assign to loop counters. The loop counters can however be freely used in expressions throughout the program.

- The restriction on the usage of loop counter variables simplifies the analysis and presentation, while still allowing a large class of programs to be effectively analyzed. Specifically, whenever the count of iterations of a loop can be expressed in a closed form in terms of constants and variables not updated in the loop, we can mimic its behaviour using our restricted loops. As a generic example, suppose we are told that the loop `for (i=exp1; Cond; i=exp2) { LoopBody; }` iterates `exp3` times, where `exp3` is an arithmetic expression in terms of constants and variables used in `exp1` and `exp2` but not updated in the loop. The behaviour of this loop can be mimicked using the following restricted loop, where `l` is a fresh variable not present in the original program: `for (l=0; l<exp3 ; l=l+1) { if(l=0) { i=exp1; } if(Cond) { LoopBody; i=exp2; } }`.

To see a specific example of this transformation, suppose the program under verification has the loop: `for (i=2*N; i>=0; i=i-2) { A[i]=i; A[i-1]=N; }`, where `N` and `i` are variables that are used but not updated in the loop body. Clearly, this loop iterates `(N+1)` times. Therefore, it can be modeled in our restricted language as follows: `for (l=0; l<N+1; l=l+1) { if(l=0) { i=2*N; } if(i>=0) { A[i]=i; A[i-1]=N; i=i-2; } }`.

- The expressions for indexing arrays are generated from the non-terminal IndE, and such expressions cannot refer to other array elements. However, this is not really a restriction on the expressive power of programs since every array index expression that depends on other array elements, say $A[e]$, can be replaced by an array index expression that depends on temporary variables, say v, that are pre-assigned to the respective array elements, viz. $A[e]$. For example, `A[B[i]] = C[D[i]];` can be rewritten as `v1 = B[i]; v2 = D[i]; A[v1] = C[v2];`.

- There are no unstructured jumps, like those effected by `goto` or `break` statements in C-like languages. The effect of a `break` statement inside a loop in a C-like language can be modeled by setting a flag, and by conditioning the execution of subsequent statements in the loop body on this flag being not set. The effect of a `break` statement in a conditional branch can also be similarly modeled. Therefore, we can mimic this behaviour of `break` statements in our programs.

  To see a specific example of this transformation, consider the loop: `for (l=0; l<N; l=l+1) { A[l]=l; if(l>N/2) { break; } }`. It can be modeled in our restricted language as: `flag=1; for (l=0; l<N; l=l+1) { if(flag) { A[l]=l; if(l>N/2) { flag=0; t=l; } } } if(flag==0) { l=t; }`, where `flag` and `t` are fresh program variables.

At times a technique may not permit nesting of loops in the input program. The grammar in Fig. 3.2 restricts programs to have only non-nested loops. Specifically, programs generated starting from StLF are loop free. The non-terminal St can generate programs with loops but their bodies are generated from StLF, thereby forbidding nesting of loops. The non-terminals AssignSt and BoolE are defined in the same way as in Fig. 3.1.

$$
\begin{array}{rcl}
\mathsf{PB} & ::= & \mathsf{St} \\
\mathsf{St} & ::= & \mathsf{AssignSt} \ | \ \mathsf{St};\mathsf{St} \ | \ \textbf{if}(\mathsf{BoolE}) \ \textbf{then} \ \mathsf{St} \ \textbf{else} \ \mathsf{St} \ | \\
 & & \textbf{for} \ (\ell = 0; \ \ell < \mathsf{UB}; \ \ell = \ell{+}1) \ \ \{ \ \mathsf{StLF} \ \} \\
\mathsf{StLF} & ::= & \mathsf{AssignSt} \ | \ \mathsf{StLF} \ ; \ \mathsf{StLF} \ | \ \textbf{if}(\mathsf{BoolE}) \ \textbf{then} \ \mathsf{StLF} \ \textbf{else} \ \mathsf{StLF}
\end{array}
$$

Figure 3.2: Grammar for Loop-free Programs

We rely on the notion of the *nesting depth* of a loop during the discussions in the subsequent chapters. We define it as follows:

Definition 3.1 *NestingDepth*(L) *is a natural number n that satisfies the following:*

- *If* L *does not appear within the scope of any loop, then NestingDepth*(L) *= 1.*
- *If* L *appears within the scope of loops* $\mathsf{L}_1, \mathsf{L}_2, \ldots, \mathsf{L}_k$*, then NestingDepth*($\mathsf{L}$) *= 1 + max(NestingDepth*(L_1)*, NestingDepth*(L_2)*, ..., NestingDepth*(L_k)*).*

3.2 Control Flow Graph

Programs generated by the grammar described in Section 3.1 are represented by a *control flow graph* (or CFG) $G = (\mathit{Locs}, \mathit{CE}, \mu)$, where *Locs* denotes the set of nodes of the program, $\mathit{CE} \subseteq \mathit{Locs} \times \mathit{Locs} \times \{\mathbf{tt}, \mathbf{ff}, \mathsf{U}\}$ represents the flow of control, and $\mu : \mathit{Locs} \rightarrow \mathsf{AssignSt} \cup \mathsf{BoolE}$ annotates every node in *Locs* with either an assignment statement (of the form $v = \mathsf{E}$ or $A[\mathsf{IndE}] = \mathsf{E}$) from those represented by $\mathsf{AssignSt}$, or a Boolean expression from those represented by BoolE.

We assume there are two distinguished nodes called Start and End (denoted n_{start} and n_{end} resp.) in *Locs*, that represent the entry and exit points of control flow for the program. An edge $(n_1, n_2, \mathit{label})$ represents flow of control from n_1 to n_2 without any other intervening node. The edge is labeled **tt** or **ff** if $\mu(n_1)$ is a Boolean condition, and it is labeled U otherwise. If $\mu(n_1)$ is a Boolean condition, there are two outgoing edges labeled **tt** and **ff** respectively, from n_1. Control flows from n_1 to n_2 along $(n_1, n_2, \mathit{label})$ only if $\mu(n_1)$ evaluates to *label*. If $\mu(n_1)$ is an assignment statement, there is a single outgoing edge from n_1, and it is labeled U. Henceforth, we use CFG to refer to a control flow graph, and use P_N to refer to both a program and its CFG, when there is no confusion.

A CFG may have cycles due to the presence of loops in the program. A *back-edge* of a loop in a CFG is an edge from a node corresponding to the last statement within the body of a loop to the node representing the corresponding loop head. An *exit-edge* is an edge from the loop head to a node outside the loop body. An *incoming-edge* is an edge to the loop head from a node outside the loop body. The program grammar shown in Fig. 3.1 (as well as in Fig. 3.2) ensures that every loop has exactly one *back-edge*, one *incoming-edge* and one *exit-edge*.

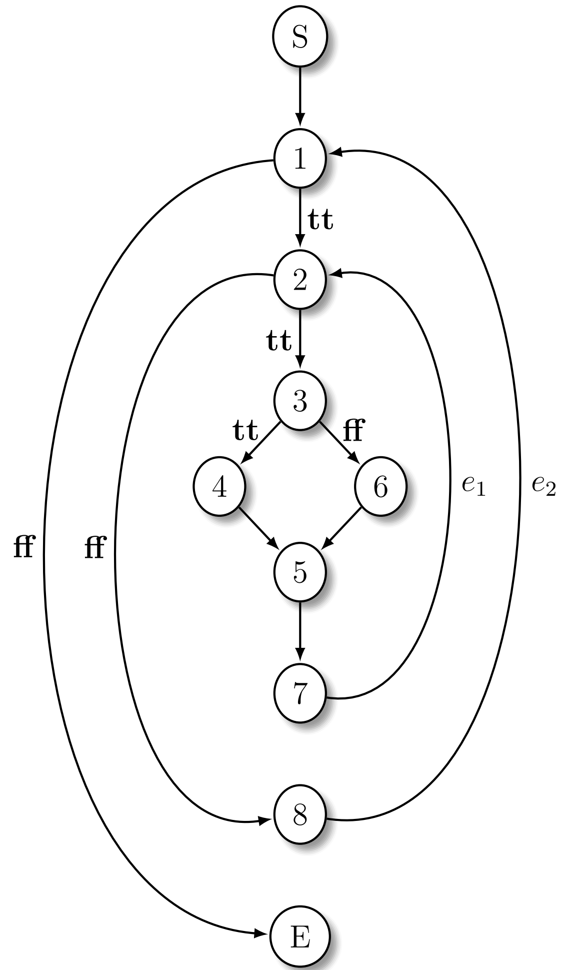


Figure 3.3: A CFG

Removing all back-edges from a CFG renders it acyclic. The target nodes of back-edges, i.e. nodes corresponding to loop heads, are called *cut-points* of the CFG. Every acyclic sub-graph of a CFG that starts from a cut-point or Start and ends at another cut-point or End, and that does not pass through any other cut-points in between and also does not include any back-edge, is called a *segment*. In the literature, a segment may also be referred to as a *region*.

Example 3.1 Consider the CFG shown in Fig. 3.3. For clarity, edges labeled U are shown unlabeled in the figure. The back-edges are $7 \to 2$ (labeled e_1) and $8 \to 1$ (labeled e_2). The incoming-edges in the CFG are $S \to 1$ and $1 \to 2$, and the exit-edges are $2 \to 8$ and $1 \to E$.

The cut-points in the CFG are nodes 1 and 2. And the segments induced by these cut-points are $S \to 1$, $1 \to 2$, $2 \to 3 \to \{4, 6\} \to 5 \to 7$, $2 \to 8$ and $1 \to E$. Note that every segment is an acyclic sub-graph of the CFG with a unique source node and a unique sink node. □

For every node n in the CFG of the program, we use $def(n)$ and $uses(n)$ to refer

to the set of scalar variables and arrays (excluding the loop counter variables) that are defined and used, respectively, in the statement or Boolean expression at n. We include the symbolic parameter N in the set $uses(n)$ if the statement at node n makes use of N. Since the parameter N cannot be re-defined by any program generated according to the grammar in Fig. 3.1 (or in Fig. 3.2), it never appears in $def(n)$ for any node n. If A represents an array in $def(n)$, we use $defIndex(A, n)$ to refer to the index expression of the element of A updated at n. Similarly, if $A \in uses(n)$, we use $useIndex(A, n)$ to refer to the set of index expression(s) of element(s) of A read at n.

A node n in a control flow graph *strictly post-dominates* a node m if all control flow paths from node m pass through n before reaching the exit node and m is not the same as n. The *immediate post-dominator* of node n is a node that strictly post-dominates n but does not strictly post-dominate any other node that strictly post-dominates n.

3.3 Modeling and Proving Verification Conditions

In this section, we present the basics of first-order logic and satisfiability modulo theories (SMT) that are necessary for underling parts of this work. The verification conditions generated by our technique are compiled into logical formulae in first-order logic. First-order logic extends propositional logic with quantifiers and the non-logical symbols. The first-order logic comprises of a set of *variables*, a set of *logical symbols* (e.g. $\wedge$, $\neg$, $\forall$), a set of *non-logical symbols* (consisting of functions $\mathcal{F}$, predicates $\mathcal{P}$ and constants $\mathcal{C}$) and the *rules* for constructing well-formed formulae. A *free variable* is one that is not bound by a quantifier. We refer to the set of non-logical symbols as a *signature* Σ.

The variables in logic take values from a specific *domain* and *axioms* specify the *interpretation* of the logical and non-logical symbols. A formula in the logic is said to be *satisfiable* if and only if it is well-formed and there exists an *assignment* of a domain element to each free variable in the formula that makes the formula evaluate to true under the given interpretation of the symbols.

When we define restrictions on the set of non-logical symbols Σ that are allowed in a formula and the interpretation that can be given to these non-logical symbols, then the restricted version of the logic is referred to as a *theory*. When we restrict the set of logical symbols or the grammar, the restricted version of the logic is referred to as a *fragment*.

The following theories and their combination are of interest to us.

Theory of Equality with Uninterpreted Functions (EUF). The signature of the theory $\mathcal{T}_{\mathcal{EUF}}$ contains a special predicate symbol "=", and countably infinite sets of uninterpreted sorts and uninterpreted function symbols. The theory consists of equality axioms, namely reflexivity, transitivity and symmetry/commutativity, that provide the interpretation to the "=" predicate symbol. The theory also contains the congruence axiom that imposes functional consistency on the uninterpreted function symbols. $\mathcal{T}_{\mathcal{EUF}}$ is decidable [KS16] and the congruence closure algorithm [Sho78] decides conjunctions of literals in the theory in polynomial time.

Theory of Linear Integer Arithmetic (LIA). The signature of the theory $\mathcal{T}_{\mathcal{LIA}}$ contains a integer sort, standard arithmetic function symbols, such as $+$ and $-$ as well as binary comparison operators (e.g. $<$). The theory consists of axioms that define the arithmetic functions and operators. $\mathcal{T}_{\mathcal{LIA}}$ is decidable [KS16].

Theory of Arrays. The signature of the theory $\mathcal{T}_{\mathcal{A}}$ contains the sorts for arrays, indices, and elements, and function symbols to read and write array elements. The theory consists of axioms that define the function symbols that operate over arrays. Several fragments of the $\mathcal{T}_{\mathcal{A}}$ have been proved decidable [Bra07].

Combination Theory. A program usually refers to arrays and variables of different data types to define various functionalities. Combination of theories is important to model such programs using SMT formulae. A combination theory $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$ is a defined using the base theories $\mathcal{T}_1$ and $\mathcal{T}_2$. The signature Σ of a combination theory $\mathcal{T}$ is the union of the signatures Σ_1 and Σ_2 of the base theories, i.e., $\Sigma = \Sigma_1 \cup \Sigma_1$. And the axioms of $\mathcal{T}$ is the union of the axioms of the $\mathcal{T}_1$ and $\mathcal{T}_2$. For convenience, we have shown the combination of only two theories. Multiple (more than two) theories can be combined simultaneously. A generic method of constructing a decision procedure for a combination theory, under several requirements from the base theories, has been proposed by Nellson-Oppen [NO79].

Various decidability results have been studied for combination theories and their fragments. We specifically note that the *array property fragment* of the combination theory $\mathcal{T}_{\mathcal{A}} \cup \mathcal{T}_{\mathcal{LIA}}$, which allows specifying universally quantified properties of arrays with some restrictions on the array indices, is decidable [Bra07].

A *decision problem* is to ascertain the satisfiability (dually validity) of a first-order logic formula with respect to given background theories or a combination thereof. We

assume the availability of a solver for solving the verification conditions generated by our technique. Specifically, we use the SMT solver Z3 [MB08] for checking the satisfiability of these first-order formulae.

3.4 Hoare Triples

A verification problem for an array-manipulating program is a parametric Hoare triple of the form $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$. We restrict our attention to problems where the formulas $\varphi(N)$ and $\psi(N)$ have special forms. Let I be a sequence of array index variables, $\boldsymbol{\alpha}$ be a quantifier-free formula in the theory of arithmetic over integers $\mathcal{T}_{\mathcal{LIA}}$, and $\boldsymbol{\beta}$ and $\boldsymbol{\eta}$ be quantifier-free formulas in the combined theory of arrays and arithmetic over integers $\mathcal{T}_{\mathcal{A}}$ $\cup$ $\mathcal{T}_{\mathcal{LIA}}$. Then, $\varphi(N)$ and $\psi(N)$ can have one of the following forms:

- Universally quantified formulas of the form $\forall I\ \big(\boldsymbol{\alpha}(I, N) \Rightarrow \boldsymbol{\beta}(\mathcal{A}, \mathcal{V}, I, N)\big)$
- Existentially quantified formulas of the form $\exists I\ \big(\boldsymbol{\alpha}(I, N) \wedge \boldsymbol{\beta}(\mathcal{A}, \mathcal{V}, I, N)\big)$
- Quantifier-free formulas of the form $\boldsymbol{\eta}(\mathcal{A}, \mathcal{V}, N)$
- Conjunctions and disjunctions of above formulas

Note that all quantification in the above formulas is on array indices and not on arrays themselves. The formula $\boldsymbol{\alpha}(I, N)$ identifies the relevant indices of the array where the property $\boldsymbol{\beta}(\mathcal{A}, \mathcal{V}, I, N)$ must hold. N is a free variable in $\varphi(\cdot)$ and $\psi(\cdot)$. The above forms for $\varphi(N)$ and $\psi(N)$ allow us to express a large class of useful pre- and post-conditions, including sortedness, which can be expressed as $\forall i\,(0 \leq i < N) \Rightarrow (A[i] \leq A[i+1])$. We use $\boldsymbol{\varphi}$ and $\boldsymbol{\psi}$ instead of $\varphi(N)$ and $\psi(N)$ when the parameterization on N is implicit but not the main focus. In the examples, we specify the pre- and post-conditions using *assume* and *assert* statements of the form **assume**($\varphi(N)$) and **assert**($\psi(N)$) respectively.

Chapter 4

Compositional Inductive Verification by Tiling

In this chapter, we present an induction-based technique to verify programs that access arrays using complex index expressions. We present an implementation of the technique in our publicly accessible tool TILER [Unab] and show its performance vis-a-vis state-of-the-art tools. A part of the work described in this chapter has been published as a conference paper in SAS 2017 [CGU17].

4.1 Introduction

Arrays are widely used in programs written in imperative languages. They are typically used to store large amounts of data in a region of memory that the programmer views as contiguous, and which she can access randomly by specifying an index (or offset). Sequential programs that process data stored in arrays commonly use looping constructs to iterate over the range of array indices of interest and access the corresponding array elements. The ease with which data can be accessed by specifying an index is often exploited by programmers to access or modify array elements at indices that change in complex ways within a loop. While this renders programming easier, it also makes automatic reasoning about such array-manipulating programs significantly harder. Specifically, the array-access patterns within loops can vary widely from program to program, and may not be easy to predict. Furthermore, since the access patterns often span large regions of the array that depend on program parameters, the array indices of interest cannot be bounded

by statically estimated small constants. Hence, reasoning about arrays by treating each array element as a scalar is not a practical option for analyzing such programs. This motivates us to ask if we can automatically infer program-dependent array-access patterns within loops, and use these patterns to simplify automatic verification of programs with loops that manipulate arrays.

A commonly used approach for proving properties of sequential programs with loops is to construct an inductive argument with an appropriate loop invariant. This involves three key steps: (i) showing that the invariant holds before entering the loop for the first time, (ii) establishing that if the invariant holds before entering the loop at any time, then it continues to hold after one more iteration of the loop, and (iii) proving that the invariant implies the desired property when the loop terminates. Steps (i) and (ii) allow us to inductively infer that the invariant holds before every iteration of the loop; the addition of step (iii) suffices to show that the desired property holds after the loop terminates. A significant body of research in automated program verification is concerned with finding invariants that allow the above inductive argument to be applied efficiently for various classes of programs.

For programs with loops that manipulate arrays, the property of interest at the end of a loop is often a universally quantified statement over array elements. Examples of such properties include:

$\forall i\ ((0 \leq i < N) \Rightarrow (A[i] \geq minVal) \wedge (A[i] \leq A[i+1]))$, and

$\forall i\ ((0 \leq i < N) \wedge (i \mod 2 = 0) \Rightarrow (A[i] = i))$

In such cases, a single iteration of the loop typically only ensures that the desired property holds over a small part of the array. Effectively, each loop iteration incrementally contributes to the overall property, and the contributions of successive loop iterations compose to establish the universally quantified property.

4.1.1 Motivating Example

Fig. 4.1 shows a C function snippet adapted from an industrial battery controller. This example came to our attention after a proprietary industry-strength static analysis tool failed to prove the quantified assertion at the end of the function. Note that the loop at line 5 in the function `BatteryController` updates an array `volArray` whose size is given by `COUNT`. In general, `COUNT` can be large, viz. 100000. The universally quantified asser-

```
1.  void BatteryController(int volArray[], int COUNT, int MIN)
2.  {
3.    int i;
4.    if(COUNT % 4 != 0) return;

5.    for(i=1; i<=COUNT/4; i++)
6.    {
7.      if(5 >= MIN) { volArray[i×4-4] = 5; }
8.      else { volArray[i×4-4] = 0; }

9.      if(7 >= MIN) { volArray[i×4-3] = 7; }
10.     else { volArray[i×4-3] = 0; }

11.     if(3 >= MIN) { volArray[i×4-2] = 3; }
12.     else { volArray[i×4-2] = 0; }

13.     if(1 >= MIN) { volArray[i×4-1] = 1; }
14.     else { volArray[i×4-1] = 0; }
15.   }
16.   assert(∀j ∈ [0, COUNT) volArray[j] ≥ MIN ∨ volArray[j] = 0);
17. }
```

Figure 4.1: A Motivating Example - `BatteryController`

tion at the end of the "for" loop requires that every element of `volArray` be either zero or at least as large as `MIN`. It is not hard to convince oneself through informal reasoning that the assertion indeed holds. The difficulty lies in proving it automatically. At the time the work described in this chapter was done, BOOSTER [AGS14] and VAPHOR [MG16] were the only publicly available tools with automated verification capabilities for proving universally quantified assertions in programs that manipulate arrays. However, neither BOOSTER [AGS14] nor VAPHOR [MG16] could prove the above assertion within 15 minutes on a desktop machine. Bounded model checking tools like CBMC [CKL04] and SMACK+CORRAL [LQL12] are able to prove this assertion for arrays with small values

of COUNT. For large arrays, viz. COUNT = 100000, these tools cannot prove the assertion within 15 minutes on a desktop machine. This is not surprising since a bounded model checker must unwind the loop in the function a large number of times if COUNT is large.

4.1.2 Overview of Verification by Tiling

We now give an informal overview of our technique and illustrate how our reasoning using tiles works on our motivating example. We consider programs P_N generated by the grammar in Fig. 3.1. The programs consist of (possibly nested) loops that manipulate arrays of parametric size N. We focus on assertions expressed as universally quantified formulas on arrays, where the quantification is over array indices. The formal syntax of such assertions is explained in Section 3.4. In our experience, assertions of this form suffice to express a large class of interesting properties of array-manipulating programs. We suggest the following approach consisting of three main steps for proving assertions in array-manipulating programs.

Generating Tiles that Cover the Range of Indices Relevant to the Property

We first *identify the region of the array where the contribution of a generic loop iteration is localized.* Informally, we call such a region a *tile* of the array. Note that depending on the program, the set of array indices representing a tile may not include all indices updated in the corresponding loop iteration. In our motivating example, for all array accesses in the ith loop iteration, the value of the index lies between $\mathtt{4*i-4}$ and $\mathtt{4*i}$. Therefore, we choose $[\mathtt{4*i-4, 4*i})$ as the tile corresponding to the ith iteration of the loop.

We check that the tiles *cover the entire range of array indices of interest.* In other words, we check that (i) every tile contains only valid array indices, and (ii) that no array index of interest in the quantified assertion is left unaccounted for in the tiles. In our motivating example, array indices range from 0 to $\mathtt{COUNT}-1$, while the loop (and hence, tile) counter i ranges from 1 to $\mathtt{COUNT}/4$. Since the ith tile comprises of four array indices $4i-4, 4i-3, 4i-2$ and $4i-1$, both the above requirements are met. In the later parts of the chapter, we refer to this vital condition as "*Covers range*".

Identifying the right tile for a given loop in the program that manipulate arrays can be challenging in general. We have developed heuristics to automate tile identification

in a useful class of programs. To understand the generic idea behind our heuristic for generating the tiles of array-access patterns, suppose the program under consideration has a single loop, and suppose the quantified property is asserted at the end of the loop. We introduce a fresh counter variable that is incremented in each loop iteration. We then use existing arithmetic invariant generation techniques, viz. [GR09, EPG+07], to identify a relation between the indices of array elements that are accessed and/or updated in a loop iteration, and the corresponding value of the loop counter. This information is eventually used to define a tile of the array for the loop under consideration. In our motivating example, we introduce a fresh auxiliary variable (say `j`) to denote the index used to update an element of `volArray`. Using tools viz.InvGen [GR09], Daikon [EPG+07], we infer $4 * \texttt{i} - 4 \leq j < 4 * \texttt{i}$, making $[4 * \texttt{i} - 4, 4 * \texttt{i})$ the tile corresponding to the ith iteration of the loop. We discuss more about tiling in Section 4.2.

Proving the Slice of a Post-condition Relevant to a Tile

Next, we *carve out a "slice" of the quantified property that is relevant to the tile identified above.* Informally, we want this slice to represent the contribution of a generic loop iteration to the overall property. The inductive step of our approach checks if a generic iteration of the loop indeed ensures this slice of the property. In the later parts of the chapter, we refer to this inductive check as "*Sliced post-condition holds inductively*".

The sliced property in case of our motivating example says that the elements of `volArray` corresponding to indices within a tile have values that are either 0 or at least `MIN`. To prove that this holds after an iteration of the loop, we first obtain a loop-free program containing a single generic iteration of the loop, and check that the elements of `volArray` corresponding to the ith tile satisfy the sliced property after the execution of the ith loop iteration. The transformed program is shown in Fig. 4.2. Note that this program has a fresh variable j. The assume statements in lines 5–6 say that i is within the expected range and that j is an index in the ith tile. Since this program is loop-free, we can use a bounded model checker like CBMC [CKL04] to prove the assertion in the transformed program.

```
1.  void BatteryController_Inst(int volArray[], int COUNT, int MIN)
2.  {
3.    int i = nondetInt(), j = nondetInt();
4.    if(COUNT % 4 != 0) return;4.

5.    assume(i>=1 && i<=COUNT/4);
6.    assume(4×i-4<=j && j<4×i);

7.    if(5 >= MIN) { volArray[i×4-4] = 5; }
8.    else { volArray[i×4-4] = 0;  }

9.    if(7 >= MIN) { volArray[i×4-3] = 7; }
10.   else { volArray[i×4-3] = 0; }

11.   if(3 >= MIN) { volArray[i×4-2] = 3; }
12.   else { volArray[i×4-2] = 0; }

13.   if(1 >= MIN) { volArray[i×4-1] = 1; }
14.   else { volArray[i×4-1] = 0;   }

15.   assert(volArray[j] ≥ MIN ∨ volArray[j] = 0);
16. }
```

Figure 4.2: Instrumented Program for Verifying that Sliced Property Holds for the Tile

Checking Non-interference of Tiles Across Loop Iterations

Finally, we check that *successive loop iterations do not interfere with each other's contributions*. In other words, once a loop iteration ensures that the slice of the property corresponding to its tile holds, subsequent loop iterations must not nullify this slice of the property. In the later parts of the chapter, we refer to this condition as "*Non-interference across tiles*".

To show non-interference, we assume that the sliced property holds for the i'^{th} tile, where $0 \leq i' < i$, before the ith loop iteration starts. This is done by including three

```
1.  void BatteryController_NonInf(int volArray[], int COUNT, int MIN)
2.  {
3.    int i = nondetInt(), j = nondetInt();
4.    int i′ = nondetInt(), j′ = nondetInt();
5.    if(COUNT % 4 != 0) return;

6.    assume(i>=1 && i<=COUNT/4);
7.    assume(4×i-4<=j && j<4×i);
8.    assume(1 <= i′ < i);
9.    assume(4×i′ - 4 <= j′ < 4×i′);
10.   assume(volArray[j′] >= MIN || volArray[j′] == 0);

11.   if(5 >= MIN) { volArray[i×4-4] = 5; }
12.   else { volArray[i×4-4] = 0;  }

13.   if(7 >= MIN) { volArray[i×4-3] = 7; }
14.   else { volArray[i×4-3] = 0; }

15.   if(3 >= MIN) { volArray[i×4-2] = 3; }
16.   else { volArray[i×4-2] = 0; }

17.   if(1 >= MIN) { volArray[i×4-1] = 1; }
18.   else { volArray[i×4-1] = 0;   }

19.   assert(volArray[j′] ≥ MIN ∨ volArray[j′] = 0);
20. }
```

Figure 4.3: Instrumented Program for Checking Non-Interference Across Tiles

additional assumptions in lines 8–10 in Fig. 4.3 that were absent while checking the sliced property. The assume statement on line 8 says that i' is within the expected range upper bounded by i. The statement on line 9 says that j' is an index in the i'^{th} tile. And the statement on line 10 assumes that the sliced post-condition for the i'^{th} tile holds. We then

assert at the end of the loop body that the sliced property for the i'^{th} tile continues to hold even after the ith iteration. This is done by replacing the assertion in line 19 of Fig. 4.2 by `assert (volArray[j'] >= MIN || volArray[j'] == 0)`, that uses the auxiliary variable j' for referring to the i'^{th} tile. Since the program in Fig. 4.3 is loop-free, as before, this assertion can be easily checked using a bounded model checker like CBMC.

Once all the three checks mentioned above, namely (i) *Covers range*, (ii) *Sliced post-condition holds inductively* and (iii) *Non-interference across tiles* succeed, we can conclude that the quantified assertion holds in the original program after the loop terminates. Note the careful orchestration of inductive reasoning to prove the sliced property, and compositional reasoning to aggregate the slices of the property to give the original quantified assertion. The *sliced property for a tile* allows us to formally capture the contributions of a generic loop iteration and inductively establish slices of the given quantified property. Formalizing the *covers range* and *non-interference* conditions allows us to show that the contributions of different loop iterations compose to yield the overall quantified property at the end of the loop. All three checks succeed for our motivating example. Our tool, that implements the reasoning described above, proves the assertion in our motivating example in less than a second.

4.1.3 Handling Nested and Sequence of Loops

In a more general scenario, the program under verification may have a sequence of loops, and the quantified property may be asserted at the end of the last loop. In such cases, we introduce a fresh counter variable for each loop, and repeat the above process to identify a tile corresponding to each loop. For our technique based on tiles to work, we also need invariants, or *mid-conditions*, between successive loops in the program. Since identifying precise invariants is uncomputable in general, we work with *candidate invariants* reported by existing off-the-shelf annotation/candidate-invariant generators. Specifically, in our implementation, we use the dynamic analysis tool DAIKON [EPG+07] that informs us of *candidate invariants* that are likely (but not proven) to hold between loops. Our algorithm then checks to see if the candidate invariants reported after every loop can indeed be proved using our technique. Only those candidates that can be proved in this way are subsequently used to compose the reasoning across consecutive loops (refer Section 4.4.1 for an example). Finally, our technique can be applied to programs with

nested loops as well. While the basic heuristic for identifying tiles remains the same in this case, the inductive argument needs to be carefully constructed when reasoning about nested loops. We discuss this in detail later in Sections 4.2.2 and 4.3.

4.1.4 Prototyping and Evaluation

We have implemented the above technique in a tool called TILER. Our tool takes as input a C function with one or more loops that manipulate arrays. It also accepts a universally quantified assertion about arrays at the end of the function. TILER automatically generates tiles of array-access patterns for each loop in the C function and tries to prove the assertion, as described above. We have applied TILER to a suite of 60 benchmarks comprised of programs that manipulate arrays in different ways. For most benchmarks where the specified assertion holds, TILER was able to prove the assertion reasonably quickly. In contrast, two state-of-the-art tools for reasoning about arrays faced difficulties and timed out on most of these benchmarks. For benchmarks where the specified assertion does not hold, TILER relies on bounded model checking to determine if an assertion violation can be detected within a few unwindings of the loops. There are of course corner cases where TILER remains inconclusive about the satisfaction of the assertion. Overall, our initial experiments suggest that our compositional reasoning can be very effective for proving assertions in a useful class of array-manipulating programs.

4.1.5 Relation to Optimization Techniques in Compilers

In compilers, the polyhedral analysis (refer Section 2.6) is widely used for parallelizing and optimizing loops in programs. The analysis is powerful enough to extract affine relations between program variables. It computes *iteration domains* and *access functions* that can be used to extract tiles for arrays accessed in loops. Further, the analysis may be useful in inferring affine relations that can be used as mid-conditions between successive loops in the program. However, several programs require non-linear relations which may be out of the reach of such techniques. The scalar evolution analysis may also be able to extract useful relations about programs. Importantly, unlike our technique, these techniques are not property driven and aim to produce the most precise relations even when they may not be necessary to prove the given program.

Another optimization that is commonly performed by optimizing compilers is loop tiling [Muc97]. There are several significant differences between our concept of a *tile* and the loop tiling optimization [Muc97]. The latter technique transforms array accesses within the loop body and explicitly partitions loops, possibly by transforming a non-nested loop into a nested one, with the aim of optimizing the memory/cache performance. The optimization is known to improve data locality. In several cases, verifying programs obtained after this transformation may be harder, especially when a non-nested loop is converted to a nested one to improve caching. Our technique neither transforms array access expressions in programs nor does it partition loops. We identify regions in an array pertaining to a loop iteration and use this information to simplify the verification goals. Consequently, the memory/cache size and its performance do not have any influence on our method of identifying tiles and their subsequent use. Compiler optimizations can in some cases simplify verification, specifically if the data dependencies in the optimized program are easier to reason with using our techniques. We assume that such optimizations are already applied on the program input to our technique.

The contributions made through this technique are summarized as follows.

- We introduce the novel concept of *tiling* an array for reasoning about quantified assertions in programs with loops that manipulate arrays.
- We present a practical algorithm for verifying a class of array-manipulating programs using the inferred tiles and prove correctness of the presented algorithm. We demonstrate the algorithm using a running example.
- We describe a prototype tool Tiler that implements our algorithms. The program transformations in the tool are built using the LLVM compiler framework that provides the CLANG front-end. The tool uses the Z3 SMT solver for discharging verification conditions and CBMC for bounded model checking of the transformed programs.
- We present an experimental evaluation on a large suite of benchmarks that manipulate arrays. From the experimental results, we observe that Tiler performs particularly well on benchmarks where the quantified assertion holds. On such benchmark programs, Tiler outperforms verification tools for array programs such as Booster, Vaphor, and the abstraction-based model checker SMACK+Corral.

The remainder of the chapter gives an elaborate description of our technique, details the implementation of the above ideas to prove quantified assertions in a useful class of array-manipulating programs in our tool TILER and presents experimental evaluation.

4.2 A Theory of Tiles

In this section, we present a theory of tiles for proving a class of universally quantified properties of programs that manipulate arrays within loops.

4.2.1 Tiling in a Simple Setting

Consider a program P_N as defined in Section 3.1 that accesses elements of an array A in a loop L. Suppose P_N has a single non-nested loop L with loop counter ℓ and loop exit condition $(\ell < \mathcal{E}_\ell)$, where $\mathcal{E}_\ell$ is an arithmetic expression involving only constants and variables not updated in L. Thus, the loop iterates $\mathcal{E}_\ell$ times, with the value of ℓ initialized to 0 at the beginning of the first iteration, and incremented at the end of each iteration. Each access of an element of A in the loop is either a *read access* or a *write access*. For example, in the program shown in Fig. 4.4, the loop L (lines 3-12) has three read accesses of A (at lines 7, 8, 9), and three write accesses of A (at lines 5, 8, 9). In order to check an assertion about the array at the end of the loop (see, for example, line 13 of Fig. 4.4), we wish to tile the array based on how its elements are updated in different iterations of the loop, reason about the effect of each loop iteration on the corresponding tile, and then compose the tile-wise reasoning to prove/disprove the overall assertion.

Note that the idea of tiling an array based on access patterns in a loop is not new, and has been used earlier in the context of parallelizing and optimizing compilers [SSK17, JK11]. However, its use in the context of verification has been limited [CCL11]. To explore the idea better, we need to formalize the notion of *tiles*.

Let $\mathsf{Indices}_A$ denote the range of indices of the array A. We assume that this is available to us; in practice, this can be obtained from the declaration of A if it is statically declared, or from the statement that dynamically allocates the array A. Let $\boldsymbol{\varphi}$ and $\boldsymbol{\psi}$ denote the pre- and post-conditions, respectively, for the loop L under consideration. For the ease of exposition, we restrict both $\boldsymbol{\varphi}$ and $\boldsymbol{\psi}$ to be universally quantified formulas with a single quantifier of the form $\forall j\ (\boldsymbol{\alpha}(j) \Rightarrow \boldsymbol{\beta}(\mathsf{A}, \mathcal{V}, j))$, where $\mathcal{V}$ denotes the set of

```
1.  void ArrayUpdate(int A[], int N, int THRESH)
2.  {
3.    for (int l=0; l<N; l=l+1) { // loop L
4.      if ((l == 0) || (l == N-1)) {
5.        A[l] = THRESH;
6.      } else {
7.        if (A[l] < THRESH) {
8.          A[l+1] = A[l] + 1;
9.          A[l] = A[l-1];
10.       } // end if
11.     } // end else
12.   } // end for
13.   assert(∀i∈[0,N) A[i] ≥ THRESH);
14. }
```

Figure 4.4: A Program with Interesting Tiling

scalar variables in the program. To keep the discussion simple, we consider ψ to be of this specific form for the time being, while ignoring the form of φ. Later, in Section 4.2.2, we relax these restrictions and show how the specific form of φ can be used to simplify the analysis further. For purposes of simplicity, we also assume that the array A is one-dimensional; our ideas generalize easily to multi-dimensional arrays, as shown later. Let Inv be a loop invariant for loop L. Clearly, if $\varphi \Rightarrow \mathsf{Inv}$ and $\mathsf{Inv} \wedge \neg(\ell < \mathcal{E}_{\mathsf{L}}) \Rightarrow \psi$, then we are already done, and tile-wise reasoning is not necessary. The situation becomes interesting when Inv is not strong enough to ensure that $\mathsf{Inv} \wedge \neg(\ell < \mathcal{E}_{\mathsf{L}}) \Rightarrow \psi$. We encounter several such cases in our benchmark suite, and it is here that our technique adds value to the existing verification flows.

A *tiling* of A with respect to L, Inv and ψ is a binary predicate $\mathsf{Tile}_{\mathsf{L},\mathsf{Inv},\psi} : \mathbb{N} \times \mathsf{Indices}_{\mathsf{A}} \rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ such that conditions T1 through T3 listed below hold. Note that these conditions were discussed informally in Section 4.1.2 in the context of our motivating example. For ease of notation, we use Tile instead of $\mathsf{Tile}_{\mathsf{L},\mathsf{Inv},\psi}$ below, when L, Inv and ψ are clear from the context. We also use "ℓ^{th} tile" to refer to all array indices in the set $\{j \mid (j \in \mathsf{Indices}_{\mathsf{A}}) \wedge \mathsf{Tile}(\ell, j)\}$.

(T1) *Covers range:* Every array index of interest must be present in some tile, and every tile contains array indices in $\mathsf{Indices_A}$. Thus, the formula $\eta_1 \wedge \eta_2$ must be valid, where $\eta_1 \equiv \forall j\,((j \in \mathsf{Indices_A}) \wedge \boldsymbol{\alpha}(j) \Rightarrow \exists \ell\,((0 \leq \ell < \mathcal{E}_\ell) \wedge \mathsf{Tile}(\ell, j)))$, and $\eta_2 \equiv \forall \ell\,((0 \leq \ell < \mathcal{E}_\ell) \wedge \mathsf{Tile}(\ell, j) \Rightarrow (j \in \mathsf{Indices_A}))$.

(T2) *Sliced post-condition holds inductively:* We define the sliced post-condition for the ℓth tile as $\boldsymbol{\psi}_{\mathsf{Tile}(\ell,\cdot)} \triangleq \forall j\,(\mathsf{Tile}(\ell, j) \wedge \boldsymbol{\alpha}(j) \Rightarrow \boldsymbol{\beta}(\mathsf{A}, \mathcal{V}, j))$. Thus, $\boldsymbol{\psi}_{\mathsf{Tile}(\ell,\cdot)}$ asserts that $\boldsymbol{\beta}(\mathsf{A}, \mathcal{V}, j)$ holds for all relevant j in the ℓth tile. We now require that if the loop invariant Inv and the sliced post-condition for the ℓ'^{th} tile for all $\ell' \in \{0, \ldots, \ell - 1\}$ hold prior to executing the ℓth loop iteration, then the sliced post condition for the ℓth tile and Inv must also hold after executing the ℓth loop iteration.

Formally, if $\mathsf{L}_{\mathrm{body}}$ denotes the body of the loop L, the Hoare triple given by $\{\mathsf{Inv} \wedge \bigwedge_{\ell':0 \leq \ell' < \ell} \boldsymbol{\psi}_{\mathsf{Tile}(\ell',\cdot)}\}\ \mathsf{L}_{\mathrm{body}}\ \{\mathsf{Inv} \wedge \boldsymbol{\psi}_{\mathsf{Tile}(\ell,\cdot)}\}$ must be valid for all $\ell \in \{0, \ldots, \mathcal{E}_\ell - 1\}$.

(T3) *Non-interference across tiles:* For every pair of iterations ℓ, ℓ' of the loop L such that $\ell' < \ell$, the later iteration (ℓ) must not falsify the sliced post condition $\boldsymbol{\psi}_{\mathsf{Tile}(\ell',\cdot)}$ rendered true by the earlier iteration (ℓ').

Formally, the Hoare triple $\{\mathsf{Inv} \wedge (0 \leq \ell' < \ell) \wedge \boldsymbol{\psi}_{\mathsf{Tile}(\ell',\cdot)}\}\mathsf{L}_{\mathrm{body}}\{\boldsymbol{\psi}_{\mathsf{Tile}(\ell',\cdot)}\}$ must be valid for all $\ell \in \{0, \ldots, \mathcal{E}_\ell - 1\}$.

Note that while tiling depends on L, Inv and $\boldsymbol{\psi}$ in general, the array-access patterns in a loop often suggests a natural tiling of array indices that suffices to prove multiple assertions $\boldsymbol{\psi}$ even when Inv by itself may not be strong enough to prove them. We illustrated this simplification in Section 4.1.2 on our motivating example. The example in Fig. 4.4 admits the tiling predicate $\mathsf{Tile}(\ell, j) \equiv (j = \ell)$ based on inspection of array-access patterns in the loop. Note that in this example, the ℓth iteration of the loop can update both $\mathsf{A}[\ell]$ and $\mathsf{A}[\ell + 1]$. However, as we show later, a simple reasoning reveals that the right tiling choice here is $(j = \ell)$, and not $(\ell \leq j \leq \ell + 1)$.

Theorem 4.1 *Suppose* $\mathsf{Tile}_{\mathsf{L},\mathsf{Inv},\boldsymbol{\psi}} : \mathbb{N} \times \mathsf{Indices_A} \rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ *satisfies conditions T1 through T3. If* $\boldsymbol{\varphi} \Rightarrow \mathsf{Inv}$ *also holds and the loop* L *iterates at least once, then the Hoare triple* $\{\boldsymbol{\varphi}\}\ \mathsf{L}\ \{\boldsymbol{\psi}\}$ *holds.*

Proof. The proof proceeds by induction on the values of the loop counter ℓ. The inductive claim is that at the end of the ℓth iteration of the loop, the post-condition $\bigwedge_{\ell':0 \leq \ell' \leq \ell} \boldsymbol{\psi}_{\mathsf{Tile}(\ell',\cdot)}$

holds. The base case is easily seen to be true from condition T2 and from the fact that $\varphi \Rightarrow \mathsf{Inv}$. Condition T3 and the fact that ℓ is incremented at the end of each loop iteration ensure that once we have proved $\psi_{\mathsf{Tile}(\ell,\cdot)}$ at the end of the ℓth iteration, it cannot be falsified in any subsequent iteration of the loop. Condition T2 now ensures that the sliced post-condition can be inductively proven for the ℓth tile. By condition T1, we also have $\bigwedge_{0 \le \ell < \mathcal{E}_\ell} \psi_{\mathsf{Tile}(\ell,\cdot)} \equiv \psi$. Since the loop L iterates with ℓ increasing from 0 to $\mathcal{E}_\ell - 1$, it follows that ψ indeed holds if Inv holds before the start of the first iteration. This is the compositional step in our approach. Putting all the parts together, we obtain a proof of $\{\varphi\}$ L $\{\psi\}$. $\square$

A few observations about the conditions are worth noting. First, note that there is an alternation of quantifiers in the check for T1. Fortunately, state-of-the-art SMT solvers like Z3 [MB08] are powerful enough to check this condition efficiently for tiles expressed as Boolean combinations of linear inequalities on ℓ and $\mathcal{V}$, as is the case for the examples in our benchmark suite. We anticipate that with further advances in reasoning about quantifiers, the check for condition T1 will not be a performance-limiting step.

The checks for T2 and T3 require proving Hoare triples with post-conditions that have a conjunct of the form $\psi_{\mathsf{Tile}(\ell,\cdot)}$. From the definition of a sliced post-condition, we know that $\psi_{\mathsf{Tile}(\ell,\cdot)}$ is a universally quantified formula. Additionally, the pre-condition for T2 has a conjunct of the form $\bigwedge_{\ell':0 \le \ell' < \ell} \psi_{\mathsf{Tile}(\ell',\cdot)}$, which is akin to a universally quantified formula. Therefore T2 and T3 can be checked using Hoare logic-based reasoning tools that permit quantified pre- and post-conditions, viz. [HB08, JSP+11]. Unfortunately, the degree of automation and scalability available with such tools is limited today. To circumvent this problem, we propose to use stronger Hoare triple checks that logically imply T2 and T3, but do not have quantified formulas in their pre- and post-conditions. Since the program, and hence $\mathsf{L}_{\mathrm{body}}$, is assumed not to have nested loops, state-of-the-art bounded model checking tools that work with quantifier-free pre- and post-conditions, viz. CBMC, can be used to check these stronger conditions. Specifically, we propose the following pragmatic replacements of T2 and T3.

(T2*) Let $\mathsf{RdAcc}_{\mathsf{L}}(\ell)$ denote the set of array index expressions corresponding to read accesses of A in the ℓth iteration of the loop L. For example, in Fig. 4.4, $\mathsf{RdAcc}_{\mathsf{L}}(\ell) = \{\ell, \ell - 1\}$. Clearly, if $\mathsf{L}_{\mathrm{body}}$ is loop-free, $\mathsf{RdAcc}_{\mathsf{L}}(\ell)$ is a finite set of expressions.

Suppose $|\ \mathsf{RdAcc}_{\mathsf{L}}(\ell)\ | = k$ and let $e_1, \ldots e_k$ denote the expressions in $\mathsf{RdAcc}_{\mathsf{L}}(\ell)$. Define $\zeta(\ell)$ to be the formula $\bigwedge_{e_k \in \mathsf{RdAcc}_{\mathsf{L}}(\ell)} \big(((0 \leq \ell_k < \ell < \mathcal{E}_\ell) \wedge \mathsf{Tile}(\ell_k, e_k) \wedge \boldsymbol{\alpha}(e_k)) \Rightarrow \boldsymbol{\beta}(\mathsf{A}, \mathcal{V}, e_k) \big)$, where ℓ_k are fresh variables not used in the program. Informally, $\zeta(\ell)$ states that if $\mathsf{A}[e_k]$ is read in the ℓth iteration of L and if e_k belongs to the ${\ell_k}^{th}$ ($\ell_k < \ell$) tile, then $\boldsymbol{\alpha}(e_k) \Rightarrow \boldsymbol{\beta}(\mathsf{A}, \mathcal{V}, e_k)$ holds.

We now require the following Hoare triple to be valid, where j is a fresh free variable not used in the program.
$\{\mathsf{Inv} \wedge (0 \leq \ell < \mathcal{E}_\ell) \wedge \zeta(\ell) \wedge \mathsf{Tile}(\ell, j) \wedge \boldsymbol{\alpha}(j)\}\ \mathsf{L}_{\text{body}}\{\mathsf{Inv} \wedge \boldsymbol{\beta}(\mathsf{A}, \mathcal{V}, j)\}$.

(T3*) Let j' and ℓ' be fresh free variables that are not used in the program. We require the following Hoare triple to be valid:
$\{\mathsf{Inv} \wedge (0 \leq \ell' < \ell < \mathcal{E}_\ell) \wedge \mathsf{Tile}(\ell', j') \wedge \boldsymbol{\alpha}(j') \wedge \boldsymbol{\beta}(\mathsf{A}, \mathcal{V}, j')\}\ \mathsf{L}_{\text{body}}\ \{\boldsymbol{\beta}(\mathsf{A}, \mathcal{V}, j')\}$

Lemma 4.1 *The Hoare triple in T2* implies that in T2. Similarly, the Hoare triple in T3* implies that in T3.*

Proof. Follows from the observation that a counterexample for validity of the Hoare triple in T2 or T3 can be used to construct a counterexample for validity of the triple in T2* or T3* respectively. Notice that Inv is an inductive invariant, hence, $\{\mathsf{Inv}\}\ \mathsf{L}_{\text{body}}\{\mathsf{Inv}\}$ holds. As a result the only way T2 can get violated is if the sliced-post condition $\boldsymbol{\psi}_{\mathsf{Tile}(\ell,\cdot)}$ gets violated. A counterexample that violates this post-condition while checking T2 refers to a specific value of ℓ and a specific value of j such that $\mathsf{Tile}(\ell, j) \wedge \boldsymbol{\alpha}(j)$ evaluates to true but $\boldsymbol{\beta}(\mathsf{A}, \mathcal{V}, j)$ evaluates to false. We use these values of ℓ and j to construct a counterexample that violates T2*. Similarly, a counterexample that violates T3 refers to specific values of ℓ' and j', which are used to construct a counterexample that violates T3*. $\square$

Observe that T2* and T3* require checking Hoare triples with quantifier-free formulas in the pre- and post-conditions. This makes it possible to use assertion checking tools that work with quantifier-free formulas in pre- and post-conditions. Furthermore, since L_{body} is assumed to be loop-free, these checks can also be discharged using state-of-the-art bounded model checkers, viz. CBMC. The scalability and high degree of automation provided by tools like CBMC make conditions T1, T2* and T3* more attractive to use.

4.2.2 Tiling in a General Setting

The above discussion was restricted to a single uni-dimensional array accessed within a single non-nested loop in a program P_N. We now relax these restrictions and show that the same technique continues to work with some adaptations.

We consider the case where P_N is a sequential composition of possibly nested loops. To analyze such programs, we identify all segments in the CFG of P_N. Let CutPts be the set of cut-points of the CFG. Recall from Section 3.2 that a segment is a sub-DAG of the CFG between a source node in $\mathsf{CutPts} \cup \{\mathsf{Start}\}$ and a sink node in $\mathsf{CutPts} \cup \{\mathsf{End}\}$. Thus, a segment s corresponds to a loop-free fragment of P_N. Let ℓ_s denote the loop counter variable corresponding to the innermost loop in which s appears. We assign $\varnothing$ to ℓ_s if s lies outside all loops in P_N. Let $\mathsf{OuterLoopCtrs}_s$ denote the set of loop counter variables of all outer loops (excluding the innermost one) that enclose (or nest) s. The syntactic restrictions of programs described in Section 3.1 ensure that ℓ_s and $\mathsf{OuterLoopCtrs}_s$ are uniquely defined for every segment s.

Suppose we are given invariants at every cut-point in P_N, where Inv_c denotes the invariant at cut-point c. We assume the invariants are of the usual form $\forall I \; (\boldsymbol{\alpha}(I) \Rightarrow \boldsymbol{\beta}(\mathcal{A}, \mathcal{V}, I))$, where I is a sequence of quantified array index variables, and $\mathcal{A}$ and $\mathcal{V}$ are sequences of array and scalar variables respectively. Let $\mathcal{A}_s$ be a sequence of arrays that are updated in the segment s between cut-points c_1 and c_2, and for which $\ell_s \neq \varnothing$. We define a tiling predicate $\mathsf{Tile}_{s,\mathsf{Inv}_{c_1},\mathsf{Inv}_{c_2}} : \mathbb{N} \times \mathsf{Indices}_{\mathcal{A}_s} \rightarrow \{\mathbf{tt}, \mathbf{ff}\}$, where $\mathsf{Indices}_{\mathcal{A}_s} = \prod_{\mathsf{A}' \in \mathcal{A}_s} \mathsf{Indices}_{\mathsf{A}'}$ plays a role similar to that of $\mathsf{Indices}_{\mathsf{A}}$ in Section 4.2.1 (where a single array A was considered). The predicate $\mathsf{Tile}_{s,\mathsf{Inv}_{c_1},\mathsf{Inv}_{c_2}}$ relates values of the loop counter ℓ_s of the innermost loop containing s to the index expressions that define the updates of arrays in $\mathcal{A}_s$ in the program segment s. The entire analysis done in Section 4.2.1 for a simple loop L can now be re-played for segment s, with Inv_{c_1} playing the role of Inv, Inv_{c_2} playing the role of $\boldsymbol{\psi}$, $\mathcal{V} \cup \mathsf{OuterLoopCtrs}_s$ playing the role of $\mathcal{V}$, and ℓ_s playing the role of ℓ. If the segment s is not enclosed in any loop, i.e. $\ell_s = \varnothing$, we need not define any tiling predicate for this segment. This obviates the need for conditions T1 and T3, and checking T2 simplifies to checking the validity of the Hoare triple $\{\mathsf{Inv}_{c_1}\} \; s \; \{\mathsf{Inv}_{c_2}\}$. In general, Inv_{c_1} and Inv_{c_2} may be universally quantified formulas. In such cases, the technique used to simplify condition T2 to T2* in Section 4.2.1 can be applied to obtain a stronger condition, say T2**, that does not involve any tile, and requires checking a Hoare triple with quantifier-free pre-

and post-conditions. If the condition checks for all segments as described above succeed, it follows from Theorem 4.1 and Lemma 4.1 that we have a proof of $\{\varphi\}$ P_N $\{\psi\}$.

Recall that in Section 4.2.1, we ignored the specific form of the pre-condition φ. φ has the same form as that of the post-condition ψ and invariants at cut-points considered above. Therefore, the above technique works if we treat φ as $\mathsf{Inv}_{\mathsf{Start}}$ and ψ as $\mathsf{Inv}_{\mathsf{End}}$.

The extension to multi-dimensional arrays is straightforward. Instead of using one index variable j for accessing arrays, we now allow a tuple of index variables $(j_1, j_2, \ldots, j_r)$ for accessing arrays. Each such variable j_i, $1 \leq i \leq r$, takes values from its own domain, say $\mathsf{Indices}_{\mathsf{A}_i}$. The entire discussion about tiles above continues to hold, including the validity of Theorem 4.1, if we replace every occurrence of an array index variable j by a sequence of variables $j_1, j_2, \ldots, j_r$ and every occurrence of $\mathsf{Indices}_{\mathsf{A}}$ by $\mathsf{Indices}_{\mathsf{A}_1} \times \mathsf{Indices}_{\mathsf{A}_2} \times \ldots \times \mathsf{Indices}_{\mathsf{A}_r}$.

4.3 Algorithms for Verification by Tiling

The discussion in the previous section suggests a three-phase algorithm, presented as routine TILEDVERIFY in Algorithm 1, for verifying quantified properties of arrays in programs with sequences of possibly nested loops that manipulate arrays. In the first phase of the algorithm, we use bounded model checking with small pre-determined loop unrollings to check for assertion violations. If this fails, we construct the CFG of the input program P_N, topologically sort its cut-points and initialize the sets of candidate invariants at each cut-point to $\emptyset$.

In the second phase, we generate candidate invariants at each cut-point c by considering every segment s that ends at c. For each such segment s, we identify the loop counter $\ell[s]$ corresponding to the innermost loop in which s appears, and the set of loop counters $\mathsf{OuterLoopCtrs}[s]$ corresponding to other loops that contain (or nest) s. Note that when the program fragment in the segment s executes, the *active* loop counter that increments from one execution of s to the next is $\ell[s]$. The loop counters in $\mathsf{OuterLoopCtrs}[s]$ can be treated similar to other scalar variables in $\mathcal{V}$ when analyzing segment s. We would like the candidate invariants identified at different cut-points to be of the form $\forall I \left(\boldsymbol{\alpha}(I) \Rightarrow \boldsymbol{\beta}(\mathcal{A}, \mathcal{V}, I)\right)$, whenever possible. We assume access to a routine FINDHEURISTICCANDIDATEINVARIANTS for this purpose. Note that the candidate

Algorithm 1 TiledVerify(P_N: program, φ: pre-condition, ψ: post-condition)

1: Let G be the CFG for program $\mathsf{P}_N = (\mathcal{A}, \mathcal{V}, \mathcal{L}, \mathsf{PB}, N)$, as defined in Section 3.1.
 ▶ `Check for shallow counterexample and initialization`
2: Do bounded model checking with pre-determined small loop unrollings;
3: **if** counterexample found **then**
4:   **return** "Post condition violated!";
5: CutPts := set of cut-points in G;
6: Remove all back-edges from G and topologically sort CutPts; ▷ Let $\sqsubseteq$ be the sorted order
7: **for** each c in CutPts **do**
8:   CandInv[c] := $\emptyset$; ▷ Set of candidate invariants at c
9: CandInv[Start] := φ; CandInv[End] := ψ; ▷ Fixed invariants at Start and End
 ▶ `Candidate invariant generation`
10: **for** each segment s from c_1 to c_2, where $c_1, c_2 \in \mathsf{CutPts} \cup \{\mathsf{Start}, \mathsf{End}\}$ and $c_1 \sqsubseteq c_2$ **do**
11:   **if** (s lies within a loop) **then**
12:     ℓ[s] := loop counter of innermost nested loop containing s;
13:     $\mathsf{OuterLoopCtrs}$[s] := loop counters of all other outer loops containing s;
14:   **else** ▷ s not in any loop
15:     ℓ[s] := $\varnothing$;
16:     $\mathsf{OuterLoopCtrs}$[s] := $\emptyset$;
17:   $\mathsf{ScalarVars}$[s] := $\mathcal{V} \cup \mathsf{OuterLoopCtr}$[s];
18:   CandInv[c_2] := CandInv[c_2] $\cup$ $\mathsf{findHeuristicCandidateInvariants}(s, c_2, \ell[s], \mathsf{ScalarVars}[s], \mathcal{A})$;
 ▶ `Tiling and verification`
19: **for** each segment s from c_1 to c_2 **do**
20:   **if** (s lies within a loop) **then**
21:     CandTile[s] := $\mathsf{findHeuristicTile}(s, \ell[s], \mathsf{ScalarVars}[s], \mathcal{A})$; ▷ Candidate tile for s
22:     Check conditions T1, T2* and T3* for CandTile[s], presented in Section 4.2.1;
23:     **if** (not timed out) AND (T1 or T3* fail) **then**
24:       Re-calculate CandTile[s] using different heuristics;
25:       **goto** 22;

26: **if** (not timed out) AND (T2* or T3* fail) AND ($c_2 \neq \mathsf{End}$) **then**

27: Re-calculate CandInv[c_2] using different heuristics;

28: **goto** 22;

29: **else** ▷ s not in any loop

30: Check condition T2**, as described in Section 4.2.2;

31: **if** (not timed out) AND (T2** fails) AND ($c_2 \neq \mathsf{End}$) **then**

32: Re-calculate CandInv[c_2] using different heuristics;

33: **goto** 30;

34: **if** timed out **then**

35: **return** "Time out! Inconclusive answer!";

36: **return** "Post-condition verified!";

invariants obtained from this routine may not actually hold at c_2. In the next phase, we check using our tile-wise reasoning whether a candidate invariant indeed holds at a cut-point, and use only those candidates that we are able to prove.

In the third phase, we iterate over every segment s between cut-point c_1 and c_2 again, and use heuristics to identify tiles. This is done by a routine FINDHEURISTICTILE. The working of our current tile generation heuristic is shown in Algorithm 2. For every array update $\mathsf{A}[e] := e'$ in segment s, the heuristic traverses the control flow graph of s backward until it reaches the entry point of s, i.e. c_1, to determine the expression e in terms of values of $\ell[s]$, $\mathcal{V}$, $\mathsf{OuterLoopCtrs}[s]$ and $\mathcal{A}$ at c_1. Let $\mathsf{UpdIndexExprs}^{\mathsf{A}}[s]$ denote the set of such expressions for updates to A within s. We identify an initial tile for A in s as $\mathsf{InitTile}^{\mathsf{A}}(\ell[s], j) \equiv \bigvee_{e \in \mathsf{UpdIndexExprs}^{\mathsf{A}}[s]}(j = e)$. It may turn out that the same array index expression appears in two or more initial tiles after this step. For example, in Fig. 4.4, we obtain $\mathsf{InitTile}^{\mathsf{A}}(\ell, j) \equiv (\ell \leq j \leq \ell + 1)$, and hence $\mathsf{InitTile}^{\mathsf{A}}(\ell, \ell+1) \wedge \mathsf{InitTile}^{\mathsf{A}}(\ell+1, \ell+1)$ is satisfiable. While the conditions T1, T2 and T3 do not forbid overlapping tiles in general (non-interference is different from non-overlapping tiles), our current heuristic for generating a tile avoids them by refining the initial tile estimates. For each expression e in $\mathsf{UpdIndexExprs}^{\mathsf{A}}[s]$, we check if $\mathsf{InitTile}^{\mathsf{A}}(\ell[s], e) \wedge \mathsf{InitTile}^{\mathsf{A}}(\ell[s] + k, e) \wedge (0 \leq \ell[s] < \ell[s] + k < \mathcal{E}_{\ell[s]})$ is satisfiable. If so, we drop e from the refined tiling predicate, denoted $\mathsf{Tile}^{\mathsf{A}}(\ell[s], \cdot)$ in Algorithm 2. This ensures that an array index expression e belongs to the tile corresponding to the largest value of the loop counter

Algorithm 2 FINDHEURISTICTILE(s : segment, ℓ: loop counter, ScalarVars: set of scalars, $\mathcal{A}$: set of arrays)

1: Let c_1 be the starting cut-point (or Start node) of s;
2: **for** each array A updated in s **do**
3: $\quad \mathsf{UpdIndexExprs}^{\mathsf{A}}[s] := \emptyset$;
4: $\quad$ **for** each update of the form $\mathsf{A}[e] := e'$ at location c in s **do** $\quad \triangleright$ e and e' are arithmetic expressions
5: $\quad\quad \widehat{e} := e$ in terms of ℓ, ScalarVars, $\mathcal{A}$ at c_1; $\quad \triangleright$ Obtained by backward traversal from c to c_1
6: $\quad\quad \mathsf{UpdIndexExprs}^{\mathsf{A}}[s] := \mathsf{UpdIndexExprs}^{\mathsf{A}}[s] \cup \{\widehat{e}\}$;
7: $\quad \mathsf{InitTile}^{\mathsf{A}}(\ell, j) := \mathsf{Simplify}\big(\bigvee_{e \in \mathsf{UpdIndexExprs}^{\mathsf{A}}[s]}(j = e)\big)$; $\quad \triangleright$ Initial estimate of tile
8: $\quad$ **for** each $e \in \mathsf{UpdIndexExprs}^{\mathsf{A}}[s]$ **do**
9: $\quad\quad$ **if** $\big(\mathsf{InitTile}^{\mathsf{A}}(\ell, e) \wedge \mathsf{InitTile}^{\mathsf{A}}(\ell + k, e) \wedge (0 \leq \ell < \ell + k < \mathcal{E}_\ell)\big)$ is satisfiable **then**
10: $\quad\quad\quad$ Remove e from $\mathsf{UpdIndexExprs}^{\mathsf{A}}[s]$;
11: $\quad \mathsf{Tile}^{\mathsf{A}}(\ell, j) := \mathsf{Simplify}\big(\bigvee_{e \in \mathsf{UpdIndexExprs}^{\mathsf{A}}[s]}(j = e)\big)$; $\quad \triangleright$ Refined tile
12: **return** $\bigwedge_{\mathsf{A} \in \mathcal{A}} \mathsf{Tile}^{\mathsf{A}}(\ell, \cdot)$;

$\ell[s]$ when it is updated. The procedure Simplify invoked in lines 7 and 11 of Algorithm 2 tries to obtain a closed form linear expression (or Boolean combination of a few linear expressions) for $\bigvee_{e \in \mathsf{UpdIndexExprs}^{\mathsf{A}}[s]}(j = e)$, if possible. In the case of Fig. 4.4, this gives the tile $(j = \ell)$, which suffices for proving the quantified assertion in this example.

The choice of the generated tile by our heuristic or the choice of candidate invariants may not always be good enough for the requisite checks (T1, T2*, T2**, T3) to go through. In such cases, Algorithm 1 allows different heuristics to be used to update the tiles and invariants. In our current implementation, we do not update the tiles, but update the set of candidate invariants by discarding candidates that cannot be proven using our tile-wise reasoning. The tiles and candidate invariants obtained in this manner may not always suffice to prove the assertion within a pre-defined time limit. In such cases, we time out and report an inconclusive answer.

```
1.  void copynswap(int a[], int b[], int S)
2.  {
3.    int acopy[S], i, tmp;
4.    for (i = 0; i < S; i++) { // loop L1
5.      acopy[i] = a[i];
6.    }
7.    for (i = 0; i < S; i++) { // loop L2
8.      tmp = a[i];
9.      a[i] = b[i];
10.     b[i] = tmp;
11.   }
12.   assert(∀i ∈ [0,S), b[i] = acopy[i]);
13. }
```

Figure 4.5: Program with an Assertion

4.4 Experimental Evaluation

In this section, we experimentally evaluate the verification efficacy of our tile-wise reasoning technique on a set of array-manipulating benchmarks.

4.4.1 Implementation

We have implemented the above technique in a tool called TILER. The tool is publicly available at [Unab]. The tool is built on top of the LLVM/CLANG [LA04] compiler infrastructure. We ensure that input C programs are adapted, if needed, to satisfy the syntactic restrictions in Section 3.1 from Chapter 3. The current implementation is *fully automated* for programs with non-nested loops, and can handle programs with nested loops semi-automatically.

Generating Candidate Invariants

We use a template-based dynamic analysis tool, called DAIKON [EPG+07], for generating *candidate* invariants. DAIKON supports linear invariant discovery among program variables and arrays, and reports invariants at the entry and exit points of functions. In order

to learn candidate quantified invariants, we transform the input program as follows. The sizes of all arrays in the program are changed to a fixed small constant, and all arrays and program variables that are live are initialized with random values. We then insert a dummy function call at each cut-point. Our transformation collects all array indices that are accessed in various segments of the program and expresses them in terms of the corresponding loop counter(s). Finally, it passes the values of accessed array elements, the corresponding array index expressions and the loop counter(s) as arguments to the dummy call, to enable DAIKON to infer candidate invariants among them. The transformed program is executed multiple times to generate traces. DAIKON learns candidate linear invariants over the parameters passed to the dummy calls from these traces. Finally, we lift the candidate invariants thus identified to quantified invariants in the natural way.

As an example, consider the input program shown in Fig. 4.5. The transformed program is shown in Fig. 4.6. In the transformed program, arrays a and b are initialized to random values. The *dummy* function call in loop $L1$ has four arguments $a[i]$, $b[i]$, $acopy[i]$ and i. Based on concrete traces, DAIKON initially detects the candidate invariants $(a_i = acopy_i)$ and $(a_i \neq b_i)$ on the parameters of the dummy function. We lift these to obtain the candidate quantified invariants $\forall i.(a[i] = acopy[i])$ and $\forall i.(a[i] \neq b[i])$. In the subsequent analysis, we detect that $\forall i, (a[i] \neq b[i])$ cannot be proven. This is therefore dropped from the candidate invariants (line 24 of Algorithm 1), and we proceed with $\forall i, (a[i] = acopy[i])$, which suffices to prove the post-condition.

Tile Generation and Checking

Tiles are generated as in Algorithm 2. Condition T1 is checked using Z3 [MB08], which has good support for quantifiers. We employ CBMC [CKL04] for implementing the checks T2*, T2** and T3*.

4.4.2 Benchmarks

We evaluated our tool TILER on 60 benchmarks from the test-suites of BOOSTER [AGS14] and VAPHOR [MG16], as well as on programs from a code base of an industrial battery controller. The benchmarks from BOOSTER and VAPHOR test-suites (Table 4.1) perform common array operations such as array initialization, reverse order initialization, incrementing array contents, finding largest and smallest elements, odd and even elements,

```
1.  void dummy(int a_i, int b_i, int acopy_i, int i, int S) {}

2.  void copynswap_inst()
3.  {
4.    int S=10, i, tmp;
5.    int a[S], b[S], acopy[S];
6.    for (i = 0; i < S; i++) { // initialization
7.      a[i] = rand();
8.      b[i] = rand();
9.    }
10.   for (i = 0; i < S; i++) { // loop L1
11.     acopy[i] = a[i];
12.     dummy(a[i], b[i], acopy[i], i, S);
13.   }
14.   for (i = 0; i < S; i++) { // loop L2
15.     tmp = a[i];
16.     a[i] = b[i];
17.     b[i] = tmp;
18.   }
19. }
```

Figure 4.6: Transformed Program Input to Daikon

array comparison, array copying, swapping arrays, swapping a reversed array, multiple swaps, and the like. Of the 135 benchmarks in this test suite, 66 benchmarks are minor variants of the benchmarks we report. For example, there are multiple versions of programs such as `copy`, `init`, `copyninit`, with different counts of sequentially composed loops. In such cases, the benchmark variant with the largest count is reported in the table. Besides these, there are 22 cases containing nested loops which can currently be handled only semi-automatically by our implementation, and 25 cases with post-conditions in a form that is different from what our tool accepts. Hence, these results are not reported here.

Benchmarks were also taken from a code base of an industrial battery controller in a

high-end automotive (Table 4.2). These benchmarks set a repetitive contiguous bunch of cells in a battery with different values based on the guard condition that gets satisfied. The updates to individual cells in the bunch, however, may be performed in a non-sequential manner in a loop iteration. The size of such a contiguous bunch of cells varies in different models. The assertion checks if the cell values are consistent with the given specification.

All our benchmarks are within 100 lines of uncommented code. The programs have a variety of tiles such as $4i - 4 \leq j < 4i$, $2i - 2 \leq j < 2i$, $j = size - i - 1$, $j = i$ etc., with the last one being the most common tile, where i denotes the loop counter and j denotes the array index accessed.

4.4.3 Experimental Setup

The experiments reported here were conducted on an Intel Core i5-3320M processor with 4 cores running at 2.6 GHz, with 4GB of memory running Ubuntu 14.04 LTS. A time-out of 900 seconds was set for Tiler, SMACK+Corral [HCE+15], Booster [AGS14] and Vaphor [MG16]. The memory limit was set to 1GB for all the tools. Spacer [KGC14] was used as the SMT solver for the Horn formulas generated by Vaphor since this has been reported to perform well with Vaphor. In addition, C programs were manually converted to mini-Java, as required by Vaphor. Since SMACK+Corral is a bounded model checker, a meaningful comparison with Tiler can be made only in cases where the benchmark violates a quantified assertion. In such cases, the verifier option `svcomp` was used for Corral. In all other cases, we have shown the symbol of † as the result from SMACK+Corral to indicate that comparison is not meaningful.

4.4.4 Results

Tables 4.1 and 4.2 show the results obtained by running the tools on the set of benchmarks. In the table, the first column gives the name of the benchmark, the second column labeled #Loops indicates the number loops (and sub-loops, if any) in the benchmark, from the third column onwards the table shows the results for the tools Tiler, SMACK+Corral, Booster, and Vaphor respectively. ✓ indicates assertion safety, ✗ indicates assertion violation, ? indicates unknown result, and $\star$ indicates unsupported construct. All the times reported in the table are in seconds. TO stands for time-out,

Benchmark	#Loops	Tiler	SMACK+Corral	Booster	Vaphor
init2ipc.c	1	✓ 0.5	†	✓ 0.01	✓ 1.0
initnincr.c	2	✓ 5.8	†	✓ 0.01	✓ 0.7
evenodd.c	1	✓ 0.4	†	✓ 0.01	✓ 0.04
revrefill.c	1	✓ 0.6	†	✓ 0.01	✓ 0.79
largest.c	1	✓ 0.4	†	✓ 0.01	✓ 0.02
smallest.c	1	✓ 0.4	†	✓ 0.01	✓ 0.02
cpy.c	1	✓ 0.6	†	✓ 0.01	✓ 2.0
cpynrev.c	2	✓ 3.8	†	✓ 3.1	✓ 5.4
cpynswp.c	2	✓ 4.2	†	✓ 12.4	✓ 1.38
cpynswp2.c	3	✓ 10.2	†	✓ 198	✓ 7.2*
01.c	1	✓ 0.44	†	✓ 0.05	✓ 0.38
02.c	1	✓ 0.65	†	✓ 0.02	✓ 2.3
06.c	2	✓ 8.15	†	✓ 0.04	✓ 0.35
27.c	1	✓ 0.41	†	✓ 0.01	✓ 0.12
43.c	1	✓ 0.45	†	✓ 0.03	✓ 0.05
maxinarr.c	1	✓ 0.51	†	✓ 0.01	✓ 0.11
mininarr.c	1	✓ 0.53	†	✓ 0.02	✓ 0.13
compare.c	1	✓ 0.44	†	✓ 0.04	✓ 0.62
palindrome.c	1	✓ 0.52	†	✓ 0.02	✓ 0.39
copy9.c	9	✓ 34.6	†	✓ 0.46	TO
init9.c	9	✓ 29.2	†	✓ 0.34	✓ 0.16
seqinit.c	1	✓ 0.45	†	✓ 0.03	✓ 0.43
nec40t.c	1	✓ 0.50	†	✓ 0.06	✓ 0.48
sumarr.c	1	✓ 0.55	†	✓ 0.56	✓ 4.2
vararg.c	1	✓ 0.42	†	✓ 0.03	✓ 0.12
find.c	1	✓ 0.52	†	✓ 0.02	✓ 0.14
running.c	1	✓ 0.62	†	✓ 0.04	✓ 0.12
revcpy.c	1	✓ 0.7	†	✓ 0.01	✓ 0.73
revcpyswp.c	2	✓ 6.3	†	✓ 0.02	TO
revcpyswp2.c	3	✓ 8.6	†	✓ 0.03	TO

Table 4.1: Results on Benchmarks from Booster & Vaphor Test-suites.

Benchmark	#Loops	Tiler	SMACK+Corral	Booster	Vaphor
copy9u.c	9	✗ 0.16	✗ 4.48	✗ 0.44	✗ 30.8
init9u.c	9	✗ 0.15	✗ 3.77	✗ 0.32	✗ 0.14
revcpyswpu.c	2	✗ 0.18	✗ 3.11	✗ 0.01	TO
skippedu.c	1	✗ 0.81	✗ 2.94	✗ 0.02	TO
mclceu.c	1	? 0.37	✗ 2.5	⋆	⋆
poly1.c	1	TO	†	✓ 15.7	TO
poly2.c	2	? 6.44	†	? 19.5	TO
tcpy.c	1	? 0.65	†	TO	✓ 25.1
skipped.c	1	✓ 1.24	†	TO	TO
rew.c	1	✓ 0.48	†	✓ 0.01	TO
rewrev.c	1	✓ 0.39	†	TO	TO
rewnif.c	1	✓ 0.49	†	✓ 0.01	TO
rewnifrev.c	1	✓ 0.28	†	✓ 0.01	TO
rewnifrev2.c	1	✓ 0.47	†	✓ 0.01	TO
pr2.c	1	✓ 0.51	†	TO	TO
pr3.c	1	✓ 0.70	†	TO	TO
pr4.c	1	✓ 0.68	†	TO	TO
pr5.c	1	✓ 1.32	†	TO	TO
pnr2.c	1	✓ 0.55	†	TO	TO
pnr3.c	1	✓ 0.98	†	TO	TO
pnr4.c	1	✓ 0.86	†	TO	TO
pnr5.c	1	✓ 1.98	†	TO	TO
mbpr2.c	2	✓ 6.48	†	TO	TO
mbpr3.c	3	✓ 9.24	†	TO	TO
mbpr4.c	4	✓ 12.75	†	TO	TO
mbpr5.c	5	✓ 18.08	†	TO	TO
nr2.c	1-1	✓ 1.48*	†	TO	TO
nr3.c	1-1	✓ 2.02*	†	TO	TO
nr4.c	1-1	✓ 2.43*	†	TO	TO
nr5.c	1-1	✓ 2.90*	†	TO	TO

Table 4.2: Results on Benchmarks from Code of an Industrial Battery Controller.

indicating that the tool did not terminate with a result within the stipulated time limit. * indicates that semi-automated experiments were performed and the execution times shown in the table correspond to the automated part of the experiments.

TILER takes about two seconds for verifying all single loop programs that satisfy their assertions. For programs containing multiple loops, 10 random runs of the program were used to generate candidate invariants using DAIKON. For extracting program traces, DAIKON suggests the tool KVASIR. However, KVASIR has a dependency on the Linux operating system due to its use of VALGRIND instrumentation framework. We have implemented an automatic trace extraction module to generate traces in DAIKON's input format. Hence, we do not rely on KVASIR for instrumentation and trace generation. The weak loop invariant Inv, mentioned in Section 4.2, was assumed to be **true**. TILER took a maximum of 35 seconds to output the correct result for each such benchmark. The execution time of TILER includes program instrumentation, trace generation, execution of DAIKON on the traces for extracting candidate invariants, translating these to `assume` statements for use in CBMC, proving the reported candidate invariants and proving the final assertion. The execution of DAIKON and proving candidate invariants took about 95% of the total execution time.

To demonstrate the application of our technique on programs with nested loops, we applied it to the last four benchmarks in Table 4.2, each of which has a loop nested inside another. We used TILER to automatically generate tiles for these programs. However, for the purposes of this demonstration, we did not automate the calls to CBMC for the class of programs with nested loops. Instead, we manually encoded the *sliced post-condition* and *non-interference* queries (see Section 4.2) and ran CBMC.

4.4.5 Analysis

BOOSTER and VAPHOR performed well on benchmarks from their respective repositories. Although VAPHOR could analyze the benchmark for reversing an array, as well as one for copying and swapping arrays, it could not analyze the benchmark for reverse copying and swapping. Since the arrays are reversed and then swapped, all array indices need to be tracked in this case, causing VAPHOR to fail. VAPHOR also could not verify most of the industrial benchmarks due to two key reasons that are not handled well by VAPHOR: (i) at least two distinguished array cells need to be tracked in these benchmarks, and (ii)

```
1. void mclceu(int a[], int N)
2. {
3.   for(i=0; i<N; i++) {
4.     if(i>>16 > 250) {
5.       a[i] = 1;
6.     } else {
7.       a[i] = 0;
8.     }
9.   }
10.  assert(∀x∈[0,N) a[x] = 0);
11.}
```

(a)

```
1. void tcpy(int a[], int tcpy[], int N)
2. {
3.   if(N % 2 != 0)  { return; }
4.   for (i=0; i<N/2; i++) {
5.     tcpy[N-i-1] = a[N-i-1];
6.     tcpy[i] = a[i];
7.   }
8.   assert(∀x∈[0,N) a[x] = tcpy[x]);
9. }
```

(b)

Figure 4.7: (a) `mclceu.c` and (b) `tcpy.c`

updates to the arrays are made using non-sequential index values.

Booster could analyze all the examples in which the assertion gets violated, except for a benchmark containing an unsupported construct (shift operator) indicated by $\star$. This is not surprising since finding a violating run is sometimes easier than proving an assertion. Booster however could not prove several other industrial benchmarks because it could not accelerate the expressions for indices at which the array was being accessed. Tiler on the other hand, was able to generate interesting tiles for almost all these benchmarks.

In our experiments, SMACK+Corral successfully generated counter-examples for all benchmarks in which the assertion was violated. As expected, it was unable to produce any conclusive results for benchmarks with parametric array sizes where the quantified assertions were satisfied. Note that we used the latest versions of the tools Booster, Vaphor and SMACK+Corral available during our experimentation in 2017. Since then these tools have not been updated.

4.4.6 Limitations

There are several scenarios under which Tiler may fail to produce a conclusive result. Tiler uses CBMC with small loop unwinding bounds to find violating runs in programs with shallow counter-examples. Consequently, when there are no short counter-examples

```
1. void poly2(int a[], int b[], int N)
2. {
3.   for(i=0; i<N; i++) {
4.     a[i] = i*i + 2;
5.   }
6.   for(j=0; j<N; j++) {
7.     b[j] = a[j] - 2;
8.   }
9.   assert(∀x∈[0,N) b[x] = x²);
10.}
```

(a)

```
1. void poly(int a[], int N)
2. {
3.   for(i=0; i<N; i++) {
4.     a[i] = i*i;
5.   }
6.   assert(∀x∈[0,N) a[x] = x²);
7. }
```

(b)

Figure 4.8: (a) poly2.c and (b) poly.c

(e.g. in mclceu.c shown in Fig. 4.7(a)), TILER reports an inconclusive answer. TILER is also unable to report conclusively in cases where the tile generation heuristic is unable to generate the right tile (e.g. in tcpy.c shown in Fig. 4.7(b)), when DAIKON generates weak mid-conditions (e.g. in poly2.c shown in Fig. 4.8(a)) or when CBMC (version 5.1) takes too long to prove conditions T2* or T3* (e.g. in poly1.c shown in Fig. 4.8(b)).

Our work is motivated by the need to prove quantified assertions in programs from industrial code bases, where we observed interesting array-access patterns. Our tile generation heuristic is strongly motivated by these patterns. There is clearly a need to develop more generic tile generation heuristics for larger classes of programs.

4.5 Comparison with Related Techniques

The VAPHOR tool [MG16] uses an abstraction to transform array-manipulating programs to array-free Horn formulas, parameterized by the number of array cells that are to be tracked. The technique relies on Horn clause solvers such as Z3 [MB08], SPACER [KGC14] and ELDARICA [RHK13] to check the satisfiability of the generated array-free Horn formulas. VAPHOR does not automatically infer the number of array cells to be tracked to prove the assertion. It also fails if the updates to the array happen at non-sequential indices, as is the case in array reverse and swap, for example. In comparison, TILER requires no input on the number of cells to be tracked and is not limited by sequential accesses. The

experiments in [MG16] show that Horn clause solvers are not always efficient on problems arising from program verification. To be efficient on a wide range of verification problems, the solvers need to have a mix of heuristics. Our work brings a novel heuristic in the mix, which may be adopted in these solvers.

Booster [AGS14] combines acceleration [BIK10, JSS14] and lazy abstraction with interpolants for arrays [ABG+12a] for proving quantified assertions on arrays for a class of programs. Interpolation for universally quantified array properties is known to be hard [JM07, MA15]. Hence, Booster fails for programs where simple interpolants are not easily computable. Fluid updates [DDA10] uses bracketing constraints, which are over- and under-approximations of indices, to specify the concrete elements being updated in an array without explicit partitioning. This approach is not property-directed and their generalization assumes that a single index expression updates the array.

The analysis proposed in [GRS05, HP08] partitions the array into symbolic slices and abstracts each slice with a numeric scalar variable. These techniques cannot easily analyze arrays with overlapping slices, and they do not handle updates to multiple indices in the array or to non-contiguous array partitions. In comparison, Tiler uses state-of-the-art SMT solver Z3 [MB08] with quantifier support [BJ15] for checking interference among tiles and can handle updates to multiple non-contiguous indices.

Abstract interpretation based techniques [LR15, CCL11] propose an abstract domain which utilizes cell contents to split array cells into groups. In particular, the technique in [LR15] is useful when array cells with similar properties are non-contiguously present in the array. All the industrial benchmarks in our test-suite are such that this property holds. Furthermore, these approaches require the implementation of abstract transformers for each specialized domain due to the subtleties in the analysis. Implementing these transformers requires significant effort, making such techniques less appealing in practice. Template-based techniques [GMT08] have been used to generate expressive invariants. However, this requires the user to supply the right templates, which may not be easy in general. In [JKD+16], a technique to scale bounded model-checking by transforming a program with arrays and possibly unbounded loops to an array-free and loop-free program is presented. This technique is not compositional, and is precise only for a restricted class of programs.

Stencil [Dat09] refers to some fixed pattern of updating array elements in loops.

Computational simulations in fluid dynamics, partial differential equations, Jacobi kernels, Gauss-Seidel method, image processing, and cellular automata, are commonly modeled using stencils. The access patterns in these stencil computations often do not satisfy the conditions required to qualify as a *tile*. Additional, state-of-the-art methods have focused on optimizing the computation performance of programs that use patterns and not on proving the correctness of such programs using automatically inferred patterns which is our primary goal.

There are some close connections between the notion of tiles as used in this chapter and similar ideas used in compilers. The loop tiling optimization [Muc97] is performed by compilers an aim of optimizing the memory/cache performance. This transformation explicitly partitions the loops, possibly by introducing nested loops, possibly to ensure fewer cache misses and reduces memory overhead. The loop tiling optimization has also been used in compilers for translating loops into the SIMD instruction set [JK11, RAL+13]. Our concept of *tiles* and its computation are different from the loop tiling compiler optimization [Muc97]. The aim of our technique is to improve the verification efficiency, while the latter is aimed at improving memory/cache performance. These goals are not necessarily aligned, making our technique orthogonal. For instance, verifying programs obtained by such partitioning of loops may be harder. On the contrary, we do not explicitly partition loops but identify regions in an array pertaining to a loop iteration and use this information to simplify the verification goals. Another noteworthy difference is that the size of memory/cache and its performance do not have any influence on our method of identifying tiles and their use during verification.

Another relevant compiler optimization is the loop induction variable analysis, for instance in the LLVM compiler, that can generate all index expressions for an array that are accessed in a loop iteration in terms of the loop counters. Note, however, that not all such expressions may be part of a *tile* (recall the tiles in Fig. 4.4). Hence, automatically generating the right *tile* still remains a challenging problem in general.

4.6 Conclusion

Programs that manipulate arrays are often extremely hard to reason about. The problem is further exacerbated when the programmer uses diverse array-access patterns in differ-

ent loops. In this chapter, we provided a theory of tiling that helps us decompose the reasoning about an array into reasoning about automatically identified tiles in the array, and then compose the results for each tile back to obtain the overall result. While generation of tiles is difficult in general, we have shown that simple heuristics are often quite effective in automatically generating tiles that work well in practice. Surprisingly, these simple heuristics allow us to analyze programs that several state-of-the-art tools choke on. Further work is needed to identify better and varied tiles for programs automatically.

Chapter 5

Verification by Full-Program Induction

In this chapter, we build from ground up a novel inductive reasoning technique to prove a sub-class of quantified and quantifier-free properties of programs that manipulate arrays. A part of the work described in this chapter has been published as a conference paper in TACAS 2020 [CGU20a] and as a journal paper in STTT [CGU22].

5.1 Introduction

We present a new verification technique, called *full-program induction*, to add to the arsenal of a portfolio of verification techniques. Specifically, we consider array-manipulating programs generated by the grammar shown in Fig. 3.2. The programs consist of non-nested loops that manipulate arrays, where the size of each array is a symbolic integer parameter N (> 0). We allow (a sub-class of) quantified and quantifier-free pre- and post-conditions that may depend on the symbolic parameter N as described in Section 3.4. Thus, the problem we wish to solve can be viewed as checking the validity of a parameterized Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ for all values of N (> 0), where the program P_N computes with arrays of size N, and N is a free variable in $\varphi(\cdot)$ and $\psi(\cdot)$.

Like earlier verification approaches [SSS00], our technique also relies on mathematical induction to reason about programs with loops. However, the way in which the inductive claim is formulated and proved differs significantly from the previous techniques. Specifically, (i) we *induct on the full program* (possibly containing multiple

loops) with parameter N and not on iterations of individual loops in the program, (ii) we perform *non-trivial correct-by-construction transformation of the given program and the pre-condition*, whenever feasible, to simplify the inductive step of reasoning, (iii) we *strengthen the pre- and post-condition simultaneously during the inductive step* using the auxiliary inductive predicates obtained by employing Dijkstra's weakest pre-condition computation, (iv) we *recursively apply* the technique to prove the inductive step and most importantly (v) we *do not require explicit or implicit loop-specific inductive invariants* to be provided by the user or generated by a solver (viz. by constrained Horn clause solvers [KBGM15, GSV18, FPMG19] or recurrence solvers [RL18, HHKR10]). The combination of these factors often reduces reasoning about a program with multiple loops to reasoning about one with fewer (sometimes even none) and "simpler" loops, thereby simplifying proof goals. In this chapter, we demonstrate this, focusing on programs with sequentially composed, but non-nested loops.

5.1.1 Motivating Example

Fig. 5.1 shows an example of one such Hoare triple, where the pre- and post-conditions are specified using `assume` and `assert` statements. This triple effectively verifies the formula $\sum_{j=0}^{i-1}\left(1+\sum_{k=0}^{j-1} 6\cdot(k+1)\right) = i^3$ for all $i \in \{0 \ldots N-1\}$, and for all $N > 0$. Although each loop in Fig. 5.1 is simple, their sequential composition makes it difficult even for state-of-the-art tools like VIAP [RL18], VeriAbs [ACC+20], FreqHorn [FPMG19], Tiler [CGU17], Vaphor [MG16], or Booster [AGS14] to prove the post-condition correct. In fact, none of the above tools succeed in automatically proving the quantified post-condition in Fig. 5.1. In contrast, our technique *full-program induction* proves the post-condition in Fig. 5.1 correct within a few seconds.

Full-program induction reduces checking the validity of the Hoare triple in Fig. 5.1 to checking the validity of two "simpler" Hoare triples, represented in Figs. 5.2 and 5.3. The base case of our inductive reasoning is shown in Fig. 5.2, where every loop in the program is statically unrolled a fixed number of times after instantiating the program parameter N to a small constant value (here $N = 1$). As the induction hypothesis, we assume that the Hoare triple $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$ holds for values of $N > 1$. Note that this assumption does not relate to a specific loop in the program, but to the entire program P_N. For the motivating example, the induction hypothesis states that

```
// assume(true)

1. for (int t1=0; t1<N; t1=t1+1) {
2.   if (t1==0) { A[t1] = 6; }
3.   else { A[t1] = A[t1-1] + 6; }
4. }

5. for (int t2=0; t2<N; t2=t2+1) {
6.   if (t2==0) { B[t2] = 1; }
7.   else { B[t2] = B[t2-1] + A[t2-1]; }
8. }

9. for (int t3=0; t3<N; t3=t3+1) {
10.  if (t3==0) { C[t3] = 0; }
11.  else { C[t3] = C[t3-1] + B[t3-1]; }
12.}

// assert(∀i ∈ [0,N) C[i] = i³)
```

Figure 5.1: Original Hoare Triple

the entire Hoare triple in Fig. 5.1, after substituting N with $N-1$, holds. Notice that the induction hypothesis is on the entire program including all three loops and not on individual loops. The inductive step of the reasoning shown in Fig. 5.3 proves the post-condition, by automatically generating the computation to be performed after the program with parameter $N-1$ has executed and strengthening the pre- and post-conditions using auxiliary predicates. Note that all the program statements in Fig. 5.3 have syntactic counterparts in Fig. 5.1, but this may not be the case in general (we demonstrate this using an example in next chapter). We conceptualize the computation in the inductive step using the notions of *difference program* and *difference pre-condition* in the later sections. Effectively, we reasoned about three sequentially composed loops in Fig. 5.1 together, without the need for any implicitly or explicitly specified loop invariants. We defer a discussion of how our technique computes these Hoare triples and how auxiliary predicates

```
// assume(true)

1.  A[0] = 6;
2.  B[0] = 1;
3.  C[0] = 0;

// assert((C[0] = 0³) and
//        (B[0] = 1³ - 0³) and
//        (A[0] = 2³ - 2×1³ + 0³))
```

Figure 5.2: Base-case Hoare Triple

```
// assume(
//  (N > 1) ∧ (C_Nm1[N-2] = (N-2)³) ∧
//  (B_Nm1[N-2] = (N-1)³ - (N-2)³) ∧
//  (A_Nm1[N-2] = N³ - 2×(N-1)³ + (N-2)³)
// )

1.  A[N-1] = A_Nm1[N-2] + 6;
2.  B[N-1] = B_Nm1[N-2] + A_Nm1[N-2];
3.  C[N-1] = C_Nm1[N-2] + B_Nm1[N-2];

// assert(
//  (C[N-1] = (N-1)³) ∧
//  (B[N-1] = N³ - (N-1)³) ∧
//  (A[N-1] = (N+1)³ - 2×N³ + (N-1)³)
// )
```

Figure 5.3: Inductive Step Hoare Triple

are generated to iteratively strengthen the pre- and post-conditions to Section 5.6, where we present the details of our algorithms.

We mention a few important things here to highlight the simplifications illustrated by the Hoare triples in Figs. 5.2 and 5.3 that resulted from the application of the full-

program induction technique on the problem in Fig. 5.1. First, the programs in Figs. 5.2 and 5.3 are loop-free. Second, their pre- and post-conditions are quantifier-free. Third, the validity of these Hoare triples (Figs. 5.2 and 5.3) can be easily proved, e.g. by bounded model checking [CBRZ01] with a back-end SMT solver like Z3 [MB08]. Fourth, the value computed in each iteration of each loop in Fig. 5.1 is data-dependent on previous iterations of the respective loops as well as on the value computed in previous loops. Even though none of these loops can be trivially translated to a set of parallel assignments, our method still succeeds in automating the inductive step of the analysis. Last, we did not require any specialized constraint solving techniques like recurrence solving, theory of uninterpreted functions or constrained Horn clause solving to verify these Hoare triples, thus making our technique orthogonal to these approaches when proving properties of array programs.

5.1.2 Beyond Loop-Invariant based Proofs

Techniques based on synthesis and use of loop invariants are popularly used to reason about programs with loops. These techniques have been successfully applied to verify different classes of array-manipulating programs, viz. [GRS05, HP08, LR15, CCL11, GMT08, SG09, BHMR07, JM07, FPMG19]. If we were to prove the assertion in Fig. 5.1 using such techniques, it would be necessary to use appropriate loop-specific invariants for each of the three loops in Fig. 5.1. The weakest loop invariants needed to prove the post-condition in this example are: $\forall i \in [0, t1)\ (A[i] = 6i + 6)$ for the first loop (lines 1-4), $\forall j \in [0, t2)\ (B[j] = 3j^2 + 3j + 1) \wedge (A[j] = 6j + 6)$ for the second loop (lines 5-8), and $\forall k \in [0, t3)\ (C[k] = k^3) \wedge (B[k] = 3k^2 + 3k + 1)$ for the third loop (lines 9-12). Notice that these invariants are quantified and have non-linear terms. Unfortunately, automatically deriving such quantified non-linear inductive invariants for each loop is far from trivial. Template-based invariant generators, viz. [FL01, EPG+07], are among the best-performers when generating such complex invariants. However, their abilities are fundamentally limited by the set of templates from which they choose. We therefore choose not to depend on inductive loop-invariants at all in our work. Instead, we make use of inductive pre- and post-conditions – a notion that is related to, yet significantly different from loop-specific invariants. Specifically, inductive pre- and post-conditions are computed for the entire program, possibly consisting of multiple loops, instead of for each loop in the program.

As is clear from the discussion above, the primary difference between a proof generated by an invariant synthesis technique and the proof generated by our method is that we no longer need loop-specific safe inductive invariants. Instead, we generate and verify the Hoare triples shown in Figs. 5.2 and 5.3 considering the entire program P_N in Fig. 5.1. Automatically generating these Hoare triples in some cases may be more difficult than automatically generating inductive invariants for each loop and vice versa. However, as demonstrated by the motivating example, there are several complex programs, where it may be easier to generate these Hoare triples than compute safe inductive invariants for individual loops. It is a considerable challenge for verification techniques to be able to automatically generate these invariants and to the best of our knowledge none of the current state-of-the-art techniques do so.

The main contributions of the chapter can be summarized as follows:

- We introduce a novel verification technique called *full-program induction* for reasoning about assertions in programs with loops that manipulate arrays of parametric size. Full-program induction does not need loop-specific invariants in order to prove assertions, even when the program contains multiple sequentially composed loops.

- We propose the notions of *difference program* and *difference pre-condition* that relates programs and properties with different parameters. These are crucially used to enable the inductive step of the analysis.

- We describe practical algorithms for performing full-program induction. We present algorithms for computing difference programs that are used to perform the inductive step of the analysis. We present simple program transformations that allow presenting the computation of difference programs in a palatable manner. We present algorithms for handling a sub-class of quantified as well as quantifier-free pre-conditions and post-conditions in our inductive verification technique.

- We present rigorous proofs of correctness for the described algorithms. We demonstrate each algorithm in detail using examples.

The remainder of the chapter is structured as follows. In Section 5.2, we give a formal overview of the full-program induction technique. In Section 5.3, we present various analysis and transformations that generate necessary information used for computing the

difference program. Section 5.4 presents the algorithm for computing the difference program and Section 5.5 presents the algorithm for computing the difference pre-condition. In Section 5.6, we present the algorithms for full-program induction, prove their correctness and demonstrate the algorithms using examples. Finally, Section 5.7 concludes.

5.2 Overview of Full-Program Induction

We now elaborate on the core idea behind the *full-program induction* technique. Our goal is to check the validity of the parameterized Hoare triple $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ for all $N > 0$. A visual representation of this Hoare triple is shown in Fig. 5.4, where the clouds represent (possibly quantified) formulas and boxes represent programs/code fragments.

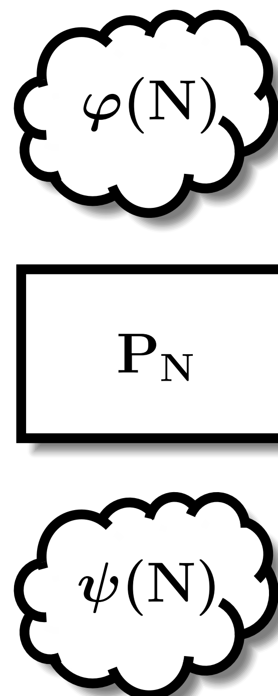


Figure 5.4: Goal

Intuitively, at a conceptual level, our approach works like any other inductive reasoning technique. However, the induction is over the entire program, via the parameter N, and not on the individual loops in the program.

We first check the base case, where we verify that the parameterized Hoare triple holds for some small values of N, say $0 < N \leq M$. We rely on an important, yet reasonable, assumption that can be stated as follows: *For every value of N (> 0), every loop in P_N can be statically unrolled a number (say $f(N)$) of times that depends only on N, to yield a loop-free program $\widehat{\mathsf{P}_N}$ that is semantically equivalent to P_N.* Note that this does not imply that reasoning about loops can be translated into loop-free reasoning. In general, $f(N)$ is a non-constant function, and hence, the number of unrollings of loops in P_N may strongly depend on N. In our experience, loops in a vast majority of array-manipulating programs (including Fig. 5.1 and all our benchmarks) satisfy the above assumption. Consequently, the base case of our induction reduces to checking a Hoare triple for a loop-free program. Checking a Hoare triple for a loop-free program is easily achieved by compiling the pre-condition, program and post-condition into an SMT formula, whose (un)satisfiability can be checked with an off-the-shelf SMT solver.

Next we hypothesize that $\{\varphi(N-1)\}$ P_{N-1} $\{\psi(N-1)\}$ holds for some $N > M$, visually depicted in Fig. 5.5. A few things are worth mentioning here. First, the entire Hoare triple is assumed not just the formula in the post-condition. Second, the assumption is not on a specific loop in the program, but the entire program P_N. Third, the change in the parameter from N to $N-1$ is uniform across the entire Hoare triple and not on a specific part thereof. We then try to show that the induction hypothesis implies $\{\varphi(N)\}$ P_N $\{\psi(N)\}$. While this sounds simple in principle, there are several technical difficulties en-route. Our contribution lies in overcoming these difficulties algorithmically for a large class of programs and assertions, thereby making *full-program induction* a viable and competitive technique for proving properties of array- manipulating programs.

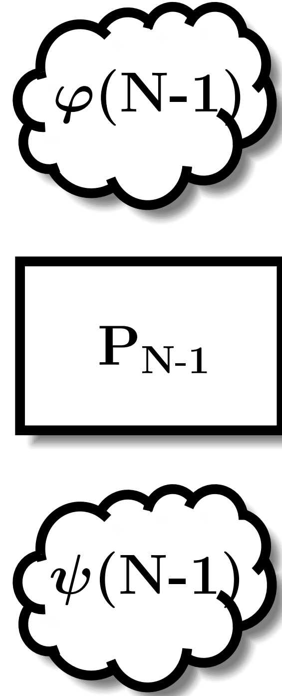


Figure 5.5: Hypothesis

The inductive step is the most complex one, and is the focus of the rest of the chapter. Recall that the inductive hypothesis asserts that $\{\varphi(N-1)\}$ P_{N-1} $\{\psi(N-1)\}$ is valid. To make use of this hypothesis in the inductive step, we must relate the validity of $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ to that of $\{\varphi(N-1)\}$ P_{N-1} $\{\psi(N-1)\}$. We propose doing this, whenever possible, via two key notions – that of "difference" program and "difference" pre-condition. Given a parameterized program P_N, intuitively the "difference" program $\partial\mathsf{P}_N$ is one such that $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ holds iff $\{\varphi(N)\}$ P_{N-1}; $\partial\mathsf{P}_N$ $\{\psi(N)\}$ holds, where ";" denotes sequential composition. Refer to Fig. 5.6 for a visual representation of the Hoare

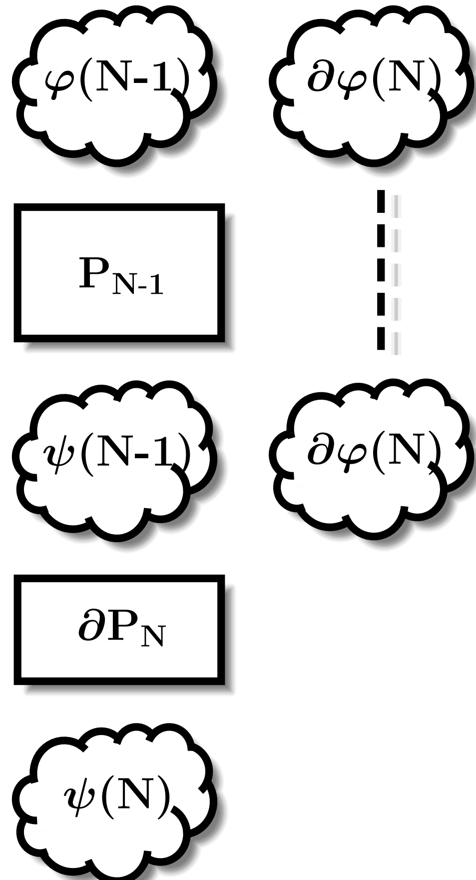


Figure 5.6: Transformations

triple after the decomposition of P_N into P_{N-1} and $\partial\mathsf{P}_N$. We will use this interpretation of a "difference" program in the subsequent parts of this chapter.

A simple way of ensuring the correctness of this transformation is by having a difference program $\partial\mathsf{P}_N$ such that the sequential composition $\mathsf{P}_{N-1};\ \partial\mathsf{P}_N$ is semantically equivalent to P_N. We define the precise notion of semantic equivalence as follows.

Definition 5.1 *Programs* P *and* Q *are said to be semantically equivalent if forall formulas* φ *and* ψ*, whenever* $\{\ \varphi\ \}\ \mathsf{P}\ \{\ \psi\ \}$ *holds, then* $\{\ \varphi\ \}\ \mathsf{Q}\ \{\ \psi\ \}$ *holds as well.*

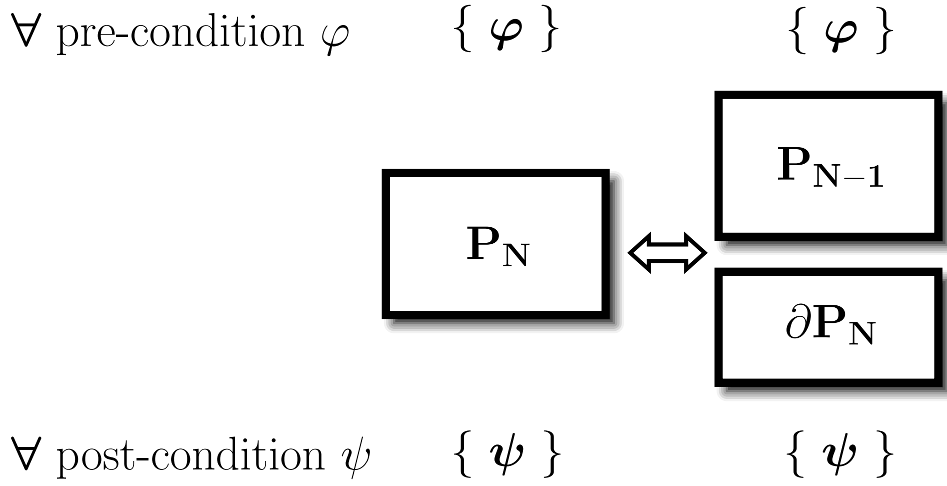


Figure 5.7: Decomposition of P_N and Semantic Equivalence

Decomposition of P_N into the program P_{N-1} and $\partial\mathsf{P}_N$ while ensuring semantic equivalence is visually depicted in Fig. 5.7. This decomposition ensures that both P_N and $\mathsf{P}_{N-1};\ \partial\mathsf{P}_N$ reach the same program state upon termination. In general, semantic equivalence of the decomposition for all possible pre- and post-condition formulas is a strong condition and is referred here only for an intuitive demonstration of its soundness. If the given parametric post-condition $\psi(N)$ is not impacted by the entire program state then the semantic equivalence alluded to above is not required to ensure correctness of our technique. For the construction of the "difference" program, we only need to ensure that the Hoare semantics of the given program P_N is preserved under the given pre-condition $\varphi(N)$ and the post-condition $\psi(N)$. Throughout the thesis, whenever we refer to semantic equivalence, we mean the preservation of Hoare semantics of the program under the given pre- and post-condition formulas.

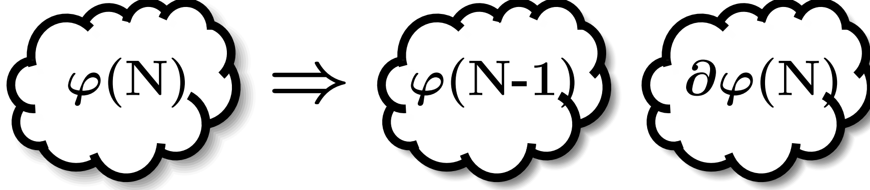


Figure 5.8: Difference Pre-condition

The "difference" pre-condition $\partial\varphi(N)$ is a formula such that the following conditions hold.

1. $\varphi(N) \Rightarrow (\varphi(N-1) \odot \partial\varphi(N))$, where the Boolean operator $\odot$ is $\wedge$ when $\varphi(N)$ is a universally quantified formula and it is $\vee$ when $\varphi(N)$ is a existentially quantified formula. We depict this decomposition of $\varphi(N)$ into $\varphi(N-1)$ and $\partial\varphi(N)$ in Fig. 5.8.

2. The execution of P_{N-1} does not affect the truth of $\partial\varphi(N)$. This can be visualized using Fig. 5.6, where the dashed line indicates the propagation of the difference pre-condition across P_{N-1} when it is not affected.

Computing the "difference" program $\partial\mathsf{P}_N$ and the "difference" pre-condition $\partial\varphi(N)$ is not easy in general. In Section 5.6, we discuss ways to overcome these problems and challenges. Assuming we have $\partial\mathsf{P}_N$ and $\partial\varphi(N)$ with the properties stated above, the proof obligation $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ can now be reduced to proving the Hoare triples $\{\varphi(N-1)\}$ P_{N-1} $\{\psi(N-1)\}$ and $\{\psi(N-1) \wedge \partial\varphi(N)\}$ $\partial\mathsf{P}_N$ $\{\psi(N)\}$ in the inductive step. Both these Hoare triples can be easily visualized from Fig. 5.6. The first Hoare triple follows from the inductive hypothesis (Fig. 5.5), and hence, is available for free. Thus, the inductive step is reduced to proving the second Hoare triple as shown in the Fig. 5.9.

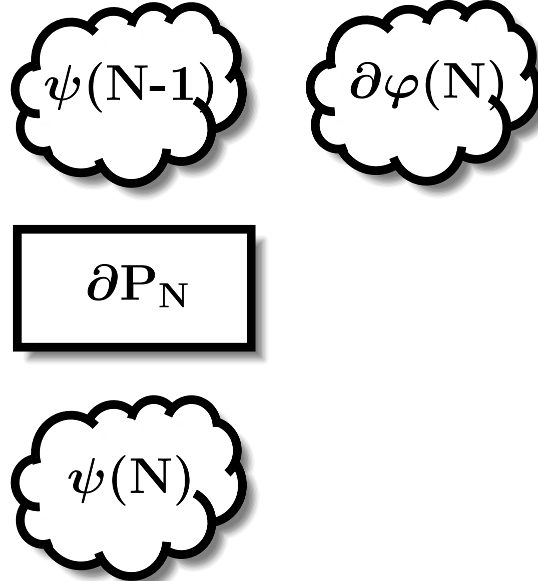


Figure 5.9: Inductive Step

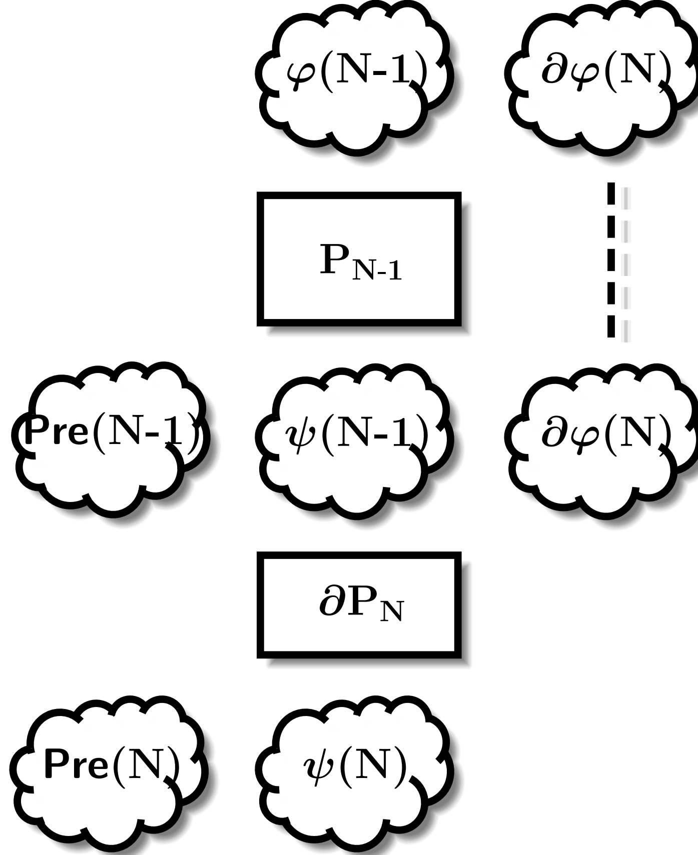


Figure 5.10: Strengthening Pre- and Post-conditions

Proving the inductive step may require strengthening the pre-condition, say by a formula $\mathsf{Pre}(N-1)$, in general. Since we are in the inductive step of mathematical induction, we formulate the new proof sub-goal in such a case as $\{(\psi(N-1) \wedge \mathsf{Pre}(N-1)) \wedge \partial\varphi(N)\}$ $\partial\mathsf{P}_N$ $\{\psi(N) \wedge \mathsf{Pre}(N)\}$. While this is somewhat reminiscent of loop invariants, observe that $\mathsf{Pre}(N)$ is *not* really a loop-specific invariant. Instead, it is analogous to computing an invariant for the entire program, possibly containing multiple loops. Specifically, the above process strengthens both the pre- and post-condition of $\{\psi(N-1) \wedge \partial\varphi(N)\}$ $\partial\mathsf{P}_N$ $\{\psi(N)\}$ simultaneously using $\mathsf{Pre}(N-1)$ and $\mathsf{Pre}(N)$, respectively. Fig. 5.10 shows the Hoare triple after the strengthening step. The strengthened post-condition of the resulting Hoare triple may, in turn, require a new pre-condition $\mathsf{Pre}'(N-1)$ to be satisfied. This process of strengthening the pre- and post-conditions of the Hoare triple involving $\partial\mathsf{P}_N$ can be iterated until a fix-point is reached, i.e. no further pre-conditions are needed for the parameterized Hoare triple to hold. While the fix-point was quickly reached for all benchmarks we experimented with, we also discuss how to handle cases where the above process may not converge easily. Note that since we effectively strengthen the pre-

condition of the Hoare triple in the inductive step, for the overall induction to go through, it is also necessary to check that the strengthened assertions hold at the end of each base case check. Automatically computing $\mathsf{Pre}(N)$ to strengthen the pre- and post-conditions of the Hoare triple may not always be straightforward, especially when the difference program $\partial \mathsf{P}_N$ has loops. In such cases, we recursively apply our technique on the Hoare triple $\{\psi(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{\psi(N)\}$ generated during the inductive step. This helps our technique converge when the generated difference program has one or more loops. We check if the recursive invocation of our technique will yield beneficial results using a progress measure influenced by several characteristics of the difference program.

The technique outlined above is called *full-program induction*, and the following theorem is the basis for the soundness of *full-program induction*.

Theorem 5.1 *Given* $\{\varphi(N)\}\ \ \mathsf{P}_N\ \ \{\psi(N)\}$*, suppose the following are true:*

1. *For* $N > 1$*,* $\{\varphi(N)\}\ \ \mathsf{P}_{N-1};\ \partial\mathsf{P}_N\ \ \{\psi(N)\}$ *holds iff* $\{\varphi(N)\}\ \ \mathsf{P}_N\ \ \{\psi(N)\}$ *holds.*
2. *For* $N > 1$*, there exists a formula* $\partial\varphi(N)$ *such that*
   - *(a)* $\partial\varphi(N)$ *does not refer to any program variable or array element modified in* P_{N-1}*, and*
   - *(b)* $\varphi(N) \Rightarrow \varphi(N-1)\ \wedge\ \partial\varphi(N)$*.*
3. *There exists an integer* $M \geq 1$ *and a parameterized formula* $\mathsf{Pre}(M)$ *such that*
   - *(a)* $\{\varphi(N)\}\ \ \mathsf{P}_N\ \ \{\psi(N)\}$ *holds for* $0 < N \leq M$*,*
   - *(b)* $\{\varphi(M)\}\ \ \mathsf{P}_M\ \ \{\psi(M) \wedge \mathsf{Pre}(M)\}$ *holds, and*
   - *(c)* $\{\psi(N-1) \wedge \mathsf{Pre}(N-1) \wedge \partial\varphi(N)\}\ \ \partial\mathsf{P}_N\ \ \{\psi(N) \wedge \mathsf{Pre}(N)\}$ *holds for* $N > M$*.*

Then $\{\varphi_N\}\ \ \mathsf{P}_N\ \ \{\psi_N\}$ *holds for all* $N \geq 1$*.*

Proof. For $0 < N \leq M$, condition 3(a) (the base case) ensures that $\{\varphi(N)\}\ \ \mathsf{P}_N\ \ \{\psi(N)\}$ holds. For $N > M$, note that by virtue of conditions 1 and 2(b), $\{\varphi(N)\}\ \ \mathsf{P}_N\ \ \{\psi(N)\}$ holds if $\{\varphi(N-1)\wedge\partial\varphi(N)\}\ \ \mathsf{P}_{N-1};\ \partial\mathsf{P}_N\ \ \{\psi(N)\wedge\mathsf{Pre}(N)\}$ holds. With $\psi(N-1)\wedge\mathsf{Pre}(N-1)$ as a mid-condition, and by virtue of condition 2(a), the latter Hoare triple holds for $N > M$ if $\{\varphi(M)\}\ \ \mathsf{P}_M\ \ \{\psi(M) \wedge \mathsf{Pre}(M)\}$ holds and $\{\psi(N-1) \wedge \mathsf{Pre}(N-1) \wedge \partial\varphi(N)\}$ $\partial\mathsf{P}_N\ \ \{\psi(N) \wedge \mathsf{Pre}(N)\}$ holds for all $N > M$. Both these triples are seen to hold by virtue of conditions 3(b) and (c). $\square$

```
// assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N; i++) {
3.    S = S + A[i];
4.  }

5.  for(i=0; i<N; i++) {
6.    A[i] = A[i] + S;
7.  }

8.  for(i=0; i<N; i++) {
9.    S = S + A[i];
10. }

// assert(S = N × (N+2))
```

Figure 5.11: Running Example

5.3 Input Programs, Pre-processing and Analysis

While the Hoare triple with the simple program with pre- and post-conditions shown in Fig. 5.1 motivates the need for our technique, it may not suffice to illustrate more nuanced features of our technique. We present a Hoare triple that acts as a running example for illustrating the ideas in this section.

Example 5.1 The program in Fig. 5.11 is generated using the grammar shown in Fig. 3.2. It updates a scalar variable `S` and an array variable `A`. The first loop adds the value of each element in array `A` to variable `S`. The second loop adds the value of `S` to each element of `A`. The last loop aggregates the updated content of `A` in `S`. The pre-condition $\varphi(N)$ is a universally quantified formula on array `A` stating that each element has the value 1. We need to establish the post-condition $\psi(N)$, which is a predicate on `S` and `N`. Note that the post-condition has non-linear terms and is quite challenging to prove. We will use the Hoare triple in Fig. 5.11 as the example of choice to illustrate the important aspects of

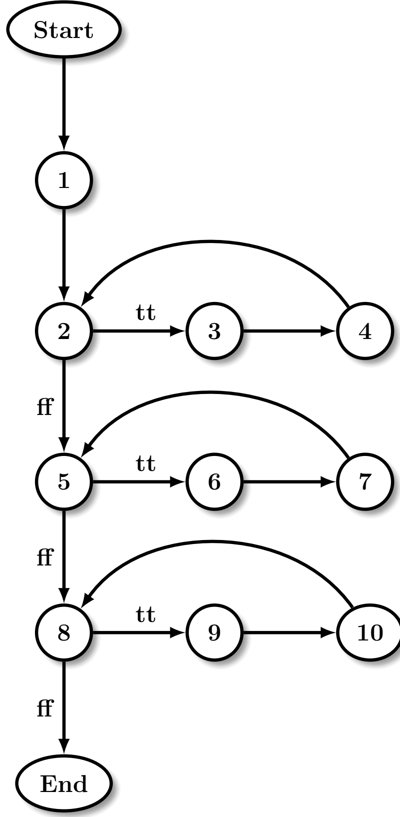


Figure 5.12: CFG of Fig. 5.11

our verification algorithm. □

Recall from Section 3.2 that we represent a program P_N using its *control flow graph* (or CFG) $G_C = (Locs, CE, \mu)$, where *Locs* denotes the set of control locations (nodes), CE are the control-flow edges, and μ annotates every node in *Locs* with an assignment statement, or a Boolean expression.

Example 5.2 The CFG of the program in Fig. 5.11 is shown in Fig. 5.12. The nodes are numbered such that they coincide with the line numbers in the program. The graph has three cycles each corresponding to a loop in the given program. $\{(1,2)$, $(2,5)$, $(5,8)\}$ are *incoming-edges*, $\{(4,2)$, $(7,5)$, $(10,8)\}$ are *back-edges* and $\{(2,5)$, $(5,8)$, $(8,E)\}$ are *exit-edges*.

Recall that $def(n)$ and $uses(n)$ denote the set of scalars and arrays defined and used at node n, respectively. At nodes $[n=]1$, 3, and 9, $def(n) = \{\texttt{S}\}$, at node 6, $def(n) = \{\texttt{A}\}$, and at nodes 3, 6, and 9, $uses(n) = \{\texttt{S, A}\}$. Since there are no uses of scalars or arrays at node 1, $uses(n)$ is an empty set. The index of `A` updated at 6 is $defIndex(\texttt{A},n)$ $= \texttt{i}$, and the set of indices of array `A` used at nodes 3, 6, and 9 is $useIndex(\texttt{A},n) = \{\texttt{i}\}$.

The *Start* node does not strictly post-dominate any other node. Node 1 strictly post-dominates *Start* and is its immediate post-dominator. Node 2 strictly post-dominates nodes 1, 4 and *Start*. Node 2 is an immediate post-dominator of 1 and 4. Node 8 strictly post-dominates all nodes except itself and *End*. Node 8 is an immediate post-dominator of 5 and 10. *End* node strictly post-dominates all nodes except itself. Similarly, the post-domination relations for other nodes can be computed. □

Prior to describing the algorithm for generating the difference program and its adaptation in the full-program induction algorithm, we present analyses and transformations that are a precursor to these algorithms.

Computing the *difference program* $\partial\mathsf{P}_N$ is a non-trivial endeavor. Fig. 5.13 presents a high level overview of the sequence of steps involved in the generation of a difference program. In the first step, we carefully rename the variables and arrays such that each loop in the renamed program refers to its own copy of variables/arrays. Note that this is similar in spirit to SSA renaming, although there are important differences that will become clear in Section 5.3.1. In the next step, we peel the last (in some cases the last few) iteration(s) of each loop in the program such that the remaining part of each loop in the peeled version of P_N iterates exactly the same number of times as the corresponding loop in P_{N-1}. We use the term *peel* to denote the last (or last few as the case may be) iteration(s) of a loop that have been removed from the loop. The motivation for this peeling is that the difference program can often be constructed by moving the peels of individual loops to the end of the program and stitching them up in appropriate ways, as will be discussed in detail in Section 5.4 and further generalized in Chapter 6. In order to ensure that the semantics of the program P_N is preserved even after moving the peels to the end of the program, we need to do a careful analysis of the data dependencies between between variables and array elements updated/read in statements within loops and those updated/read in the peeled iterations. This is achieved, in the third step, by computing a customized data dependence graph, details of which are presented in Section 5.3.3. In general, variables and array elements in the program P_N can have data/control dependencies on the parameter N beyond those attributable to the iteration counts of loops being possibly determined by N. We call such variables/array elements as "affected" by N and identify them, in the fourth step, using a special data-flow analysis and the data dependencies computed above. Details of this analysis are presented in Section

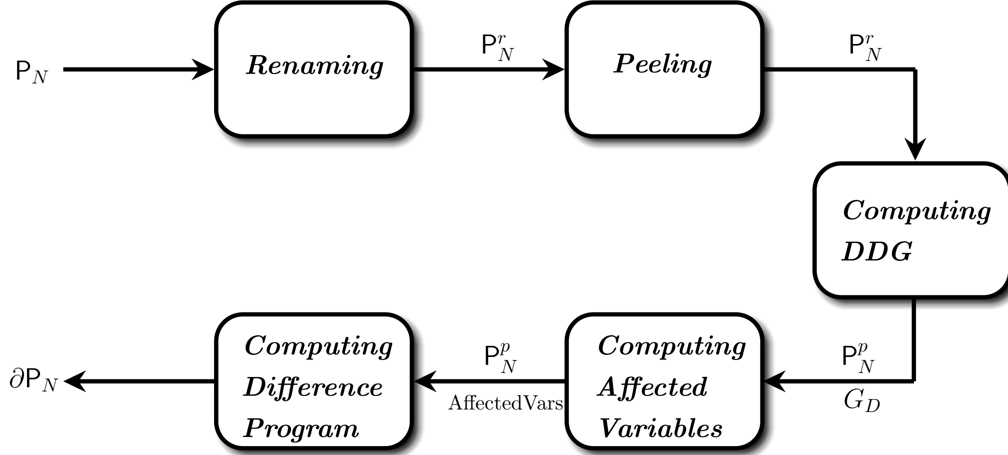


Figure 5.13: Sequence of Steps for Computing the Difference Program $\partial\mathsf{P}_N$.

5.3.4. Finally, in the last step, use the information about data dependencies and affected variables to compute the difference program $\partial\mathsf{P}_N$.

5.3.1 Renaming Variables and Arrays

Recall that our proposed approach requires us to construct a difference program $\partial\mathsf{P}_N$ such that $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ holds iff $\{\varphi(N)\}\ \mathsf{P}_{N-1};\ \partial\mathsf{P}_N\ \{\psi(N)\}$ holds (condition 1 of Theorem 5.1). A natural (though not necessary) way to do this is to construct $\partial\mathsf{P}_N$ such that both P_N and $\mathsf{P}_{N-1};\ \partial\mathsf{P}_N$ modify all relevant scalar variables and arrays in exactly the same way. Note, however, that P_N may update the same scalar variable or array in multiple sequentially composed loops. Therefore, when P_{N-1} terminates and $\partial\mathsf{P}_N$ starts executing (in $\mathsf{P}_{N-1};\ \partial\mathsf{P}_N$), we may no longer have access to the values of scalar variables and arrays that resulted after individual loops in P_{N-1} terminated. In general, this makes it difficult to construct $\partial\mathsf{P}_N$ compositionally from the peels of individual loops while ensuring that $\mathsf{P}_{N-1};\ \partial\mathsf{P}_N$ has the same effect as P_N on all relevant scalar variables and arrays. To circumvent this problem, we propose to pre-process P_N such that each loop in P_N updates its own "private" copy of scalar variables and arrays. We add *glue code* to copy the values of these scalar variables and arrays after one loop ends and before the next one begins. We also rename the variables/arrays referred in the post-condition $\psi(N)$ to their versions corresponding to the last loop in the program. As we show later, this eases the construction of $\partial\mathsf{P}_N$, and also helps in inductive strengthening of the pre- and post-conditions.

It is important to note here that static single assignment (SSA) [RWZ88] is a well-

known technique for renaming scalar variables such that a variable is updated at most once in a program. Similarly, array SSA renaming has been studied earlier in the context of compilers to achieve similar goals [KS98]. Unlike SSA renaming, we do not have the stringent requirement of a single update in the whole program. For our method to function successfully, we only require each loop to update its own copy of a scalar/array variable. We note that these well-studied techniques can be easily adapted for our purposes.

In the following discussion, we often need to refer to scalar variables and arrays in a uniform way. For notational clarity, we use vA (as opposed to v for a scalar variable and A for an array) as a combined symbolic name to refer to a scalar variable or array, depending on the context. Function RENAME, presented in Algorithm 3, performs the renaming task. In this function, we first create a copy of the CFG of program P_N. Let this copy be denoted P^r_N (line 1). Next, we transform the CFG of P^r_N by collapsing all nodes and edges in the body of each loop into a single node identified with the loop-head. The function COLLAPSELOOPBODY does this transformation in line 2. This is done for ease of presenting the core idea underlying our renaming strategy. The transformation allows us to view a loop in P_N as a single collapsed control flow node in the CFG. Recall that our grammar disallows nesting of loops. Therefore, after the invocation of COLLAPSELOOPBODY, the resulting CFG is a finite directed acyclic graph (DAG). This DAG has finitely many paths, and along each such path, there is a total ordering of all collapsed loops appearing along the path.

Let the *start* and *end* nodes of the CFG of P^r_N be denoted n_{start} and n_{end} respectively. We begin the renaming transformation with the start node n_{start}. We rename each scalar/array vA to $vA^{n_{start}}$ in $\mu^r(n_{start})$ at node n_{start} on line 3. We call $vA^{n_{start}}$ the version of vA corresponding to n_{start}; the notation for versions of vA corresponding to other nodes in the CFG is similar. Since the pre-condition refers to the same variables/arrays as in the start node, we rename each variable/array vA referred in the pre-condition $\varphi(N)$ to $vA^{n_{start}}$. Let the resulting renamed pre-condition be denoted $\varphi^r(N)$ (line 4).

We add n_{start} to the worklist at line 6 and begin the traversal of the collapsed CFG. The while loop in lines 7–18 performs a breadth-first top-down traversal of the DAG representing the CFG, starting with n_{start}. The function processes one node at a time from the worklist. For node n, we use the sub-routine SUCC for obtaining a list of its successors. The loop in lines 9–18 processes each successor n' of n at a time. We rename

Algorithm 3 RENAME(($Locs, CE, \mu$): program P_N, $\varphi(N)$: pre-condition, $\psi(N)$: post-condition)

1: Let P_N^r denote $(Locs^r, CE^r, \mu^r)$, where $Locs^r := Locs$, $CE^r := CE$, and $\mu^r := \mu$;
2: $\mathsf{P}_N^r :=$ COLLAPSELOOPBODY(P_N^r);
▶ $Locs^r, CE^r$ `and` μ^r `are defined in the natural way in the resulting CFG of` P_N^r `after collapsing loops. For every (collapsed) loop-head node` n, $\mu^r(n)$ `gives the entire collapsed loop.`
3: Rename each scalar/array vA that appears in P_N to $vA^{n_{start}}$ in $\mu^r(n_{start})$;
4: $\varphi^r(N) := \varphi(N)$ after renaming each scalar/array vA to $vA^{n_{start}}$;
5: $\mathsf{GlueNodes} := \emptyset$;
6: $\mathsf{WorkList} := (n_{start})$; ▷ Add the start node to the worklist
7: **while** $\mathsf{WorkList}$ is not empty **do**
8: Remove a node n from head of $\mathsf{WorkList}$;
9: **for** each node $n' \in$ SUCC(n) **do** ▷ SUCC gives all the successors of a node
10: Rename each scalar/array vA to $vA^{n'}$ in $\mu^r(n')$;
11: $n_{glue} :=$ FRESHNODE(); ▷ Glue node to be inserted
12: $Locs^r := Locs^r \cup \{n_{glue}\}$;
13: $c :=$ Label of edge from n to n' in P_N^r;
14: $CE^r := CE^r \setminus \{(n, n', c)\}$;
15: $CE^r := CE^r \cup \{(n, n_{glue}, c), (n_{glue}, n', \mathsf{U})\}$;
▶ `With abuse of notation,` n_{glue} `is labeled by potentially multiple assignment statements that copy values between versions of scalar/array variables in line 16 below. Here,` vA `refers to scalars/arrays. Loops are introduced if needed (refer page 89) to copy array elements.`
16: $\mu^r(n_{glue}) := (vA^{n'} = vA^n)$;
17: $\mathsf{GlueNodes} := \mathsf{GlueNodes} \cup \{n_{glue}\}$;
18: $\mathsf{WorkList} :=$ APPENDTOLIST($\mathsf{WorkList}$, n');
19: $\psi^r(N) := \psi(N)$ with each scalar/array vA renamed to $vA^{n_{end}}$;
20: $\mathsf{P}_N^r :=$ UNCOLLAPSELOOPBODY(P_N^r);
21: **return** $\langle \mathsf{P}_N^r, \varphi^r(N), \psi^r(N), \mathsf{GlueNodes} \rangle$

each scalar/array vA to $vA^{n'}$ at node n' (line 10). Note that when n' is a loop-head, this amounts to renaming all scalars/arrays in the body of the loop as well. Subsequently, we create a fresh node, which we call a *glue node*, denoted n_{glue} (line 11). We insert n_{glue} between the nodes n and n' in lines 12–15. To ensure the correct flow of data, we add program statements in $\mu^r(n_{glue})$ to copy values of all scalars/arrays from their versions corresponding to n to their respective versions corresponding to n' (line 16). For every scalar variable v^n, this amounts to introducing a statement $v^{n'} = v^n$; at node n_{glue}. For every array A, we introduce the loop `for(i=0; i<f(N); i++) {` $A^{n'}$`[i] =` A^n`[i]; }` at n_{glue}, where `f(N)` denotes the size of array A in P_N . With slight abuse of notation, all program statements for effecting this copying operation are added to $\mu^r(n_{glue})$. Line 17 collects all the glue nodes n_{glue} in the set GlueNodes. At line 18, we append the worklist with the successors n' of the current node n. Nodes are added to the worklist only after they are processed by our renaming function. The loop continues until no further nodes are left to be processed.

To conclude the renaming process, we rename the variables/array referred to in the post-condition $\psi(N)$ to their versions corresponding to the end node n_{end}, resulting in the renamed post-condition $\psi^r(N)$ (line 19). Finally, we re-introduce the nodes and edges in the loop body of each loop in the program using the function UnCollapseLoopBody in line 20. The function returns the renamed program, the renamed pre- and post-conditions and the set of glue nodes at line 21.

The following lemmas state some important properties of function Rename.

Lemma 5.1 *Let n be a node in the collapsed CFG of P_N (and hence of P^r_N). In every execution of P^r_N in which control flows through n, no scalar variable or array renamed vA^n is updated after the execution exits node n.*

Proof. Since the collapsed CFG of P^r_N is acyclic, once control flow exits node n, it cannot come back to either n or to any node n' that has a control flow path to n. The proof now follows from the observation that renaming ensures that any scalar variable/array renamed vA^n can only be updated in glue nodes immediately leading to node n or in node n itself. □

For convenience of exposition, we will henceforth refer to the property formalized in Lemma 5.1 as the *"no-overwriting"* property of renamed programs. For a node n that

corresponds to a collapsed loop in the collapsed CFG of P_N^r, we abuse notation and use $\mu^r(n)$ to denote the entire loop (including the entire loop-body) corresponding to the loop-head n in the subsequent discussion.

Lemma 5.2 *$\{\varphi(N)\}$ P_N $\{\psi(N)\}$ holds iff $\{\varphi^r(N)\}$ P_N^r $\{\psi^r(N)\}$ holds.*

Proof. Consider the CFG of P_N^r obtained in line 2 of Algorithm 3, i.e. after invocation of CollapseLoopBody but before any glue node has been added. Since this CFG is a DAG, there are finitely many, say M, paths from the start node n_{start} to the end node n_{end} in this DAG. Let the set of all such paths be Π. For every path $\pi \in \Pi$, there is a finite number, say $\nu(\pi)$, of linearly ordered nodes along π. Let these nodes be called $n_{start} = n_{\pi,1}, n_{\pi,2}, \ldots, n_{\pi,\nu(\pi)} = n_{end}$, where $n_{\pi,s}$ is the predecessor of $n_{\pi,s+1}$ along π, for $1 \leq s < \nu(\pi)$.

Recall that at the end of line 2, we have $\mu^r(n) = \mu(n)$ for all CFG nodes n that are not loop-heads. For a loop-head n, however, the label $\mu^r(n)$ gives the entire collapsed loop, while $\mu(n)$ gives only the conditional statement in the original loop-head. For clarity of exposition, we abuse notation and use $\mu(n)$ for the labels of all nodes n (including collapsed loop-head nodes) in P_N^r at the end of line 2, when there is no confusion.

For an execution trace of the program P_N^r that starts from a state satisfying $\varphi(N)$ and that corresponds to the path $\pi \in \Pi$, let $\mathsf{Inv}_{\pi,s}$ be an invariant that holds after the statement(s) in $\mu(n_{\pi,s})$ has/have been executed, for every $s \in \{1, \ldots, \nu(\pi) - 1\}$. If, for some $i \in \{1, \ldots, \nu(\pi)\}$, no execution trace of P_N^r can reach the node $n_{\pi,i}$ along π, we assume that $\mathsf{Inv}_{\pi,i} = \mathbf{ff}$. It now follows that $\{\varphi(N)\}$ $\mu(n_{\pi,1});\ \cdots;\ \mu(n_{\pi,\nu(\pi)})$ $\{\psi(N)\}$ holds for the execution of P_N^r along $\pi \in \Pi$ iff the following Hoare triples hold:

- H1: $\{\varphi(N)\}\ \mu(n_{\pi,1})\ \{\mathsf{Inv}_{\pi,1}\}$
- H2: $\{\mathsf{Inv}_{\pi,s-1}\}\ \mu(n_{\pi,s})\ \{\mathsf{Inv}_{\pi,s}\}$ for all $s \in \{2, \ldots, \nu(\pi) - 1\}$
- H3: $\{\mathsf{Inv}_{\pi,\nu(\pi)-1}\}\ \mu(n_{\pi,\nu(\pi)})\ \{\psi(N)\}$

Therefore, $\{\varphi(N)\}$ P_N^r $\{\psi(N)\}$ holds iff H1, H2 and H3 hold for all $\pi \in \Pi$. However, P_N^r obtained at line 2 of Algorithm 3 is really the same as P_N, since the only change between the CFGs of P_N and P_N^r is that loops have been collapsed into their respective loop-heads in P_N^r, and the label of the loop-head has been set to the entire loop, including

the loop-body. It follows therefore that $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ holds iff H1, H2 and H3 hold for all $\pi \in \Pi$.

Let us now focus on the program P^r_N obtained after the while loop of lines 7–18 terminates. By virtue of the way glue nodes are inserted in lines 11–16 of Algorithm 3, no cycles are introduced in the CFG of P^r_N in any iteration of the while loop. Therefore, the CFG that started as a DAG in line 2 continues to remain a DAG on termination of the while loop. Let Π' denote the set of paths from n_{start} to n_{end} in the resulting DAG. For notational convenience, we use $g_{\pi,s}$ to denote the glue node inserted between nodes $n_{\pi,s}$ and $n_{\pi,s+1}$ in the original CFG. Clearly, for every path $\pi : (n_{start} = n_{\pi,1}, \ldots, n_{\pi,\nu(\pi)} = n_{end})$ in Π, there exists a path $\pi' : (n_{start} = n_{\pi,1}, g_{\pi,1}, \ldots, g_{\pi,\nu(\pi)-1}, n_{\pi,\nu(\pi)} = n_{end})$ in Π', and vice versa. This defines a natural bijection between paths in Π and those in Π'; it follows that $|\Pi'| = |\Pi|$.

Let $\mathsf{Inv}_{\pi,s} \Uparrow t$ denote the invariant $\mathsf{Inv}_{\pi,s}$ referred to earlier, but with all scalars/arrays renamed to their versions corresponding to $n_{\pi,t}$. The interpretations of $\varphi(N) \Uparrow t$ and $\psi(N) \Uparrow t$ are similar, when the path π is implicit from the context. Recall that $\varphi^r(N)$ and $\psi^r(N)$ denote the pre- and post-conditions obtained after renaming the n_{start} and n_{end} nodes at lines 4 and 19 respectively. From line 16, we know that $\mu^r(g_{\pi,s})$ copies the values of all scalars/arrays from their versions for $n_{\pi,s}$ to their corresponding versions for $n_{\pi,s+1}$. Therefore, $\{\mathsf{Inv}_{\pi,s} \Uparrow s\}$ $\mu^r(g_{\pi,s})$ $\{\mathsf{Inv}_{\pi,s} \Uparrow s+1\}$ holds for all $s \in \{1, \ldots, \nu(\pi) - 1\}$ and for all $\pi \in \Pi$.

It can now be seen that for each path $\pi' \in \Pi'$, if the corresponding path in Π is π, then $\{\varphi^r(N)\}$ $\mu^r(n_{\pi',1});\ \cdots;\ \mu^r(n_{\pi',\nu(\pi')})$ $\{\psi^r(N)\}$ holds iff $\{\varphi^r(N)\}$ $\mu^r(n_{\pi,1});\ \mu^r(g_{\pi,1});\ \cdots;$ $\mu^r(g_{\pi,\nu(\pi)-1});\ \mu^r(n_{\pi,\nu(\pi)})$ $\{\psi^r(N)\}$ holds. This, in turn, holds iff all of the following hold for the path π:

- H1′: $\{\varphi(N) \Uparrow 1\}$ $\mu^r(n_{\pi,1});\ \mu^r(g_{\pi,1})$ $\{\mathsf{Inv}_{\pi,1} \Uparrow 2\}$
- H2′: $\{\mathsf{Inv}_{\pi,s-1} \Uparrow s\}$ $\mu^r(n_{\pi,s});\ \mu^r(g_{\pi,s})$ $\{\mathsf{Inv}_{\pi,s} \Uparrow s+1\}$ for all $s \in \{2, \ldots, \nu(\pi) - 1\}$.
- H3′: $\{\mathsf{Inv}_{\pi,\nu(\pi)-1} \Uparrow \nu(\pi)\}$ $\mu^r(n_{\pi,\nu(\pi)})$ $\{\psi(N) \Uparrow \nu(\pi)\}$

Noting the correspondence between paths in Π and those in Π', we conclude that $\{\varphi^r(N)\}$ P^r_N $\{\psi^r(N)\}$ holds iff H1′, H2′ and H3′ hold for all $\pi \in \Pi$.

It is easy to see that these Hoare triples can be further decomposed. For example, H2′ gets decomposed into the Hoare triples $\{\mathsf{Inv}_{\pi,s-1} \Uparrow s\}$ $\mu^r(n_{\pi,s})$ $\{\mathsf{Inv}_{\pi,s} \Uparrow s\}$ and

$\{\mathsf{Inv}_{\pi,s} \Uparrow s\}\ \mu^r(g_{\pi,s})\ \{\mathsf{Inv}_{\pi,s} \Uparrow s+1\}$. After the decomposition of H2′, the first Hoare triple follows from H2 and the second Hoare triple is the one on the glue nodes as previously stated. Since renaming all variables in the pre-condition, post-condition and program fragment in a Hoare triple does not change the validity of the triple, conditions H1, H2 and H3 holding for all $\pi \in \Pi$ is equivalent to conditions H1′, H2′ and H3′ holding for all $\pi \in \Pi$. Therefore, $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ holds iff $\{\varphi^r(N)\}\ \mathsf{P}^r_N\ \{\psi^r(N)\}$ holds. □

While function Rename, as shown in Algorithm 3, gives the core idea behind our renaming strategy, there are significant optimizations that can be done to reduce the size and complexity of the renamed program. For example, Rename introduces a separate version of all scalar/array variables and inserts glue code for copying values between different versions of these variables, even when these variables are not updated by the CFG nodes under consideration. In such cases, the redundant versions of scalar/array variables and the corresponding copy statements in the glue code can be optimized away while still ensuring correct data flow. We end this subsection with an illustration of the program transformation achieved by applying the renaming strategy discussed above on our running example.

Example 5.3 Consider the program shown in Fig. 5.11. This program has multiple sequentially composed loops that update a scalar `S` and an array `A`. The transformed program after renaming the scalar and array using function Rename is shown in Fig. 5.14(a), where we have used simple names for the renamed versions of `S` and `A` (instead of using the naming convention in Algorithm 3) for ease of readability. Notice that the algorithm renames `S` and `A` in the second loop to `S1` and `A1` respectively, and renames the same variables in the third loop to `S2` and `A2` respectively. The function also adds glue code (lines 5-7 and 11-13), shown in blue in Fig. 5.14(a), to copy values from one version of the renamed scalar and array to another version of the same.

An optimized renaming of `S` and `A` for this example is shown in Fig. 5.14(b). This avoids creating new versions of `S` and `A` for statements and loops that do not update `S` and `A` respectively. Moreover, values are read directly from the version of `S` and `A` that reaches the access location. This helps in reducing the glue code required for renaming quite significantly. Specifically, we create a new version `A1` for array `A`. However, instead of adding glue code that copies the content of array `A` into `A1`, we directly update array `A1`

```
// assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N; i++) {
3.    S = S + A[i];
4.  }

// Lines 5-7 are glue code
5.  S1 = S;
6.  for(i=0; i<N; i++)
7.    A1[i] = A[i];
8.  for(i=0; i<N; i++) {
9.    A1[i] = A1[i] + S1;
10. }

// Lines 11-13 are glue code
11.  S2 = S1;
12.  for(i=0; i<N; i++)
13.    A2[i] = A1[i];
14.  for(i=0; i<N; i++) {
15.    S2 = S2 + A2[i];
16. }

// assert(S2 = N × (N+2))
```

(a)

```
// assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N; i++) {
3.    S = S + A[i];
4.  }

5.  for(i=0; i<N; i++) {
6.    A1[i] = A[i] + S;
7.  }

// Optimized glue code
8.  S1 = S;

9.  for(i=0; i<N; i++) {
10.   S1 = S1 + A1[i];
11. }

// assert(S1 = N × (N+2))
```

(b)

Figure 5.14: Renaming (a) Unoptimized and (b) Optimized

in the second loop and directly read values from array `A`. Similarly, we rename the variable `S` to `S1` in the last loop and add the glue statement `S1 = S;` before the loop. Note that we do not create a new version of `S` for the second loop, since this loop simply reads the value of S and does not update it. The variable `S` referred to in the post-condition is renamed to `S1` accordingly. □

Algorithm 4 PEELALLLOOPS($(Locs, CE, \mu)$: program P_N)

1: $\mathsf{P}_N^p := (Locs^p, CE^p, \mu^p)$, where $Locs^p = Locs$, $CE^p = CE$, $\mu^p = \mu$; ▷ P_N^p is a copy of P_N
2: $\mathsf{PeelNodes} := \varnothing$;
3: **for** each loop $L \in \text{LOOPS}(\mathsf{P}_N^p)$ **do**
4: Let $k_L(N)$ be the expression for iteration count of L in P_N^p;
5: $\mathsf{PeelCount} := \text{SIMPLIFY}(k_L(N) - k_L(N-1))$;
6: **if** PeelCount is non-constant **then**
7: **throw** "Failed to peel non-constant number of iterations";
8: $\langle \mathsf{P}_N^p, Locs' \rangle := \text{PEELSINGLELOOP}(\mathsf{P}_N^p, L, k_L(N-1), \mathsf{PeelCount})$;
▶ We assume availability of function PEELSINGLELOOP, for example, from a compiler framework like LLVM. It transforms loop L so that last PeelCount iterations of L are peeled. Updated CFG and newly created CFG nodes for the peeled iterations are returned.
9: $\mathsf{PeelNodes} := \mathsf{PeelNodes} \cup Locs'$;
10: **return** $\langle \mathsf{P}_N^p, \mathsf{PeelNodes} \rangle$;

5.3.2 Peeling the Loops

Recall from Section 5.2 that our induction strategy requires us to use $\mathsf{P}_{N-1}; \partial\mathsf{P}_N$ in place of P_N when proving the Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$. In general, the parameter N may determine the number of times each loop in P_N iterates (see, for example, Fig. 5.11). Therefore, the count of iterations of a loop in P_{N-1} may differ from the corresponding count in P_N. Relating P_N and P_{N-1} requires taking into account such differences of loop iterations. Towards this end, we transform P_N by *peeling* the last few iterations of each loop as needed, so that corresponding loops in P_{N-1} and the transformed P_N iterate the same number of times. This is done by function PEELALLLOOPS shown in Algorithm 4. The algorithm first makes a copy, viz. P_N^p, of the non-collapsed input CFG P_N. Let $\text{LOOPS}(\mathsf{P}_N^p)$ denote the set of loops of P_N^p, and let $k_L(N)$ and $k_L(N-1)$ denote the number of times loop L iterates in P_N^p and P_{N-1} respectively. The difference $k_L(N) - k_L(N-1)$, computed in line 5, gives the extra iteration count of loop L in P_N^p. If this difference is not a constant, we currently report a failure of our technique (line 7). For example, consider a

loop in P_N with the counter i initialized to 0 and the loop termination condition "$i < N^2$". The corresponding loop in P_{N-1} has the same initialization but the termination condition is "$i < (N-1)^2$". Thus, $k_L(N) = N^2$ and $k_L(N-1) = (N-1)^2$ and the difference of these iteration counts is "$2 \times N + 1$". Our technique is unable to handle such cases currently. Note that such cases cannot arise if the upper bounds of all loops in P_N are linear functions of N.

The routine PEELSINGLELOOP transforms loop L of P^p_N as follows: it replaces the termination condition $(\ell < k_L(N))$ of L by $(\ell < k_L(N-1))$. It also peels the last $(k_L(N) - k_L(N-1))$ iterations of L and adds control flow edges such that the peeled iterations are executed immediately after the loop body is iterated $k_L(N-1)$ times. Effectively, PEELSINGLELOOP peels the last $(k_L(N) - k_L(N-1))$ iterations of loop L in P^p_N. The transformed CFG is returned as the updated P^p_N in line 8. In addition, PEELSINGLELOOP also returns the set $Locs'$ of all CFG nodes newly added while peeling the loop L. We accumulate these newly added nodes for loops in the set PeelNodes in line 9. Henceforth, we call all nodes in PeelNodes as *peeled nodes*, all other nodes in the CFG as *non-peeled nodes*, and the CFG resulting from the invocation of PEELALLLOOPS as a *peeled program*. This function PEELALLLOOPS returns the peeled program P^p_N and the set of peeled nodes PeelNodes in line 10.

We now state several useful properties of peeled programs.

Lemma 5.3 *Let $n \in Locs^p$ be a node in the peel of loop L, and let n_h be the loop-head of loop L. For every $n' \in Locs^p$ that is not in the peel, if there is a control flow path in P^p_N from n' to n, the path necessarily passes through n_h.*

Proof. The proof follows from the observation that the peel of a loop L must necessarily execute after the loop L has itself executed $k_L(N-1)$ times. Hence, the sole predecessor of the first node in the peel must be the loop-head node n_h. It follows that every control flow path from n' to n where n' is not in the peel must pass through n_h. □

Peeling of loops can destroy the no-overwriting property (as mentioned in Lemma 5.1), since the same variable/array can get updated in a loop L and also in its peel. However, a weaker variant of the no-overwriting property continues to hold, as described below.

Lemma 5.4 *Let n be a node in the collapsed CFG of P_N. In every execution of the renamed and peeled program P^p_N in which control flows through n, the following hold.*

1. *If n is not a collapsed node, the no-overwriting property as in Lemma 5.1 holds for all scalar variables/arrays vA^n.*

2. *If n is a node representing a collapsed loop L, the scalar variable/array vA^n is not updated at any subsequent node along the execution, except possibly in the peel of L.*

Proof. Follows from the same reasoning as used in the proof of Lemma 5.1. □

We will henceforth refer to the property formalized in Lemma 5.4 as the *"no-overwriting"* property of the renamed and peeled program P_N^p.

Lemma 5.5 *In the peeled program P_N^p, each conditional branch node in a peel of a loop has an immediate post-dominator in the same peel.*

Proof. The syntactic restrictions on the input program, imposed by the grammar shown in Fig. 3.2, do not admit *break, continue, goto, exit* and *return* statements. Since a loop body is also syntactically a complete program, conditional branch nodes in the body of the loop, if any, always have an immediate post-dominator node within the body of the same loop. The peel of a loop is obtained by creating a copy of the loop body (using the function PEELSINGLELOOP invoked on line 8 of routine PEELALLLOOPS in Algorithm 4). Thus, the conditional branch nodes, if any, in the peel have an immediate post-dominator node within the same peel. □

Finally, the following lemma asserts that peeling does not change the Hoare semantics of programs.

Lemma 5.6 $\{\varphi_N\}\ \mathsf{P}_N\ \{\psi_N\}$ *holds iff* $\{\varphi_N\}\ \mathsf{P}_N^p\ \{\psi_N\}$ *holds.*

Proof. Follows immediately from the observation that peeling each loop preserves the semantics of the program. □

Example 5.4 We execute function PEELALLLOOPS on the renamed version of our running example, shown in Fig. 5.14(b). The resulting program is shown in Fig. 5.15. The algorithm first computes the number of iterations to be peeled from a loop in the program, given by PeelCount. The upper bound expression of each loop in the program is N. Hence, the number of iterations to be peeled is $N - (N - 1) = 1$. In other words, only

```
// assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N-1; i++) {
3.    S = S + A[i];
4.  }
5.  S = S + A[N-1];

6.  for(i=0; i<N-1; i++) {
7.    A1[i] = A[i] + S;
8.  }
9.  A1[N-1] = A[N-1] + S;

10. S1 = S;
11. for(i=0; i<N-1; i++) {
12.   S1 = S1 + A1[i];
13. }
14. S1 = S1 + A1[N-1];

// assert(S1 = N × (N+2))
```

Figure 5.15: Program with Loops Peeled

the last iteration is to be peeled from each loop. The function appends the statements in the peeled iteration after each loop and updates the upper bound expressions of each loop in the resulting program, as shown in Fig. 5.15. The algorithm also returns the set of peeled nodes, i.e. CFG nodes corresponding to the statements at lines 5, 9, and 14. □

5.3.3 Tracking Data Dependencies

As discussed in Section 3.2, the flow of control in a program can be conveniently represented by a CFG. A CFG, however, does not immediately provide information about data dependencies between program statements. We use a separate *data dependence graph*, or

DDG, to summarize data dependencies between relevant statements in a program. Our primary purpose in constructing such a DDG is to understand the dependencies of and from statements that are executed in P_N but not in P_{N-1}. These are related to the peeled statements described in Section 5.3.2, and determine what must eventually go into the difference program $\partial\mathsf{P}_N$, so that P_N and P_{N-1}; $\partial\mathsf{P}_N$ have the same effect on arrays and scalar variables.

While there are several notions of data dependence used in the literature (see [Tow76, Kuc78] for details), we use a fairly simple notion that best serves our purpose. We say that there is a read-after-write data dependence from n_1 to n_2 if the statement at n_2 uses a data value that is potentially generated by the statement at n_1. There is another kind of data dependence that is peculiar to our approach that also needs special handling. It may so happen that the glue code inserted between two nodes by Algorithm 3 has a loop, say L_1, that updates an array A that is also subsequently updated in another loop, say L_2 in non-glue code. If the peel of L_1 potentially updates an element of A that is also updated in the non-peeled part of L_2, then we have a write-after-write dependence between a statement in the peel of a (glue) loop and subsequent statement in the non-peeled part of another (non-glue) loop. We call such a dependence *non-peeled-write-after-peeled-write* dependence. Since we intend to move peels to the end of the program to construct a difference program, this kind of dependence poses a problem. Therefore, we explicitly identify such *non-peeled-write-after-peeled-write* dependencies below. Given the way Algorithm 3 operates, it is easy to see that such a dependence can only arise for arrays and not for scalars.

Note that in the above case when we have a glue loop followed by a non-glue loop updating the same array, there may also be write-after-write dependencies between the non-peeled (resp. peeled) part of the glue loop and the non-peeled (resp. peeled) part of the non-glue loop. However, such dependencies are preserved if we move peels of all loops to the end of the program to construct a difference program. Therefore, such write-after-write dependencies do not pose any problem for our purposes, and hence we do not keep track of them. Furthermore, due to the way Algorithm 3 operates, it can be seen that write-after-read dependencies can never arise between nodes of the collapsed CFG.

Formally, a DDG is a directed graph $G_D = (Locs, DE, \mu)$, where $Locs$ and μ are exactly as in the definition of a CFG, and $DE \subseteq Locs \times Locs$ represents *read-after-write*

and *non-peeled-write-after-peeled-write* dependencies between statements in the program. Since our primary interest is in using data dependencies to and from peeled statements for purposes of constructing difference programs, and since loops have a very specific form in our programs of interest (the grammar in Fig. 3.2 allows only loop counter to be updated in a loop-head node), it suffices to restrict our attention to data dependencies between distinct non-loop-head nodes in the peeled program.

Let n and n' be two CFG (hence also DDG) nodes. A conservative way of generating DDG edges is to add the edge (n, n') to DE if there is a control flow path $\pi : (n = n_1, n_2, \ldots, n_{t-1}, n_t = n')$ in the CFG such that one of the following conditions hold.

D1: $\big(\mathit{def}(n) \cap \mathit{uses}(n')\big) \setminus \bigcup_{i=2}^{t-1} \mathit{def}(n_i)$ contains a scalar variable v, or

D2: $\mathit{def}(n)$ contains an array A such that

(a) Either of the following conditions hold:

i. $A \in \mathit{uses}(n')$ and there is a common value that the index expression $\mathit{defIndex}(A, n)$ and some index expression in $\mathit{useIndex}(A, n')$ can have.

ii. $n \in \mathsf{PeelNodes}$ and $n' \notin \mathsf{PeelNodes}$ and $A \in \mathit{def}(n')$ and there is a common value that both the index expressions $\mathit{defIndex}(A, n)$ and $\mathit{defIndex}(A, n')$ can have.

(b) Some elements of A are potentially not updated along the path π.

Lemma 5.7 *For $n, n' \in Locs$ such that n' is reachable from n in the CFG, if neither condition D1 nor condition D2 holds, then there is no read-after-write or non-peeled-write-after-peeled-write dependence from n to n'.*

Proof. We prove the lemma by contradiction. Suppose, if possible, neither D1 nor D2 holds and yet there is a *read-after-write* dependence due to the data value generated at n being potentially used at n'. There are two cases to consider.

- If the data value pertains to a scalar variable v that is updated at n and accessed at n', then there must be a control flow path π from n to n' along which v is not updated at any intermediate node. This implies condition D1 is satisfied – a contradiction!

- Suppose the data value pertains to an element of array A that is updated at n and accessed at n'. Let the index expression of the array element updated at n be e, and let the corresponding index expression of the same element accessed at n' be e'. Clearly, both e and e' can assume the same value (the concrete index of the element of A under consideration), and there is a control flow path π from n to n' along which this specific array element has not been updated. This implies that both the conditions D2(a)(i) and D2(b) are satisfied, and hence condition D2 is satisfied – a contradiction!

Suppose, if possible, neither D1 nor D2 holds and yet there is a *non-peeled-write-after-peeled-write* dependence from n to n'. Suppose the data value pertains to an element of array A that is updated at nodes n and n' where $n \in \mathsf{PeelNodes}$ and $n' \notin \mathsf{PeelNodes}$. Let the index expression of the array element updated at n be e, and let the corresponding index expression of the same element updated at n' be e'. Clearly, both e and e' can assume the same value (the concrete index of the element of A under consideration), and there is a control flow path π from n to n' along which this specific array element has not been updated. This implies that both the conditions D2(a)(ii) and D2(b) are satisfied, and hence condition D2 is satisfied – a contradiction! □

Condition D2(b) above is not easy to check in general. However, for programs generated by the grammar in Fig. 3.2, it is possible to detect that condition D2(b) is violated in special cases. As an example, if there is a loop that updates all elements of array A in every control flow path from n to n', then indeed condition D2(b) is violated. For our purposes, we use this special case as a sufficient condition to detect violation of condition D2(b), and conservatively assume that the condition is potentially satisfied in all other cases. Needless to say, a more precise analysis can be done with additional computational effort to reduce the degree of conservativeness in the above check for condition D2(b). We defer such an improved analysis to future work. We now look at how condition D2(a) is checked. Recall from Section 3.1 that the array indices in our programs can only be expressions in terms of constants, scalar variables, the loop counter variables and the parameter N. Furthermore, our programs do not have nested loops. Therefore, at most one loop counter variable can appear in an array index expression. Specifically, if e is the index expression $defIndex(A, n)$, and if node n is part of a loop L with loop counter

ℓ, then e depends in general on ℓ, N and a set of scalar constants. Otherwise, i.e. if node n is not part of a loop, e depends on N and a set of scalar constants. A similar reasoning applies for array index expression(s) in $useIndex(A, n')$ as well. Condition D2(a) is satisfied if the constraint $(e = e')$ has a model, i.e. is satisfiable, for some index expression $e' \in useIndex(A, n')$, subject to the following conditions:

- Loop counters ℓ and ℓ' must have values within their respective loop bounds.
- If both n and n' are part of the same loop, then $\ell \leq \ell'$ (update at n cannot happen in an iteration after access at n').
- Every scalar variable v that appears in both e and e' and is updated along some control flow path from n to n' is renamed in e' to a fresh variable (since the values of v in e and e' may be different).

Function ComputeDDG, shown in Algorithm 5, constructs the DDG for an input program P_N represented using its CFG $(Locs, CE, \mu)$. We use the notation $n \overset{X}{\rightsquigarrow} n'$ to denote that there is a control flow path from n to n' in the CFG that passes through intermediate nodes in $X \subseteq Locs$. ComputeDDG proceeds by initializing the set of data dependence edges DE to $\emptyset$, and by checking for every pair of distinct non-loop-head nodes (n, n') such that $n \overset{Locs}{\rightsquigarrow} n'$, whether condition D1 or D2 referred to above is satisfied. If either one of the conditions is satisfied, it adds (n, n') to DE (lines 38–39).

The set S of scalar variables and arrays that potentially introduce data dependence from n to n' is initialized to $def(n) \cap uses(n')$ in line 5. If n is a peeled node and n' is a non-peeled node, then we append S with $def(n) \cap def(n')$ in line 7. Subsequently, the check for D1 is done in the loop in lines 8–12. In each iteration of this loop, we choose a scalar variable v from the set S and check whether there exists a control flow path from n to n' such that no intermediate node along the path updates v. The latter check is implemented by first collecting all nodes n'' (other than n and n') that does not update v in the set $NDefV$ (line 9). If there is a control flow path from n to n' that passes through intermediate nodes in $NDefV$, the value of v updated at n can reach the use of v at n'. In this case, there is a potential data dependence of n' on n through v and condition D1 is satisfied. We therefore set $D1Sat$ to True (line 11) and abort the search over additional scalar variables v in S (line 12). Otherwise, there is no dependency of n' on n through v.

Algorithm 5 COMPUTEDDG(($Locs$, CE, μ): program P_N, $\mathsf{PeelNodes}$: peeled statements)

1: $DE := \emptyset$;

2: **for** each $n, n' \in Locs \setminus LoopHeads$ s.t. $n \neq n'$ and $n \overset{Locs}{\rightsquigarrow} n'$ **do** ▷ n' reachable from n in CFG

3: $D1Sat := \mathsf{False}$; ▷ Initializing flags to indicate if condition D1/D2 holds

4: $D2Sat := \mathsf{False}$;

5: $S := def(n) \cap uses(n')$; ▷ Set of scalar variables/arrays that potentially introduce data dependence between n and n'

6: **if** $n \in \mathsf{PeelNodes} \wedge n' \notin \mathsf{PeelNodes}$ **then**

7: $S := S \cup \big(def(n) \cap def(n')\big)$; ▷ For non-peeled-write-after-peeled-write dependence

▶ `Check for condition D1`

8: **for** each scalar variable $v \in S$ **do**

9: $NDefV := \{n'' \mid n'' \neq n, n'' \neq n', v \notin def(n'')\}$; ▷ Nodes other than n, n' that do not define v

10: **if** $n \overset{NDefV}{\rightsquigarrow} n'$ **then** ▷ Potential data dependence (n, n') due to variable v

11: $D1Sat := \mathsf{True}$;

12: **break**;

▶ `Check for condition D2 if D1 is not already satisfied`

13: **if** (**not** $D1Sat$) **then**

14: **for** each array $A \in S$ **do**

15: $\phi_n := \mathsf{True}$; ▷ Initializing loop bound constraints for n and n'

16: $\phi_{n'} := \mathsf{True}$;

17: **if** n is part of loop L with loop counter ℓ **then**

18: $\phi_n := (0 \leq \ell < k_L)$;

19: **if** n' is part of a loop L' with loop counter ℓ' **then**

20: $\phi_{n'} := (0 \leq \ell' < k_{L'})$;

21: **if** L' same as L **then**

22: $\phi_{n'} := \phi_{n'} \wedge (\ell \leq \ell')$; ▷ n and n' in the same loop

23: $e := defIndex(A, n)$; ▷ Index expression used to update element of A at n

24: $U := useIndex(A, n')$;

25: **if** $n \in \mathsf{PeelNodes} \wedge n' \notin \mathsf{PeelNodes}$ **then**

26: $U := U \cup \{defIndex(A, n')\}$;

27: **for** each scalar variable v that appears in both e and some $e' \in U$ **do**

28: $DefV := \{n'' \mid n'' \neq n, n'' \neq n', v \in def(n'')\}$; ▷ Nodes other than n, n' that define v

29: **if** $\exists n'' \in DefV \wedge n \rightsquigarrow n'' \wedge n'' \rightsquigarrow n'$ **then** ▷ v modified along a path from n to n'

30: $e' := e'[v/v_{fresh}]$; ▷ Rename variable v in e' with fresh variable v_{fresh}

31: **if not** $\text{IsSAT}\big(\phi_n \wedge \phi_{n'} \wedge \bigvee_{e' \in U}(e = e')\big)$ **then**

32: **continue**; ▷ D2(a) violated for array A

33: **if** $\exists$ loop L'' with loop-head n'' s.t. $(n, n'$ not in $L'') \wedge (n \overset{Locs \setminus \{n''\}}{\not\rightsquigarrow} n')$ **then**

34: **if** L'' necessarily updates all elements of A **then**

35: **continue**; ▷ Condition D2(b) violated for array A

36: $D2Sat := \mathsf{True}$; ▷ Potential data dependence (n, n') due to array A

37: **break**;

38: **if** $\big(D1Sat \vee D2Sat\big)$ **then**

39: $DE := DE \cup \{(n, n')\}$;

40: **return** $(Locs, DE, \mu)$;

If the flag $D1Sat$ is not set to True even after iterating over all scalar variables in S, we turn to checking if condition D2 can be satisfied. Towards this end, we iterate over all array names A remaining in S, and formulate a constraint to check if condition D2(a) is satisfied (lines 15–32). This condition effectively checks if it is possible for the index expression $defIndex(A, n)$ to have the same value as any index expression $e' \in useIndex(A, n')$ or if it is possible for the index expression $defIndex(A, n)$ to have the same value as the index expression $defIndex(A, n')$ when n is a peeled node and n' is a non-peeled node. If not, the update/read of array A at n and n', cannot be for the same element, and hence, there is no data dependence (n, n') through A. As discussed above, to check if condition D2(a) is satisfied, we must conjoin loop bound constraints for loop counter variables in

case n or n' is present in a loop (lines 15–22), and rename every scalar variable v in expressions $e' \in \mathit{useIndex}(A, n')$ that also appears in the index expression $\mathit{defIndex}(A, n)$, if v is potentially re-defined in a control flow path from n to n'. The renaming of scalar variables, if needed, is done in lines 23–30. The call to IsSat in line 31 is an invocation of an SMT solver that tells us whether the constraint fed to it as argument is satisfiable, i.e. has a model. If not, condition D2(a), and hence D2, is violated. Otherwise, we check in lines 33 and 34 if there exists a loop L'' not containing n and n' that necessarily executes as control flows from n to n' ($n \overset{Locs\setminus\{n''\}}{\not\rightsquigarrow} n'$ checks this), and in which all elements of the array A are updated. Recall from the grammar in Fig. 3.2 that all loops in our programs are **for** loops with a loop counter that increments by 1 in each operation, and cannot be updated in the body of the loop. For such programs, it is sometimes easy to identify if a loop L'' is indeed updating all elements of an array A. If we cannot determine whether L'' necessarily updates all elements of A, we conservatively assume that it does not and the check in line 34 fails. If both the checks in lines 33 and 34 succeed, we conclude that condition D2(b), and hence D2, has been violated. In all other cases, we conservatively assume that D2 is satisfied, and set $\mathit{D2Sat}$ to True in line 36. In such cases, we also abort the search over additional array variables A in S.

Lemma 5.8 *Given a program represented as $(Locs, CE, \mu)$, let $(Locs, DE, \mu)$ be the DDG computed by* ComputeDDG*. For every pair of distinct non-loop-head nodes $n, n' \in Locs$, if $(n, n') \notin DE$, there is no read-after-write or non-peeled-write-after-peeled-write data dependence from n to n'.*

Proof. Since function ComputeDDG implements the checks for conditions D1, D2(a) and D2(b) in a straightforward manner, the proof follows from Lemma 5.7. □

We conclude this subsection with an illustration of DDG edges computed by ComputeDDG for our running example.

Example 5.5 Our running example with peeled loops is shown in Fig. 5.15. The CFG for this program is shown using solid edges in Fig. 5.16. For convenience of exposition, we have named nodes such that node n_i in the CFG of this program corresponds to the statement at line i of the peeled program, with two special nodes n_{start} and n_{end}, as usual. If we execute function ComputeDDG on this CFG, we obtain the data dependence edges

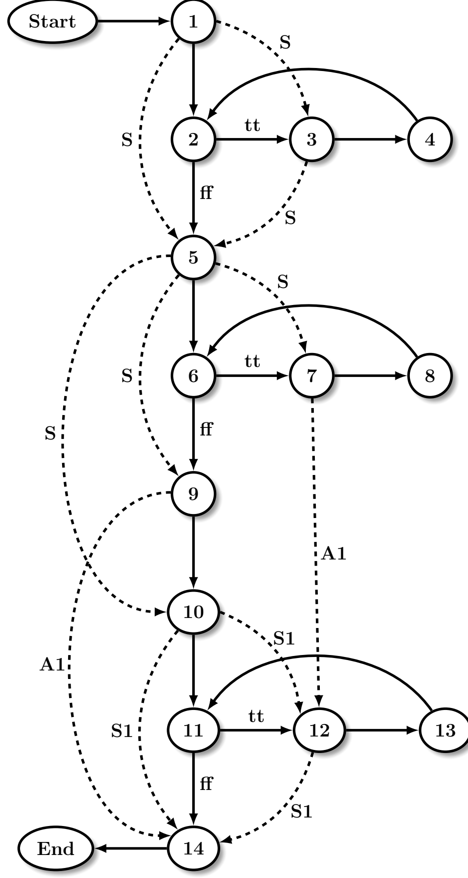


Figure 5.16: CFG (solid edges) and DDG (dashed edges) of the Peeled Program in Fig. 5.15

shown using dashed edges in Fig. 5.16. For ease of understanding, each DDG edge (n, n') is also labeled by a scalar variable/array that is responsible for the data dependence of n' on n. Thus, DDG edges $(n_1, n_3), (n_1, n_5), (n_3, n_5), (n_5, n_7), (n_5, n_9)$ and (n_5, n_{10}) represent data dependence through the scalar variable `S` and edges $(n_{10}, n_{12}), (n_{10}, n_{14}), (n_{12}, n_{14})$ represent data dependence through the scalar variable `S1`. In all these cases, condition $D1$ holds. Similarly, DDG edges $(n_7, n_{12}), (n_9, n_{14})$ represent data dependence through the array `A1`, since conditions D2(a) and D2(b) hold in these cases. Note that edge (n_7, n_{14}) (resp. (n_9, n_{12})) is not added although $\texttt{A1} \in \mathit{def}(n_7) \cap \mathit{uses}(n_{14})$ (resp. $\in \mathit{def}(n_9) \cap \mathit{uses}(n_{12})$) because condition D2(a) fails in this case. To see why D2(a) fails, notice that $\phi_7 := 0 \leq \texttt{i} < \texttt{N} - 1$ and $\phi_{14} := \mathsf{True}$. The expressions used to define and access array `A` are $\mathit{defIndex}(\texttt{A}, 7) := \texttt{i}$ and $\mathit{useIndex}(\texttt{A}, 14) := \{\texttt{N}-1\}$. The constraint $0 \leq \texttt{i} < \texttt{N}-1 \wedge \texttt{i} = \texttt{N} - 1$, computed in line 31 of Algorithm 5, is unsatisfiable. This violates D2(a)(i). Note that, in this example there are no non-peeled-write-after-peeled-write dependencies. ☐

Relation to Existing Techniques in Compilers

Several existing compilers generate *program dependence graph*, or PDG from a given input program, and a DDG can be extracted from such a PDG [FOW87]. Standard data-flow analysis techniques are usually used to identify data dependencies when constructing a PDG [FOW87, HR92]. One needs to be particularly careful when identifying dependence between statements updating and accessing array elements, since it is not only the same array name that must be involved in the update and access, but also the same element in the array. The problem is further compounded by the fact that array indices can be arbitrary expressions in general. While vectorizing compilers can compute precise dependencies with array index expressions using sophisticated *dependence tests* [KA01], it is not always the case that these are implemented in general-purpose compilers. A conservative generation of DDG may contain spurious data dependence edges, which, in our context, can lead to the construction of a difference program that is more complex than what is needed.

Another abstraction widely used in compilers for representing data dependencies is the polyhedral model (refer Section 2.6). The model has three main parts *iteration domains*, *scheduling functions* and *access functions* that represent each execution instance of a statement in the program as an integer point in a polyhedron. Since the polyhedral model can facilitate the extraction and representation of data dependencies, it is possible to use the dependence polyhedron instead of our DDG. The analysis, however, may at times be more conservative than required by our techniques. For example, several accesses to different memory locations may be conservatively treated as may-accesses. We believe that our way of refining the data dependencies by formulating a satisfiability problem can be used to strengthen the polyhedral analysis, specifically to refine the access function by resolving as many may-accesses as possible.

The polyhedral model has also been used to check the validity of program transformations by ensuring the preservation of the identified data dependencies. The program transformations performed by our techniques, as a sub-part of our verification strategy, are much more aggressive than those performed by compilers (refer Sections 5.4, 6.2 and 7.3.1). The correctness proofs for such transformations are, however, similar in spirit.

The programs input to our techniques may not always satisfy the constraints under which the polyhedral analysis produces fruitful results. Further, the polyhedral model

is often implemented in certain parallelizing compilers and may not be available in all general-purpose compilers. This would require the design of tools to be based on a specific parallelizing compiler.

5.3.4 Identifying "Affected" Variables

Recall that every loop L originally present in P_N iterates $k_L(N-1)$ times in P_{N-1} and $k_L(N)$ times in P_N. The iterations missed by P_{N-1} are represented by the peeled statements computed by function PEELALLLOOPS. It is natural to expect the difference program $\partial\mathsf{P}_N$ to contain the peeled statements, perhaps with some adaptations, if P_{N-1}; $\partial\mathsf{P}_N$ is to have the same effect as P_N on all relevant scalar variables and arrays. However, statements that are present in both P_N and P_{N-1} may also differ in their semantics, and therefore require "rectification" in $\partial\mathsf{P}_N$. For example, statements like `x = N;` and `if(x > N)` in P_N become `x = N-1;` and `if(x > N-1)` respectively, in P_{N-1}. Clearly, the corresponding statements in P_N and P_{N-1} in the above examples have different semantics. We say that such statements, though present in both P_{N-1} and P_N, are "affected" by the parameter N, and potentially need to be "rectified" in $\partial\mathsf{P}_N$. Our goal in this subsection is to identify all relevant scalar variables/arrays that are potentially affected in this sense, i.e. they are updated by versions of the same statement in P_N and P_{N-1} but can potentially result in different values being assigned due to the change in the parameter N. We use the data dependence information computed in the previous subsection to identify such variables and arrays, which we also call *affected variables/arrays*. Once these variables/arrays are identified, we can proceed to generate the difference program $\partial\mathsf{P}_N$ for effecting any rectification that may be needed.

Function COMPUTEAFFECTED, shown in Algorithm 6, computes the set of affected scalar variables/arrays of a peeled program P_N^p, represented by its CFG $(Locs^p, CE^p, \mu^p)$. Besides the CFG, the function also takes as input the set of CFG nodes corresponding to peeled statements, denoted PeelNodes. Recall that such a set is obtained when function PEELALLLOOPS is invoked. COMPUTEAFFECTED starts by constructing the data dependence graph using function COMPUTEDDG (line 1). The set of data dependence edges thus obtained is represented by DE^p. We use AffectedVars to denote the set of affected variables and arrays of P_N, and initialize it to the empty set in line 2. We also maintain a list of nodes n in WorkList such that the semantics of the program statement in $\mu(n)$

Algorithm 6 COMPUTEAFFECTED($(Locs^p, CE^p, \mu^p)$: peeled program P_N^p, PeelNodes: peeled statements)

1: $(Locs^p, DE^p, \mu^p) :=$ COMPUTEDDG($(Locs^p, CE^p, \mu^p)$, PeelNodes);
2: AffectedVars $:= \emptyset$; ▷ Initialize AffectedVars
   ▶ `Initialize WorkList with non-peeled nodes of CFG that either use` N `directly or have data dependence on peeled nodes.`
3: WorkList $:= \{n \mid n \in Locs^p \setminus \mathsf{PeelNodes}, N \in uses(n)$ or $\exists n'.\ n' \in \mathsf{PeelNodes} \wedge (n', n) \in DE^p\}$;
4: Processed $:= \emptyset$;
5: **while** WorkList is not empty **do**
6:     Remove a node n from the head of WorkList;
7:     Processed := Processed $\cup \{n\}$;
8:     **if** $\mu(n)$ is an assignment statement **then**
9:         AffectedVars := AffectedVars $\cup\ def(n)$; ▷ $def(n)$ potentially affected by N
10:         **for** all n' s.t. $n' \in Locs^p \setminus$ PeelNodes, $n' \notin$ Processed and $(n, n') \in DE^p$ **do**
11:             WorkList := APPENDTOLIST(WorkList, n');
12:     **else if** $\mu(n)$ is a branch condition **then**
13:         **for** all n' s.t. $n' \in Locs^p \setminus$ PeelNodes, $n' \notin$ Processed and $n \overset{Locs^p}{\rightsquigarrow} n'$ **do**
14:             WorkList := APPENDTOLIST(WorkList, n');
15:     **else**
16:         **continue**; ▷ n is a loop-head; do not do anything for loop-heads
17: **return** AffectedVars;

is potentially affected (directly or indirectly) by N. This worklist is initialized in line 3 with all non-peeled nodes n that either (i) have N in $uses(n)$, or (ii) are potentially data dependent on a peeled node, i.e. $\exists n'.\ n' \in$ PeelNodes and $(n', n) \in DE^p$. The exclusion of peeled nodes from the worklist is justified by the observation that these statements are present in P_N^p but not in P_{N-1}. Therefore, these must necessarily appear (possibly with modifications) in $\partial\mathsf{P}_N$, and no additional analysis is needed to identify these statements or variables/arrays updated by them. We also keep track of all non-peeled nodes that have been processed so far in the set Processed, initialized in line 4.

The loop in lines 5–16 iterates over the worklist, processing one node at a time

to identify affected scalar variables and arrays. We remove the node n at the head of the worklist and add it to Processed in lines 6–7. If $\mu^p(n)$ is an assignment statement, we conservatively consider the scalar variable or array updated at n to be potentially affected (marked in line 9). We also add all as-yet unprocessed nodes n' that have a data dependence on n to the worklist in line 11. This accounts for nodes that are potentially affected because they use a value that is generated at node n. If node n corresponds to a conditional branch statement, we conservatively consider all non-peeled nodes n' that are reachable from n in the CFG of P_N^p as potentially affected by N. If such a node n' has not been processed yet, we add it to the worklist in line 14. Finally, if n corresponds to a loop-head, we skip the identification of affected variables from n (line 16). This is justified since the special form of loops allowed by the grammar in Fig. 3.2 permits only loop counter variables to be updated in a loop-head, and loop counter variables are not relevant for the post-conditions we wish to prove. The overall set of affected scalar variables/arrays is iteratively computed until there are no nodes left in the worklist to process. Since the CFG of P_N^p has only a finite number of nodes, and since no node is processed more than once (thanks to the book-keeping done using the set Processed), the loop in lines 5–16 is guaranteed to terminate.

Lemma 5.9 *Let P_N^p be a peeled program fed as input to the function* ComputeAffected*. Let n be a node in P_N^p such that some scalar variable/array in $uses(n)$ is transitively data/control dependent on N or on a peeled node in P_N^p. Then n is added to* WorkList *during the execution of function* ComputeAffected*.*

Proof. A transitive data/control dependence as referred to in the lemma can be represented by a sequence of nodes $n_{i_1}, n_{i_2}, \ldots, n_{i_k} (= n)$, where either (i) $k = 1$ and $N \in uses(n)$ or (ii) $k > 1$ and $uses(n_{i_j})$ is data/control dependent on $n_{i_{j-1}}$, for $2 \leq j \leq k$. By Lemma 5.8, in case (ii), there is an edge from $n_{i_j - 1}$ to n_{i_j}, for $2 \leq j \leq k$, in the data dependence graph computed at line 1 of function ComputeAffected. We now prove the claim by induction on k.

We consider two base cases of the induction. If $k = 1$, we know that $N \in uses(n)$. Hence, n is added to WorkList in line 3 of the function ComputeAffected. If $k = 2$, either n_{i_1} is a peeled node or $N \in uses(n_{i_1})$. In the former case, $n = n_{i_2}$ is added to WorkList in line 3 of ComputeAffected. Otherwise, n_{i_1} is added to WorkList in line 3 and must

be removed from WorkList in a later iteration before function ComputeAffected terminates. In the iteration in which n_{i_1} is removed from WorkList, the node $n = n_{i_2}$ is added to WorkList either due to data dependence (line 11 of ComputeAffected) or control dependence (line 14 of ComputeAffected) of $uses(n)$ on n_{i_1}.

We next hypothesize that, for every transitive data/control dependence on N or on a peeled node, represented by the sequence of nodes $(n_{i_1}, n_{i_2}, \ldots, n_{i_j})$, where $2 \leq j \leq k-1$, the node n_{i_j} is added to WorkList during the execution of ComputedAffected.

For the inductive step, consider a transitive data/control dependence on N or on a peeled node, represented by the sequence of nodes $(n_{i_1}, n_{i_2}, \ldots, n_{i_k})$, where $n_{i_k} = n$. This implies that $uses(n_{i_{k-1}})$ is also data/control dependence on N or on a peeled node. Now by the inductive hypothesis, $n_{i_{k-1}}$ must be added to WorkList in some iteration of the loop in lines 5 – 16 of ComputedAffected. In the iteration in which $n_{i_{k-1}}$ is removed from WorkList, the node $n = n_{i_k}$ is added to WorkList either due to data dependence (line 11 of ComputeAffected) or control dependence (line 14 of ComputeAffected) of $uses(n_{i_k})$ on $n_{i_{k-1}}$. □

In order to study additional properties of function ComputeAffected, we need to introduce some additional notation. Given a peeled program P_N^p generated by PeelAllLoops, let $\mathsf{P}_N^\star$ denote the program obtained by removing the peels of all loops from P_N^p. Clearly, $\mathsf{P}_N^\star$ is identical to the un-peeled program P_N (fed as input to PeelAllLoops), but with all instances of N in upper bound expressions of loops replaced by $N-1$. The program $\mathsf{P}_N^\star$ is also closely related, but not identical, to P_{N-1}. Indeed, P_{N-1} has all occurrences of N (not just those appearing in upper bound expressions of loops) in P_N replaced by $N-1$, whereas only upper bound expressions of loops are modified to obtain $\mathsf{P}_N^\star$. As an example, the loop in Fig. 5.17(a) in the program P_N transforms to the loop in Fig. 5.17(b) in the program $\mathsf{P}_N^\star$ and to the loop in Fig. 5.17(c) in the program P_{N-1}.

Since there is a bijection between the nodes in the CFGs of P_N and P_{N-1}, and

```
for(l=0; l<N; l=l+1)
  if(x < N)
    x = x + N;
```
(a)

```
for(l=0; l<N-1; l=l+1)
  if(x < N)
    x = x + N;
```
(b)

```
for(l=0; l<N-1; l=l+1)
  if(x < N-1)
    x = x + N-1;
```
(c)

Figure 5.17: A loop in (a) P_N, (b) $\mathsf{P}_N^\star$, and (c) P_{N-1}

similarly between the nodes in the CFGs of P_N and $\mathsf{P}_N^\star$, there exists a bijection between the nodes in the CFGs of P_{N-1} and $\mathsf{P}_N^\star$ as well. It is also easy to see that since programs generated by the grammar in Fig. 3.2 only allow loop bound expressions that depend on constants and N, if L and L' are corresponding loops in P_{N-1} and $\mathsf{P}_N^\star$ respectively, then both L and L' iterate exactly the same number, i.e. $k_L(N-1)$, times.

Let $n_0, n_1, n_2, \ldots, n_t$ denote nodes in the CFG of P_{N-1}, and let $n'_0, n'_1, n'_2, \ldots, n'_t$ denote the corresponding nodes (per the bijection) in the CFG of $\mathsf{P}_N^\star$. For notational convenience, we let n_0 and n_t be the start and end nodes respectively of the CFG of P_{N-1}, and similarly for n'_0 and n'_t. Let σ denote an arbitrary initial state, i.e. valuation of all scalar variables and arrays, from which we wish to start executing P_{N-1} and $\mathsf{P}_N^\star$. Since programs generated by the grammar in Fig. 3.2 are deterministic, there is exactly one control flow path, say π, in the CFG of P_{N-1} that corresponds to the execution of P_{N-1} starting from σ. A similar argument holds for $\mathsf{P}_N^\star$, and let π' be the corresponding path in its CFG. In the following discussion, we use n_{i_j} to denote the jth node starting from n_{i_0} in π, where $i_0 = 0$ and $0 \le j < |\pi|$. The interpretation of n'_{i_j} in the context of π' is analogous.

Lemma 5.10 *Let π (resp. π') be the path in the CFG of P_{N-1} (resp. $\mathsf{P}_N^\star$) that corresponds to the execution of P_{N-1} (resp. $\mathsf{P}_N^\star$) starting from the state σ. Let $\widehat{\pi} : (n_{i_0}, n_{i_1}, \ldots, n_{i_j})$ be a prefix of the path π. Suppose, upon termination of* ComputeAffected*, no conditional branch node in $\widehat{\pi}$ is present in* Processed*. Then, $(n'_{i_0}, n'_{i_1}, \ldots, n'_{i_j})$ must be a prefix of π'.*

Proof. Recall that the CFGs of P_{N-1} and $\mathsf{P}_N^\star$ are identical (including all loop bounds) except possibly for the usage of N or an expression involving N in some conditional branches and/or assignment statements. Note that by definition, none of these CFGs contain any peeled nodes. Since π and π' start at corresponding nodes n_0 and n'_0 in the respective CFGs, every subsequent node n_{i_j} in π must be matched by the corresponding node n'_{i_j} in π', until a branch node is encountered along one of the paths and the branch condition potentially depends on N. To prove the lemma, it therefore suffices to show that no conditional branch node n_{i_k} in $\widehat{\pi}$ has any transitive data/control dependence on N.

Let n_{i_k} be a conditional branch node in $\widehat{\pi}$. By Lemma 5.9, if any scalar variable/array in $uses(n_{i_k})$ is transitively data/control dependent on N, then n_{i_k} must be added to

$\mathsf{WorkList}$ at some point during the execution of function ComputeAffected. Consequently, n_{i_k} must also be removed from $\mathsf{WorkList}$ (line 6) and added to $\mathsf{Processed}$ (line 7) before ComputeAffected terminates. However, this violates the premise of the claim, i.e. n_{i_k} is not present in $\mathsf{Processed}$ on termination of ComputeAffected. Therefore, no scalar variable/array in $uses(n_{i_k})$ can be transitively data/control dependent on N. This completes the proof of the lemma. □

Lemma 5.11 *Let P_N^p be a peeled program fed as input to the function* ComputeAffected*. Let vA be a scalar variable/array that is absent from* $\mathsf{AffectedVars}$ *when* ComputeAffected *terminates. If P_{N-1} and $\mathsf{P}_N^\star$ are executed starting from the same state σ, then vA has the same value on termination of both programs.*

Proof. We prove the lemma by contradiction. If possible, let σ be a state (i.e, valuation of variables and arrays) such that vA has different values on termination of P_{N-1} and $\mathsf{P}_N^\star$, when both programs are executed starting from σ. As before, we use π and π' to denote the paths in the CFGs of P_{N-1} and $\mathsf{P}_N^\star$ respectively, that correspond to the execution of the respective programs starting from σ. Note that by definition, none of these CFGs contain any peeled nodes. We consider the following cases.

- If none of π and π' updates vA, the value of vA at the end of execution of the two programs is the same as the value it had in σ. Clearly, the lemma holds in this case.

- Suppose node n_{i_j} in π updates vA. We define $Branch(\pi, n_{i_j})$ to be the set of all nodes n_{i_k} in the prefix of π ending at n_{i_j} such that n_{i_k} corresponds to a conditional branch statement. Similarly, $Dep(\pi, n_{i_j})$ is defined to be the set of all nodes n_{i_k} in the same prefix of π such that there is a path through data dependency edges in DE^p from node n_{i_k} to either n_{i_j} or to one of the nodes in $Branch(\pi, n_{i_j})$.

  If possible, let n_{i_k} be a node in $Branch(\pi, n_{i_j})$ such that the branch condition in $\mu^p(n_{i_k})$ has a (possibly transitive) data dependence on N. By Lemma 5.9, n_{i_k} must be added to $\mathsf{WorkList}$ sometime during the execution of ComputeAffected. Since n_{i_j} is reachable from n_{i_k} along π, this further implies that n_{i_j} must be added to $\mathsf{WorkList}$, and subsequently $vA \in def(n_{i_j})$ must be added to $\mathsf{AffectedVars}$ during the execution of ComputeAffected. However, we know that vA is not present in $\mathsf{AffectedVars}$ on termination of ComputeAffected. Therefore, no branch condition in any node n_{i_k} in $Branch(\pi, n_{i_j})$ can be transitively data dependent on N.

It follows that no node in $\mathit{Branch}(\pi, n_{i_j})$ can be added to WorkList during the execution of COMPUTEAFFECTED. Hence, none of them can be present in Processed on termination of COMPUTEAFFECTED. By Lemma 5.10, it now follows that if $(n_{i_0}, n_{i_1}, \ldots, n_{i_j})$ is a prefix of π, then $(n'_{i_0}, n'_{i_1}, \ldots, n'_{i_j})$ must be a prefix of π'.

Since the scalar variable/array vA is updated at n_{i_j} in π, the statement at n'_{i_j} in π' must also update vA. Therefore, (i) vA is updated at n'_{i_j} in π', and (ii) for every node n_{i_k} in $\mathit{Dep}(\pi, n_{i_j}) \cup \mathit{Branch}(\pi, n_{i_j})$, the corresponding node n'_{i_k} is present in $\mathit{Dep}(\pi', n'_{i_j}) \cup \mathit{Branch}(\pi', n'_{i_j})$.

Finally, we argue that no node in $\mathit{Dep}(\pi, n_{i_j})$ can be transitively data dependent on N. Indeed, if this was not the case, by Lemma 5.9, n_{i_j} would be added to WorkList during the execution of COMPUTEAFFECTED, and hence vA would be added to AffectedVars. However, this violates the premise that vA is absent from AffectedVars. Combining this with the result obtained above, we find that as far as path π is concerned, no node in $\mathit{Dep}(\pi, n_{i_j}) \cup \mathit{Branch}(\pi, n_{i_j})$ is transitively data dependent on N. Since modifications, if any, in statements at corresponding nodes of P_{N-1} and $\mathsf{P}^{\star}_{N}$ only involve replacing $N-1$ by N, such modifications preserve the dependence of every node on N. Therefore, no node in $\mathit{Dep}(\pi', n'_{i_j}) \cup \mathit{Branch}(\pi', n'_{i_j})$ transitively depends on N. This implies that the statements labeling nodes in $\mathit{Dep}(\pi, n_{i_j}) \cup \mathit{Branch}(\pi, n_{i_j})$ in π are identical to the statements labeling the corresponding nodes in $\mathit{Dep}(\pi', n'_{i_j}) \cup \mathit{Branch}(\pi', n'_{i_j})$.

Since both P_{N-1} and $\mathsf{P}^{\star}_{N}$ start from the same state σ, the values of vA computed by P_{N-1} after executing the sequence of statements corresponding to $(n_{i_0}, n_{i_1}, \ldots, n_{i_j})$ must therefore be identical to that computed by $\mathsf{P}^{\star}_{N}$ after executing the sequence of statements corresponding to $(n'_{i_0}, n'_{i_1}, \ldots, n'_{i_j})$. This proves the lemma.

- The case when node n'_{i_j} in π' updates vA is analogous to the above case. □

For a variable/array vA that is not identified as affected, vA cannot be in the *def* set of a non-peeled node that either (i) has a transitive data dependence on N or on a peeled node, or (ii) has a control flow path from a branch node that, in turn, has a transitive data dependence on N or on a peeled node. The following lemma formalizes this property.

Lemma 5.12 *Let* P_N^p *be a peeled program fed as input to the function* ComputeAffected*. Let* vA *be a scalar variable/array that is absent from* AffectedVars *after* ComputeAffected *terminates. Then,* $vA \notin def(n)$ *for every non-peeled node* n *in* P_N^p *such that some scalar variable/array in* $uses(n)$ *is transitively data/control dependent on* N *or on a peeled node in* P_N^p.

Proof. Consider an arbitrary non-peeled node n in P_N^p such that some scalar variable/array in $uses(n)$ has a transitive data/control dependence on N or on a peeled node. Then, by Lemma 5.9, n must be added to WorkList sometime during the execution of ComputeAffected. Consequently, n must also be removed from WorkList (line 6 of ComputeAffected). If n is an assignment node, then $def(n)$ is added to the set AffectedVars (line 9 of ComputeAffected). But since vA is absent from AffectedVars, it follows that $vA \notin def(n)$. □

For clarity of exposition, we will henceforth refer to the property formalized in Lemma 5.12 as the *"not-affected"* property in our arguments.

Example 5.6 Consider the peeled program in Fig. 5.15 along with its DDG in Fig. 5.16. Recall that nodes in the DDG are named such that node n_i corresponds to the statement at line i of the peeled program. The value of variable `S` computed in the peeled node (line 5 in Fig. 5.15) of the first loop is used to define array `A1` in the body of the second loop (line 7). Furthermore, the value of variable `S` computed in line 5 is used to initialize the value of `S1` in line 10. Thus, the algorithm initializes the worklist with the non-peeled nodes n_7 and n_{10} that are data dependent on the peeled node n_5. Array `A1` and variable `S1` updated at n_7 and n_{10} respectively are marked as affected in line 9 of ComputeAffected. We then add non-peeled nodes that have a data dependence on n_7 and n_{10} to the worklist in line 11 of the algorithm. Since array `A1` is used in node n_{12} in the third loop to define the variable `S1`, n_{12} is added to the worklist. Subsequently, the variable `S1` updated at n_{12} is marked as affected. No further non-peeled nodes have any data dependence on n_{12} and the worklist therefore becomes empty. Function ComputeAffected therefore terminates with {`A1`, `S1`} as the set of potentially affected variables/arrays. Note that variable `S` updated in the first loop (line 3 of Fig. 5.15) is not marked as affected since its value does not transitively depend on N or on any variable/array updated in peeled nodes. □

5.4 Computing the Difference Program $\partial\mathsf{P}_N$

In Section 5.3, we computed the ingredients to generate $\partial\mathsf{P}_N$ from P_N such that P_N and P_{N-1}; $\partial\mathsf{P}_N$ have the same effect on scalar variables and arrays of interest. For notational convenience, in the remainder of this subsection, we use P_N to denote the renamed version of a given program, and P^p_N to denote the peeled version of the renamed program.

Generating the difference program $\partial\mathsf{P}_N$ from the renamed and peeled program using the set of affected variables computed previously is still a daunting task. In order to help the reader better visualize the computation of difference program as well as to simplify the proof of correctness, we present steps involved in the computation of $\partial\mathsf{P}_N$ as a sequence of simple program transformations. Fig. 5.18 presents a high level overview of this sequence of transformations. We start with a peeled program P^p_N. We first canonicalize it to a program T^p_N that consists of a sequence of statements of a specific form (explained in Section 5.4.1). The statements in T^p_N corresponding to the peels of loops in P^p_N are then moved to the end of T^p_N to obtain the program T^o_N. The resulting program T^o_N can be viewed as the program $\mathsf{P}^\star_N$ followed by the peels of all loops in P^p_N. For purposes of the present discussion, we informally call the block of statements corresponding to the peels of all loops as $\mathsf{Peel}(\mathsf{P}_N)$. A formal definition of $\mathsf{Peel}(\mathsf{P}_N)$ and an algorithm for computing $\mathsf{Peel}(\mathsf{P}_N)$ from P_N are given in Section 5.4.4. Note that, if variables/arrays of interest are not those identified as affected by ComputeAffected, $\mathsf{P}^\star_N$ can be replaced with P_{N-1}. This allows us to obtain the difference program as $\mathsf{Peel}(\mathsf{P}_N)$. In the subsequent sections, we present each transformation in detail, describe the programs generated by them and prove that they preserve the overall semantics of the program as far as the variables/arrays of interest are concerned. This allows us to show that for a large class of programs wherein the variables/arrays of interest have specific properties, it is possible to use just the peels of loops in P^p_N as the difference program. This simplifies the computation of $\partial\mathsf{P}_N$ significantly.

We continue to use vA to denote a scalar variable or an array depending on the context. If vA is an array, the discussion below applies to every individual element $vA[j]$, where j is an index in the allowed range of indices of array vA. However, for notational convenience, we use vA (and not $vA[j]$) to refer to such an array element in the lemmas below. Note that this implies that the proof, once completed, applies to an arbitrary

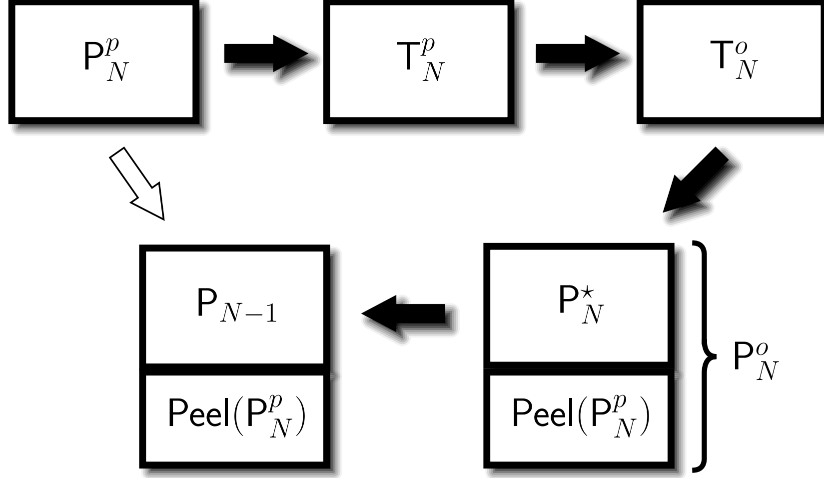


Figure 5.18: Sequence of Program Transformations to Decompose P^p_N into P_{N-1}; Peel(P^p_N)

element of the array vA, and hence to the whole of vA.

5.4.1 Canonicalizing the Program

In this section, we describe a simple transformation of the program P^p_N that allows us to view the program as a *linear sequence of statements* of a specific form. The transformation is only meant for purposes of simplifying the proofs of lemmas in the subsequent subsections and making them more approachable.

For every program P_N that can be generated by the grammar shown in Fig. 3.2, we rewrite the corresponding peeled program P^p_N as a linear sequence of statements of the form:

$$\textbf{if}(\mathsf{C})\ \textbf{then}\ \mathsf{S}\ \textbf{else}\ \mathsf{skip},$$

where skip is shorthand for the assignment statement `x = x;` for an arbitrary scalar variable `x` in P^p_N. The program fragment S is either (i) a loop, or (ii) a peel of a loop, or (iii) an assignment statement outside loops and peels in P^p_N. The conditional expression C is a conjunction of Boolean expressions along the True (resp. False) branches of all the conditional branch nodes b within the scope of which the program fragment S occurs in the program P^p_N. Since skip does not change values of any variables or arrays, we omit the else part in our subsequent discussion for notational clarity. Henceforth, we refer to statements in this form as guarded statements.

We now describe the function TRANSFORM in Algorithm 7 that canonicalizes the

Algorithm 7 TRANSFORM$((Locs^p, CE^p, \mu^p)$: peeled program $\mathsf{P}^p_N)$

1: $\mathsf{T}^p_N := (Locs, CE, \mu)$, where $Locs = \emptyset$, $CE = \emptyset$, $\mu = \emptyset$;
2: $\mathsf{P}^p_N :=$ COLLAPSELOOPBODY(P^p_N);
3: $\mathsf{P}^p_N :=$ COLLAPSELOOPPEEL(P^p_N);
   ▶ `The collapsed CFG of` P^p_N `is a directed acyclic graph (DAG).`
4: $\mathsf{BranchNodes} := \{n \in Locs^p \mid n$ is a branch node$\}$;
5: $n_{prev} :=$ FRESHNODE(); ▷ Create a start node and designate it as the previous node
6: $Locs := Locs \cup n_{prev}$;
7: $\mathsf{WorkList} :=$ TOPOLOGICALSORT$(Locs^p)$;
8: **while** $\mathsf{WorkList}$ is not empty **do**
9:     Remove a node n from head of $\mathsf{WorkList}$;
10:     **if** $n \notin \mathsf{BranchNodes}$ **then** ▷ n can a collapsed loop, collapsed peel or an assignment
11:         $\mathsf{C} := \mathsf{True}$;
12:         **for** each branch node $b \in \mathsf{BranchNodes}$ **do**
13:             $ipd :=$ IMMEDIATEPOSTDOMMINATOR(b);
14:             **if** $b \overset{Locs^p}{\rightsquigarrow} n \wedge \neg \left(b \overset{Locs^p}{\rightsquigarrow} ipd \wedge ipd \overset{Locs^p}{\rightsquigarrow} n\right)$ **then**
15:                 **if** $\exists n' \in Locs^p.\ (b, n', \mathbf{tt}) \in CE^p \wedge n' \overset{Locs^p}{\rightsquigarrow} n$ **then**
16:                     $\mathsf{C} := \mathsf{C} \wedge \mu^p(b)$;
17:                 **else**
18:                     $\mathsf{C} := \mathsf{C} \wedge \neg\mu^p(b)$;
19:         $n_{fresh} :=$ FRESHNODE();
20:         $\mu(n_{fresh}) :=$ 'if(C) then $\mu^p(n)$';
21:         $CE := CE \cup (n_{prev}, n_{fresh}, \mathsf{U})$;
22:         $Locs := Locs \cup n_{fresh}$;
23:         $n_{prev} := n_{fresh}$;
24: $\mathsf{T}^p_N :=$ UNCOLLAPSELOOPBODY(T^p_N);
25: $\mathsf{T}^p_N :=$ UNCOLLAPSELOOPPEEL(T^p_N);
26: **return** T^p_N;

input program P^p_N. Note that P^p_N is the program obtained after renaming the variables and arrays in the program P_N using the function RENAME (as described in Section 5.3.1)

and peeling all loops in the renamed program using the function PEELALLLOOPS (as described in Section 5.3.2).

We use T_N^p to denote the program generated as a result of this transformation. The function first creates an empty CFG for the program T_N^p in line 1. We then use the function COLLAPSELOOPBODY (line 2) to collapse the nodes and edges in each loop of the program P_N^p into its loop-head and the function COLLAPSELOOPPEEL (line 3) to collapse the nodes and edges in the peel of each loop in the program P_N^p into a single node. We collect all the conditional branch nodes of the peeled program P_N^p in the set BranchNodes in line 4. As the first step towards creating the program T_N^p, we create a new start node in the CFG of T_N^p in line 5, and mark it as the previous node n_{prev}. Subsequently, new nodes added to the set of nodes *Locs* of the program T_N^p during the transformation and are always linked to n_{prev}. We use the function TOPOLOGICALSORT to sort the set of nodes in the program P_N^p in a topological order. The sorted list is stored in WorkList in line 7.

The loop in lines 8–23 iterates over the sorted list of nodes in the worklist, processing one node at a time to generate the guarded assignment form for each node in the CFG that is either a collapsed loop, a collapsed peel, or an assignment statement. We remove a node n at the head of the worklist in line 9. For all nodes n that are not branch nodes (line 10), we transform the statement at node n into its guarded assignment form. In lines 11–18, we generate the guard condition C for node n, and in lines 19–23, we create the guarded assignment statement with the appropriate program fragment and insert it into the CFG of T_N^p.

For nodes n that are not within the scope of any conditional branch nodes, the conditional expression C in the guard must be set to True. Hence, we initialize the guard condition C to True in line 11. The loop in lines 12–18 accumulates the conditional expressions at each branch node b that has n within its scope. A node n is within the scope of a branch node b if there is a path from the node b to the node n that does not go thorough the immediate post-dominator of b. This condition is checked in line 14. Next, in line 15, we check if there is a path along the True edge of branch b leading upto node n. If so, we conjunct the conditional expression at node b with condition C (line 16). Otherwise, there is a path along the False edge of branch b leading upto the node n. Then, we conjunct the negated conditional expression at node b with condition C (line 18).

Once the guard condition C is computed, we create and insert the guarded statement in the CFG. We first create a fresh node n_{fresh} in line 19. We label the guarded assignment statement at node n_{fresh} in line 20 using the computed guard condition C and the statement at node n in P_N^p as the program fragment S. We then add an edge in CFG between the previous node n_{prev} and the new node n_{fresh} in line 21. Next, we add the node n_{fresh} to the list of nodes in the CFG (line 22) and mark it as the previous node (line 23).

Finally, we re-introduce the nodes and edges in the body of each loop using the function UnCollapseLoopBody in line 24 and the nodes and edges in the peel of each loop using the function UnCollapseLoopPeel in line 25. The function returns the canonicalized program T_N^p at line 26.

Note that if a variable/array element is used in the conditional expression at a branch node b, then it must have been updated (if at all) in an assignment node that has a control flow path to b. From the no-overwriting property as stated in Lemma 5.4, it now follows that the same variable/array element cannot be updated in any node that has a control flow path from b. This includes all nodes within the scope of branch b. This interesting property allows us to prove the following lemma on the function Transform.

Lemma 5.13 *Let T_N^p be the canonicalized version of P_N^p computed using function* Transform. *Let n be a node corresponding an assignment statement in the CFG of P_N^p (and hence T_N^p). Let π (resp. π') be a path in the collapsed CFG of P_N^p (resp. T_N^p) starting from the state σ. Then the following hold.*

1. *n is reached along π iff n is also reached along π'.*

2. *The program state σ_n is computed at node n along π iff the program state σ_n is computed at node n along π'.*

Proof. We consider a path π in the collapsed CFG of P_N^p corresponding to an execution starting from σ. Let n be a node corresponding to an assignment statement along path π. Let $\widehat{\pi}$ be the prefix of π that ends at n. We prove by induction on the length of $\widehat{\pi}$ that n is also reached along π' and the program state computed at n along π is the same as the state at n along π'. The proof crucially uses the fact that the guards of all statements in T_N^p that do not correspond to statements in nodes along π evaluate to False. The reasons for this are (i) every such guard has a conjunct that is the negation of

some branch condition b that evaluates to True along π, (ii) the consequence of the no-overwriting property stated above, and (iii) sequencing of statements in topological index order in T^p_N. In particular, (ii) and (iii) above ensure that the values of all variables/array elements used in the branch node b along π are the same as the corresponding values used in b along π'.

The converse direction of the proof is similar. □

We now informally state several properties about the canonicalized program T^p_N:

1. Every node labeled with an assignment statement in the program P^p_N appears exactly at one unique location in the program T^p_N. Thus, there exists a bijection function F^{P} that maps the assignment nodes in T^p_N to the corresponding assignment nodes in P^p_N and a bijection function F^{T} that maps the assignment nodes in P^p_N to the corresponding assignment nodes in T^p_N.

2. Every node (including conditional branch nodes) within a loop body/peel in the program P^p_N appears exactly at one unique location in the program T^p_N. Thus, there exists a bijection function G^{P} that maps the nodes within loop body/peel in T^p_N to the corresponding nodes in P^p_N and a bijection function G^{T} that maps the nodes within loop body/peel in P^p_N to the corresponding nodes in T^p_N.

3. A dependence edge (n_1, n_2) exists between nodes with assignment statements n_1 and n_2 in the program T^p_N if and only if the dependence edge $(F^{\mathsf{P}}(n_1), F^{\mathsf{P}}(n_2))$ exists in the program P^p_N.

4. A dependency edge (b, n) exists between a conditional branch node b and a node with assignment statement n in the program T^p_N if and only if either b and n both occur within the same loop body/peel of a loop and the dependency edge $(G^{\mathsf{P}}(b), F^{\mathsf{P}}(n))$ exists in P^p_N, or if n appears in the program fragment S in a guarded assignment statement of the form '**if**(C) **then** S **else** skip', where node b refers to the conditional branch node labeled with the condition C. In the latter case, there is a dependency edge $(b_i, F^{\mathsf{P}}(n))$ from at least one conditional branch node b_i in the program P^p_N whose condition is used to form the condition C at node b in T^p_N.

Observe that after the transformation, there are potentially many more conditional branch nodes in the CFG of T^p_N as compared to the CFG of P^p_N. Note that renaming

(as described in Section 5.3.1) ensures that each use of a variable/array gets mapped to exactly one definition along each path in the program. Thus, due to renaming, the variables/arrays used in a conditional expression are not modified along any branch of the conditional statement. These facts allow the function TRANSFORM in Algorithm 7 to use the same conditional expression $\mu(b)$ of a branch node b of program P_N^p at several program locations in the canonicalized program T_N^p where the expression $\mu(b)$ is replicated as a conjunct in the conditional expression C.

We end this subsection with an illustration of the program transformation achieved by applying the technique discussed above on an example with loops and branch statements.

Example 5.7 Consider the peeled program P_N^p shown in Tab. 5.1(a). The variables and arrays of the input program are renamed (as described in Section 5.3.1) ensuring that along each path of the program the value of a variable or an array element is accessible till the end of the path and all the loops in the program are peeled (as described in Section 5.3.2). We number the lines in the program such that the statements in loops (and in peels) have the same line number but with an alphabet appended when multiple statements are present. This numbering allows us to refer to the program statements with a consistent line number even after collapsing loops and peels. P_N^p has three peeled loops, `L1` (lines $4a$ and $4b$), `L2` (lines $8a$ and $8b$) and `L3` (lines $11a$ and $11b$). Loops `L1` and `L2` along with their peels are within the scope of the conditional branch statement in line 2. Loop `L3` and its peel are not within the scope of any branch statement.

Tab. 5.1(b) shows the program T_N^p generated by our transformation. To distinguish the statements in P_N^p from those in T_N^p, the line numbers of statements in T_N^p are suffixed with a prime symbol. As can be seen, T_N^p is a linear sequence of guarded statements. The line numbers of each guarded statement in T_N^p matches the line number of the corresponding statement in P_N^p. Notice that every assignment statement in P_N^p appears exactly at one unique location in T_N^p. □

5.4.2 Reordering the Peels

We now reorder the statements in the program T_N^p such that all guarded statements corresponding to peels of loops are executed after all other guarded statements. However,

Table 5.1: (a) Program P_N^p, (b) Transformed Program T_N^p, (c) Transformed & Reordered Program T_N^O and (d) Program P_N^o / Program $\mathsf{P}_N^\star$; $\mathsf{Peel}(\mathsf{P}_N^p)$

```
1.   S = 10;
2.   if(S > 5) {
3.     S1 = 1;
4a.    for(i=0; i<N-1; i++)  //Loop L1
4b.      A1[i] = A[i] + 1;
5.     A1[N-1] = A[N-1] + 1; //Peel of L1
6.   } else {
7.     S1 = 20;
8a.    for(i=0; i<N-1; i++)  //Loop L2
8b.      A1[i] = A[i];
9.     A1[N-1] = A[N-1];     //Peel of L2
10.  }
11a. for(i=0; i<N-1; i++)    //Loop L3
11b.   A2[i] = A1[i] + S1;
12.  A2[N-1] = A1[N-1] + S1; //Peel of L3
```

(a)

```
1'.  if(true) S = 10;
3'.  if(S > 5) S1 = 1;
4'.  if(S > 5) Loop L1;
5'.  if(S > 5) Peel of L1;
7'.  if(!(S > 5)) S1 = 20;
8'.  if(!(S > 5)) Loop L2;
9'.  if(!(S > 5)) Peel of L2;
11'. if(true) Loop L3;
12'. if(true) Peel of L3;
```

(b)

```
1'.  if(true) S = 10;
3'.  if(S > 5) S1 = 1;
4'.  if(S > 5) Loop L1;
7'.  if(!(S > 5)) S1 = 20;
8'.  if(!(S > 5)) Loop L2;
11'. if(true) Loop L3;

5'.  if(S > 5) Peel of L1;
9'.  if(!(S > 5)) Peel of L2;
12'. if(true) Peel of L3;
```

(c)

```
1.   S = 10;
2a.  if(S > 5) {
3.     S1 = 1;
4a.    for(i=0; i<N-1; i++)  //Loop L1
4b.      A1[i] = A[i] + 1;
6a.  } else {
7.     S1 = 20;
8a.    for(i=0; i<N-1; i++)  //Loop L2
8b.      A1[i] = A[i];
10a. }
11a. for(i=0; i<N-1; i++)    //Loop L3
11b.   A2[i] = A1[i] + S1;

2b.  if(S > 5) {
5.     A1[N-1] = A[N-1] + 1; //Peel of L1
6b.  } else {
9.     A1[N-1] = A[N-1];     //Peel of L2
10b. }
12.  A2[N-1] = A1[N-1] + S1; //Peel of L3
```

(d)

the relative ordering among the guarded statements corresponding to peels is preserved. We use T_N^o to denote the program obtained after this reordering.

Example 5.8 Continuing with the canonicalized program T_N^p shown in Tab. 5.1(b), the reordered program T_N^o is shown in Tab. 5.1(c). The line numbers follow the same pattern described in Example 5.7. Notice that in T_N^o the guarded statements corresponding to peels at lines $5'$, $9'$ and $12'$ appear after all other statements in the program. □

Let vA be a variable/array in P_N^p that is not identified as affected by function ComputeAffected. The following lemmas establish that if programs T_N^p and T_N^o are executed from the same state σ, they always compute the same value of vA. Specifically, Lemma 5.14 shows that all data dependencies that potentially have a bearing on the value of vA are the same in T_N^p and T_N^o. Lemma 5.15 uses this to show that the value of vA computed by T_N^p and T_N^o are the same. For clarity of exposition in the following discussion, when we say that there is a data dependence path from n_i to n_j in a program, we mean that there is a path from n_i to n_j in the DDG of the program. Similarly, when we say that there is a control dependence path from n_i to n_j, we mean that there is a data dependence path from n_i to a conditional branch node within whose scope n_j lies.

Lemma 5.14 *Let vA be a scalar variable/array that is absent from* AffectedVars *when* ComputeAffected *is invoked on P_N^p. Let n be a node in P_N^p such that $vA \in def(n)$. For every node n' in P_N^p, there is a data/control dependence path from n' to n in T_N^p iff there exists such a path in T_N^o.*

Proof. Consider nodes n and n' in P_N^p (hence also in T_N^p and T_N^o). We consider two cases.

1. Suppose there is a data/control dependence path from n' to n in T_N^p. From our construction of T_N^p, we know that there exists such a data/control dependence path in P_N^p as well. We now show that such a data/control dependence path also exists in T_N^o by considering two sub-cases.

   (a) Suppose n is a non-peeled node in P_N^p. Since vA is not identified as affected, by the not-affected property (Lemma 5.12), we have that n' is not a peeled node. Therefore, both n and n' are non-peeled nodes. Since the relative ordering of all non-peeled nodes is preserved by our reordering transformation, the data/control dependence between n and n' continues to exist in T_N^o as well.

(b) Suppose n is a peeled node in P_N^p. If n' is also a peeled node, then since the relative ordering among the peeled nodes is preserved by reordering, the data/control dependence between n and n' continues to exist in T_N^o. On the other hand, if n' is a non-peeled node, since all non-peeled nodes precede all peeled nodes after reordering, the data/control dependence exists in T_N^o in this case as well.

2. Suppose there is a data/control dependence path from n' to n in T_N^o. We show that such a dependence path exists in T_N^p by considering the following sub-cases.

   (a) If both n' and n are non-peeled (resp. peeled) nodes, then since reordering does not change the relative ordering of the non-peeled (resp. peeled) nodes, the dependence is present in T_N^p.

   (b) The case where n is a non-peeled node and n' is peeled node cannot arise, since all non-peeled nodes appear before peeled nodes in T_N^o.

   (c) If n' is a non-peeled node and n is a peeled node, there are two further sub-cases. If n' is ordered before n in T_N^p then the dependence is present in T_N^p as well. Otherwise, we ask if the dependence from n' to n in T_N^o is a read-after-write or write-after-write. Since n is ordered before n' in T_N^p, by the no-overwriting property (Lemma 5.4), both n and n' cannot update the same renamed variable. Therefore, the dependence from n' to n in T_N^o cannot be write-after-write, and hence must be read-after-write. This requires a variable/array element in $uses(n)$ to also be present in $def(n')$. Such a variable/array element must have been updated (if at all) prior to its use in node n. Once again, by the no-overwriting property, this variable/array element cannot be updated by n', which appears after n in T_N^p. This completes the proof. □

Lemma 5.15 *Let vA be a scalar variable/array that is absent from* AffectedVars *upon invocation of* ComputeAffected *on P_N^p. If T_N^p and T_N^o are executed from the same state σ, then vA has the same value on termination of both programs.*

Proof. Follows from Lemma 5.14 and the fact that reordering does not change the individual guarded statements in the canonicalized program T_N^p. □

5.4.3 De-canonicalizing the Reordered Program

Recall that our aim is to decompose the program P_N^p into two program fragments, the program P_{N-1} and the difference program $\partial\mathsf{P}_N$. We have seen above that the reordering step already achieves the purpose of moving the guarded statements corresponding to peels to the end of the program, providing a good candidate for the difference program $\partial\mathsf{P}_N$. However, the part of the reordered program that precedes the guarded statements corresponding to peels may not have syntactic similarity with P_{N-1} in general. In order to remedy this situation, we now "undo" the canonicalization process (as described in Section 5.4.1) that allowed us to view the program P_N^p as a linear sequence of guarded statements. Specifically, we transform the guarded statements back to statements of the form that were present in P_N^p to begin with. We do this separately for the guarded statements corresponding to peels, and for the part of T_N^o that precedes these, so that we obtain a program fragment that is syntactically similar to P_{N-1} followed by a difference program. In the subsequent discussion, we call the resulting de-canonicalized program P_N^o.

Example 5.9 Consider the reordered program T_N^o from the example shown in Tab. 5.1(c). The program P_N^o shown in Tab. 5.1(d) is obtained by de-canonicalization. The line numbers follow the pattern similar to the program P_N^p as described in Example 5.7. It is worth noticing that, the statements corresponding to peels of loops appear after all other statements in P_N^o and part of program P_N^o that precedes the peels is syntactically similar to P_{N-1}. □

Notice that after de-canonicalization, there may be more conditional branch nodes in the CFG of P_N^o as compared to the CFG of P_N^p but fewer conditional branch nodes as compared to the CFG of T_N^o. Since the canonicalization transformation is semantics preserving (Lemma 5.13), the reverse transformation is also semantics preserving, establishing the correctness of de-canonicalization. Following lemma formalizes this fact.

Lemma 5.16 *Let P_N^o be the de-canonicalized version of T_N^o. Let n be a node corresponding an assignment statement in the CFG of T_N^o (and hence P_N^o). Let π (resp. π') be a path in the collapsed CFG of T_N^o (resp. P_N^o) starting from the state σ. Then the following hold.*

1. *n is reached along π iff n is also reached along π'.*

2. *The program state σ_n is computed at node n along π iff the program state σ_n is computed at node n along π'.*

Proof. The proof is similar to that shown in Lemma 5.13. $\square$

The program P^o_N obtained after de-canonicalization satisfies properties similar to those stated during the transformation to the canonicalized program T^p_N in Section 5.4.1. Below, we state these properties informally:

1. Every node labeled with an assignment statement in the program T^o_N appears exactly at one unique location in the program P^o_N. Thus, there exists a bijection function F^{P} that maps the assignment nodes in T^o_N to the corresponding assignment nodes in P^o_N and a bijection function F^{T} that maps the assignment nodes in P^o_N to the corresponding assignment nodes in T^p_N.

2. Every node (including conditional branch nodes) within a loop body/peel in the program P^o_N appears exactly at one unique location in the program T^o_N. Thus, there exists a bijection function G^{P} that maps the nodes within loop body/peel in T^o_N to the corresponding nodes in P^o_N and a bijection function G^{T} that maps the nodes within loop body/peel in P^o_N to the corresponding nodes in T^o_N.

3. A dependence edge (n_1, n_2) exists between nodes with assignment statements n_1 and n_2 in the program T^o_N if and only if the dependence edge $(F^{\mathsf{P}}(n_1), F^{\mathsf{P}}(n_2))$ exists in the program P^o_N.

4. A dependency edge (b_i, n) exists between a conditional branch node b_i and a node with assignment statement n in the program P^o_N if and only if either b_i and n both occur within the same loop body/peel of a loop and the dependency edge $(G^{\mathsf{T}}(b_i), F^{\mathsf{T}}(n))$ exists in T^o_N, or if n appears in the program fragment S in a guarded assignment statement of the form '**if**(C) **then** S **else** skip', where node b refers to the conditional branch node labeled with the condition C. In the latter case, there is a dependency edge $(b, F^{\mathsf{T}}(n))$ from a conditional branch node b in the program T^o_N whose condition C has a conjunct with the expression at the node b_i from P^o_N.

5.4.4 Peels of Loops as the Difference Program

Recall from Section 5.3.4 that $\mathsf{P}^{\star}_{N}$ is effectively P_N with the peels removed. This is exactly what we get by de-canonicalizing the part of T^{o}_{N} that precedes the guarded statements corresponding to peels. Therefore, the program P^{o}_{N} can be viewed as the sequential composition of $\mathsf{P}^{\star}_{N}$ and the de-canonicalized version of guarded statements corresponding to peels of loops in T^{o}_{N}.

Definition 5.2 *The de-canonicalized version of the guarded statements corresponding to peels of loops in* T^{o}_{N} *is called* $\mathsf{Peel}(\mathsf{P}^{p}_{N})$*.*

It follows from Definition 5.2 that P^{o}_{N} can be written as $\mathsf{P}^{\star}_{N}$; $\mathsf{Peel}(\mathsf{P}^{p}_{N})$. It turns out that $\mathsf{Peel}(\mathsf{P}^{p}_{N})$ can be constructed directly from P_N without having to go through canonicalization, reordering and de-canonicalization. We now describe an algorithm for constructing the program $\mathsf{Peel}(\mathsf{P}^{p}_{N})$. We start with the CFG of the peeled program P^{p}_{N}. Recall from Section 5.3.2 that $\mathsf{PeelNodes}$ denotes the set of peeled nodes in P_N. Let $\mathsf{CondNodes}$ be the set of all non-peeled conditional branch nodes b such that there is a peeled node n within the scope of the branch b. In other words, if d denotes the immediate post-dominator of b, there is a path from b to d that passes through n. We define *Locs* to be the set $\mathsf{PeelNodes} \cup \mathsf{CondNodes}$. Only these nodes in the CFG of P^{p}_{N} are relevant for the construction of $\mathsf{Peel}(\mathsf{P}^{p}_{N})$.

The function ProgramPeel, presented in Algorithm 8, generates the program $\mathsf{Peel}(\mathsf{P}^{p}_{N})$ with the peeled nodes and their enclosing branches. The function takes the peeled program P^{p}_{N} and the set of peeled nodes $\mathsf{PeelNodes}$ (computed by function PeelAllLoops in Section 5.3.2). The program $\mathsf{Peel}(\mathsf{P}^{p}_{N})$ inherits the skeletal structure of the peeled program P^{p}_{N} as we copy the CFG of the input program P^{p}_{N} in line 1. We next compute the set $\mathsf{CondNodes}$ consisting of the conditional branch nodes that enclose a peel of a loop in the peeled program P^{p}_{N} in line 2. We retain only the peeled nodes and the nodes in the set $\mathsf{CondNodes}$, in line 3. The loop in lines 4–9 iterates over each node that must be retained and ensure that its out-going edges end up in a node that is retained. The loop in lines 5–9 iterates over each out-going edge (n, n', c) that got removed and does the following. We first compute the set of nodes in P^{p}_{N} that post-dominate the node n' using the function Post-Dominators (line 6). Only the nodes in $\mathsf{PDNodes} \cap \mathit{Locs}$ are retained in the CFG of $\mathsf{Peel}(\mathsf{P}_N)$. We then identify the node n'' that is strictly post-

Algorithm 8 ProgramPeel($(Locs^p, CE^p, \mu^p)$: peeled program P_N^p, PeelNodes: peeled statements)

1: $\mathsf{Peel}(\mathsf{P}_N^p) := (Locs, CE, \mu)$, where $Locs := Locs^p$, $CE := CE^p$, and $\mu := \mu^p$;

2: $\mathsf{CondNodes} := \{n \mid n \overset{Locs^p}{\rightsquigarrow} n' \wedge n' \overset{Locs^p}{\rightsquigarrow} n''\}$ where n is a non-peeled conditional branch node, n' is a peeled node and n'' is the immediate post-dominator of n;

3: $Locs := \mathsf{PeelNodes} \cup \mathsf{CondNodes} \cup \{n_{start}, n_{end}\}$; ▷ Retain only the peeled nodes, their enclosing non-peeled branches, and the start and end nodes

4: **for** each node $n \in Locs$ **do**

5:     **for** each edge $(n, n', c) \in CE^p$ **do**

6:         PDNodes := Post-Dominators(n');

7:         Let n'' be the node that is strictly post-dominated by each node in PDNodes $\cap$ $Locs$;

8:         $CE := CE \setminus \{(n, n', c)\}$;

9:         $CE := CE \cup \{(n, n'', c)\}$;

10: **return** $\mathsf{Peel}(\mathsf{P}_N^p)$;

dominated by each node in the set of retainable post-dominators of n' (line 7). We then remove the edge (n, n', c) (line 8) and add the edge (n, n'', c) (line 9). Finally, the function returns the program $\mathsf{Peel}(\mathsf{P}_N^p)$ in line 10.

Example 5.10 Consider the peeled program P_N^p shown in Tab. 5.1(a). The program has a non-peeled conditional branch statement on line 2. The peeled statements on lines 6 and 13 are within the scope of the conditional branch statement on line 2. Thus, CondNodes = { 2 } and $Locs$ = { 2b, 5, 6b, 9, 10b, 12 }. The program $\mathsf{Peel}(\mathsf{P}_N^p)$ is the program fragment consisting of the nodes in the set $Locs$ in Tab. 5.1(d). Notice that the non-peeled conditional branch node in P_N^p (on line 2) that has the peeled nodes within its scope is retained in $\mathsf{Peel}(\mathsf{P}_N^p)$ along with the peels of loops. □

Recall that our goal is to decompose the given program P_N such that P_N and $\mathsf{P}_{N-1};\ \partial\mathsf{P}_N$ have the same effect on scalar variables and arrays of interest. By Lemma 5.11, for variables/arrays vA that are absent from AffectedVars, if we execute P_{N-1} and $\mathsf{P}_N^\star$ starting from the same state σ, then vA has the same value on termination of both programs. This allows us to relate the computation in the programs P_N^o, $\mathsf{P}_N^\star$; $\mathsf{Peel}(\mathsf{P}_N^p)$ and P_{N-1}; $\mathsf{Peel}(\mathsf{P}_N^p)$ as formalized in the lemma below.

Lemma 5.17 *Let* P_N^p *be a peeled program and let* vA *be a scalar variable/array in* P_N^p *that is absent from* AffectedVars *when* ComputeAffected *is executed on* P_N^p. *If* P_N^o *and* P_{N-1}; $\mathsf{Peel}(\mathsf{P}_N^p)$ *are executed from the same state* σ, *then* vA *has the same value on termination of both programs.*

Proof. We break the proof in two parts. We first show that if P_N^o and $\mathsf{P}_N^\star$; $\mathsf{Peel}(\mathsf{P}_N^p)$ are executed starting from the same state σ, then vA has the same value on termination of both programs. This follows easily from Lemmas 5.13, 5.15 and 5.16.

Next we show that if $\mathsf{P}_N^\star$; $\mathsf{Peel}(\mathsf{P}_N^p)$ and P_{N-1}; $\mathsf{Peel}(\mathsf{P}_N^p)$ are executed from the same state σ, then vA has the same value on termination of both programs. We prove this part by case analysis.

Suppose the last update to vA in $\mathsf{P}_N^\star$; $\mathsf{Peel}(\mathsf{P}_N^p)$ happens in a non-peeled node in $\mathsf{P}_N^\star$. Then, the proof follows immediately from Lemma 5.11.

Suppose the last update to vA in $\mathsf{P}_N^\star$; $\mathsf{Peel}(\mathsf{P}_N^p)$ happens in a peeled node n in $\mathsf{Peel}(\mathsf{P}_N^p)$. Let S denote the set of variables/arrays vA' such that the updated value of vA at node n depends on the values of each $vA' \in S$ on termination of $\mathsf{P}_N^\star$. There are two sub-cases to consider.

If no $vA' \in S$ is identified as affected by ComputeAffected, then by Lemma 5.11 the value of every such vA' is the same after termination of $\mathsf{P}_N^\star$ and P_{N-1}. This implies that the value of vA is also same after termination of $\mathsf{P}_N^\star$; $\mathsf{Peel}(\mathsf{P}_N^p)$ and P_{N-1}; $\mathsf{Peel}(\mathsf{P}_N^p)$.

Now consider the case where some $vA' \in S$ is identified as affected by ComputeAffected. Let L be a loop in P_N^p from which the node n is peeled. From the construction of peeled nodes, we know that for every node n in the peel of L there is a corresponding node n' in the "uncollapsed" body of loop L such that the *def* and *uses* sets of the two nodes n and n' coincide. Since the update to vA at node n depends on vA' that is identified as affected, the update to vA at node n' in loop L must also depend on the affected variable/array vA'. However, this would cause ComputeAffected to identify vA as an affected variable. This leads to a contradiction since we know vA is not affected. This completes the proof. □

Lemma 5.17 allows us to use $\mathsf{Peel}(\mathsf{P}_N^p)$ as the difference program $\partial\mathsf{P}_N$ if none of the scalar variables and arrays of interest are identified as affected by ComputeAffected. This holds true in the case where the post-condition $\psi^r(N)$ does not refer to any affected

Algorithm 9 SyntacticDiff($\varphi(N)$: pre-condition)

1: **if** $\varphi(N)$ is of the form $\forall i \in \{0 \dots N\}\ \widehat{\varphi}(i)$ **then**
2:     **if** $\varphi(N) \Rightarrow \varphi(N-1)$ is invalid **then**
3:         **throw** "Unable to compute the difference pre-condition";
4:     $\partial\varphi(N) := \widehat{\varphi}(N)$;
5: **else if** $\varphi(N)$ is of the form $\exists i \in \{0 \dots N\}\ \widehat{\varphi}(i)$ **then**
6:     **if** $\varphi(N-1) \Rightarrow \varphi(N)$ is invalid **then**
7:         **throw** "Unable to compute the difference pre-condition";
8:     $\partial\varphi(N) := \widehat{\varphi}(N)$;
9: **else if** $\varphi(N)$ is of the form $\varphi^1(N) \wedge \cdots \wedge \varphi^k(N)$ **then**
10:     $\partial\varphi(N) :=$ SyntacticDiff$(\varphi^1(N)) \vee \cdots \vee$ SyntacticDiff$(\varphi^k(N))$;
11: **else if** $\varphi(N)$ is of the form $\varphi^1(N) \vee \cdots \vee \varphi^k(N)$ **then**
12:     $\partial\varphi(N) :=$ SyntacticDiff$(\varphi^1(N)) \vee \cdots \vee$ SyntacticDiff$(\varphi^k(N))$;
13: **else**
14:     $\partial\varphi(N) :=$ True;
15: **if** P_{N-1} updates scalars or array elements in $\partial\varphi(N)$ **then**
16:     $\partial\varphi(N) :=$ True;
17: **return** $\partial\varphi(N)$;

variable/array. Note that using $\mathsf{Peel}(\mathsf{P}_N^p)$ as the difference program $\partial\mathsf{P}_N$ works even if there are other variables/arrays (not of interest) that are affected.

5.5 Computing the Difference Pre-condition $\partial\varphi$(N)

We now present a syntactic routine, called SyntacticDiff, in Algorithm 9 for generation of the difference pre-condition $\partial\varphi(N)$. Although this suffices for all our experiments, for the sake of completeness, in Section 6.3.2, we present a more sophisticated algorithm for generating $\partial\varphi(N)$ simultaneously with $\mathsf{Pre}(N)$.

Formally, given $\varphi(N)$, the function SyntacticDiff from Algorithm 9 generates a formula $\partial\varphi(N)$ such that $\varphi(N) \Rightarrow (\varphi(N-1) \odot \partial\varphi(N))$, where $\odot$ is $\wedge$ when $\varphi(N)$ is a universally quantified formula and is $\vee$ when $\varphi(N)$ is a existentially quantified formula. Observe that if such a $\partial\varphi(N)$ exists for universally quantified formulas $\varphi(N)$, then $\varphi(N)$

$\Rightarrow \varphi(N-1)$ must hold. Similarly, if such a $\partial\varphi(N)$ exists for existentially quantified formulas $\varphi(N)$, then $\varphi(N-1) \Rightarrow \varphi(N)$ must hold. Therefore, we can use the validity of $\varphi(N) \Rightarrow \varphi(N-1)$ and $\varphi(N-1) \Rightarrow \varphi(N)$, as a test to decide the existence of $\partial\varphi(N)$ for universally and existentially quantified formulas respectively.

Algorithm 9 incorporates the scenarios described above and boolean combinations thereof. When $\varphi(N)$ is of the syntactic form $\forall i \in \{0 \ldots N\}\ \widehat{\varphi}(i)$, we first check the validity of $\varphi(N) \Rightarrow \varphi(N-1)$ in line 2. If this test fails, we report failure using the **throw** statement in line 3. Otherwise, $\partial\varphi(N)$ is set to $\widehat{\varphi}(N)$ in line 4. Similarly, when $\varphi(N)$ is of the syntactic form $\exists i \in \{0 \ldots N\}\ \widehat{\varphi}(i)$, then $\partial\varphi(N)$ is set to $\widehat{\varphi}(N)$ in line 8, after checking the validity of the $\varphi(N-1) \Rightarrow \varphi(N)$ (line 6). If the test in line 6 fails, again we report failure using the **throw** statement in line 7. When $\varphi(N)$ is of the syntactic form $\varphi^1(N) \wedge \cdots \wedge \varphi^k(N)$, $\partial\varphi(N)$ is computed by taking the difference of each individual conjunct and *disjuncting* them as $\partial\varphi^1(N) \vee \cdots \vee \partial\varphi^k(N)$ (line 10). Note that this operation results in an over-approximation of the difference pre-condition. When $\varphi(N)$ is of the form $\varphi^1(N) \vee \cdots \vee \varphi^k(N)$, $\partial\varphi(N)$ is computed by taking the difference of each individual disjunct as $\partial\varphi^1(N) \vee \cdots \vee \partial\varphi^k(N)$ (line 12). Finally, if $\varphi(N)$ does not belong to any of these syntactic forms (line 13) or if condition 2(a) of Theorem 5.1 is violated by the $\partial\varphi(N)$ computed in this manner (line 15), then we over-approximate $\partial\varphi_N$ by True in lines 14 and 16.

Lemma 5.18 *The difference pre-condition $\partial\varphi(N)$ computed by* SyntacticDiff *is such that (i) $\varphi(N) \Rightarrow (\varphi(N-1) \odot \partial\varphi(N))$, where $\odot$ is $\wedge$ if $\varphi(N)$ is a universally quantified formula, and $\odot$ is $\vee$ if $\varphi(N)$ is an existentially quantified formula, and (ii) P_{N-1} does not modify variables/arrays in $\partial\varphi(N)$.*

Proof. Condition (i) follows from the checks implemented in lines 2 and 6 of function SyntacticDiff. The check in line 15 ensures condition (ii). This concludes the proof.□

Example 5.11 Consider the pre-condition $\varphi(N) := \forall i \in [0, N)\ A[i] = 1 \vee \forall i \in [0, N)\ A[i] = 2$. SyntacticDiff in Algorithm 9 enters the recursive case in line 12. The recursive invocations with inputs $\varphi_1(N) := \forall i \in [0, N)\ A[i] = 1$ and $\varphi_2(N) := \forall i \in [0, N)\ A[i] = 2$ compute the difference pre-conditions $\partial\varphi_1(N) := A[N-1] = 1$ and $\partial\varphi_2(N) := A[N-1] = 2$ respectively. On returning from the recursive case, the algorithm stores the formula $A[N-1] = 1 \vee A[N-1] = 2$ in $\partial\varphi(N)$. □

Example 5.12 Consider the pre-condition $\varphi(N) := \exists i \in [0, N)\ A[i] \geq 100 \wedge \exists j \in [0, N)\ A[j] \leq 10$. The difference pre-condition computed by function SYNTACTICDIFF in Algorithm 9 is $\partial\varphi(N) := A[N-1] \geq 100 \vee A[N-1] \leq 10$. Notice that the computed difference pre-condition is an over-approximation. Had we computed the difference pre-condition as $\partial\varphi(N) := A[N-1] \geq 100 \wedge A[N-1] \leq 10$, then it would have resulted in a contradiction. □

Example 5.13 For pre-condition formulas $\varphi(N) := \forall i \in [0, N)\quad A[i] = N$ and $\varphi(N) := \exists i \in [0, N)\quad A[i] = N$ the validity checks at lines 2 and 6 respectively in Algorithm 9 fail. Hence, the algorithm terminates with out being able to compute an appropriate pre-condition. □

5.6 Verification using Full-Program Induction

In the previous sections, we discussed the algorithms for generating the two components crucial to full program induction: the *difference program* $\partial \mathsf{P}_N$ and the *difference pre-condition* $\partial\varphi(N)$. Before describing the *full-program induction* algorithm, however, we present the strategy for computing the formula $\mathsf{Pre}(N)$ for strengthening pre- and post-conditions.

5.6.1 Generating the Formula $\mathsf{Pre}(\mathbf{N}-\mathbf{1})$

We use Dijkstra's weakest pre-condition computation to obtain $\mathsf{Pre}(N-1)$ after the "difference" pre-condition $\partial\varphi(N)$ and the "difference" program $\partial \mathsf{P}_N$ have been generated. The weakest pre-condition can always be computed using quantifier elimination engines in state-of-the-art SMT solvers like Z3 if $\partial \mathsf{P}_N$ is loop-free. In such cases, we use a set of heuristics to simplify the calculation of the weakest pre-condition before harnessing the power of the quantifier elimination engine. If $\partial \mathsf{P}_N$ contains a loop, it may still be possible to obtain the weakest pre-condition if the loop does not affect the post-condition. Otherwise, we compute as much of the weakest pre-condition as can be computed from the non-loopy parts of $\partial \mathsf{P}_N$, and then try to recursively solve the problem by invoking full-program induction on $\partial \mathsf{P}_N$ with appropriate pre- and post-conditions.

Example 5.14 We apply Dijkstra's weakest pre-condition computation on the Hoare triple from the example in Fig. 5.1. The Hoare triple in Fig. 5.3 shows the strengthened pre- and post-condition formulas. In this example, we use the convention that all variables and arrays of P_{N-1} have the suffix $_Nm1$ (for N-minus-1), while those of P_N have the suffix $_N$. The first application of weakest pre-condition computation generates the formula $B_Nm1[N-2] = (N-1)^3 - (N-2)^3$ on array B in the program. We substitute N with $N-1$ and rename array B_Nm1 to B_N generating the formula $B_N[N-1] = N^3 - (N-1)^3$. We use this formula to strengthen the post-condition of $\mathsf{Peel}(\mathsf{P}_N^p)$ as shown in Fig. 5.3. Effectively, the formula for strengthening the post-condition of P_N can be viewed as naturally lifted to its quantified form $\forall i \in [0, N)\ \ B_N[i] = N^3 - (N-1)^3$.

Re-applying weakest pre-condition computation generates the formula $A_Nm1[N-2] = N^3 - 2 \times (N-1)^3 + (N-2)^3$ on array A in the program. This formula is used to further strengthen the pre-condition as shown in Fig. 5.3. Again, we substitute N with $N-1$ and rename array A_Nm1 to A_N generating the formula $A_Nm1[N-1] = N^3 - 2 \times N^3 + (N-1)^3$. We use this formula to further strengthen the post-condition of $\mathsf{Peel}(\mathsf{P}_N^p)$ as shown in Fig. 5.3. Effectively, the formula for the post-condition of P_N can be viewed as naturally lifted to its quantified form $\forall i \in [0, N)\ \ A_N[i] = (N+1)^3 - 2 \times N^3 + (N-1)^3$.$\square$

5.6.2 The Full-program Induction Algorithm

The basic version of the algorithm for full-program induction is presented as the function FPIVerify-Basic in Algorithm 10. The main steps of this algorithm are: checking conditions 3(a), 3(b) and 3(c) of Theorem 5.1 (lines 1, 20 and 15 resp.), calculating the weakest pre-condition of the relevant part of the post-condition (line 18), and strengthening the pre-condition and post-condition with the weakest pre-condition thus calculated (line 19).

We first check the base case of the analysis (line 1). If the check fails, we have found a valid counter-example and the algorithm terminates in line 3 after reporting the result to the user. We then rename the variables and arrays in the program P_N as well as the pre- and post-conditions using Algorithm 3 and collect the set of glue nodes (line 4). Next, we peel each loop in the renamed program P_N^r and collect the list of peeled nodes (line 5) using Algorithm 4. Then, we compute the set of *affected* variables (line 6) using Algorithm 6. If the number of affected variables (line 7) is non-zero then we report that

Algorithm 10 FPIVERIFY-BASIC(P_N: program, $\varphi(N)$: pre-condition, $\psi(N)$: post-condition)

1: **if** Base case check $\{\varphi(1)\}$ P_1 $\{\psi(1)\}$ fails **then**
2:     **print** "Counterexample found!";
3:     **return** False;
4: $\langle \mathsf{P}^r_N, \varphi(N), \psi(N), \mathsf{GlueNodes}\rangle := \textsc{Rename}(\mathsf{P}_N,\ \varphi(N),\ \psi(N))$; ▷ Renaming as described in Section 5.3.1
5: $\langle \mathsf{P}^p_N, \mathsf{PeelNodes}\rangle := \textsc{PeelAllLoops}(\mathsf{P}^r_N)$;
6: $\mathsf{AffectedVars} := \textsc{ComputeAffected}(\mathsf{P}^p_N, \mathsf{PeelNodes})$;
7: **if** $|\mathsf{AffectedVars}| > 0$ **then**
8:     **return** False; ▷ Unable to prove using full-program induction
9: $\partial\mathsf{P}_N := \textsc{ProgramPeel}(\mathsf{P}^p_N,\ \mathsf{PeelNodes})$;
10: $\partial\varphi(N) := \textsc{SyntacticDiff}(\varphi(N))$;
11: $i := 0$;
12: $\mathsf{Pre}_i(N) := \psi(N)$;
13: $c_\mathsf{Pre}_i(N) :=$ True; ▷ Cumulative conjoined pre-condition
14: **do**
15:     **if** $\{c_\mathsf{Pre}_i(N-1) \wedge \psi(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{c_\mathsf{Pre}_i(N) \wedge \psi(N)\}$ **then**
16:         **return** True; ▷ Assertion verified
17:     $i := i + 1$;
18:     $\mathsf{Pre}_i(N-1) := \textsc{LoopFreeWP}(\mathsf{Pre}_{i-1}(N), \partial\mathsf{P}_N)$; ▷ Dijkstra's WP
19:     $c_\mathsf{Pre}_i(N) := c_\mathsf{Pre}_{i-1}(N) \wedge \mathsf{Pre}_i(N)$;
20: **while** Base case check $\{\varphi(1)\}$ P_1 $\{c_\mathsf{Pre}_i(1)\}$ passes;
21: **return** False; ▷ Failed to prove by full-program induction

we are unable to verify the program using full-program induction (line 8).

We compute the difference program $\partial\mathsf{P}_N$ in line 9 using the function PROGRAMPEEL from Section 5.4 as none of the scalar variables and arrays of interest are in AffectedVars. We then compute the difference pre-condition $\partial\varphi(N)$ in line 10 using the function SYNTACTICDIFF (described in Section 5.5).

The loop in lines 14–20 iteratively checks if the assertion can be proved. We check

the inductive step in line 15. In case the loop terminates via the `return` statement in line 16, the inductive claim has been successfully proved. Otherwise, we compute Dijkstra's weakest pre-condition $\mathsf{Pre}_i(N-1)$ in line 18. Note that, the formula $\mathsf{Pre}_i(N-1)$ strengthens the pre-condition and the same formula $\mathsf{Pre}_i(N)$, but with $N-1$ substituted with N, strengthens the post-condition. The variable $\mathsf{Pre}_i(N)$ is initialized to $\psi(N)$ in line 12. The variable $c_\mathsf{Pre}_i(N-1)$ accumulates the weakest pre-condition formulas in each loop iteration and is initialized to True in line 13. The computed strengthening $\mathsf{Pre}_i(N)$ is conjoined with the variable $c_\mathsf{Pre}_{i-1}(N)$ in line 19. Since the weakest pre-condition computed in every iteration of the loop ($\mathsf{Pre}_i(N-1)$ in line 18) is conjoined to strengthen the inductive pre-condition ($c_\mathsf{Pre}_i(N-1)$ in line 19), it suffices to compute the weakest pre-condition of $\mathsf{Pre}_{i-1}(N)$ (instead of $c_\mathsf{Pre}_i(N) \wedge \psi(N)$) in line 18. Possibly multiple iterations of strengthening of pre- and post-conditions is effected by the loop in lines 14–20. After each strengthening, the base case is checked again in line 20 with the strengthened pre- and post-conditions. If the loop terminates by a violation of the condition in line 20, we report that verification by full-program induction failed.

Theorem 5.2 *Upon successful termination, if algorithm* FPIVerify-Basic *returns* True*, then* $\{\varphi_N\}\ \ \mathsf{P}_N\ \ \{\psi_N\}$ *holds for all* $N \geq 1$*.*

Proof. Verifying the given Hoare triple requires establishing the conditions mentioned in Theorem 5.1. The invocation of the functions Rename and PeelAllLoops in lines 4 and 5 resp. preserves the semantics of the program (refer the Lemmas 5.2 and 5.6 resp.). The function ProgramPeel invoked at line 9 ensures condition 1 of Theorem 5.1 (refer Lemma 5.17). The call to SyntacticDiff at line 10 in FPIVerify-Basic computes the difference pre-conditions that satisfy conditions 2(a) and 2(b) (refer Lemma 5.18). The conditions 3(a) and 3(b) of Theorem 5.1 are checked on lines 1 and 20 respectively. The check at line 15 ensures that the return statement at line 16 executes only when condition 3(c) of Theorem 5.1 is ensured. Hence, from Theorem 5.1, we conclude that $\{\varphi_N\}\ \ \mathsf{P}_N\ \ \{\psi_N\}$ holds for all $N \geq 1$. □

Example 5.15 Consider the Hoare triple in the example from Fig. 5.1. The function FPIVerify-Basic in Algorithm 10 first checks the base-case. The Hoare triple for the base-case check shown in Fig. 5.2 is obtained by substituting N with the value 1. The algorithm then computes the program $\mathsf{Peel}(\mathsf{P}_N)$ as shown in the Hoare triple in Fig. 5.3.

Since the pre-condition in the given Hoare triple (Fig. 5.1) is True, the computed difference pre-condition is also True. The inductive step for this example is not immediately proved. Hence, the algorithm iteratively computes the weakest pre-conditions to strengthen the pre- and post-condition formulas (as described in example 5.14), checking the base-case and the inductive step in each iteration. The given post-condition is proved after two iterations of strengthening the pre- and post-condition using the Hoare triple shown in Fig. 5.3. □

5.7 Conclusion

We presented a novel property-driven verification technique, called *full-program induction*, that performs induction over the entire program via parameter N. Significantly, our analysis obviates the need for loop-specific invariants during verification. The technique automatically computes the difference program as the peels of loops and enclosed branches if any when the variables and arrays of interest are not affected. Interestingly, we only need to ensure the Hoare semantics of the given program under the given pre- and post-condition formulas and do not require full semantic equivalence. The technique computes useful difference pre-conditions during the inductive step of the reasoning for a class of pre-condition formulas. The full-program induction technique is general and can be applied to array-manipulating programs that store integers, matrices, polynomials, vectors and so on. This makes the technique capable of verifying APIs used in machine learning and cryptography libraries.

Next chapter generalizes the difference computation in our technique to verify programs when variables and arrays of interest are identified as affected by the analysis. The computation of the difference pre-conditions is generalized and performed along with the simultaneous strengthening of pre- and post-conditions. The next chapter also presents an implementation of the generalized algorithm in detail as well as the experimental results.

Chapter 6

Generalizing Difference Computation

In this chapter, we extend the full-program induction technique described in Chapter 5 to verify programs that access variables/arrays that are identified as affected. Specifically, we generalize the difference computation algorithm by taking into account the affected variables and arrays. We also present two different extended versions of the full-program induction technique that add to the capabilities of its base version. We present an implementation of the technique in our publicly accessible tool VAJRA [CGU20b] and show its performance vis-a-vis state-of-the-art tools. A part of the work described in this chapter has been published as a conference paper in TACAS 2020 [CGU20a] and as a journal paper in STTT [CGU22].

6.1 Introduction

Programs vary widely in terms of accessing scalar variables and arrays within loops. When none of the variables and arrays of interest are identified as affected (by the function COMPUTEAFFECTED described in Section 5.3.4) then the difference program ($\mathsf{Peel}(\mathsf{P}_N^p)$ computed in Section 5.4.4) consisting of only the peeled iterations of loops and their enclosing branches suffices to prove the given post-condition using full-program induction. However, programs may access variables/arrays that are indeed identified as affected. Hence, the full-program induction algorithm described in Section 5.6.2 may not always allow us to verify assertions in the given program. In this section, we give an overview of the *full-program induction* technique extended with the generalized difference program computation to verify programs that access affected variables/arrays.

We demonstrate the working of the technique by focusing on the programs generated by the grammar in Fig. 3.2. Recall that these programs manipulate arrays of parametric size in sequentially composed, but non-nested loops. We allow a sub-class of quantified and quantifier-free pre- and post-conditions that may depend on the symbolic parameter N. As previously stated, we view the problem as checking the validity of a parameterized Hoare triple $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ for all values of N (> 0), where the program P_N is parameterized in N, size of arrays in P_N is a function of N and N is a free variable in $\varphi(\cdot)$ and $\psi(\cdot)$.

6.1.1 Motivating Examples

We present a couple of examples that motivate the need for generalizing the method for computing the difference programs and use the generalized version to extend the full-program induction technique. The programs in these examples access variables/arrays that are identified as affected by our analysis. We first highlight the challenges in verifying the assertion using full-program induction when the given program accesses affected variables/arrays. Then, we demonstrate the inductive step of the analysis using the difference programs computed using the generalized procedure. The first example illustrates the need for the generalization of the difference program computation due to the presence of "affected" scalar variables. The second example is used as a running example in this chapter. The example illustrates the fact that loops may be retained in the difference program and later simplified to enable the inductive step of our analysis for programs where both arrays and scalars are identified as "affected".

Fig. 6.1 presents a Hoare triple with a simple program and pre- and post-conditions specified using `assume` and `assert` statements respectively. The program P_N in the Hoare triple from Fig. 6.1 performs the summation of the content of `A` into scalars `S1` and `S2` in sequentially composed loops. The post-condition states that `S2 == 7` $\times$ N, for all $N > 0$. Notice that value of `S2` depends on the final value of `S1` that is computed in the peeled iteration of the second loop. Thus, our affected variable analysis (described in Section 5.3.4) will identify variable `S2` as affected. Although the program and the post-condition in Fig. 6.1 are simple, the difference program consisting of only the peeled iterations of loops (as computed in Section 5.4.4) does not suffice. The generalization described in this chapter takes the affected variables into account while computing the

```
// assume(true)

1.  for (int t1=0; t1<N; t1=t1+1) {
2.    A[t1] = 3;
3.  }

4.  S1 = 0;
5.  for (int t2=0; t2<N; t2=t2+1) {
6.    S1 = S1 + A[t2] + 1;
7.  }

8.  S2 = S1;
9.  for (int t3=0; t3<N; t3=t3+1) {
11.   S2 = S2 + A[t3];
12. }

// assert(S2 = 7×N)
```

Figure 6.1: Hoare Triple using an Affected Variable

difference programs. The full-program induction enable with this generalization proves the post-condition in Fig. 6.1 within a few seconds.

As previously described in Chapter 5, full-program induction reduces checking the validity of the Hoare triple in Fig. 6.1 to checking the validity of two "simpler" Hoare triples, represented in Figs. 6.2(a) and 6.2(b). The base case is shown in Fig. 6.2(a), where every loop in the program is statically unrolled a fixed number of times after instantiating the program parameter N to 1. As the induction hypothesis, we assume that the Hoare triple $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$ holds for values of $N > 1$. As previously stated, this assumption does not relate to a specific loop in the program, but to the entire program P_N. For the first motivating example, the induction hypothesis states that the entire Hoare triple in Fig. 6.1, after substituting N with $N-1$, holds.

The inductive step of the reasoning is shown by the Hoare triple in Fig. 6.2(b). In this step, we prove the post-condition, by automatically generating the computation to be

```
// assume(true)

1. A[0] = 3;

2. S1 = 0;
3. S1 = S1 + A[0] + 1;

4. S2 = S1;
5. S2 = S2 + A[0];

// assert(S2 = 7)
```

(a)

```
// assume((N > 1) ∧ S2_Nm1 = 7×(N-1))

1. A[N-1] = 3;

2. S1 = S1_Nm1 + A[N-1] + 1;

3. S2 = S2_Nm1 + (S1 - S1_Nm1);
4. S2 = S2 + A[N-1];

// assert(S2 = 7×N)
```

(b)

Figure 6.2: (a) Base-case Hoare Triple and (b) Inductive-step Hoare Triple

performed after the program with parameter $N-1$ has executed. Since the program P_N defines as well as uses the affected variable S2, the program $\mathsf{Peel}(\mathsf{P}^p_N)$ (described in Section 5.4.4) does not suffice as the difference program $\partial\mathsf{P}_N$. Note that the statement at line 3 in Fig. 6.2(b) "rectifies" the value of the variable S2 using the difference of the value of S1 in programs P_N and P_{N-1}. This statement would be absent from $\mathsf{Peel}(\mathsf{P}^p_N)$. Also note that this statement does not have a syntactic counterpart in Fig. 6.1. In general, there may be multiple statements in the difference program computed during the inductive step that do not have syntactic counterparts in the original program.

Now consider the Hoare triple shown in Fig. 6.3(a) (replicated here from Example 5.1 in Section 5.3). The program updates a scalar variable S and an array variable A. The first loop adds the value of each element in array A to variable S. The second loop adds the value of S to each element of A. The last loop aggregates the updated content of A in S. The pre-condition $\varphi(N)$ is a universally quantified formula on array A stating that each element has the value 1. We need to establish the post-condition $\psi(N)$, which is a predicate on S and N. Note that the post-condition with non-linear terms makes it quite challenging to prove the Hoare triple. State-of-the-art tools like VIAP [RL18], VeriAbs [ACC+20], FreqHorn [FPMG19], Tiler [CGU17], Vaphor [MG16], and Booster [AGS14] are unable to prove the post-condition correct in this example. In contrast, the

```
// assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N; i++) {
3.     S = S + A[i];
4.  }

5.  for(i=0; i<N; i++) {
6.     A[i] = A[i] + S;
7.  }

8.  for(i=0; i<N; i++) {
9.     S = S + A[i];
10. }

// assert(S = N × (N+2))
```

(a)

```
// assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N; i++) {
3.     S = S + A[i];
4.  }

5.  for(i=0; i<N; i++) {
6.     A1[i] = A[i] + S;
7.  }

8.  S1 = S;
9.  for(i=0; i<N; i++) {
10.    S1 = S1 + A1[i];
11. }

// assert(S1 = N×(N+2))
```

(b)

Figure 6.3: (a) Running Example and (b) Its Renamed Version

full-program induction technique with generalized difference computation described in this chapter proves the post-condition in Fig. 6.3(a) correct within a few seconds.

Since the program P_N updates the same scalar variable `S` and the array `A` in multiple sequentially composed loops, we rename the scalars and arrays such that each loop in P_N updates its own copy of scalar variables and arrays using the function RENAME described in Section 5.3.1. This ensures that when P_{N-1} terminates we have access to the values of these variables and arrays after each loop in the program. Example 5.3 of Section 5.3.1 in Chapter 5 gives a detailed account of renaming our running example. For convenience, we replicate the renamed version of the Hoare triple from Fig. 5.14(b) in Fig. 6.3(b). We will use the Hoare triple in Fig. 6.3(b) as our running example to illustrate important aspects of our technique.

The weakest loop invariants needed to prove the post-condition for the program in

Fig. 6.3(b) are: $\forall j \in [0, i)\ (A[j] = 1) \wedge (S = j)$ for the first loop (lines 2-4), $\forall k \in [0, i)\ (A1[k] = N + 1) \wedge (A[k] = 1) \wedge (S = N)$ for the second loop (lines 5-7), and $\forall l \in [0, i)\ (A1[l] = N + 1) \wedge (S1 = l \times (N + 1) + N)$ for the third loop (lines 9-11). Unfortunately, automatically deriving such quantified non-linear inductive invariants for each loop is far from trivial. We now describe how full-program induction verifies the post-condition in this example.

In the base case of our inductive reasoning, we instantiate the parameter N to a small constant value (say $N = 1$). As a result, every loop in the program P_N in Fig. 6.3(b) can be statically unrolled a fixed number of times. The resulting Hoare triple can be easily compiled to a first-order logic formula and verified using an SMT solver. As the induction hypothesis, we assume that the Hoare triple $\{\varphi(N-1)\}\ \ \mathsf{P}_{N-1}\ \ \{\psi(N-1)\}$, shown in Fig. 6.4(a), holds for values of $N > 1$. This Hoare triple is obtained by substituting N with $N - 1$ in the entire Hoare triple in Fig. 6.3(b).

The Hoare triple in Fig. 6.4(b), consisting of the difference program, is computed during the inductive step. Intuitively, the difference program $\partial\mathsf{P}_N$ recovers the effect of the computation in P_N on all scalar variables and arrays after the computation in P_{N-1} has been performed. It includes the iterations of a loop in P_N that are missed by P_{N-1}. When program statements are impervious to the value of N, the values computed in such statements are the same in P_N and P_{N-1}, and hence, they may not need any modification. However, $\partial\mathsf{P}_N$ may contain code to "rectify" values of variables and arrays that have different values at corresponding statements in P_N vis-a-vis P_{N-1}. The code, possibly consisting of loops, to rectify the values of variables and arrays is further simplified whenever possible. Consequently, not all program statements of P_N in Fig. 6.3(b) may have a syntactic counterpart in Fig. 6.4(b) and vice-versa. The inductive step may not be immediately established, in which case we strengthen the pre- and post-conditions using automatically inferred auxiliary predicates as shown in Fig. 6.4(b).

We defer a discussion of how our technique computes the programs used in the inductive step in the Hoare triples shown in Figs. 6.2(b) and 6.4(c) to Section 6.2, where we present algorithms for computation of the difference program (Section 6.2.1) and its simplification (Section 6.2.2).

```
// assume(∀i∈[0,N-1) A[i] = 1)

1.  S_Nm1 = 0;
2.  for(i=0; i<N-1; i++) {
3.    S_Nm1 = S_Nm1 + A[i];
4.  }

5.  for(i=0; i<N-1; i++) {
6.    A1_Nm1[i] = A[i]+S_Nm1;
7.  }

8.  S1 = S;
9.  for(i=0; i<N-1; i++) {
10.   S1_Nm1=S1_Nm1+A1_Nm1[i];
11. }

// assert(S1_Nm1 = (N-1)×(N+1))
```

(a)

```
// assume(N>1 ∧ A[N-1] = 1
// ∧ S1_Nm1 = (N-1)×(N+1)
// ∧ ∀i∈[0,N-1) A1_Nm1[i] = N
// ∧ S_Nm1 = N-1)

1.  S = S_Nm1 + A[N-1];

2.  for(i=0; i<N-1; i++) {
3.    A1[i] = A1_Nm1[i] + 1;
4.  }
5.  A1[N-1] = A[N-1] + S;

6.  S1 = S1_Nm1 + A[N-1];
7.  S1 = S1 + (N-1);
8.  S1 = S1 + A1[N-1];

// assert(S1 = N×(N+2) ∧ S = N
// ∧ ∀i∈[0,N) A1[i] = N+1)
```

(b)

Figure 6.4: (a) Induction Hypothesis Hoare Triple on P_{N-1} and (b) Inductive Step Hoare Triple on $\partial\mathsf{P}_N$ after Simplification and Strengthening

6.1.2 Instantiation of Full-Program Induction in Vajra

We have implemented the full-program induction technique in a prototype tool called VAJRA. Written in C++, the tool is built on top of a compiler framework (LLVM/CLANG [LA04]) and uses an off-the-shelf SMT solver (Z3 [MB08]) at the back-end. Our experiments show that the full-program induction technique is able to solve several difficult problem instances, which other techniques either fail to solve, or can solve only with the help of sophisticated recurrence solvers. VAJRA is significantly more efficient as compared to other tools on a set of benchmarks.

Needless to say, various approaches have their own strengths and limitations, and

the right choice always depends on the problem at hand. Full-program induction is no exception, and despite its several strengths, it has its own limitations, which we discuss in detail in Section 6.5.3.

The full-program induction technique is orthogonal to other verification approaches proposed in literature, making it suitable to be a part of an arsenal of verification techniques. It has already been incorporated within a verification tool, namely VeriAbs [ACC+20]. Since the 2020 edition of the international software verification competition (SV-COMP), VeriAbs invokes full-program induction (via our tool Vajra) in its pipeline of techniques for verifying programs with arrays from the set of benchmarks in the verification competition (refer [ACC+20]).

The main contributions of the chapter can be summarized as follows:

- We elaborate the generalized algorithms for computing the difference program and the difference pre-condition alongside weakest pre-condition computation. We extend the *full-program induction* technique with these generalized algorithms.
- We present generalizations of the full-program induction technique to programs with multiple parameters and loops with increasing and/or decreasing loop counters.
- We present a new algorithm to compute a progress measure, based on the characteristics of the difference program. This gives a measure of how easy it is to prove the inductive step of our technique using constraint solving based techniques like bounded model checking.
- We give rigorous proofs of correctness for the new and improved algorithms. We demonstrate the algorithms using examples.
- We describe a prototype tool Vajra that implements the algorithms for performing full-program induction, using (i) the compiler framework LLVM/CLANG for analysis and transformation of the input program and (ii) an off-the-shelf SMT solver, viz. Z3, at the back-end to discharge verification conditions.
- We present an extensive experimental evaluation on a large suite of benchmarks that manipulate arrays. Vajra outperforms the state-of-the-art tools VIAP, VeriAbs, Booster, Vaphor, and FreqHorn, on the set of benchmark programs.

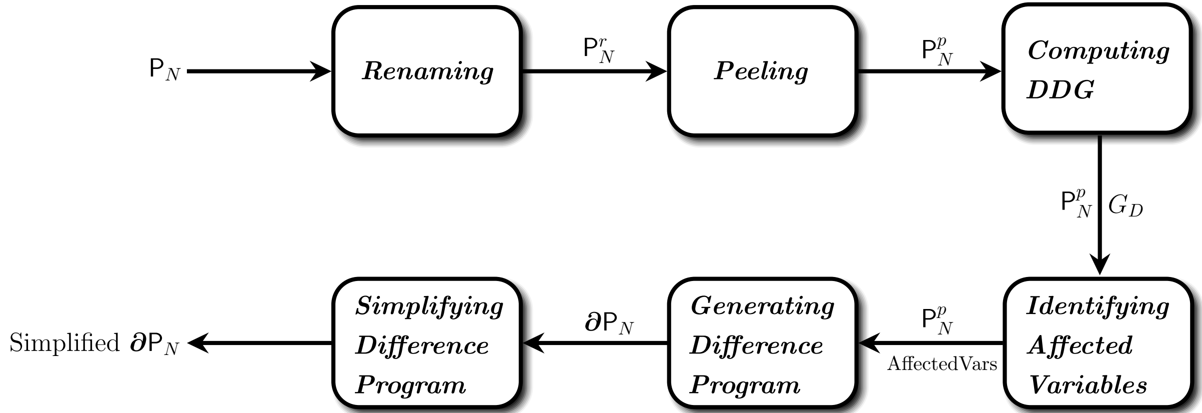


Figure 6.5: Sequence of Steps to Compute the Simplified Difference Program $\partial\mathsf{P}_N$

The remainder of the chapter is organized as follows. Section 6.2 presents the generalized difference program computation algorithm for programs that access affected variables/arrays. In Section 6.3, we extend the full-program induction algorithm to incorporate the generalized difference program computation algorithm. We present two different extensions of the full-program induction algorithm that have a recursive invocation to iteratively simplify the computed difference program and the difference pre-condition. In Section 6.4, we present an algorithm to check whether the recursive application of our technique will eventually be able to verify the given program. Section 6.5 presents the generalizations of full-program induction in different problem settings. In Section 6.6, we present the implementation of our technique in VAJRA, its evaluation on a set of benchmarks and comparison vis-a-vis state-of-the-art tools. We discuss related techniques from literature in Section 6.7. Section 6.8 presents the concluding remarks.

6.2 Computing the Difference Program

In the previous chapter, we proved that using $\mathsf{Peel}(\mathsf{P}^p_N)$ consisting only of peeled iterations of loops and their enclosing branches suffices as the difference program $\partial\mathsf{P}_N$ when the variables/arrays of interest are absent from $\mathsf{AffectedVars}$. However, if some of the variables/arrays of interest are indeed identified as affected by COMPUTEAFFECTED (described in Section 5.3.4), we must include additional code in the difference program that effectively "rectifies" the values of affected variables as computed by P_{N-1}.

Computing the *difference program* $\partial\mathsf{P}_N$ is, in general, a non-trivial task. We build on the ideas used in Section 5.4 and discuss the computation of the difference programs $\partial\mathsf{P}_N$

when variables/arrays of interest are identified as affected. Fig. 6.5 presents a high level overview of the sequence of steps involved in the computation of a difference program. We have already seen the initial steps, including renaming, peeling, computing the data dependence graph and identifying affected variables, in Sections 5.3.1–5.3.4.

We now give an informal overview of the difference program computation. We first move the peels of loops to the end of the program P_N^p and use the information about data dependencies and affected variables computed previously to appropriately stitch and modify them to obtain an unoptimized version of the difference program $\partial\mathsf{P}_N$. In general, this modification may involve adding carefully constructed loops in the difference program itself. It turns out that the difference program obtained in this way can often be significantly optimized using simple optimization techniques. This includes things like pruning superfluous computational steps and accelerating loops among others. We include this optimization as the last step in our flow for generating the difference program. Since peeling a program preserves its semantics (Lemma 5.6), in this section, we use the notations P_N and P_N^p interchangeably to denote the program under analysis. Similarly, we use $\partial\mathsf{P}_N$ and $\partial\mathsf{P}_N^p$ interchangeably to denote the difference program.

For better visualization of the computation of difference programs and to simplify the correctness proofs, we divide the computation of $\partial\mathsf{P}_N$ into a sequence of simple program transformations, similar to the one described in Section 5.4. This sequence of transformations and the programs generated after each transformation are shown in Fig. 6.6. We first canonicalize the peeled program P_N^p to the program T_N^p with linear sequence of guarded statements, using the method described in Section 5.4.1. Now the program T_{N-1} can be obtained by substituting the symbolic parameter N with $N-1$. We now present an intuitive description for the computation of the program T_N^∂, when the variables/arrays of interest in T_N^p are identified as affected.

We compute the program T_N^∂ such that $\{\varphi(N)\}\ \ \mathsf{T}_N^p\ \ \{\psi(N)\}$ holds iff $\{\varphi(N)\}$ T_{N-1}; $\mathsf{T}_N^\partial\ \ \{\psi(N)\}$ holds, where ";" denotes sequential composition. This ensures a parallel between the programs T_N^∂ and $\partial\mathsf{P}_N^p$ as well as the continuity of our technique as described in Section 5.2. We start with the CFG of the peeled program T_N^p. We begin by creating the CFG of T_N^∂ as a copy of the CFG of T_N^p. Since the CFGs of T_N^p and T_N^∂ are a copy of each other, there is a bijection between the nodes in T_N^p and T_N^∂. Next we compute the set of affected variables AffectedVars using the function

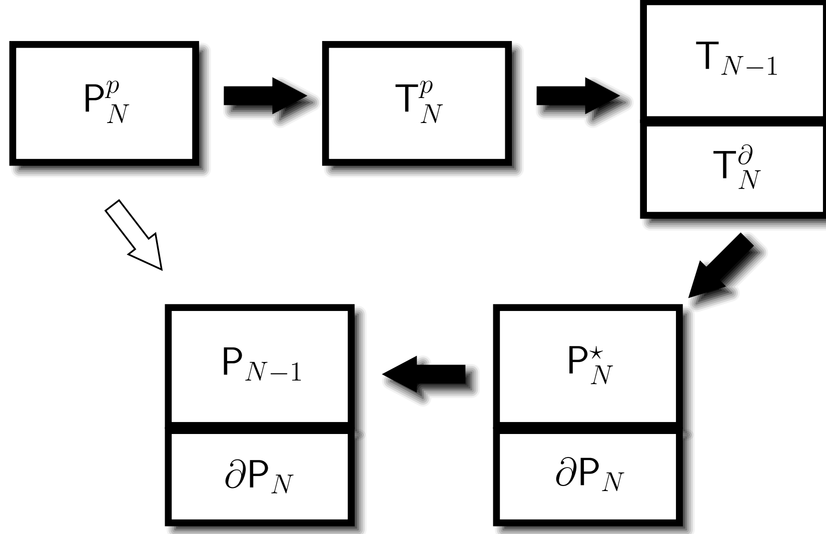


Figure 6.6: Sequence of Program Transformations to Decompose P_N^p into P_{N-1}; ∂P_N

ComputeAffected. We iterate over each node in the CFG of T_N^∂. We retain in the CFG of T_N^∂ all the peeled nodes PeelNodes and their enclosing conditional branch nodes CondNodes (as computed in Section 5.4.4), as well as a subset of the non-peeled nodes. The non-peeled nodes that update an affected variable/array are retained in the CFG, albeit with a modified assignment statement labeled at the node. These modified statements "rectify" the values of affected variables/arrays updated at the retained non-peeled nodes such that these variables/arrays take the same value at the end of programs T_{N-1}; T_N^∂ and T_N^p. We elaborate on how such statements are computed in the next subsection. The non-peeled nodes that do not access any affected variables are removed from the CFG of T_N^∂. When no assignment statements are retained in a loop body, then we remove the loop from the CFG. Notice that while we remove nodes from the CFG of T_N^∂, we do not introduce new nodes during this entire process. As a result, we get a injective mapping from the nodes in T_N^∂ to the nodes in T_N^p.

Since the de-canonicalization of T_N^p to P_N^p (described in Section 5.4.3) is effectively simple and semantics preserving (Lemma 5.16), we de-canonicalize the programs T_{N-1} and T_N^∂ to the programs P_{N-1} and ∂P_N respectively. Hence, the difference computation described above for the canonicalized program T_N^p naturally extends to the program P_N^p. In the next subsection, we describe the computation to generate the modified statements at the retained non-peeled nodes and present an algorithm to compute the difference program ∂P_N. We also present its proof of correctness.

6.2.1 Generating $\partial \mathsf{P_N}$ for Programs with Affected Variables

We propose to construct a difference program whose CFG is "inherited" from the collapsed CFG of the renamed and peeled program P_N^p. We perform simplifications of this control flow structure while computing the difference program $\partial \mathsf{P}_N$. Specifically, all assignment statements that update an affected variable/array but were earlier removed while constructing $\mathsf{Peel}(\mathsf{P}_N^p)$, are retained with a possibly changed expression in the right hand side of the assignment. As a part of the simplifications, some redundant loops and statements may be removed. But the rest of the structure of $\partial \mathsf{P}_N$ stays the same as P_N^p. Specifically, we do not introduce new nodes in the difference program, and there is a natural injective mapping, say γ, from the nodes in the CFG of the difference program $\partial \mathsf{P}_N$ to a unique node in the peeled program P_N^p. Thus, these modifications to the CFG of the difference program $\partial \mathsf{P}_N$ ensure that its control structure is still similar to that of the peeled program P_N^p. As previously stated in Chapter 5, for every non-peeled node in the CFG of P_N^p, there is a corresponding node in the CFG of P_{N-1} (resp. $\mathsf{P}_N^\star$) and vice-versa. This extends the injective mapping from each node in the difference program to a unique node in P_{N-1} (resp. $\mathsf{P}_N^\star$).

To understand how the right hand side expressions of assignments may need to be changed when constructing $\partial \mathsf{P}_N$, consider an execution of each of P_N and P_{N-1}; $\partial \mathsf{P}_N$ starting from the same initial state σ.

Definition 6.1 *For every node n in the CFG of $\partial \mathsf{P}_N$ and for every variable/array element vA we say that vA has a* rectified value *at n if its value at n matches the value of vA at $\gamma(n)$. Otherwise, we say that vA has an* unrectified value *at n.*

For every node n in $\partial \mathsf{P}_N$ that updates an affected variable/array vA of interest, we modify the right hand side of the assignment (if necessary) such that the right hand side expression evaluates to the rectified value of vA at n. This expression is constructed in such a manner that it uses the unrectified value of vA at n in its computation of the rectified value. This construction allows us to establish an important property of the resulting program $\partial \mathsf{P}_N$: every variable/array vA of interest has its rectified value at every node n in $\partial \mathsf{P}_N$.

We now elaborate on how we construct the modified right hand side expression of an assignment statement at node n in $\partial \mathsf{P}_N$ that updates the affected variable/array vA. We

assume that we have access to the rectified and unrectified values of all variables/arrays vA' used in the right hand side expression of the assignment statement at node $\gamma(n)$ in P_N. The easiest way to do this would be to construct the right hand side expression exclusively in terms of the rectified values of vA' at node n. Note that, this results in a difference program that is as complex as the original program P_N. This defeats our purpose, since full-program induction can succeed only if $\partial\mathsf{P}_N$ is "simpler" than P_N. Therefore, we do not use this naive method and present an operator algebra to compute of the rectified value of vA updated in the assignment statement in terms of its unrectified value and the "difference" between the rectified and unrectified values of other variables/arrays that have a data dependence to vA.

Let $\circ$ be a binary operator on a set $\mathcal{S}$ that denotes the domain of values of variables/arrays in P_N. We say that e is the right identity element of $\circ$ if $v \circ e = v$ and e is the left identity element if $e \circ v = v$ for each $v \in \mathcal{S}$. We call $v^{-\circ}$ the right inverse element of v under $\circ$ if $v \circ v^{-\circ} = e$ and we call it the left inverse element if $v^{-\circ} \circ v = e$ for each $v \in \mathcal{S}$. We say that $\circ$ is an associative operator if $(u \circ v) \circ w = u \circ (v \circ w)$. We say that $\circ$ is a commutative operator if $u \circ v = v \circ u$. When the operator $\circ$ is associative, $(u \circ v)^{-\circ}$ $= (v^{-\circ} \circ u^{-\circ})$.

For the following lemmas, we assume that $\circ$ is an associative operator, there exists a left identity element under $\circ$ in $\mathcal{S}$ and each element in $\mathcal{S}$ has a right inverse under $\circ$. Suppose w is an affected variable/array of interest. Let w_{N-1}, u_{N-1}, v_{N-1} denote the values of w, u and v at the end of execution of P_{N-1}. Let w_N, u_N, v_N be the rectified values of w, u and v at node n.

Lemma 6.1 *Let n be a node in $\partial\mathsf{P}_N$ such that the statement at $\gamma(n)$ in P_N is $w := u \circ v$. Then, the rectified value of w is computed as $w_N := w_{N-1} \circ ((v_{N-1})^{-\circ} \circ ((u_{N-1})^{-\circ} \circ u_N) \circ v_N)$.*

Proof. We proceed as follows:

1. $w_N := e \circ w_N$
2. $w_N := (w_{N-1} \circ (w_{N-1})^{-\circ}) \circ w_N$
3. $w_N := w_{N-1} \circ ((w_{N-1})^{-\circ} \circ w_N)$
4. $w_N := w_{N-1} \circ ((u_{N-1} \circ v_{N-1})^{-\circ} \circ (u_N \circ v_N))$

5. $w_N := w_{N-1} \circ ((v_{N-1})^{-\circ} \circ (u_{N-1})^{-\circ} \circ (u_N \circ v_N))$

6. $w_N := w_{N-1} \circ ((v_{N-1})^{-\circ} \circ ((u_{N-1})^{-\circ} \circ u_N) \circ v_N)$ □

Suppose $\circ$ is additionally a commutative operator. Then the following lemmas hold.

Lemma 6.2 *Let n be a node in $\partial \mathsf{P}_N$ such that the statement at $\gamma(n)$ in P_N is $w := u \circ v$. Then, the rectified value of w is computed as $w_N := w_{N-1} \circ (u_N \circ (u_{N-1})^{-\circ}) \circ (v_N \circ (v_{N-1})^{-\circ})$.*

Proof. We continue from the proof of Lemma 6.1 and proceed as follows:

6. $w_N := w_{N-1} \circ ((v_{N-1})^{-\circ} \circ ((u_{N-1})^{-\circ} \circ u_N) \circ v_N)$

7. $w_N := w_{N-1} \circ (((u_{N-1})^{-\circ} \circ u_N) \circ (v_{N-1})^{-\circ} \circ v_N)$

8. $w_N := w_{N-1} \circ ((u_{N-1})^{-\circ} \circ u_N) \circ ((v_{N-1})^{-\circ} \circ v_N)$

9. $w_N := w_{N-1} \circ (u_N \circ (u_{N-1})^{-\circ}) \circ (v_N \circ (v_{N-1})^{-\circ})$ □

To use the equation from Lemma 6.2 for statements with non-commutative operators such as $\{-, \div\}$ often used in practice, we perform a simple transformation that allows us to use commutative operators in-place of non-commutative ones. As an example of this transformation, consider the expressions $u - v$ and u/v. We transform them into the expressions $u + (-v)$ and $u \times (1/v)$ respectively. This allows us to use the equation in Lemma 6.2 when for every element $v \in \mathcal{S}$, the elements $-v$ and $1/v$ are also in $\mathcal{S}$.

Lemma 6.3 *Let n be a node in $\partial \mathsf{P}_N$ such that the statement at $\gamma(n)$ in P_N is $w := w \circ v$. Then, the rectified value of w is computed as $w_N := w_N \circ (v_N \circ (v_{N-1})^{-\circ})$ along with the presumption $w_N = w_{N-1}$.*

Proof. Using the given presumption $w_N = w_{N-1}$, we have the definition of the identity element as: $e = w_N \circ (w_N)^{-\circ} = w_N \circ (w_{N-1})^{-\circ}$.

As the first step, we use the result from Lemma 6.2 and proceed as follows:

1. $w_N := w_{N-1} \circ (w_N \circ (w_{N-1})^{-\circ}) \circ (v_N \circ (v_{N-1})^{-\circ})$

2. $w_N := w_{N-1} \circ e \circ (v_N \circ (v_{N-1})^{-\circ})$

3. $w_N := w_{N-1} \circ (v_N \circ (v_{N-1})^{-\circ})$

4. $w_N := w_N \circ (v_N \circ (v_{N-1})^{-\circ})$ □

It is worth noting that the rectification described in Lemmas 6.1, 6.2 and 6.3 applies not only when the set $\mathcal{S}$ is integers, i.e. integers are stored as array elements but even when the set consists of *matrices, vectors, and polynomials*. When matrices are stored as array elements, such arrays are called *tensors*. These are extensively used in machine learning algorithms. Further, it applies to interesting operators such as $\oplus$, $+$ *mod* x, $\times$ *mod* y as well as to other interesting algebraic structures. It is also worth mentioning that for a restricted class of programs our technique extends to computing the differences of programs that manipulate heaps.

The routine ProgramDiff presented in Algorithm 11 shows how the difference program is computed. In line 1, we peel each loop in the program P_N and collect the list of peeled nodes using function PeelAllLoops from Algorithm 4 (refer Section 5.3.2). Then, in line 2, we compute the set of affected variables using function ComputeAffected from Algorithm 5 (refer Section 5.3.4). The difference program $\partial\mathsf{P}_N$ inherits the skeletal structure of the peeled program P^p_N after peeling each loop (line 3). Next, we collapse all nodes and edges in the body of each loop into a single node identified with the loop-head in the CFG of P^p_N using the function CollapseLoopBody in line 4. The collapsed CFG of the resulting program $\partial\mathsf{P}_N$ is a DAG with finitely many paths. We then initialize a worklist of CFG nodes with n_{start} in line 5.

The while loop in lines 7–29 performs a breadth-first top-down traversal over the DAG of $\partial\mathsf{P}_N$ starting from the node n_{start} and processes one node at a time. We first remove a node n from the worklist in line 8. We store the nodes that are already processed by our algorithm in Processed (that is initialized in line 6). We add the node n removed from the worklist to Processed in line 9. Next, the loop in lines 10–11 appends each successor n' of n to the worklist that is not already processed. We use the routine Succ to obtain the list of successors of node n.

If n is a peeled node, then we retain it as is in the difference program (line 13). Otherwise, we check if any scalar variable/array used at node n is affected at line 14. We have defined the sub-routine HasAffectedVars that checks if the scalar variable/array defined at node n is affected.

Algorithm 11 PROGRAMDIFF($(Locs, CE, \mu)$: renamed program P_N, GlueNodes: set of glue nodes)

1: $\langle (Locs^p, CE^p, \mu^p), \mathsf{PeelNodes} \rangle :=$ PEELALLLOOPS($(Locs, CE, \mu)$);
2: AffectedVars := COMPUTEAFFECTED($(Locs^p, CE^p, \mu^p)$, PeelNodes);
3: $\partial\mathsf{P}_N := (Locs', CE', \mu')$, where $Locs' := Locs^p$, $CE' := CE^p$, and $\mu' := \mu^p$;
4: $\partial\mathsf{P}_N :=$ COLLAPSELOOPBODY($\partial\mathsf{P}_N$);
   ▶ `Function` COLLAPSELOOPBODY `collapses the nodes and edges of each loop into its loop-head. Now the (collapsed) CFG of` $\partial\mathsf{P}_N$ `is a DAG.`
5: WorkList := (n_{start}); ▷ Add the start node to the worklist
6: Processed := $\emptyset$;
7: **while** WorkList is not empty **do**
8:     Remove a node n from head of WorkList;
9:     Processed := Processed $\cup \{n\}$;
10:     **for** each node $n' \in$ SUCC(n) $\setminus$ Processed **do**
11:         WorkList := APPENDTOLIST(WorkList, n');
12:     **if** $n \in$ PeelNodes **then**
13:         **continue**; ▷ Difference computation not required
14:     **else if** HASAFFECTEDVARS(n, AffectedVars) **then**
15:         **if** $n \in$ GlueNodes **then**
16:             **continue**; ▷ Retain the glue loop
17:         **else if** n is a loop-head **then**
18:             $L :=$ UNCOLLAPSELOOPBODY(n);
   ▶ `Function UnCollapseLoopBody restores the collapsed nodes and edges of each loop.`
19:             **for** each node $n' \in$ NODES(L) **do**
20:                 **if** HASAFFECTEDVARS(n', AffectedVars) **then**
21:                     $\mu'(n') :=$ NODEDIFF(n', μ, AffectedVars);
22:                 **else**
23:                     $\partial\mathsf{P}_N :=$ REMOVENODE(n', $\partial\mathsf{P}_N$); ▷ No affected variables at node n'
24:         **else**

```
25:         μ'(n) := NodeDiff(n, μ, AffectedVars);
26:     else                                              ▷ No affected variables at node n
27:         CondNodes := Set of all branch nodes with a peeled node in its scope;
28:         if n ∉ CondNodes then
29:             ∂P_N := RemoveNode(n, ∂P_N);
30: return ∂P_N;
```

```
HasAffectedVars( n: node, AffectedVars: set of affected variables )
1: if ∃vA such that vA ∈ def(n) and vA ∈ AffectedVars then
2:     return True;
3: else
4:     return False;
```

```
NodeDiff( n: node, μ: node labeling function, AffectedVars: set of affected variables )
1: if μ(n) is of the form w_N := r_N^1 op r_N^2 then
2:     return w_N := w_{N-1} ∘ (r_N^1 ∘ (r_{N-1}^1)^{-∘}) ∘ (r_N^2 ∘ (r_{N-1}^2)^{-∘});    ▷ Refer Lemma 6.2
3: else if μ(n) is of the form w_N := w_N op r_N^1 wherein w_N is a scalar then
4:     return w_N := w_N ∘ (r_N^1 ∘ (r_{N-1}^1)^{-∘});                                  ▷ Refer Lemma 6.3
5: else                                                 ▷ μ(n) is a conditional statement C_N
6:     if (∃v s.t. v ∈ uses(n) and v ∈ AffectedVars) ∨ (C_N ≠ C_{N-1} is satisfiable) then
7:         throw "Branch conditions in P_N and P_{N-1} may not evaluate to same value";
8:     else
9:         return μ(n);
```

For nodes n that refer to an affected variable/array, we do the following. We check if a node n is a glue node that refers to an affected variable/array in line 16 and retain such nodes as is in the difference program. Otherwise, we check if the node n corresponds to a loop-head in line 17. We uncollapse the nodes corresponding to a loop-head that represent the entire loop in line 18. We assume that the sub-routine Nodes(L) returns the set of CFG nodes in loop L. Next, the loop in lines 19–23 iterates over all nodes n' in the body of L and process one node at a time. In line 20, we check if the variable/array updated at node n' is affected using function HasAffectedVars, and compute its

rectified value in line 21, using the function NodeDiff. If the variable/array defined at n' is not identified as affected, then we remove from $\partial\mathsf{P}_N$ nodes n' that do not update an affected variable/array using the routine RemoveNode in line 23. For a non-peeled node n that does not correspond to a loop-head, we compute the rectified value of an affected variable/array defined at n in line 25, using the function NodeDiff.

For nodes n that are not peeled nodes and do not update an affected variable/array, we do the following. We compute the set CondNodes of conditional branch nodes that have at least one peeled node within its scope in line 27. In line 29, we remove from $\partial\mathsf{P}_N$ nodes n (including collapsed loop nodes) that do not update an affected variable/array and are not in the set CondNodes, using the routine RemoveNode, as they do not need any rectification.

The sub-routine NodeDiff computes the statements that rectify values of variables/arrays updated at a node. It determines the type of statement (assignment, aggregation or branch condition) at the given node and acts accordingly. For assignment statements, we compute the rectified value as shown in Lemma 6.2 and for aggregating statements, we compute the rectified value as shown in Lemma 6.3. For the nodes representing a conditional branch in $\partial\mathsf{P}_N$, we determine if its conditional expression evaluates to the same value in P_N and P_{N-1}. If so, the conditional branch is retained as is in $\partial\mathsf{P}_N$. Otherwise, currently our technique cannot compute $\partial\mathsf{P}_N$ and we report a failure using the **throw** statement.

To explain the intuition behind the steps of Algorithm 11, we use the convention that all variables and arrays of P_{N-1} have the suffix `_Nm1` (for N-minus-1), while those of P_N have the suffix `_N`. This allows us to express variables/arrays of P_N in terms of the corresponding variables/arrays of P_{N-1} in a systematic way in $\partial\mathsf{P}_N$, given that the intended composition is P_{N-1}; $\partial\mathsf{P}_N$.

For assignment statements, we compute the rectified values as follows. For every assignment statement of the form `v = E;` in L, a corresponding statement is generated in $\partial\mathsf{P}_N$ that expresses `v_N` in terms of `v_Nm1` and the difference (or ratio) between versions of variables/arrays that appear as sub-expressions in `E` in P_{N-1} and P_N.

While the implementation is currently restricted to simple arithmetic operators $(+, -, \times, \div)$, specifically for the ease of implementation and its use in practice, as previously stated, our rectification method is general and applies to several operators beyond

the ones mentioned here. The following example shows the computation of rectified values of variables/arrays updated in simple program statements.

Example 6.1 The statement `A_N[i] = B_N[i] + v_N;` in P_N gives rise to the statement `A_N[i] = A_Nm1[i] + (B_N[i] + (- B_Nm1[i])) + (v_N + (- v_Nm1));` in $\partial\mathsf{P}_N$ that rectifies the value of `A_N[i]`. Similarly, the statement `A_N[i] = B_N[i] * v_N;` in P_N gives rise to the statement `A_N[i] = A_Nm1[i] * (B_N[i] * (1/B_Nm1[i])) * (v_N * (1/v_Nm1));` under the assumption `B_Nm1[i] * v_Nm1` $\neq 0$. □

The program P_N may have statements that aggregate/accumulate values in scalars. This kind of statement requires special processing when generating the difference program $\partial\mathsf{P}_N$. The next example shows the computation of rectified values of variables/arrays in statements that accumulate values in scalar variables.

Example 6.2 Consider the loop `for(i=0; i<N; i++)` { `sum_N = sum_N + A_N[i];` } in program P_N. The difference `A_N[i] + (- A_Nm1[i])` is aggregated over all indices from 0 through $N-2$. In this case, the loop in $\partial\mathsf{P}_N$ that rectifies the value of `sum_N` has the following form: `sum_N = sum_Nm1; for (i=0; i<N-1; i++)` { `sum_N = sum_N + (A_N[i] + (- A_Nm1[i]));` }. A similar aggregation for multiplicative ratios can also be shown. □

Conditional branch statements pose a considerable challenge to the computation of difference programs. A branch condition may evaluate to different outcomes in P_N and P_{N-1}, for the same value of N. When this happens, programs P_N and P_{N-1} execute totally unrelated blocks of statements. In such situations, it is immensely difficult to rectify the values of variables/arrays computed along the unrelated branches, and hence, our algorithm avoids doing so. Only when we can determine that the condition evaluates to the same value in P_N and P_{N-1}, we rectify values of variables/arrays computed along the corresponding branches. Next we present examples with branch conditions to highlight this.

Example 6.3 Consider the conditional branch statement `if(t3 == 0)` (as shown in line 10 of Fig. 5.1 in Chapter 5). The branch condition evaluates to the same value in P_N and P_{N-1} because the condition has no dependence on N. Therefore, the branch statement is used as is during the computation of the difference program. However, recall

that since the arrays accessed in the program are not affected, none of the loops are retained in the difference program (shown in Fig. 5.3).

Consider another conditional branch statement `if(A[i] == N)` in P_N. The corresponding statement in P_{N-1} is `if(A[i] == N-1)`. Clearly, the conditions in these statements do not evaluate to the same value in P_N and P_{N-1}. Thus, our algorithm flags a failure to compute the difference program and terminates. □

```
x = N;
y = N;
for(i=0; i<N; i++) {
  if(x == y)
    A[i] = i;
}
```

(a)

```
x = N-1;
y = N-1;
for(i=0; i<N-1; i++) {
  if(x == y)
    A[i] = i;
}
```

(b)

Figure 6.7: Example Program (a) P_N and (b) P_{N-1}

There are programs where conditional branch statements with dependence on N evaluate to the same value. For example, consider the program P_N in Fig. 6.7(a). The program P_{N-1} is shown in Fig. 6.7(b). While the branch condition (indirectly) depends on the value of N, it evaluates to the same value in P_N and P_{N-1}, since the amount of change in the value of variables `x` and `y` used in the branch condition is same. Our algorithm can successfully compute the difference program in such cases.

The restriction on branch conditions that use affected variables/arrays can be further relaxed by handling the case when the condition evaluates to True in P_{N-1} and to False in P_N by restoring the values of variables/arrays to their values at the predecessor of the branch node. However, when a branch condition evaluates to False in P_{N-1} and to True in P_N, the entire computation within the branch has to be performed again in $\partial\mathsf{P}_N$ instead of just executing the rectification code. For example, consider the branch statement, `if(i < N) Loop;`. If the branch condition `i` $<$ `N-1` in P_{N-1} evaluates to False, then the condition `i` $<$ `N` in P_N definitely evaluates to True. This will require the difference program to execute the entire computation performed by the code fragment `Loop;` and not just the difference program corresponding to `Loop;`. This will hamper the progress

guarantees on the class of programs that our technique can verify. Hence, we currently avoid handling these cases in the algorithms and consider them as a part of future work.

We now prove the soundness of the routine ProgramDiff from Algorithm 11. For the following lemma, we assume that $\partial\mathsf{P}_N$ is the difference program computed when function ProgramDiff is invoked on the renamed program P_N. Suppose both P_N^p and $\mathsf{P}_{N-1};\ \partial\mathsf{P}_N$ are executed from the same initial state σ. We assume $\pi : (n_0, n_1, \ldots, n_k)$ to be the path in the CFG of $\partial\mathsf{P}_N$ corresponding to the execution of the difference program from the state obtained after P_{N-1} has terminated (in $\mathsf{P}_{N-1};\ \partial\mathsf{P}_N$). We assume $\pi' : (n'_0, n'_1, \ldots, n'_k)$ to be the corresponding path in the CFG of the peeled program P_N^p.

Lemma 6.4 *For every node n_j in π, the rectified values of all variables/array elements used at n_j during the execution of $\partial\mathsf{P}_N$ are identical to the values of the same variables/array elements at the corresponding node n'_j during the execution of P_N^p.*

Proof. If vA is not identified as an affected variable/array by function ComputeAffected, the result follows from the proof of Lemma 5.17 and the no-overwriting property of renaming.

If vA is identified as an affected variable, we induct on the length of π. The only difference in this case is that we also need to consider the assignment statements modified by function NodeDiff at lines 21 and 25 of function ProgramDiff. Lemmas 6.1, 6.2 and 6.3 guarantee the correctness of the rectified value of vA computed by these additional statements, given the unrectified value of vA and rectified and unrectified values of all variables and array elements used in the right hand side of the assignment. By the inductive hypothesis, the rectified values of the latter set of variables and array elements are indeed available. By the no-overwriting property, the unrectified values of vA and all other variables and arrays used in the right hand side of the assignment are also available. Therefore, the correct rectified value of vA is computed at each node in π.

Finally, note that once a rectified value is generated at a non-glue node in the difference program, renaming ensures that it is not re-defined by subsequent statements in the difference program. Therefore, rectified values, once computed in the difference program, are available for use at subsequent nodes in the execution path. Putting the above parts together completes the proof. □

Theorem 6.1 *$\partial \mathsf{P}_N$ generated by* PROGRAMDIFF *is such that, for all $N > 1$, $\{\varphi(N)\}$ P_{N-1}; $\partial \mathsf{P}_N$ $\{\psi(N)\}$ holds iff $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ holds.*

Proof. Lemma 5.6 guarantees that $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ holds iff $\{\varphi(N)\}$ P^p_N $\{\psi(N)\}$ holds. Furthermore, Lemma 6.4 ensures that for every state σ satisfying $\varphi(N)$, if we execute P^p_N and P_{N-1}; $\partial \mathsf{P}_N$ starting from σ, then if the final state after termination of P^p_N satisfies $\psi(N)$, so does the final state after termination of P_{N-1}; $\partial \mathsf{P}_N$. This proves the theorem. □

```
// assume(∀i∈[0,N) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<N-1; i++) {
3.    S = S + A[i];
4.  }
5.  S = S + A[N-1];

6.  for(i=0; i<N-1; i++) {
7.    A1[i] = A[i] + S;
8.  }
9.  A1[N-1] = A[N-1] + S;

10. S1 = S;
11. for(i=0; i<N-1; i++) {
12.   S1 = S1 + A1[i];
13. }
14. S1 = S1 + A1[N-1];

// assert(S1 = N × (N+2))
```

(a)

```
// assume(∀i∈[0,N) A[i] = 1)

1.  S = S_Nm1 + A[N-1];

2.  for(i=0; i<N-1; i++) {
3.    A1[i] = A1_Nm1[i] + (S - S_Nm1);
4.  }
5.  A1[N-1] = A[N-1] + S;

6.  S1 = S1_Nm1 + (S - S_Nm1);
7.  for(i=0; i<N-1; i++) {
8.    S1 = S1 + (A1[i] - A1_Nm1[i]);
9.  }
10. S1 = S1 + A1[N-1];

// assert(S1 = N × (N+2))
```

(b)

Figure 6.8: (a) Peeled Program and (b) Difference Program

Example 6.4 We illustrate the difference computation performed by the routine ProgramDiff in Algorithm 11 on the peeled program shown in Fig. 6.8(a) (replicated from Fig. 5.15 of Section 5.3.2 in Chapter 5). Fig. 6.8(b) shows the difference program obtained after executing the algorithm on the peeled program. Notice that while some program statements in Fig. 6.8(b) are syntactically similar to corresponding statements in Fig. 6.8(a), the additional statements (e.g. at lines 3, 6 and 8) in Fig. 6.8(b) have no syntactic counterpart in Fig. 6.8(a). For the first loop, since variable `S` is not affected, only the peeled iteration is retained. Since `A1` and `S1` both are affected, the statements within the second and the third loop are replaced with difference statements that rectify their values, along with inserting the peeled statements for both loops. Initialization of variable `S1` is also replaced with its difference statement since it depends on variable `S`.□

6.2.2 Simplifying the Difference Program

While we have described a simple strategy to generate a difference program $\partial\mathsf{P}_N$ above, this may lead to unoptimized as well as redundant statements in the naively generated difference program. Our implementation aggressively optimizes $\partial\mathsf{P}_N$ and removes redundant code, renaming variables/arrays as needed. The routine SimplifyDiff in Algorithm 12 simplifies program statements that compute rectified values, removes redundant loops from the difference program and substitutes loops with the summarized statements computed using acceleration. This helps in $\partial\mathsf{P}_N$ having fewer and simpler loops in a lot of cases. Below, we describe these optimizations and illustrate them using examples.

Since the generation of statements that compute rectified values is not fully optimized, these statements may have expressions that can be further simplified using the values computed in other statements in the generated difference program $\partial\mathsf{P}_N$. The function Simplify performs this optimization aggressively and simplifies the statements in the difference program (lines 3–4 in Algorithm 12). Let us take an example to illustrate the effect of the Simplify function.

Example 6.5 Suppose the difference program $\partial\mathsf{P}_N$ has statements of the form `B_N[i] = B_Nm1[i] + expr1;` and `v_N = expr2*v_Nm1;`. If `expr1` and `expr2` are constants or functions of N and loop counters, then expressions such as `(B_N[i] - B_Nm1[i])` and `(v_N/v_Nm1)` can often be simplified from the statements in the difference program. The

Algorithm 12 $\textsc{SimplifyDiff}((Locs, CE, \mu)$: difference program $\partial\mathsf{P}_N)$

1: $\partial\mathsf{P}'_N := (Locs', CE', \mu')$, where $Locs' := Locs$, $CE' := CE$, and $\mu' := \mu$;
2: **for** each loop $L \in \textsc{Loops}(\partial\mathsf{P}'_N)$ **do**
3:     **for** each node $m \in \textsc{Nodes}(L)$ **do**
4:         $\mu'(m) := \textsc{Simplify}(\mu'(m))$; ▷ Simplify the statement
5:     $(n_1, n, c) := \textsc{IncomingEdge}(L)$; ▷ c is the label of the edge from n_1 to n
6:     $(n, n_2, \mathbf{ff}) := \textsc{ExitEdge}(L)$;
7:     **if** body of L is of the form $w_N := w_N$ `op` $expr$, wherein w_N is a scalar variable **then**
8:         $n_{acc} = \textsc{FreshNode}()$;
9:         **if** `op` $\in \{+, -\}$ **then**
10:             $\mu'(n_{acc}) := (w_N := w_N$ `op` $\textsc{Simplify}(k_L(N-1) \times expr))$; ▷ Accelerated statement
11:         **else if** `op` $\in \{\times, \div\}$ **then**
12:             $\mu'(n_{acc}) := (w_N := w_N$ `op` $\textsc{Simplify}(expr^{k_L(N-1)}))$; ▷ Accelerated statement
13:         **else**
14:             **throw** "Specified operator not handled";
15:         $CE' := CE' \cup \{(n_1, n_{acc}, c), (n_{acc}, n_2, \mathsf{U})\} \setminus \{(n_1, n, c), (n, n_2, \mathbf{ff})\}$;
16:         $Locs' := Locs' \cup \{n_{acc}\} \setminus \textsc{Nodes}(L)$;
17:     **if** body of L is of the form $w_N := w_{N-1}$ or $w_N := w_N$ **then** ▷ Remove redundant loops
18:         $CE' := CE' \cup \{(n_1, n_2, c)\} \setminus \{(n_1, n, c), (n, n_2, \mathbf{ff})\}$;
19:         $Locs' := Locs' \setminus \textsc{Nodes}(L)$;
20: **return** $\partial\mathsf{P}'_N$;

expression `expr1` is substituted for `B_N[i] - B_Nm1[i]` and `expr2` for `v_N/v_Nm1` respectively. □

The difference program $\partial\mathsf{P}_N$ may contain loops that perform redundant computation, for example, copying values across versions of an array corresponding to P_N and P_{N-1}, due to the simplification of the statements that compute its rectified value. We remove such

loops from $\partial\mathsf{P}_N$ in lines 17–19 of Algorithm 12. Let us illustrate this with an example.

Example 6.6 Suppose the difference program $\partial\mathsf{P}_N$ has the loop `for(i=0; i<N-1; i++) { A_N[i] = A_Nm1[i]; }` where `A_N` is not used subsequently in the program. Such loops can be removed from the difference program, as these loops only copy values from the version of array `A` in P_{N-1} to its version in P_N, and hence, are redundant. □

The difference program $\partial\mathsf{P}_N$ may also contain loops that compute values of variables that can be accelerated. We perform this optimization in lines 7–16 of SIMPLIFYDIFF in Algorithm 12. We first check if the body of a loop L is in the specific form eligible for this optimization in line 7. If so, we create a fresh node in line 8 to replace L. Lines 9–14 of Algorithm 12 label the fresh node with the accelerated statement. If we encounter operators that are not supported, then we report a failure of our technique using the **throw** statement in line 14. Next, we replace the loop with the fresh node in lines 15–16. We demonstrate this optimization with the following example.

Example 6.7 Suppose the difference program has the loop `for(i=0; i<N-1; i++) { sum = sum + 1; }`. The semantics of the loop can be summarized using the accelerated statement `sum = sum + (N-1);`. SIMPLIFYDIFF removes this loop from the program and introduces the accelerated statement instead. □

In the following lemma, we use $\partial\mathsf{P}'_N$ to denote the program generated by SIMPLIFYDIFF.

Lemma 6.5 $\{\varphi(N)\}\ \mathsf{P}_{N-1};\ \partial\mathsf{P}'_N\ \{\psi(N)\}$ *holds iff* $\{\varphi(N)\}\ \mathsf{P}_{N-1};\ \partial\mathsf{P}_N\ \{\psi(N)\}$ *holds.*

Proof. Follows trivially from the fact that SIMPLIFYDIFF in Algorithm 12 optimizes program statements, removes only redundant statements/loops, and replaces loops using semantically equivalent accelerated statements. □

Example 6.8 We illustrate the application of the simplification routine SIMPLIFYDIFF from Algorithm 12 on our running example. The program in Fig. 6.9 is obtained after simplification of $\partial\mathsf{P}_N$ in Fig. 6.8. The difference terms are replaced with the simplified expressions from the difference program itself. Notice that the loop that rectifies the value of `S1` is accelerated and the statement obtained after this optimization is shown in line 7. □

```
// assume(∀i∈[0,N) A[i] = 1)

1.  S = S_Nm1 + A[N-1];

2.  for(i=0; i<N-1; i++) {
3.    A1[i] = A1_Nm1[i] + 1;
4.  }
5.  A1[N-1] = A[N-1] + S;

6.  S1 = S1_Nm1 + A[N-1];
7.  S1 = S1 + (N-1);
8.  S1 = S1 + A1[N-1];

// assert(S1 = N × (N+2))
```

Figure 6.9: Simplified Difference Program

6.3 Extensions to Full-Program Induction

In this section, we discuss two different versions of the *full-program induction* algorithm that expand its capabilities beyond the base version FPIVerify-Basic presented in Section 5.6.2 of Chapter 5.

6.3.1 The Recursive Full-program Induction Algorithm

The full-program induction algorithm presented as routine FPIVerify in Algorithm 13 is an extended version of Algorithm 10 from Section 5.6.2. The important steps of the algorithm include checking conditions 3(a), 3(b) and 3(c) of Theorem 5.1 (lines 1, 23 and 12), calculating the weakest pre-condition of the relevant part of the post-condition (line 15), recursively invoking our routine FPIVerify with the strengthened pre- and post-conditions (line 18), and accumulating the the weakest pre-condition predicates thus calculated for strengthening the pre- and post-conditions (line 22). We now discuss the algorithm in detail.

We first check the base case of the analysis in line 1. The base case of our induction

Algorithm 13 FPIVERIFY(P_N: program, $\varphi(N)$: pre-condition, $\psi(N)$: post-condition)

1: **if** Base case check $\{\varphi(1)\}$ P_1 $\{\psi(1)\}$ fails **then**
2:     **print** "Counterexample found!";
3:     **return** False;
4: $\langle \mathsf{P}_N, \varphi(N), \psi(N), \mathsf{GlueNodes}\rangle :=$ RENAME(P_N, $\varphi(N)$, $\psi(N)$); ▷ Renaming as described in Section 5.3.1
5: $\partial\varphi(N) :=$ SYNTACTICDIFF($\varphi(N)$);
6: $\partial\mathsf{P}_N :=$ PROGRAMDIFF(P_N, GlueNodes);
7: $\partial\mathsf{P}_N :=$ SIMPLIFYDIFF($\partial\mathsf{P}_N$); ▷ Simplify and Accelerate loops
8: $i := 0$;
9: $\mathsf{Pre}_i(N) := \psi(N)$;
10: $c_\mathsf{Pre}_i(N) :=$ True; ▷ Cumulative conjoined pre-condition
11: **do**
12:     **if** $\{c_\mathsf{Pre}_i(N-1) \wedge \psi(N-1) \wedge \partial\varphi(N)\}$ $\partial\mathsf{P}_N$ $\{c_\mathsf{Pre}_i(N) \wedge \psi(N)\}$ **then**
13:         **return** True; ▷ Assertion verified
14:     $i := i + 1$;
15:     $\mathsf{Pre}_i(N-1) :=$ LOOPFREEWP($\mathsf{Pre}_{i-1}(N), \partial\mathsf{P}_N$); ▷ Dijkstra's WP sans WP-for-loops
16:     **if** no new $\mathsf{Pre}_i(N-1)$ obtained **then** ▷ Can happen if $\partial\mathsf{P}_N$ has a loop
17:         **if** CHECKPROGRESS(P_N, $\partial\mathsf{P}_N$) **then**
18:             **return** FPIVERIFY($\partial\mathsf{P}_N$, $c_\mathsf{Pre}_{i-1}(N-1) \wedge \psi(N-1) \wedge \partial\varphi(N)$, $c_\mathsf{Pre}_{i-1}(N) \wedge \psi(N)$);
19:         **else**
20:             **return** False; ▷ Failed to prove by full-program induction
21:     **else**
22:         $c_\mathsf{Pre}_i(N) := c_\mathsf{Pre}_{i-1}(N) \wedge \mathsf{Pre}_i(N)$;
23: **while** Base case check $\{\varphi(1)\}$ P_1 $\{c_\mathsf{Pre}_i(1)\}$ passes;
24: **return** False; ▷ Failed to prove by full-program induction

reduces to checking a Hoare triple for a loop-free program, as previously explained (in Section 5.2). This is achieved by compiling the pre-condition, the program and the post-

condition into a first-order logic formula. The validity of this formula can be checked using an off-the-shelf back-end SMT solver, such as Z3. If the check fails, we have found a valid counter-example and the algorithm terminates in line 3 after reporting the result to the user.

Next, we rename the variables and arrays in the program P_N as well as the pre- and post-conditions (using Algorithm 3 described in Section 5.3.1) and collect the set of glue nodes (line 4). Then, in line 5, we compute the difference pre-condition $\partial\varphi(N)$ using the function SyntacticDiff (described in Section 5.5). We then compute the difference program $\partial\mathsf{P}_N$ in line 6 using the function ProgramDiff from Section 6.2. Note that this function can compute the difference program when the scalar variables and arrays of interest are identified as affected (refer Section 5.3.4 from Chapter 5 for our method of identifying affected variables). In line 7, we simplify the statements in the computed difference program, remove redundant statements and try to accelerate loops, if any, using the routine SimplifyDiff from Algorithm 12.

Unlike the base version of the algorithm, the loop in lines 11–23, that iteratively checks if the assertion can be proved, has a recursive invocation in line 18. Once the base case succeeds, we check the inductive step in line 12. If the loop terminates via the `return` statement in line 13, the inductive claim has been successfully proved. Otherwise, in line 15, we compute Dijkstra's weakest pre-condition of the formula $\mathsf{Pre}_i(N)$, over the difference program. The formula $\mathsf{Pre}_i(N)$ is initialized to $\psi(N)$ in line 9. We denote the computed weakest pre-condition as $\mathsf{Pre}_i(N-1)$. Note that, the formula $\mathsf{Pre}_i(N-1)$ strengthens the pre-condition and the same formula $\mathsf{Pre}_i(N)$, but with N substituted for $N-1$, strengthens the post-condition. The variable $c_\mathsf{Pre}_i(N-1)$, initialized to True in line 10, accumulates weakest pre-condition formulas in each loop iteration (line 22).

In case no further weakest pre-conditions can be generated, as checked in line 16, we recursively invoke FPIVerify on $\partial\mathsf{P}_N$ in line 18. Prior to recursive invocation, we check if it will be beneficial using function CheckProgress, in line 17. Discussion about the routine CheckProgress is deferred to Section 6.4. The recursive invocation helps in situations where the computed difference program $\partial\mathsf{P}_N$ has loops. To present an example of this scenario, we modify the program in Fig. 5.1 by having the statement `C[t3] = N;` (instead of `C[t3] = 0;`) in line 10. In this case, $\partial\mathsf{P}_N$ retains a loop that rectifies the value of `C[t3]` corresponding to its computation in the third loop in Fig. 5.1. The recursive

invocation of full-program induction on $\partial \mathsf{P}_N$ as input for the mentioned example results in a loop-free difference program. If the check at line 17 reports that further application of full-program induction will not yield any benefits then we report the failure of our technique in line 20.

When weakest pre-condition computation succeeds, we conjoin the computed strengthening predicate $\mathsf{Pre}_i(N)$ with the variable $c_\mathsf{Pre}_{i-1}(N)$ in line 22. Since the weakest pre-condition ($\mathsf{Pre}_i(N-1)$ in line 15) computed in every iteration of the loop is conjoined to strengthen the inductive pre-condition ($c_\mathsf{Pre}_i(N-1)$ in line 22), it suffices to compute the weakest pre-condition of $\mathsf{Pre}_{i-1}(N)$ (instead of $c_\mathsf{Pre}_i(N) \wedge \psi(N)$) in line 15. Possibly multiple iterations of the loop in lines 11–23 are required to strengthen the pre- and post-conditions. After each iteration, the base case is checked again in line 23 with the strengthened pre- and post-conditions. If the loop terminates due to violation of the base-case with the strengthened post-condition (line 23), we report the failure of our method by returning False in line 24.

Lemma 6.6 *Upon successful termination, if function* FPIVerify *returns* True*, then* $\{\varphi_N\}\ \mathsf{P}_N\ \{\psi_N\}$ *holds for all* $N \geq 1$.

Proof. Verifying the given Hoare triple requires establishing the conditions mentioned in Theorem 5.1. The function Rename in line 4 preserves the semantics of the program (refer Lemma 5.2). The functions ProgramDiff invoked at line 6 and SimplifyDiff invoked at line 7 ensure condition 1 of Theorem 5.1 (refer Theorem 6.1 and Lemma 6.5). The call to SyntacticDiff in line 5 in FPIVerify computes the difference pre-conditions that satisfy conditions 2(a) and 2(b) (refer Lemma 5.18). The conditions 3(a) and 3(b) of Theorem 5.1 are checked in lines 1 and 23 respectively. The check in line 12 ensures that the return statement in line 13 executes only when condition 3(c) of Theorem 5.1 is ensured. Similarly, the statement in line 18 returns True only if the recursive call to FPIVerify proves all conditions in Theorem 5.1. Hence, we conclude that $\{\varphi_N\}\ \mathsf{P}_N\ \{\psi_N\}$ holds for all $N \geq 1$. □

Example 6.9 We execute function FPIVerify on the example, shown in Fig. 6.3(b). The algorithm first checks the base-case by instantiating N with value 1 as shown in the Hoare triple from Fig. 6.10(a). Then the scalar and array variables are renamed

```
// assume(∀i∈[0,1) A[i] = 1)

1.  S = 0;
2.  for(i=0; i<1; i++) {
3.    S = S + A[i];
4.  }

5.  for(i=0; i<1; i++) {
6.    A[i] = A[i] + S;
7.  }

8.  for(i=0; i<1; i++) {
9.   S = S + A[i];
10. }

// assert(S = 3)
```

(a)

```
// assume(A[N-1] = 1)              //∂φ(N)
// assume(S1_Nm1 = (N-1)×(N+1)) //ψ(N-1)
// assume(∀i∈[0,N-1) A1_Nm1[i] = N)
// assume(S_Nm1 = N-1)

1.  S = S_Nm1 + A[N-1];

2.  for(i=0; i<N-1; i++) {
3.    A1[i] = A1_Nm1[i] + 1;
4.  }
5.  A1[N-1] = A[N-1] + S;

6.  S1 = S1_Nm1 + A[N-1];
7.  S1 = S1 + (N-1);
8.  S1 = S1 + A1[N-1];

// assert(S1 = N×(N+2))          //ψ(N)
// assert(∀i∈[0,N) A1[i] = N+1)
// assert(S = N)
```

(b)

Figure 6.10: (a) Base-case (b) Inductive Step with Strengthening of Pre- and Post-conditions

using function Rename (described in Algorithm 3 from Section 5.3.1). We then compute the difference pre-condition using function SyntacticDiff in Algorithm 9 on the given pre-condition $\varphi(N) := \forall i \in [0, N)\ A[i] = 1$. The difference pre-condition computed as $\partial\varphi(N) := A[N-1] = 1$ is shown in Fig. 6.10(b) using an assume statement. The difference program computed using the function ProgramDiff as shown in Fig. 6.8 is further simplified using the routine SimplifyDiff as shown in Fig. 6.9. The simplified difference program is used in the inductive step as shown in Fig. 6.10(b). The algorithm then checks the inductive step after assuming the post-condition $\psi(N-1)$ from the inductive hypothesis. We use a back-end SMT solver to check the satisfiability of the inductive step. The check does not immediately succeed, indicating that we need a stronger pre-condition.

The algorithm then employs Dijkstra's weakest pre-condition computation (described in Section 5.6.1) to strengthen the pre- and post-conditions simultaneously. The first application of weakest pre-condition computation generates pre-condition on array `A1` as the formula $A1[N-1] = N+1$. This is naturally lifted to its quantified form $\forall i \in [0, N)$ $A1[i] = N+1$ and used to simultaneously strengthen the post-condition. We substitute N with $N-1$ and rename the array to get the formula $\forall i \in [0, N) \quad A1_Nm1[i] = N$, which is used to strengthen the pre-condition. The algorithm again checks the base-case which succeeds but is unable to establish the inductive step of the analysis. We re-apply the weakest pre-condition computation and generate the predicates $S = N$ and $S_Nm1 = N-1$ that further strengthen the pre- and post-conditions. At this point the base-case and the inductive step succeed proving the given post-condition using full-program induction. The Hoare triple in Fig. 6.10(b) shows the difference pre-condition $\partial\varphi(N)$, post-condition $\psi(N)$, formula $\psi(N-1)$ from the induction hypothesis as well as the strengthened pre- and post-condition formulas. □

6.3.2 Full-Program Induction with Formula Decomposition

While the algorithm FPIVerify suffices for all of our experiments, it may not always be the case. Specifically, even if $\partial \mathsf{P}_N$ is loop-free, the analysis may exit the loop in lines 11–23 of FPIVerify by violating the base case check in line 23. To handle (at least partly) such cases, we propose the following strategy. Whenever a (weakest) pre-condition $\mathsf{Pre}_i(N-1)$ is generated, instead of using it directly to strengthen the current pre- and post-conditions, we "decompose" it into two formulas $\mathsf{Pre}'_i(N-1)$ and $\partial\varphi'_i(N)$ with a two-fold intent: (a) potentially weaken $\mathsf{Pre}_i(N-1)$ to $\mathsf{Pre}'_i(N-1)$, and (b) potentially strengthen the difference formula $\partial\varphi(N)$ to $\partial\varphi'_i(N) \wedge \partial\varphi(N)$. The checks for these intended usages of $\mathsf{Pre}'_i(N-1)$ and $\partial\varphi'_i(N)$ are implemented in lines 3, 4, 5, 11 and 16 of routine FPIDecomposeVerify, shown as Algorithm 14. This routine is meant to be invoked as FPIDecomposeVerify(i) after each iteration of the loop in lines 11–23 of routine FPIVerify (so that $\mathsf{Pre}_i(N)$, $c_\mathsf{Pre}_i(N)$ etc. are initialized properly). In general, several "decompositions" of $\mathsf{Pre}_i(N)$ may be possible, and some of them may work better than others. FPIDecompseVerify permits multiple decompositions to be tried through the use of the functions NextDecomposition and HasNextDecomposition. The names of both these functions intuitively indicates their meaning. Lines 19–22 of FPIDecom-

Algorithm 14 FPIDecomposeVerify(i : integer)

1: **do**
2: $\langle \mathsf{Pre}'_i(N-1), \partial\varphi'_i(N)\rangle := \textsc{NextDecomposition}(\mathsf{Pre}_i(N-1))$;
3: Check if (a) $\partial\varphi'_i(N) \wedge \mathsf{Pre}'_i(N-1) \Rightarrow \mathsf{Pre}_i(N-1)$,
4: (b) $\varphi(N) \Rightarrow \varphi(N-1) \odot (\partial\varphi'_i(N) \wedge \partial\varphi(N))$, ▷ where $\odot \in \{\wedge, \vee\}$
5: (c) P_{N-1} does not update any variable or array element in $\partial\varphi'_i(N)$
6: **if** any check in lines 3 - 5 fails **then**
7: **if** $\textsc{HasNextDecomposition}(\mathsf{Pre}_i(N-1))$ **then**
8: **continue**;
9: **else**
10: **return** False;
11: **if** $\{c_\mathsf{Pre}_{i-1}(N-1) \wedge \psi(N-1) \wedge \mathsf{Pre}_i(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{c_\mathsf{Pre}_{i-1}(N) \wedge \psi(N) \wedge \mathsf{Pre}'_i(N)\}$ **then**
12: **return** True; ▷ Assertion verified
13: **else**
14: $c_\mathsf{Pre}_i(N) := c_\mathsf{Pre}_{i-1}(N) \wedge \mathsf{Pre}'_i(N)$;
15: $\mathsf{Pre}_{i+1}(N-1) := \textsc{LoopFreeWP}(\mathsf{Pre}'_i(N), \partial\mathsf{P}_N)$; ▷ Dijkstra's WP sans WP-for-loops
16: **if** $\{\varphi(1)\}\ \mathsf{P}_1\ \{c_\mathsf{Pre}_i(1) \wedge \mathsf{Pre}_{i+1}(1)\}$ holds **then**
17: $prev_\partial\varphi(N) := \partial\varphi(N)$;
18: $\partial\varphi(N) := \partial\varphi'_i(N) \wedge \partial\varphi(N)$;
19: **if** $\textsc{FPIDecomposeVerify}(i+1)$ **then** ▷ Recursive invocation
20: **return** True; ▷ Assertion verified
21: **else**
22: $\partial\varphi(N) := prev_\partial\varphi(N)$;
23: $i := i + 1$;
24: **while** $\textsc{HasNextDecomposition}(\mathsf{Pre}_i(N-1))$;
25: **return** False; ▷ Failed to prove by full-program induction

poseVerify implement a simple back-tracking strategy, allowing a search of the space of decompositions of $\mathsf{Pre}_i(N-1)$. Observe that when we use FPIDecomposeVerify, we simultaneously compute a difference formula $(\partial\varphi'_i(N) \wedge \partial\varphi(N))$ and an inductive pre-

condition $(c_\mathsf{Pre}_{i-1}(N) \wedge \mathsf{Pre}'_i(N))$.

Lemma 6.7 *Upon successful termination, if function* FPIDecomposeVerify *returns* True*, then* $\{\varphi_N\}\ \ \mathsf{P}_N\ \ \{\psi_N\}$ *holds for all* $N \geq 1$.

Proof. The conditions mentioned in Theorem 5.1 are a pre-requisite to verifying the given Hoare triple. Condition 1 of Theorem 5.1 is ensured by difference computation (functions ProgramDiff and SimplifyDiff) in FPIVerify. Conditions 2(a) and 2(b) are established in FPIVerify (via the call to function SyntacticDiff) and the checks on lines 3–5 in FPIDecomposeVerify ensure that these conditions continue to hold. Further, FPIVerify also ensures conditions 3(a) and 3(b) before it invokes FPIDecomposeVerify. Now, the check in line 11 in FPIDecomposeVerify ensures condition 3(c) of Theorem 5.1. Similarly, the statement in line 20 in FPIDecomposeVerify returns True only if the recursive call to FPIDecomposeVerify proves all the conditions in Theorem 5.1. Hence, we conclude that $\{\varphi_N\}\ \ \mathsf{P}_N\ \ \{\psi_N\}$ holds for all $N \geq 1$. □

6.4 Checking Progress

Recall from Section 6.3.1 that given the parameterized Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$, our technique recursively computes difference programs until the given post-condition $\psi(N)$ is proved. The difference computation must eventually result in programs $\partial\mathsf{P}_N$ that can be easily verified without the need of further applying inductive reasoning or indicate otherwise. In this section, we define a progress measure that can be used to check if the difference computation will eventually simplify the program to the extent that it can be verified using a back-end SMT solver. The measure is based on the characteristics of the difference programs computed by our technique.

Ranking functions have been traditionally used to show program termination [DGG00, CS01, CS02, PR04]. We use the notion of ranking functions to measure the progress that our technique has made towards verifying the given post-condition using the difference programs. Several different criteria have been used in the literature to define ranking functions. Our ranking function links with each difference program a value from a well-founded domain. We assign the minimal rank to programs that can be effectively verified, for example using a back-end SMT solver. A difference program gets a smaller rank compared to another difference program if it is "closer" (in a natural way) to programs that

can be proved. Here, we list some criteria that can be used for defining the ranking function for our technique based on the syntactic changes in difference programs vis-a-vis the given program.

The main hurdle in proving the given Hoare Triple, $\{\varphi(N)\} \quad \mathsf{P}_N \quad \{\psi(N)\}$, are the loops in the given program P_N. Once the difference program is loop-free, the post-condition in such programs can be easily verified by an SMT solver and our technique is no longer required to recursively apply induction any further for proving such programs. Thus, the difference programs for which our technique terminates are loop-free programs and programs in which loops can be accelerated or optimized away with known techniques. Hence, reduction in the number of loops in the difference program $\partial\mathsf{P}_N$ vis-a-vis the given program P_N is the main criterion to measure progress in our technique.

Further, the difference computation can potentially reduce the dependence on the value of N. For programs with expressions that do not directly or indirectly [1] rely on N, the difference program consists of only the peeled iterations of loops. Clearly, when the difference program $\partial\mathsf{P}_N$ is impervious to the value of N, additional code to rectify the values of variables is no longer required in the subsequent recursive invocations. This indicates that we have made progress. We thus use the presence of variables in the program whose value directly or indirectly depends on the value of N as another criteria to measure progress. As previously stated in Section 5.3.4, if the value of a variable/array depends on N or on a value computed in a peeled statement, then we call such variables/arrays as *affected variables/arrays*. Our technique computes the set of affected variables during each recursive attempt to verify the post-condition. The difference program must rectify the values of these affected variables/arrays. When the difference program has no affected variables, the verification attempt can be terminated after the next invocation of our technique.

We also consider the complexity of expressions in the program and use it as a criteria for measuring progress. For every expression appearing in assignment statements, its expression complexity can be defined in many possible ways. Once this complexity is defined for expressions, we can take the maximum complexity of all the expressions as the expression complexity of the entire program. For programs with polynomial expressions,

[1] By indirect dependence, we mean the dependence via another value computed in a peeled or non-peeled statements in the program.

we can use the highest degree of the affected variables/arrays in the expression as the expression complexity. Similarly, several other criteria can be used to define the expression complexity. These include nesting levels of array indices, size/weight of the expression trees in $\partial \mathsf{P}_N$ vis-a-vis P_N, number of variables, operators and constants in the expressions and so on. It is worth pointing out that such notions have been previously studied in term rewriting systems [Der82, Les82].

Note that each criterion discussed so far, including the number of loops, the number of affected variables and the expression complexity, is well-founded. Hence, the domain of values represented by their Cartesian product is also well-founded and represents a lexicographic ordering on the difference programs computed by our method. Progress is guaranteed if each recursive invocation of our technique in the cycle reduces this measure assigned by such a ranking function. We argue that the cycle of recursive invocations to our technique must eventually terminate, as there are no infinite descending chains of elements in the well-founded domain. We present an algorithm that can compute values from this domain on the fly and return the result of the comparison between the computed quantities. Note that no user intervention is required for checking progress.

Algorithm 15 CHECKPROGRESS(P_N: program, $\partial \mathsf{P}_N$: difference program)

1: $\mathsf{LoopList} := \text{LOOPS}(\mathsf{P}_N)$;
2: $\mathsf{LoopList}' := \text{LOOPS}(\partial \mathsf{P}_N)$;
3: **if** $\#\mathsf{LoopList}' < \#\mathsf{LoopList}$ **then**
4:     **return** True;
5: $\mathsf{AffectedVars} := \text{COMPUTEAFFECTED}(\mathsf{P}_N)$;
6: $\mathsf{AffectedVars}' := \text{COMPUTEAFFECTED}(\partial \mathsf{P}_N)$;
7: **if** $\#\mathsf{AffectedVars}' < \#\mathsf{AffectedVars}$ **then**
8:     **return** True;
9: $\mathsf{EC} := \text{EXPRESSIONCOMPLEXITY}(\mathsf{P}_N)$;
10: $\mathsf{EC}' := \text{EXPRESSIONCOMPLEXITY}(\partial \mathsf{P}_N)$;
11: **if** $\mathsf{EC}' < \mathsf{EC}$ **then**
12:     **return** True;
13: **return** False;

The routine CHECKPROGRESS in Algorithm 15 is used for checking progress after the

difference program is computed. The algorithm is based on the change in the number of loops, number of affected variables and the expression complexity of the given program P_N vis-a-vis the difference program $\partial\mathsf{P}_N$. First, we compute the number of loops in programs P_N and $\partial\mathsf{P}_N$. We compare the number of loops in P_N and $\partial\mathsf{P}_N$ in line 3. If the difference program has fewer loops than P_N, then we return True concluding that the $\partial\mathsf{P}_N$ is simpler to verify than the given program. Note that, we do not consider the glue loops in the difference program that were introduced during the renaming step to copy values across versions. If the number of loops does not decrease in an invocation of our technique, we check if the number of affected variables has decreased. We compute the set of affected variables in P_N and $\partial\mathsf{P}_N$ using the routine ComputeAffected from Algorithm 6. In line 7, we compare the number of affected variables in both the programs. The algorithm returns True if the difference program has fewer affected variables than P_N. Subsequently, we check if the expressions in the difference program $\partial\mathsf{P}_N$ are "simpler", and easier to reason with, than P_N. We assume the availability of a routine ExpressionComplexity that can compute this complexity measure for programs P_N and $\partial\mathsf{P}_N$. In line 11 we check if the expression complexity of the difference program $\partial\mathsf{P}_N$ is less than that of the given program P_N, in which case the algorithm returns True. If none of these criteria are met, then the algorithm returns False.

Lemma 6.8 *If* CheckProgress *in Algorithm 15 returns* True*, then the difference program $\partial\mathsf{P}_N$ is "simpler" to verify (using the full-program induction technique) as compared to the given program P_N.*

Proof. The difference program $\partial\mathsf{P}_N$ has strictly less loops than P_N when the check in line 3 is satisfied. In this case, verifying $\partial\mathsf{P}_N$ is simpler than verifying P_N. Further, reduction in the number of affected variables/arrays means less code is retained to rectify their values. Hence, when the check on line 7 is satisfied, $\partial\mathsf{P}_N$ is simpler than P_N. By Lemma 5.17, when none of the variables/arrays of interest are identified as affected, only the peeled iterations of loops (referred as $\mathsf{Peel}(\mathsf{P}_N)$) suffice as the difference program $\partial\mathsf{P}_N$. This also makes verifying $\partial\mathsf{P}_N$ simpler as compared to P_N. Similarly, the last condition ensures that the expressions in the difference program are easier to reason with than the given program P_N. Further, these characteristics of the program and the ordering among them as specified by CheckProgress forms a lexicographic ranking function [PR04]. Hence,

these quantities are bound to reduce with each application of our technique, making the difference program simpler to verify each time. This concludes the lemma. □

Lemma 6.9 *The routine* FPIVERIFY *in Algorithm 13 eventually terminates.*

Proof. Function FPIVERIFY presented in Algorithm 13 can execute in infinite recursion only when the invocation of CHECKPROGRESS in line 17 returns True infinitely often. From difference program computation, we know that the number of loops and affected variables/arrays in the difference program $\partial\mathsf{P}_N$ never increase beyond their counts in the given program P_N, they either decrease or remain the same. Further, if the expression complexity of all the statements that update an affected variable/array does not decrease then our method returns False, and consequently we report failure. Thus, these three characteristics with the specified ordering among them form a lexicographic ranking function [PR04]. Since the value of the lexicographic ranking function strictly decreases in each recursive application of our method, it ensures that function FPIVERIFY eventually terminates. □

6.5 Generalization to Different Problem Settings

For brevity and ease of explanation, we have presented our technique in simple settings. We have so far considered Hoare triples that have a single parameter N. In this section, we show how our technique can be adapted to Hoare triples with multiple parameters as well as peeling loops in different directions for our inductive reasoning. We also state the limitations of our technique.

Based on the ideas previously described, our technique can already verify several interesting scenarios in programs. Our technique can verify programs that manipulate arrays of different sizes as well as loops with non-uniform termination conditions that are a linear function of N. It does so by computing a (possibly different) *peel count* for each loop that manipulates different arrays. For the ease of presentation, our algorithm computes the rectified values of variables/arrays in statements with a single operator. When the program statements have two or more operators, such statements can be split into multiple statements, by introducing temporary variables such that each statement has a single operator, and then computing the difference program using our algorithm.

6.5.1 Multiple Independent Program Parameters

Consider proving Hoare triples with multiple parameters $N_1, N_2, ..., N_k$. Suppose that the values of these parameters are independent of each other. Verifying Hoare triples for all values of these parameters can be done by inducting on one program parameter at a time while keeping the other parameters fixed. We explain this with the help of a simple example with two parameters. To prove that the Hoare Triple $\{\varphi(N_1, N_2)\}$ P_{N_1,N_2} $\{\psi(N_1, N_2)\}$ for all $N_1 \geq a \wedge N_2 \geq b$, we prove the following three sub-goals. First, in the base-case we prove that the triple $\{\varphi(a, b)\}$ $\mathsf{P}_{a,b}$ $\{\psi(a, b)\}$ holds. Second, induction over the parameter N_1, where we assume the Hoare Triple $\{\varphi(k, N_2)\}$ P_{k,N_2} $\{\psi(k, N_2)\}$ holds with $k \geq a \wedge N_2 \geq b$, and prove the Hoare Triple $\{\varphi(k+1, N_2)\}$ P_{k+1,N_2} $\{\psi(k + 1, N_2)\}$, treating N_2 as a symbolic parameter unchanged during the induction. Third, induction over the parameter N_2, where we assume that the Hoare Triple $\{\varphi(N_1, l)\}$ $\mathsf{P}_{N_1,l}$ $\{\psi(N_1, l)\}$ holds where $N_1 \geq a \wedge l \geq b$, and prove the Hoare Triple $\{\varphi(N_1, l + 1)\}$ $\mathsf{P}_{N_1,l+1}$ $\{\psi(N_1, l + 1)\}$, treating N_1 as a symbolic parameter unchanged in the induction. This can be easily extended to Hoare triples with more than two parameters. For programs that manipulate arrays of different independent sizes, we treat each variable representing the symbolic size of arrays as a parameter. As described above, our technique verifies such programs by inducting on each parameter one at a time.

6.5.2 Handling Loops with Increasing/Decreasing Counters

Recall that the difference program $\partial\mathsf{P}_N$ is sequentially composed with P_{N-1}. Earlier, we have been peeling the last iterations of the loops in P_N so that P_{N-1} and P_N have the same number of iterations in each loop. However, there are programs where peeling the last iterations of loops may not be possible such that our technique can compute a difference program. In such cases, we may need to peel the initial iterations of the loops in the program. As an example, consider a loop where the value of the loop counter decreases in each iteration. A possible way is to rotate these loops to fit the template of loops defined in our grammar and then apply our technique. However, not all loops are such that they can be rotated easily using the standard loop transformation techniques. For such loops, we may need to peel it at the beginning. We peel the initial iterations of these loops and add the code that rectifies values of variables computed in the loop after the peeled

iterations such that P_{N-1}; $\partial\mathsf{P}_N$ is semantically equivalent to P_N. Thus, by peeling initial iterations of loops when computing the difference program, our technique can be easily adapted to programs with such loops in a sound way.

6.5.3 Limitations

There are several scenarios under which the full-program induction technique may not produce a conclusive result.

Program computation with side-effects may make it difficult to compute the difference program such that there is a clear separation between the program P_{N-1} and the rest of the computation that can make up P_N. Computation that results in side-effects includes I/O operations, allocation, de-allocation and modification of heap memory and other operations that modify the environment which is not local to the given program. When the given program is not free of such side-effects, our technique may not be able to decompose it into P_{N-1} and $\partial\mathsf{P}_N$. Note that we only disallow the computation that impacts the post-condition to be proved. In our experience, a large class of array-manipulating programs are naturally free of side-effects. In particular, the programs discussed in this chapter (including Fig. 5.1) and those used for experimentation are free of side-effects.

Currently our technique is unable to verify programs with branch conditions that are dependent on the parameter N. Computing the difference program becomes cumbersome in such cases. This stems from the fact that the branch condition may evaluate to different outcomes in P_N and P_{N-1}, for the same value of N, and hence, may require us to compute the difference of two arbitrary pieces of code blocks. We identify such cases while computing the difference program in Algorithm 11 and suspend our verification attempt on line 7 of the routine NodeDiff. Note that this does not include the loop conditions, which are handled by peeling the loop. To illustrate this case, consider the Hoare triple shown in Fig. 6.11. The first loop in the program initializes array `A` and the second loop updates array `A` within a branch statement with the conditional expression `N%2 == 0`. It is easy to see that this branch condition will evaluate to different outcomes in P_N and P_{N-1}. As a result, it is difficult for our technique to compute a difference program. Invariant generation techniques may be better suited for verifying this example. The weakest loop invariants needed to prove the post-condition in this example are: $\forall j \in [0, i)\ (A[j] = 0)$ for the first loop and $\forall k \in [0, i)\ (A[k]\%2 = N\%2)$ for the second loop.

```
// assume(true)

1.  for(i=0; i<N; i++) {
2.    A[i] = 0;
3.  }

5.  for(i=0; i<N; i++) {
6.    if( N%2 == 0 ) {
7.      A[i] = A[i] + 2;
8.    } else {
9.      A[i] = A[i] + 1;
10.   }
11. }

// assert(∀i ∈ [0,N) A[i]%2 = N%2)
```

Figure 6.11: Challenge Example

The difference program includes all peeled iterations of P_N that are missed in P_{N-1}. Hence, our technique needs to know the symbolic upper bound on the value of the loop counter to be able to compute the number of iterations to be peeled from the program. Further, when programs have loops with non-linear termination conditions, the construction of the difference program becomes challenging. The number of peeled iterations itself may be a function of N and possibly result in a loop in the difference program. For example, consider a loop in P_N with the counter i initialized to 0 and the loop termination condition "$i < N^2$". The corresponding loop in P_{N-1} has the same initialization but the termination condition is "$i < (N-1)^2$". "$2 \times N + 1$" iterations must be peeled from this loop. For such loop conditions, an entire loop appears as the peel in the difference program. Since the number of iterations to be peeled is not a constant number, currently while computing this peel (in line 7 of Algorithm 4), our technique reports a failure to handle such programs. Further, our grammar restricts the shape of loops that can be verified using our technique. Most notably, we analyze programs with only non-nested loops. We have designed a variant of the full-program induction technique [CGU21b]

that can verify a class of programs with nested loops. The technique greatly simplifies the computation of difference programs. It infers and uses relations between two slightly different versions of the program during the inductive step. We refer the interested reader to Chapter 7 that describes the relational full-program induction technique [CGU21b].

The inductive reasoning may remain inconclusive when the rank of the difference programs, as defined in Sect. 6.4, does not reduce during the successive invocations to verify the post-condition using our technique. Continuous reduction in the rank/progress measure is crucial to the success of full-program induction. When no progress is observed, we suspend the verification attempt in line 20 of the routine FPIVerify in Algorithm 13. Though the ranking functions can be defined in many possible ways, there are programs that pose a challenge in computing the difference program in a way that the rank of the computed difference program does not reduce. However, such programs are rarely seen in practice.

Our technique may fail to verify a correct program if the heuristics used for weakest pre-condition either fail or return a pre-condition that causes violation of the base-case checked on line 23 of the routine FPIVerify in Algorithm 13.

Apart from the conceptual limitations mentioned above, our prototype implementation has a few limitations. We currently support expressions in assignment statements with only $\{+, -, \times, \div\}$ operators. In the implementation we support a single program parameter and peel only the last iterations of loops. Despite all these limitations, our experiments show that full-program induction performs remarkably well on a large suite of benchmarks.

6.5.4 Relation to Techniques in Compilers

In compilers, the polyhedral analysis and scalar evolution (refer Section 2.6) are widely used techniques, especially for parallelization and program optimization. In this section, we present the strengths and weaknesses of such techniques vis-a-vis full-program induction and explore possible synergies with our technique.

Strengths

As stated in the previous section, our grammar restricts the shape of loops that can be verified using our technique, disallowing non-nested loops. The primary reason for this

```
// assume(true)
1. S = 0;
2. for(i=0; i<N; i++) {
3.   A[i] = 0;
4. }
5. for(k=0; k<N; k++) {
6.   S = k;
7.   for(l=0; l<N; l++) {
8.     S = S + 1;
9.   }
10.  A[k] = A[k] + S;
11.}
// assert(∀x∈[0,N) A[x] = N + x)
```

Figure 6.12: An Example with a Nested Loop

restriction are the challenges associated with the computation of difference programs. We present an example of a program with a nested loop that is out of scope for the full-program induction technique.

Consider the Hoare triple in Fig. 6.12. The first loop in the program initializes array A. The second loop in the program is a nested loop, where the inner loop computes a recurrence on a scalar variable S that is reinitialized at the beginning of each iteration of the outer loop. The value of S is added to each element of A in the corresponding iteration of the outer loop. The pre-condition of this program is true; the post-condition asserts that each element of array A has the value N. The nested loop in this program is a bottleneck for the computation of the difference program, hindering the application of full-program induction.

Several analysis techniques, such as those used in compiler domain, may be able to infer useful relations for such programs with nested loops. In this case, the polyhedral analysis as well as the scalar evolution analysis may be able to infer affine relations between the program variables.

```
// assume(true)

1. S = 0;
2. X = 0;
3. for(i=0; i<N; i++) {
4.   if( i == X + 2*sqrt(X) + 1 ) {
5.     S = S + i;
6.     X = i;
7.   }
8. }

// assert(∃ k. k² ≤ N < (k+1)² ∧ S = k×(k+1)×(2k+1)/6)
```

Figure 6.13: An Interesting Example

Weakness

The polyhedral analysis and scalar evolution may have a hard time inferring relations in programs with non-affine computations, especially when the computation is performed using only affine expressions. For the example in Fig. 5.1, these analyses may be able to capture quantified affine expressions over the arrays used in the program, the large bulk of the analysis effort is still delegated to the back-end solver which in many cases are unable to reason with given set of formulas. The inference task becomes even trickier in cases where the computation performed within loops is guarded by complex branch conditions. In such cases, these analysis techniques will be unable to infer useful facts about the program.

Consider the Hoare triple in Fig. 6.13. The program first initializes the scalars `S` and `X` to 0. The loop in the program has a branch statement with a complicated conditional expressions parametric in the loop counter `i` and the scalar variable `X`. The loop adds the value of `i` the the scalar variable `S` and assigns `i` to `X` only when the branch condition evaluates to true. The pre-condition of this program is `true`; the existentially quantified post-condition asserts that the value of `S` is the sum of all perfect squares between 0 to N.

Full-program induction can easily verify this program. Interestingly, none of the

variables in the program are "affected". Hence, computing the difference program is easy for full-program induction. The difference program consists of only the peeled iterations the loop in Fig. 6.13. Further, the inductive hypothesis already supplies the reasoning in the inductive step with some useful facts about the program. This lifts a significant amount of analysis effort that the back-end SMT solver has to put in.

Synergies with Full-Program Induction

Our techniques are orthogonal to most analysis and verification techniques in literature. We can explore various avenues to identify the synergies between the polyhedral analysis and recurrence solving techniques like scalar evolution and our technique. An interesting line of work is to use the relations inferred by these techniques to enrich different steps of the full-program induction technique. In fact, the relation full-program induction described in Chapter 7 can directly use the inferred relations during the inductive step of reasoning. Even the tile-wise reasoning presented in Chapter 4 can use such relations (as already described in Section 4.1.5). The inferred affine relations may also be useful in simplifying the difference program. In the future, we plan on using these analysis as a pre-processing step to optimize the programs for verification in a subsequent enhancement of our technique. We also consider several synergistic opportunities for the construction and use of data dependencies which are also computed during polyhedral analysis in our techniques (described in Section 5.3.3).

6.6 Experimental Evaluation

In this section, we present an extensive experimental evaluation of the *full-program induction* technique on a large set of array-manipulating benchmarks.

6.6.1 Implementation

We have implemented our technique in a prototype tool called VAJRA. Our tool and the benchmarks used in the experiments are publicly available at [CGU20b]. VAJRA takes a C program in SV-COMP format as input. The tool, written in `C++`, is built on top of the LLVM/CLANG [LA04] 6.0.0 compiler infrastructure. We use CLANG front-end to obtain LLVM bitcode. Several normalization passes such as constant propagation, dead

code elimination, static single assignment (SSA) for renaming variables and arrays, loop normalization for running loop-dependent passes that identify program constructs such as loop counters, lower bound and upper bound expressions, branch conditions and so on, are performed on the bitcode. We use Z3 [MB08] v4.8.7 as the SMT solver to prove the validity of the parametric Hoare triples for loop-free programs and to compute weakest pre-conditions. We have also implemented a Gaussian elimination based procedure that propagates array equalities and simplifies select store nests in the generated SMT formula to compute weakest pre-conditions.

6.6.2 Benchmarks

We have evaluated Vajra on a test-suite of 231 benchmarks inspired from different algebraic functions that compute polynomials as well as a standard array operations such as copy, min, max and compare. Of these there are 121 safe benchmarks and 110 unsafe benchmarks. All our programs take a symbolic parameter N which specifies the size of each array as well as the number of times each loop executes. Several benchmarks in the test-suite follow different types of templates wherein either the number of loops in the program increases or they use potentially different data values. Program from the first kind of templates allow us to gauge the scalability aspect of our technique as the number of loops in the program increases. Programs from the latter templates allow for checking the robustness of our technique to the content of arrays and scalars.

Assertions in the benchmarks are either universally quantified or quantifier free safety properties. The predicates in these assertions are (in-)equalities over scalar variables, array elements, and possibly non-linear polynomial terms over N. Although our technique can handle some classes of existentially quantified assertions as discussed in Sect. 5.5, all the examples considered for our experiments have universally quantified or quantifier-free assertions. The approach described in the paper is naturally applicable to programs with such assertions, given that the underlying SMT solver can discharge the verification conditions containing formulas with existential quantification and quantifier alternation when a loop-free difference program is automatically computed. Handling post-conditions with existential quantification and quantifier alternation are part of future work.

Tool	Success	CE	Inconclusive	TO
Safe				
Vajra	110	0	11	0
VIAP	58	0	2	61
VeriAbs	50	1	0	70
Booster	36	27	17	41
VapHor	27	9	2	83
FreqHorn	26	0	19	76
Unsafe				
Vajra	0	109	1	0
VIAP	1	108	0	1
VeriAbs	0	102	0	8
Booster	0	84	15	11
VapHor	1	106	1	2
FreqHorn	0	99	0	11

Figure 6.14: Summary of the Experimental Results

6.6.3 Experimental Setup

All experiments were performed on a Ubuntu 18.04 machine with 16GB RAM and running at 2.5 GHz. We have compared our tool VAJRA against the verifiers for array programs VIAP (v1.1) [RL18], VERIABS (v1.3.10) [ACC+20], BOOSTER (v0.2) [AGS14], VAPHOR (v1.2) [MG16] and FREQHORN (v.0.5) [FPMG19]. C programs were manually converted to mini-Java as required by VAPHOR and CHC formulae as required by FREQHORN. Since FREQHORN does not automatically find counterexamples, so we used the supplementary tool EXPL from its repository on unsafe benchmarks as recommend by them. We have used the same version of VERIABS that was used to perform the experiments in [CGU20a], since the later version of VERIABS invokes our tool VAJRA in its pipeline for verifying array programs (refer [ACC+20]). A timeout of 100 seconds was set for these experiments.

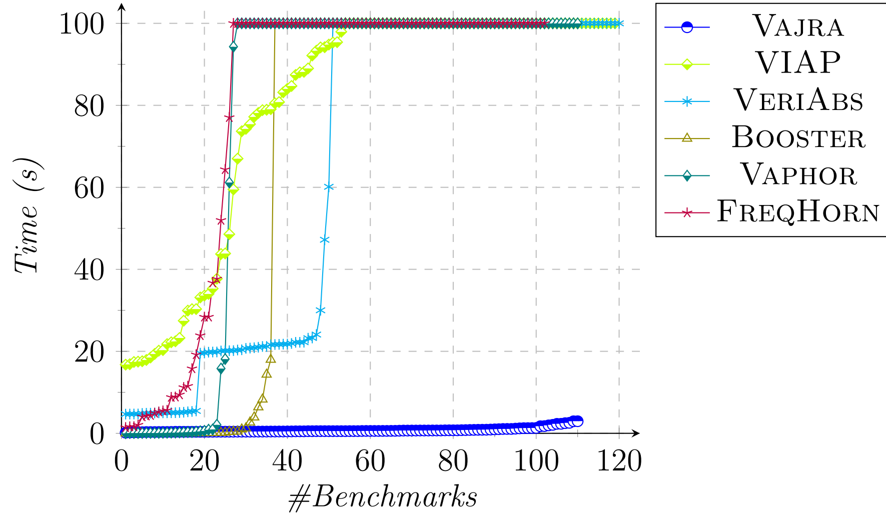


Figure 6.15: Quantile Plot Showing the Performance of the Tools on *Safe* Benchmarks

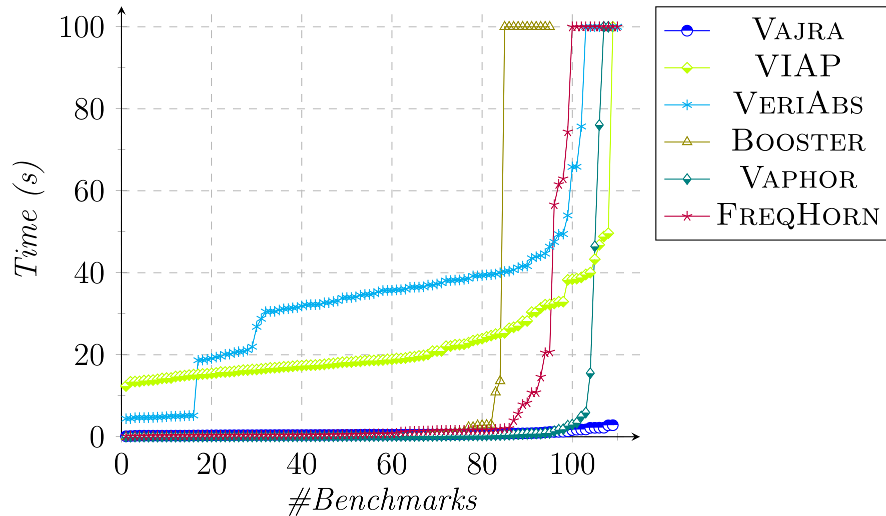


Figure 6.16: Quantile Plot Showing the Performance of the Tools on *Unsafe* Benchmarks

6.6.4 Summary of the Results

We executed all six tools on the entire set of 231 benchmarks. A table with the summary of obtained results is shown in Fig. 6.14. We present the results on safe and unsafe benchmarks separately for a fair representation of each tool on the set of benchmarks.

6.6.5 Analysis on Safe Benchmarks

Vajra verified 110 safe benchmarks, compared to 58 verified by VIAP, 50 by VeriAbs, 36 by Booster, 27 by Vaphor and 26 by FreqHorn. Vajra was inconclusive on 11 benchmarks. The reasons for the inability of our tool to generate a conclusive result are

as follows: (1) the difficulty in computing a difference program due to the presence of a branch condition dependent on N or complex operations such as modulo, (2) difficulty in computing the required strengthening of the pre- and post-conditions and (3) the back-end SMT solver returning an inconclusive result.

Vajra verified 52 benchmarks on which VIAP diverged, primarily due to the inability of VIAP's heuristics to get closed form expressions. VIAP verified 5 benchmarks that could not be verified by the current version of Vajra due to syntactic limitations. Vajra, however, is two orders of magnitude faster than VIAP on programs that were verified by both (refer Fig. 6.15).

Vajra proved 60 benchmarks on which VeriAbs diverged. VeriAbs ran out of time on programs where loop shrinking and merging abstractions were not strong enough to prove the assertions. VeriAbs reported 1 program as unsafe due to the imprecision of its abstractions and it proved 3 benchmarks that Vajra could not.

Vajra verified 74 benchmarks that Booster could not. Booster reported 27 benchmarks as unsafe due to imprecise abstractions, its fixed-point computation engine reported unknown result on 17 benchmarks and it ended abruptly on 41 benchmarks. Booster also proved 2 benchmarks that could not be handled by the current version of Vajra due to syntactic limitations.

Vajra verified 83 benchmarks that Vaphor could not. The distinguished cell abstraction technique implemented in Vaphor is unable to prove safety of programs, when the value at each array index needs to be tracked. Vaphor reported 9 programs unsafe due to imprecise abstraction, returned unknown on 2 programs and ended abruptly on 83 programs. Vaphor proved 2 benchmarks that Vajra could not.

Vajra verified 84 programs on which FreqHorn diverged, especially when constants and terms that appear in the inductive invariant are not syntactically present in the program. FreqHorn ran out of time on 76 programs, reported unknown result or ended abruptly on 19 benchmarks. FreqHorn verified a benchmark with a single loop that Vajra could not.

All the benchmarks that are uniquely solved by Vajra have multiple sequentially composed loops and/or a form of aggregation/cross-iteration dependence via a scalar variable or an array.

Instance	Loops	zerosum	zerosum-const	zerosum-m	zerosum-const-m
1	3	0.54	0.49	–	–
2	5	0.87	0.88	0.86	0.92
3	7	1.28	1.27	1.26	1.21
4	9	1.73	1.94	2.07	1.76
5	11	2.32	2.56	2.31	2.48
6	13	–	–	2.94	2.95

Figure 6.17: Template-wise Analysis of the Results for the 'zerosum' Benchmarks

6.6.6 Analysis on Unsafe Benchmarks

Vajra disproved 109 benchmarks, compared to 108 disproved by VIAP, 102 by VeriAbs, 84 by Booster, 106 by Vaphor and 99 by expl, the supplementary tool that comes with FreqHorn. Vajra was unable to disprove 1 benchmark.

Vajra disproved 1 benchmark which VIAP could not. VIAP concluded 1 benchmark as safe and timed out on 1 benchmark. Even on unsafe benchmarks, Vajra, is an order of magnitude faster than VIAP (refer Fig. 6.16). Vajra disproved 7 benchmarks which VeriAbs could not. VeriAbs ran out of time on 8 programs.

Vajra disproved 25 benchmarks that Booster could not. Booster reported unknown result on 15 benchmarks and it timed out on 11 benchmarks. Vajra disproved 3 benchmarks that Vaphor could not. Vaphor is proved 1 program as safe, returned unknown on 2 programs and timed out on 2 programs. Vajra disproved 10 programs which expl could not. expl ran out of time on 11 programs.

6.6.7 Performance Comparison

The quantile plots in Figs. 6.15 and 6.16 show the performance of each tool on all the safe and unsafe benchmarks respectively in terms of time taken to produce the result. Vajra verified/disproved each benchmark within three seconds. In comparison, as seen from the plots, other tools took significantly more time in proving the programs.

As mentioned previously, the test-suite has several benchmarks that are instantiated from different templates. For such instantiated benchmarks that only change the data values in the instances of the templates, we did not see any change in the performance of

Vajra. Hence, we do not discuss them further. We now discuss the results for a set of templates where the number of loops in the benchmarks instantiated from them increase. In Tab. 6.17, we present the results of executing Vajra on the benchmarks instantiated from the 'zerosum' templates. The first column indicates the benchmark instance number, the second column indicates the number of loops in the instantiated benchmark, columns three to six indicate the benchmark template name that is instantiated and give the time (in seconds) taken by Vajra to prove the given assertion in the benchmark instance. It can be seen from the table that as the number of loops increase in the benchmark, our tool requires more time in solving the benchmark. This is primarily attributed to pre- and post-condition strengthening step in our technique that requires our technique to infer and prove auxiliary predicates iteratively during the inductive step.

6.7 Comparison with Related Techniques

Earlier work on inductive techniques can be broadly categorized into those that require loop-specific invariants to be provided or automatically generated, and those that work without them. Requiring a "good" inductive invariant for every loop in a program effectively shifts the onus of assertion checking to that of invariant generation. Among techniques that do not require explicit inductive invariants or mid-conditions for each loop, there are some that require loop invariants to be implicitly generated by a constraint solver. These include techniques based on constrained Horn clause solving [KBGM15, GSV18, FPMG19, MG16], acceleration and lazy interpolation for arrays [AGS14] and those that use inductively defined predicates and recurrence solving [GGK20, RL18, HHKR10], among others.

QUIC3 [GSV18], FreqHorn [FPMG19] and the technique in [KBGM15] infer universally quantified inductive invariants of array programs specified as Constrained Horn Clauses. QUIC3 [GSV18] extends the IC3 framework to a combination of SMT theories and performs lazy quantifier instantiations. FreqHorn [FPMG19] infers universally quantified invariants over arrays within its syntax-guided synthesis framework and can reason with complex array index expressions by adopting the *tiling* [CGU17] ideas.

Vaphor [MG16] transforms array programs to array-free Horn formulas. Their technique is parameterized by the number of array cells to be tracked resulting in an

eager quantifier instantiation. BOOSTER [AGS14] combines acceleration [BIK10, JSS14] and lazy abstraction with interpolation for arrays [ABG+12a]. Performing interpolation to infer universally quantified array properties is difficult [JM07, MA15]. The technique does not always succeed, especially for programs where simple interpolants are difficult to compute [CGU17].

VIAP [RL18] translates the program to an array-free quantified first order logic formula in the theory of equality and uninterpreted functions using the scheme proposed in [Lin16]. They use several tactics to simplify the generated formula and apply induction over array indices to prove the property. Unlike our method, it does not have heuristics for finding additional pre-conditions that are required for the induction proof to succeed which our method successfully infers. [GGK20] uses theorem provers to introduce and prove lemmas that implicitly capture inductive loop invariants at arbitrary points in the program described in trace logic. Thanks to the impressive capabilities of modern constraint solvers and the effectiveness of carefully tuned heuristics for stringing together multiple solvers, approaches that rely on constraint solving have shown a lot of promise in recent years. However, at a fundamental level, these formulations rely on solving implicitly specified loop invariants garbed as constraint solving problems.

Template-based techniques [GMT08, SG09, BHMR07] search for inductive invariants by instantiating the parameters of a fixed set of templates within the abstract interpretation framework. They can generate invariants with alternating quantifiers, however, the user must supply invariant templates and the cost of generating invariants is quite high.

A large number of techniques have been proposed in literature that use induction [DM97, BC00, ES03, GLD09, CJRS13, RK15, UTS17] and its pragmatically more useful version k-induction [SSS00, DMRS03, HT08, DKR10, KT11, DHKR11, BDW15, BJKS15, GIC17, KVGG19, ARG+21, YBH21]. These techniques generate and use loop invariants, especially when aimed at verifying safety properties of programs. In contrast, our novel technique does not rely on generation or use of loop-specific invariants and differs significantly from these methods in the way in which the inductive step is formulated using the computed difference programs and difference pre-conditions.

There are yet other inductive techniques, such as that in [SB12, CGU17, ISIRS20, CGU21b], that truly do not depend on loop invariants being generated. In fact, the technique of [SB12] comes closest to our work in principle. However, [SB12] imposes

severe restrictions on the input programs to move the peel of one loop across the next sequentially composed loop such that the program with the peeled loops composed with the program fragment consisting of only the peeled iterations is semantically equivalent to the input program. They call these restrictions on the input programs as *commutativity of statements*. In practice, such restrictive conditions and data dependencies are not satisfied by a large class of programs. For instance, the examples in Fig. 5.1, Fig. 5.11 and Fig. 6.1 do not meet these restrictions. The technique of [SB12] is thus applicable only to a small part of the program-assertion space over which our technique works.

The tiling [CGU17] technique, described in Chapter 4, for verifying universally quantified properties of array programs reasons one loop at a time and applies only when loops have simple data dependencies across iterations (called *non-interference* of tiles in [CGU17]). The method effectively uses a slice of the post-condition of a loop as an inductive invariant. In the case of sequentially composed loops, it also requires strong enough mid-conditions to be automatically generated or supplied by the user. Our full-program induction technique circumvents all of these requirements.

The method proposed in [ISIRS20] proves programs correct by induction on a rank, chosen as the size of program states. It constructs a safety proof by automatically synthesizing a squeezing function that can map higher-ranked states to a lower-ranked state, while ensuring that original states are faithfully simulated by their squeezed counterparts. This allows the method to shrink program traces of unbounded length, limiting the reasoning to only minimally-ranked states. A guess-and-check approach combined with heuristics for making educated guesses is employed for computing the squeezing functions necessary to prove a given program. Successful synthesis of a squeezing function is equivalent to establishing the inductive step. These functions can be quite useful in practice, for example, to prove programs that may not have a first-order representable loop invariant. In general, squeezing functions are not easy to synthesize and automatically searching for such functions is a non-trivial and an exceedingly time consuming task. Further, the squeezing functions can only consist of commutative and invertible operations, restricting their applicability. The technique may be used in tandem with the classical loop invariant based methods. In comparison, our technique generates and uses difference invariants in an explicit inductive step and it does not rely on generation and use of squeezers to shrink the state space of the program.

Another technique presented in [CGU21b] also performs induction on the entire program and is a parallel line of our work. Full-program induction forms the basis of their technique but the way in which the inductive step is formulated differs significantly from ours as described in [CGU21b]. The method coins the term *difference invariants* that relate variables/arrays in two slightly different versions of the given program. They use just the peeled iterations of loops as difference programs and amend the inductive reasoning using difference invariants. The technique supports nested loops as well as branch conditions with value dependent on the program parameter N. The prototype tool DIFFY [CGU21a] implements the method. We believe that there are programs for which [CGU21b] may not be able to successfully infer and use difference invariants, but full-program induction (with its recursive invocation ability) will be able to verify the post-conditions in such programs.

There are several techniques that approximate program computation during verification. [MIG+21] has proposed a counterexample-guided abstraction refinement scheme for programs that manipulate arrays. Their idea relies on prophecy variables to refine the abstraction. VERIABS [ACC+20] is an abstraction-based verifier to prove properties of programs. It implements a portfolio of abstractions that enable the tool to leverage bounded model checking. These abstractions tend to restrict the array-manipulating loops to a fixed number of (possibly initial) iterations. The tool makes a series of attempts to prove the property and uses program features to choose the next abstraction/strategy to be applied. Fluid updates [DDA10] uses bracketing constraints, which are over- and under-approximations of indices, to specify the concrete elements being updated in an array without explicit partitioning. While their abstraction is independent of the given property, they assume that only a single index expression updates the array in each loop, severely restricting the technique. Analyses proposed in [GRS05, HP08] partition the array into symbolic slices and abstracts each slice with a numeric scalar variable. Abstract interpretation based techniques [LR15, CCL11] propose an abstract domain which utilizes cell contents to split array cells into groups. In particular, the technique in [LR15] is useful when array cells with similar properties are non-contiguously present in the array. These approaches require the implementation of abstract transformers for each specialized domain which is not a necessity with our framework. Other techniques for analyzing array-manipulating programs include [JM07, JSP+11].

Program differencing [PK82], program integration [HPR89] and differential static analysis [LVH10] have been studied in literature for various purposes. Incremental computation of expensive expressions [LST98], optimizing the execution time of programs that manipulate arrays [LSLR05], reducing the cost of regression testing [Bin92] and checking data-structure invariants [SB07] are some applications of such techniques. SymDiff [LHKR12] tool, based on differential static analysis [LVH10], displays semantic differences between different program versions and checks equivalence. However, the method neither supports checking quantified post-conditions nor does it support loops and arrays of potentially unbounded size. Unfortunately, these techniques do not always generate code fragments that are well suited for property verification, especially when the input programs manipulate arrays. To the best of our knowledge, full-program induction is the first technique to successfully employ difference computation customized for verification in an inductive setting.

Full-program induction also offers several other advantages over the existing techniques. For instance, it can reason with different quantifiers over multiple variables, it does not require implementation of specialized abstract domains for handling quantified formulas and it can enable the use of existing tools and techniques for reasoning over arrays. We believe that verification tools need to have an arsenal of techniques to be able to efficiently prove a wide range of challenging problems. Our novel technique, full-program induction, is a suitable fit for such an arsenal and has been adopted by verifiers such as VeriAbs in practice. Since the 2020 edition of the international software verification competition (SV-COMP), VeriAbs [ACC+20] invokes our tool Vajra in its pipeline of tools for verifying programs with arrays from the set of benchmarks in the competition.

6.8 Conclusion

We presented a generalized difference program computation method for performing full-program induction on a larger class of programs. Significantly, the generated difference program can rectify the values of affected variables and arrays, allowing the technique to verify programs with affected variables and arrays. Moreover, the full-program induction technique can be recursively applied to simplify the difference program when it is not loop-free. We generalized the computation of the difference pre-conditions, which is performed

with the simultaneous strengthening of pre- and post-conditions. The generalized methods can be applied to array-manipulating programs that store integers, matrices, polynomials, vectors and so on, making it capable of verifying APIs used in machine learning and cryptography libraries. We described a prototype implementation of the technique in our tool Vajra. Experiments show that Vajra performs remarkably well vis-a-vis state-of-the-art tools for analyzing array-manipulating programs.

Possible directions of future work include investigations into possible ways of incorporating automatically generated and externally supplied invariants during our analysis, especially for computing simpler difference programs and handling programs with nested loops. Automated support for handling assertions with existential quantification and quantifier alternation and for verifying heap-manipulating programs as well as programs that operate on tensors using our technique. Investigations into the use of synthesis-based techniques for automatically computing the difference programs and adapting them to programs from various interesting domains forms another line of work. Improvements to the algorithms for simultaneous strengthening of pre- and post-conditions can be considered.

Chapter 7

Relational Full-Program Induction

In Chapters 5 and 6 we described the *full-program induction* technique [CGU20a, CGU22] that verifies properties of programs that manipulate arrays by computing difference programs and difference pre-conditions during the inductive step. Computing difference programs is, in general, a challenging task and feasible only for a restricted class of programs. In this chapter, we propose a novel and practically efficient induction-based technique called *relational full-program induction*, that advances the state-of-the-art in automating the inductive step when reasoning about array-manipulating programs. Relational full-program induction greatly simplifies difference computation by using only the peels of loops as the difference program. The technique generates two slightly different versions of a program, and then infers specific kinds of relations between the corresponding variables of the two versions to aid the formulation of the inductive step. This allows us to automatically verify interesting properties of a class of array-manipulating programs that are beyond the reach of state-of-the-art induction-based techniques, viz. [CGU22, CGU20a, RL18, SB12]. The verification tool VAJRA [CGU20b] that implements full-program induction is a part of the portfolio of techniques in VERIABS [ACC+20] – the winner of SV-COMP 2021 in the ReachSafety-Arrays sub-category and is the closest related work for reasoning about array-manipulating programs. Interestingly, the relational full-program induction technique presented in this chapter addresses several key limitations of full-program induction implemented in VAJRA, thereby making it possible to analyze a much larger class of array-manipulating programs than can be done by VERIABS. Significantly, this includes programs with nested loops that have hitherto been beyond the reach of automated techniques that use mathematical induction [CGU22, CGU20a, RL18, SB12].

Relational full-program induction can thus be viewed as a significant generalization of full-program induction. A part of the work described in this chapter has been published as a conference paper in CAV 2021 [CGU21b].

7.1 Introduction

In this chapter, we present *relational full-program induction* and highlight its difference vis-a-vis the full-program induction technique. A key innovation in relational full-program induction is the construction of two slightly different versions of a given program that have identical control flow structures but slightly different data operations. We automatically identify relations, between corresponding variables in the two versions of a program at key control flow points. Interestingly, these relations often turn out to be significantly simpler than inductive invariants required to prove the property directly. This is not entirely surprising, since these relations depend less on what individual statements in the programs are doing, and more on the difference between what they are doing in the two versions of the program. We show how the two versions of a given program can be automatically constructed, and how differences in individual statements can be analyzed to infer simple relational invariants. We introduce the notion of *difference invariants* that are relational invariants specified using only the difference of values of two versions of a variable/array. Finally, we show how these difference invariants can be used to simplify the reasoning in the inductive step of the relational full-program induction technique.

We consider programs P_N parameterized by a symbolic integer N (> 0) consisting of (possibly nested) loops that manipulate arrays generated by the grammar in Fig. 3.1. The sizes of arrays in the program are also parametric in N. We verify (a sub-class of) quantified and quantifier-free properties of the form specified in Section 3.4 that may depend on the symbolic parameter N. We view the verification problem as one of proving the validity of a parameterized Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ for all values of N (> 0), where N is a free variable in $\varphi(\cdot)$ and $\psi(\cdot)$.

7.1.1 Motivation and Examples

To illustrate the kind of programs that are amenable to our relational full-program induction technique, consider the program shown in Fig. 7.1, adapted from an SV-COMP

```
// assume(true)
1. S = 0; F = 1;
2. for(i=0; i<N; i++) {
3.  S = S + 1;
4.  if(A[i] >= 0) B[i] = 1;
5.  else B[i] = 0;
6. }
7. for(j=0; j<N; j++) {
8.  if(S == N) {
9.   if(A[j] >= 0 && !B[j]) F = 0;
10.  if(A[j] < 0 && B[j]) F = 0;
11. }
12.}
// assert(F = 1)
```

Invariant after the assume statement

$\mathtt{S} = \star \ \wedge \ \mathtt{F} = \star \ \wedge \ \forall \mathtt{x} \in [0, \mathtt{N}) \ (\mathtt{A}[\mathtt{x}] = \star \ \wedge \ \mathtt{B}[\mathtt{x}] = \star)$

Invariant after executing statements on line 1

$\mathtt{S} = 0 \ \wedge \ \mathtt{F} = 1 \ \wedge \ \forall \mathtt{x} \in [0, \mathtt{N}) \ (\mathtt{A}[\mathtt{x}] = \star \ \wedge \ \mathtt{B}[\mathtt{x}] = \star)$

Invariant at loop head on line 2

$\mathtt{S} = \mathtt{i} \ \wedge \ \mathtt{F} = 1 \ \wedge \ \forall \mathtt{x} \in [0, \mathtt{N}) \ \mathtt{A}[\mathtt{x}] = \star \ \wedge$
$\forall \mathtt{x} \in [0, \mathtt{i}) \ \big((\mathtt{A}[\mathtt{x}] \geq 0 \ \Rightarrow \ \mathtt{B}[\mathtt{x}] = 1) \ \wedge$
$(\mathtt{A}[\mathtt{x}] < 0 \Rightarrow \mathtt{B}[\mathtt{x}] = 0)\big)$

Invariant after executing the first loop

$\mathtt{S} = \mathtt{N} \ \wedge \ \mathtt{F} = 1 \ \wedge \ \forall \mathtt{x} \in [0, \mathtt{N}) \ \mathtt{A}[\mathtt{x}] = \star \ \wedge$
$\forall \mathtt{x} \in [0, \mathtt{N}) \ \big((\mathtt{A}[\mathtt{x}] \geq 0 \ \Rightarrow \mathtt{B}[\mathtt{x}] = 1) \ \wedge$
$(\mathtt{A}[\mathtt{x}] < 0 \Rightarrow \mathtt{B}[\mathtt{x}] = 0)\big)$

Figure 7.1: Motivating Example - I

benchmark. This program has a couple of sequentially composed loops that update arrays and scalars. The scalars `S` and `F` are initialized to 0 and 1 respectively before the first loop starts iterating. Subsequently, the first loop computes a recurrence in variable `S` and initializes elements of the array `B` to 1 if the corresponding elements of array `A` have non-negative values, and to 0 otherwise. The outermost branch condition in the body of the second loop evaluates to true only if the program parameter N and the variable `S` have same values. The value of `F` is reset based on some conditions depending on corresponding entries of arrays `A` and `B`. The pre-condition of this program is `true`; the post-condition asserts that `F` is never reset in the second loop. The invariants required to prove the assert are shown in the right flank of Fig. 7.1, where $\star$ indicates a non-deterministic value.

State-of-the-art techniques find it difficult to prove the assertion in this program. Specifically, Vajra [CGU20b] (that implements full-program induction [CGU20a, CGU22]) is unable to prove the property, since it cannot reason about the branch condition (in the second loop) whose value depends on the program parameter N. VeriAbs [ACC+20], which employs a sequence of techniques such as loop shrinking, loop pruning, and inductive reasoning using Vajra is also unable to verify the assertion shown in this program.

	Invariant after the assume statement
`// assume(true)`	$\mathtt{S} = \star \ \wedge\ \forall \mathtt{x} \in [\mathtt{0}, \mathtt{N})\ \mathtt{A}[\mathtt{x}] = \star$
`1. S = 0;`	
`2. for(i=0; i<N; i++) {`	Invariant at loop head on line 2
`3.   A[i] = 0;`	$\mathtt{S} = \mathtt{0} \ \wedge\ \forall \mathtt{x} \in [\mathtt{0}, \mathtt{i})\ \mathtt{A}[\mathtt{x}] = \mathtt{0}$
`4. }`	
`5. for(j=0; j<N; j++) {`	Invariant at loop head on line 5
`6.   S = S + 1;`	$\mathtt{S} = \mathtt{j} \ \wedge\ \forall \mathtt{x} \in [\mathtt{0}, \mathtt{N})\ \mathtt{A}[\mathtt{x}] = \mathtt{0}$
`7. }`	
`8. for(k=0; k<N; k++) {`	Invariant at loop head on line 8
`9.   for(l=0; l<N; l++) {`	$\mathtt{S} = \mathtt{N} \ \wedge\ \forall \mathtt{x} \in [\mathtt{0}, \mathtt{k})\ \mathtt{A}[\mathtt{x}] = \mathtt{k} + \mathtt{S} \ \wedge\ \forall \mathtt{x} \in [\mathtt{k}, \mathtt{N})\ \mathtt{A}[\mathtt{x}] = \mathtt{k}$
`10.    A[l] = A[l] + 1;`	
`11   }`	Invariant at loop head on line 9
`12.   A[k] = A[k] + S;`	$\mathtt{S} = \mathtt{j} \ \wedge\ \forall \mathtt{x} \in [\mathtt{0}, \mathtt{k})\ \mathtt{A}[\mathtt{x}] = \mathtt{k} + \mathtt{S} \ \wedge\ \forall \mathtt{x} \in [\mathtt{k}, \mathtt{N})\ \mathtt{A}[\mathtt{x}] = \mathtt{k}\ \wedge$
`13.}`	$\forall \mathtt{x} \in [\mathtt{0}, \mathtt{l})\ (\mathtt{l} < \mathtt{k} \Rightarrow \mathtt{A}[\mathtt{x}] = \mathtt{k} + \mathtt{1} + \mathtt{S}) \wedge (\mathtt{l} \geq \mathtt{k} \Rightarrow \mathtt{A}[\mathtt{x}] = \mathtt{k} + \mathtt{1})$
`// assert(∀x∈[0,N) A[x] = 2×N)`	$\wedge\ \forall \mathtt{x} \in [\mathtt{l}, \mathtt{N})\ (\mathtt{l} < \mathtt{k} \Rightarrow \mathtt{A}[\mathtt{x}] = \mathtt{k} + \mathtt{S}) \wedge (\mathtt{l} \geq \mathtt{k} \Rightarrow \mathtt{A}[\mathtt{x}] = \mathtt{k})$

Figure 7.2: Motivating Example - II

Indeed, the loops in this program cannot be merged as the final value of `S` computed by the first loop is required in the second loop; hence loop shrinking does not help. Also, loop pruning does not work due to the complex dependencies in the program and the fact that the exact value of the recurrence variable `S` is required to verify the program. Subsequent abstractions and techniques applied by VeriAbs from its portfolio are also unable to verify the given post-condition. VIAP [RL18] translates the program to a quantified first-order logic formula in the theory of equality and uninterpreted functions [Lin16]. The tool applies a sequence of tactics to simplify and prove the generated formula. These tactics include computing closed forms of recurrences, induction over array indices and the like to prove the property. However, its sequence of tactics is unable to verify this example within our time limit of 1 minute.

Benchmarks with nested loops are a long standing challenge for most verifiers. Consider the program shown in Fig. 7.2 with a nested loop in addition to sequentially composed loops. The first loop initializes entries in array `A` to 0. The second loop aggregates

a constant value in the scalar `S`. The third loop is a nested loop that updates array `A` based on the value of `S`. The entries of `A` are updated in the inner as well as outer loop. The property asserts that on termination, each array element equals twice the value of the parameter N. The invariants required to prove the assert are shown in the right flank of Fig. 7.2, where $\star$ indicates a non-deterministic value.

While the inductive reasoning of VAJRA and the tactics in VIAP do not support nested loops, the sequence of techniques used by VERIABS is also unable to prove the given post-condition in this program. In sharp contrast, our prototype tool DIFFY is able to verify the assertions in both these programs automatically within a few seconds. This illustrates the power of the inductive technique proposed in this chapter.

7.1.2 Instantiation of the Technique in Diffy

We have implemented the relational full-program induction technique in a prototype tool called DIFFY. The tool is written in C++. It is built using the LLVM/CLANG [LA04] compiler framework and the SMT solver Z3 [MB08]. We perform extensive experiments using the relational full-program induction technique implemented in DIFFY. We compare its performance vis-a-vis state-of-the-art tools for verifying properties of array programs that have participated in SV-COMP. Our tool DIFFY is significantly more efficient as compared to other tools on a class of programs and is able to solve many difficult problem instances on which other tools run out of resources. As is usual, each approach has its own strengths and limitations; DIFFY being no exception.

The main contributions of the chapter can be summarized as follows:

- We present a novel method of performing *relational full-program induction*, to prove interesting properties of a class of programs that manipulate arrays.
- We introduce the concept of *difference invariants* that are relational invariants specified using only the difference of values of two versions of a variable/array. The crucial inductive step in the novel technique computes and uses relations between variables and arrays from two slightly different versions of the same program.
- We describe the algorithms for relational full-program induction. We present the transformations of the input program for the inductive step of the analysis and the

techniques to infer simple difference invariants from two slightly different versions of the same program.

- We describe a prototype tool DIFFY that implements the relational full-program induction technique. The tool is built on top of the LLVM/CLANG compiler framework and the Z3 SMT solver.

- We compare DIFFY vis-a-vis state-of-the-art tools for verification of C programs that manipulate arrays on a large set of benchmarks. We demonstrate that DIFFY significantly outperforms the winners of SV-COMP 2019, 2020 and 2021 in the ReachSafety-Arrays sub-category.

7.2 Overview of Relational Full-Program Induction

In this section, we provide an overview of the main ideas underlying the relational full-program induction technique. We then compare and contrast the relational full-program induction technique that infers and uses specialized relational invariants, called *difference invariants*, from the full-program induction technique (described in Chapters 5 and 6) that uses difference programs. We assume that we are given a parameterized pre-condition $\varphi(N)$, and our goal is to establish the parameterized post-condition $\psi(N)$, for all $N > 0$. To keep the exposition simple, we consider the program P_N, shown in the first column of Fig. 7.3, where N is a symbolic parameter denoting the sizes of arrays `a` and `b`.

The technique inducts over the entire program, via the program parameter N, and not on the individual loops in the program. We first check the base case of the induction, by verifying that the parameterized Hoare triple holds for some small values of N, say $0 < N \leq M$. We assume that every loop in P_N can be statically unrolled a number (say $f(N)$) of times that depends only on N, to yield a loop-free program $\widehat{\mathsf{P}_N}$ that is semantically equivalent to P_N. The base case is checked using an off-the-shelf SMT solver, such as Z3, after compiling the pre-condition, the loop-free program and the post-condition into an SMT formula.

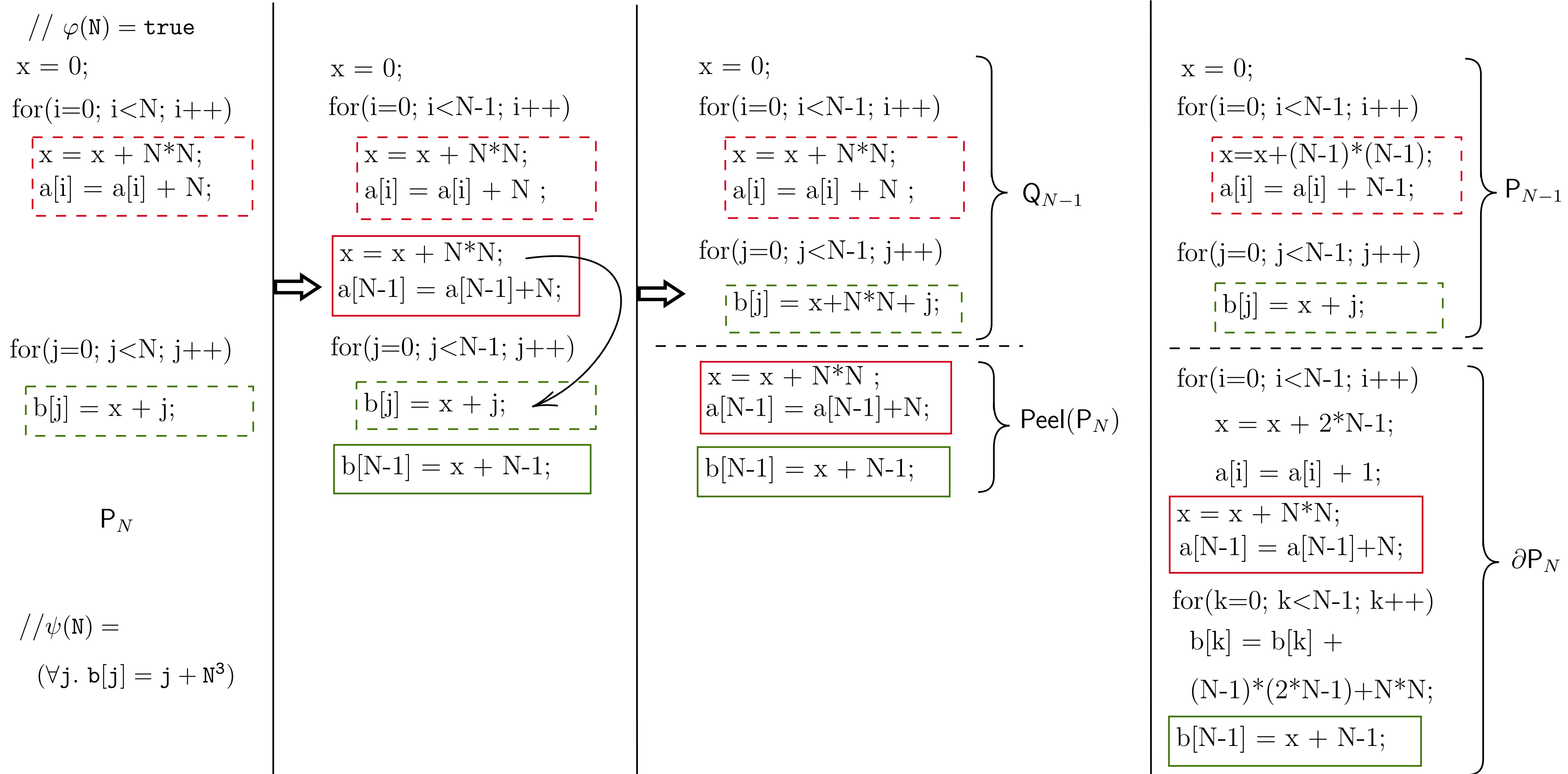


Figure 7.3: Pictorial Depiction of our Program Transformations

Next, we hypothesize that the entire Hoare triple $\{\varphi(N-1)\}$ P_{N-1} $\{\psi(N-1)\}$ holds for some $N > M$ after substituting the parameter N with $N-1$ uniformly across the entire Hoare triple. We then want to relate the induction hypothesis with the Hoare triple $\{\varphi(N)\}$ P_N $\{\psi(N)\}$. However, we intend to relate the programs P_N and P_{N-1} via a program $\mathsf{Peel}(\mathsf{P}_N)$ that consists of only the peeled iterations of loops. This is intended to significantly simplify the computation of difference programs.

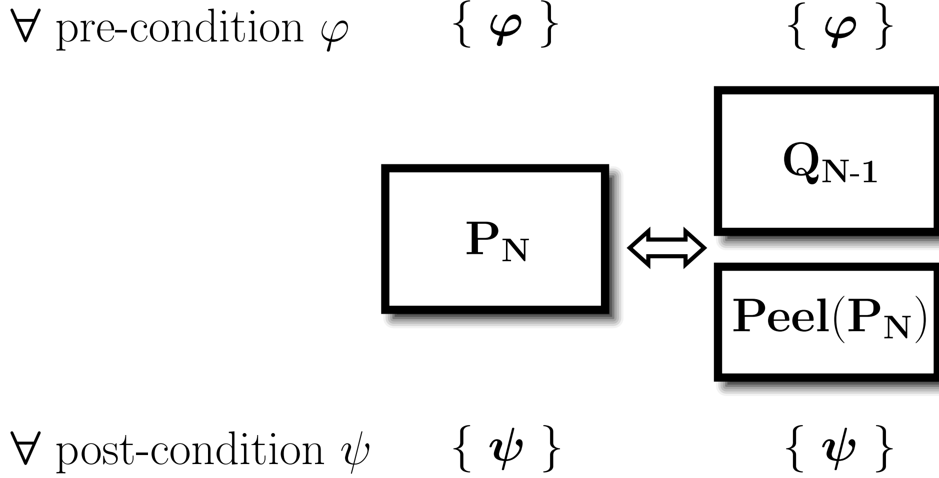


Figure 7.4: Decomposition of P_N and Semantic Equivalence

The inductive step proceeds as follows. Given P_N, we first algorithmically construct two programs Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$, such that P_N is semantically equivalent to Q_{N-1}; $\mathsf{Peel}(\mathsf{P}_N)$, as shown in Fig. 7.4. Intuitively, Q_{N-1} is the same as P_N, but with all loop bounds that depend on N now modified to depend on $N-1$ instead. Note that this is different from P_{N-1}, which is obtained by replacing *all uses* (not just in loop bounds) of N in P_N by $N-1$. As we will see, this simple deviation in the verification strategy makes the generation of the difference program $\mathsf{Peel}(\mathsf{P}_N)$ significantly simpler in relational full-program induction than the generation of the difference program $\partial\mathsf{P}_N$ in full-program induction, as described in Chapter 6, for a larger class of programs.

The third column of Fig. 7.3 shows Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$ generated by our algorithm for the program P_N in the first column of the figure. It is illustrative to compare these with P_{N-1} and $\partial\mathsf{P}_N$ shown in the fourth column of Fig. 7.3. Notice that Q_{N-1} has the same control flow structure as P_{N-1}, but is not semantically equivalent to P_{N-1}. In fact, Q_{N-1} and P_{N-1} may be viewed as closely related versions of the same program.

To use the information from the induction hypothesis during the induction step, we

must relate the computation in the programs Q_{N-1} and P_{N-1}. Let V_Q and V_P denote the set of variables of Q_{N-1} and P_{N-1} respectively. We assume V_Q is disjoint from V_P, and analyze the joint execution of Q_{N-1} and P_{N-1} starting from a state satisfying $\varphi(N)$. The purpose of this analysis is to compute a relational invariant $D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$ that relates corresponding variables in Q_{N-1} and P_{N-1} at the end of their joint execution.

The relational invariant $D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$ can be computed using a variety of methods. The problem of computing the predicate $D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$ that specifically relates corresponding variables in Q_{N-1} and P_{N-1} is reminiscent of (yet, different from) translation validation [Nec00, ZPFG02, ZP08, BCK11, SSCA13, DB17, GRB20], and indeed, our calculation of $D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$ is motivated by techniques from the translation validation literature. We introduce the concept of *difference invariants* to specialize this relation between the values of corresponding variables/arrays in Q_{N-1} and P_{N-1} by considering only the difference of their values. These invariants can be modeled using simple templates that relate only the difference between corresponding variable/array in Q_{N-1} and P_{N-1}. Automated synthesis algorithms based on the guess-and-check paradigm that take into account program syntax and behaviours can also be used for inferring these difference invariants. An important finding of our study is that corresponding variables in Q_{N-1} and P_{N-1} are often related by simple expressions and predicates on N from the class of difference invariants, regardless of the complexity of P_N, $\varphi(N)$ or $\psi(N)$. Indeed, in all our experiments, we did not need to go beyond quadratic expressions on N to compute $D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$. Of the 157 safe benchmarks, 6 benchmarks required difference invariants with quadratic terms and the rest required difference invariants with only linear terms. For unsafe benchmarks, we do not require computation of difference invariants since the analysis concludes after the base-case gets violated.

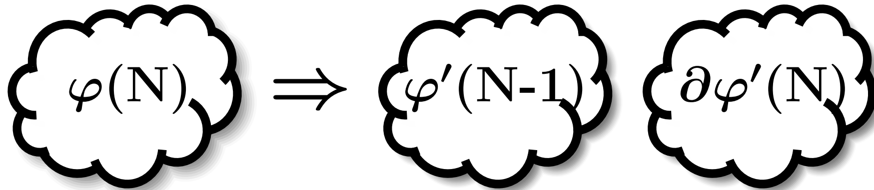


Figure 7.5: Difference Pre-condition

Some pre-conditions $\varphi(N)$ do not admit any $\partial\varphi(N)$ such that $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$. It is, however, often easy to compute formulas $\varphi'(N-1)$ and $\Delta\varphi'(N)$ in such

cases such that $\varphi(N) \Rightarrow \varphi'(N-1) \wedge \Delta\varphi'(N)$ (Fig. 7.5), and the variables/array elements in $\Delta\varphi'(N)$ are not modified by either P_{N-1} or Q_{N-1}. For example, if we were to consider a (new) pre-condition $\varphi(N) \equiv \big(\bigwedge_{i=0}^{N-1} A[i] = N\big)$ for the program P_N shown in the first column of Fig. 7.3, then we have $\varphi'(N-1) \equiv \big(\bigwedge_{i=0}^{N-2} A[i] = N\big)$ and $\Delta\varphi'(N) \equiv \big(A[N-1] = N\big)$. We assume the availability of such a $\varphi'(N-1)$ and $\Delta\varphi'(N)$ for the given $\varphi(N)$. This significantly relaxes the requirement on pre-conditions and allows a much larger class of Hoare triples to be proved using our technique vis-a-vis that of full-program induction.

Once the steps described above are completed, we have $\Delta\varphi'(N)$, $\mathsf{Peel}(\mathsf{P}_N)$ and $D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$. It can now be shown that if the inductive hypothesis, i.e. $\{\varphi(N-1)\}$ P_{N-1} $\{\psi(N-1)\}$ holds, then proving $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ reduces to proving $\{\Delta\varphi'(N) \wedge \psi'(N-1)\}$ $\mathsf{Peel}(\mathsf{P}_N)$ $\{\psi(N)\}$, as shown in Fig. 7.6, where $\psi'(N-1) \equiv \exists V_\mathsf{P}\big(\psi(N-1) \wedge D(V_\mathsf{Q}, V_\mathsf{P}, N-1)\big)$. A few points are worth emphasizing here. First, if $D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$ is obtained as a set of equalities, the existential quantifier in the formula $\psi'(N-1)$ can often be eliminated simply by substitution. We can also use quantifier elimination capabilities of modern SMT solvers, viz. Z3 [MB08], to eliminate the quantifier, if needed. Second, recall that unlike $\partial\mathsf{P}_N$ generated by the full-program induction technique (refer Section 6.2), $\mathsf{Peel}(\mathsf{P}_N)$ is guaranteed to be "simpler" than P_N, and is indeed loop-free if P_N has no nested loops. Therefore, proving $\{\Delta\varphi'(N) \wedge \psi'(N-1)\}$ $\mathsf{Peel}(\mathsf{P}_N)$ $\{\psi(N)\}$ is typically significantly simpler than proving $\{\psi(N-1) \wedge \partial\varphi(N)\}$ $\partial\mathsf{P}_N$ $\{\psi(N)\}$.

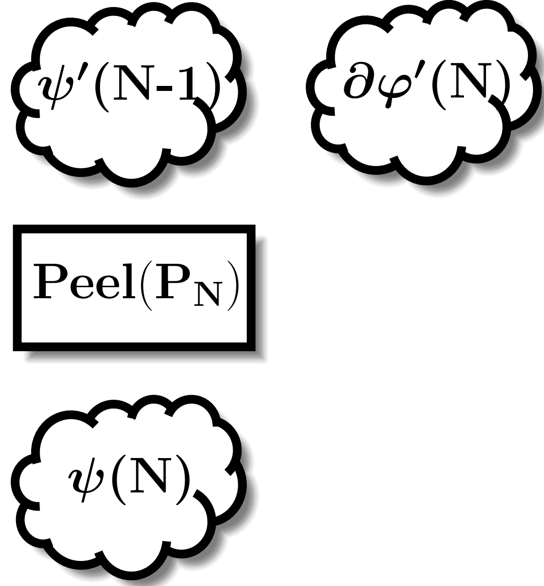


Figure 7.6: Inductive Step

Finally, it may happen that the pre-condition in $\{\Delta\varphi'(N) \wedge \psi'(N-1)\}$ $\mathsf{Peel}(\mathsf{P}_N)$ $\{\psi(N)\}$ is not strong enough to yield a proof of the Hoare triple. In such cases, we need to strengthen the existing pre-condition by a formula, say $\xi'(N-1)$, such that the strengthened pre-condition implies the weakest pre-condition of $\psi(N)$ under $\mathsf{Peel}(\mathsf{P}_N)$. Having a simple structure for $\mathsf{Peel}(\mathsf{P}_N)$ (e.g., loop-free for the entire class of programs for which

full-program induction works) makes it significantly easier to compute the weakest pre-condition. Note that $\xi'(N-1)$ is defined over the variables in V_{Q}. In order to ensure that the inductive proof goes through, we need to strengthen the post-condition of the original program by $\xi(N)$ such that $\xi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1) \Rightarrow \xi'(N-1)$. Computing $\xi(N-1)$ requires a special form of logical abduction that ensures that $\xi(N-1)$ refers only to variables in V_P. However, if $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ is given as a set of equalities (as is often the case), $\xi(N-1)$ can be computed from $\xi'(N-1)$ simply by substitution. This process of strengthening the pre-condition and post-condition may need to iterate a few times until a fixed point is reached, similar to what happens in the inductive step of full-program induction. Note that the fixed point iterations may not always converge (verification is undecidable in general). However, in our experiments, convergence always happened within a few iterations. If $\xi'(N-1)$ denotes the formula obtained on reaching the fixed point, the final Hoare triple to be proved is $\{\xi'(N-1) \wedge \Delta\varphi'(N) \;\wedge\; \psi'(N-1)\}$ $\mathsf{Peel}(\mathsf{P}_N)$ $\{\xi(N) \wedge \psi(N)\}$, as shown in Fig. 7.7, where $\psi'(N-1) \equiv \exists V_{\mathsf{P}}\big(\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)$. Having a simple (often loop-free) $\mathsf{Peel}(\mathsf{P}_N)$ significantly simplifies the above process.

7.2.1 Relation to Full-Program Induction

In this section, we compare and contrast our technique with full-program induction [CGU20a, CGU22], presented in Chapters 5 and 6. Other techniques based on mathematical induction (on N) have been proposed in literature to solve the class of problems that we focus on, for example, the method in [SB12]. However, since the full-program induction technique is significantly more general and subsumes the previous techniques, henceforth we only refer to full-program induction when talking about the earlier techniques that use induction to prove program properties. As with any induction-based technique, full-program induction consists of three steps. First, it checks if the *base case* holds, i.e. if the Hoare triple $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ holds for small values of N, say $1 \leq N \leq M$, for some $M > 0$. Next, it assumes that the *inductive hypothesis* $\{\varphi(N-1)\}$ P_{N-1} $\{\psi(N-1)\}$ holds for some $N \geq M+1$. Finally, in the *inductive step*, it shows that if the inductive hypothesis holds, so does $\{\varphi(N)\}$ P_N $\{\psi(N)\}$. It is not hard to see that the inductive step is the most crucial step in this style of reasoning. It is also often the limiting step, since not all programs and properties allow for efficient inferencing of $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ from $\{\varphi(N-1)\}$ P_{N-1} $\{\psi(N-1)\}$.

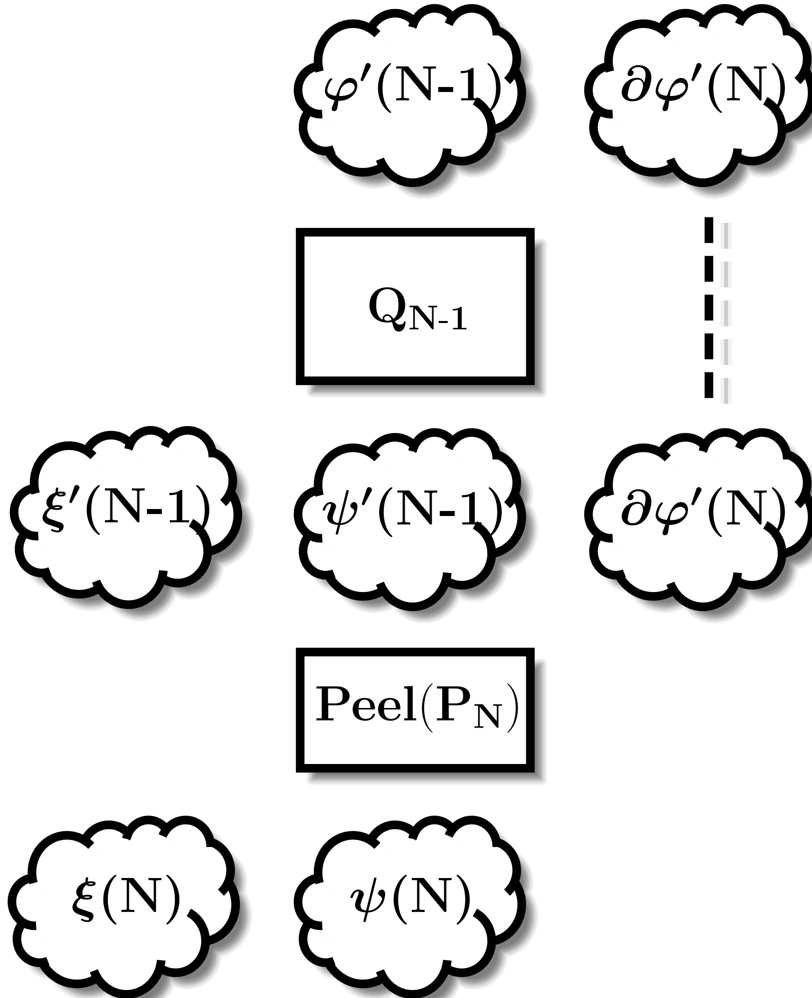


Figure 7.7: Strengthening Pre- and Post-conditions

Like full-program induction, the relational full-program induction technique uses induction on N to prove the Hoare triple $\{\varphi(N)\}\ \ \mathsf{P}_N\ \ \{\psi(N)\}$ for all $N > 0$. Hence, our base case and inductive hypothesis are the same as those in full-program induction. However, our reasoning in the crucial inductive step is significantly different from that in the full-program induction technique, and this is where our primary contribution lies. As we show later, not only does this allow a much larger class of programs to be efficiently verified compared to the prior techniques in the literature, it also permits reasoning about classes of programs with nested loops, that are beyond the reach of the full-program induction technique.

In order to better understand our contribution with relational full-program induction and its difference vis-a-vis the full-program induction technique, we present a quick recap of the inductive step used in the latter technique. The inductive step in full-program induction crucially relies on finding a "difference program" $\partial\mathsf{P}_N$ and a "difference pre-

condition" $\partial\varphi(N)$ such that: (i) P_N is semantically equivalent to P_{N-1}; $\partial\mathsf{P}_N$, where ';' denotes sequential composition of programs, (ii) $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$, and (iii) no variable/array element in $\partial\varphi(N)$ is modified by P_{N-1}. As shown in Section 5.2, once $\partial\mathsf{P}_N$ and $\partial\varphi(N)$ satisfying these conditions are obtained, the problem of proving $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ can be reduced to that of proving $\{\psi(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{\psi(N)\}$. This approach can be very effective if (i) $\partial\mathsf{P}_N$ is "simpler" (e.g. has fewer loops or strictly less deeply nested loops) than P_N and can be computed efficiently, and (ii) a formula $\partial\varphi(N)$ satisfying the conditions mentioned above exists and can be computed efficiently.

The requirement of P_N being semantically equivalent to P_{N-1}; $\partial\mathsf{P}_N$ is a very stringent one, and finding such a program $\partial\mathsf{P}_N$ is non-trivial in general. The full-program induction technique provides a set of syntax-guided conditionally sound heuristics for computing $\partial\mathsf{P}_N$. Unfortunately, when these conditions are violated, there are no known algorithmic techniques to generate $\partial\mathsf{P}_N$ in a sound manner. Suppose a difference program $\partial\mathsf{P}_N$ is found (even if in an ad-hoc manner), it may be as "complex" as P_N itself. This makes the full-program induction less effective for analyzing such programs. As an example, the fourth column of Fig. 7.3 shows P_{N-1} followed by one possible $\partial\mathsf{P}_N$ that ensures P_N (shown in the first column of the same figure) is semantically equivalent to P_{N-1}; $\partial\mathsf{P}_N$. Notice that $\partial\mathsf{P}_N$ in this example has two sequentially composed loops, just like P_N had. In addition, the assignment statement in the body of the second loop uses a more complex expression than that present in the corresponding loop of P_N. Proving $\{\psi(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{\psi(N)\}$ may therefore not be any simpler (perhaps even more difficult) than proving $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$.

In addition to the difficulty of computing $\partial\mathsf{P}_N$, it may be not be possible to find a formula $\partial\varphi(N)$ such that $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$, as required by the full-program induction technique. This can happen for a class of pre-condition formulas, such as $\varphi(N) \equiv \left(\bigwedge_{i=0}^{N-1} A[i] = N\right)$. Notice that there is no $\partial\varphi(N)$ that satisfies $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$ in this case. In such cases, the full-program induction technique cannot be used at all, even if P_N, $\varphi(N)$ and $\psi(N)$ are such that there exists a trivial proof of $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$.

The inductive step proposed in the relational full-program induction technique (in this chapter) largely mitigates the above problems, thereby making it possible to efficiently reason about a much larger class of programs than that possible using the full-program

induction technique.

While generating Q_{N-1} and $\mathsf{Peel}(P_N)$ in relational full-program induction may sound similar to generating P_{N-1} and $\partial\mathsf{P}_N$ from full-program induction, there are fundamental differences between the two approaches. First, as previously noted, P_{N-1} is semantically different from Q_{N-1}. Similarly, $\mathsf{Peel}(\mathsf{P}_N)$ is also semantically different from $\partial\mathsf{P}_N$. Second, we provide an algorithm for generating Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$ that works for a significantly larger class of programs than the full-program induction technique that generates and uses difference programs $\partial\mathsf{P}_N$. Specifically, the algorithm in this chapter works for all programs amenable to the full-program induction technique, and also for programs that violate the restrictions imposed by the grammar (refer Fig. 3.2 in Section 3.1) and the conditional heuristics for computing the difference program $\partial\mathsf{P}_N$ (refer Section 6.2). For example, we can algorithmically generate Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$ even for a class of programs with arbitrarily nested loops – a program feature explicitly disallowed by the grammar in Fig. 3.2. Third, we guarantee that $\mathsf{Peel}(\mathsf{P}_N)$ is "simpler" than P_N in the sense that the maximum nesting depth of loops in $\mathsf{Peel}(\mathsf{P}_N)$ is *strictly less* than that in P_N. Thus, if P_N has no nested loops (all programs amenable to analysis by the full-program induction technique belong to this class), $\mathsf{Peel}(\mathsf{P}_N)$ is guaranteed to be loop-free. As demonstrated by the fourth column of Fig. 7.3, no such guarantees can be given for $\partial\mathsf{P}_N$ generated by the full-program induction technique. This is a significant difference, since it greatly simplifies the analysis of $\mathsf{Peel}(\mathsf{P}_N)$ vis-a-vis that of $\partial\mathsf{P}_N$.

When given the choice between these techniques one has to ponder over the trade-offs between the two techniques. Observe that the inductive hypothesis can be used as giving a part of proof in the inductive step of full-program induction for free. On the contrary, the induction hypothesis needs to be carefully massaged before it can be used as a part of the in the inductive step in relational full-program induction. Depending on the given program, this factor may give a performance edge to one of the techniques.

7.3 Inductive Verification using the Relational Full-Program Induction Technique

Before presenting the algorithms for performing verification using relational full-program induction, we describe an important pre-processing step that renames all scalar and array

variables in the program as follows. We rename each scalar variable using the well-known Static Single Assignment (SSA) [RWZ88] technique, such that the variable is written at (at most) one location in the program. We also rename arrays in the program such that each loop updates its own version of an array and multiple writes to an array element within the same loop are performed on different versions of that array. We use techniques for array SSA [KS98] renaming studied earlier in the context of compilers, for this purpose. In the subsequent exposition, we assume that scalar and array variables in the program are already SSA renamed, and that all array and scalar variables referred to in the pre- and post-conditions are also expressed in terms of SSA renamed arrays and scalars.

The key steps in the application of the relational full-program induction technique, as discussed in Section 7.2, are

1. Generation of Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$ from a given P_N.

2. Generation of $\varphi'(N-1)$ and $\Delta\varphi'(N)$ from a given $\varphi(N)$.

3. Generation of the difference invariant $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$, given $\varphi(N-1)$, $\varphi'(N-1)$, Q_{N-1} and P_{N-1}.

4. Proving $\{\Delta\varphi'(N) \;\wedge\; \exists V_{\mathsf{P}}\big(\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)\}$ $\mathsf{Peel}(\mathsf{P}_N)$ $\{\psi(N)\}$, possibly by generation of $\xi'(N-1)$ and $\xi(N)$ to strengthen the pre- and post-conditions, respectively.

We now discuss techniques for solving each of these sub-problems.

7.3.1 Generating Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$

To concretize the intuition behind our algorithm, we first describe how the programs Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$ are computed for a given program P_N. Consider the program P_N from our motivating example shown in the first column of Fig. 7.3. The second column of this figure shows the program obtained from P_N by peeling the last iteration of each loop of the program. Clearly, the programs in the first and second columns are semantically equivalent. Since there are no nested loops in P_N, the peels (shown in solid boxes) in the second column are loop-free program fragments. For each such peel, we identify variables/array elements modified in the peel and used in subsequent non-peeled parts of the program. For example, the variable `x` is modified in the peel of the first loop and used

in the body of the second loop, as shown by the arrow in the second column of Fig. 7.3. We replace all such uses (if needed, transitively) by expressions on the right-hand side of assignments in the peel until no variable/array element modified in the peel is used in any subsequent non-peeled part of the program. Thus, the use of `x` in the body of the second loop is replaced by the expression `x + N*N` in the third column of Fig. 7.3. The peeled iteration of the first loop can now be moved to the end of the program, since the variables modified in this peel are no longer used in any subsequent non-peeled part of the program. Repeating the above steps for the peeled iteration of the second loop, we get the program shown in the third column of Fig. 7.3. This effectively gives a transformed program that can be divided into two parts: (i) a program Q_{N-1} that differs from P_N only in that all loops are truncated to iterate $N-1$ (instead of N) times, and (ii) a program $\mathsf{Peel}(\mathsf{P}_N)$ that is obtained by concatenating the peels of loops in P_N in the same order in which the loops appeared in P_N. It is not hard to see that if we execute the program P_N, shown in the first column of Fig. 7.3, and the program Q_{N-1}; $\mathsf{Peel}(\mathsf{P}_N)$, shown in the third column of Fig. 7.3, starting from a state σ, and if their executions terminate, then we end up in the same program state.

Notice that the construction of Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$ was fairly straightforward, and did not require any complex reasoning. It is easy to extend this to arbitrary sequential compositions of non-nested loops. Having all variables and arrays renamed, such that each loop accesses its own version, makes it particularly easy to carry out the substitution exemplified by the arrow shown in the second column of Fig. 7.3. In sharp contrast, construction of $\partial\mathsf{P}_N$ (using Algorithm 11 from Chapter 6) requires non-trivial reasoning, and produces a program with two sequentially composed loops, as shown in the bottom half of fourth column of Fig. 7.3.

The case of programs with nested loops is challenging and requires an additional discussion. We start by considering a simple example shown in Fig. 7.8(a). This program has two sequentially composed loops at the top level. The second loop also has a nested loop within it. Following the same ideas we used in Section 5.3.2 for constructing P_N^p, we can peel each of the outer-most loops in Fig. 7.8(a) to obtain the program in Fig. 7.8(b). Notice that the peel of each outer loop is placed immediately after the loop with its bound reduced to $N-1$. It is clear that programs in Fig. 7.8(a) and Fig. 7.8(b) are semantically equivalent (refer Lemma 5.6). The process of transforming the program in

```
x = 0;
for(i=0; i<N; i++)
  x = x + N;

for(i=0; i<N; i++) {
  B[i] = x;
  for(j=0; j<N; j++)
    A[i][j] = N;
}
```

(a)

```
x = 0;
for(i=0; i<N-1; i++)
  x = x + N;
x = x + N;
for(i=0; i<N-1; i++) {
  B[i] = x;
  for(j=0; j<N; j++)
    A[i][j] = N;
}
B[N-1] = x;
for(j=0; j<N; j++)
  A[N-1][j] = N;
```

(b)

```
x = 0;
for(i=0; i<N-1; i++)
  x = x + N;
x = x + N;
for(i=0; i<N-1; i++) {
  B[i] = x;
  for(j=0; j<N-1; j++)
    A[i][j] = N;
  A[i][N-1] = N;
}
B[N-1] = x;
for(j=0; j<N; j++)
  A[N-1][j] = N;
```

(c)

```
x = 0;
for(i=0; i<N-1; i++)
  x = x + N;

for(i=0; i<N-1; i++) {
  B[i] = x + N;
  for(j=0; j<N-1; j++)
    A[i][j] = N;
}

x = x + N;
for(i=0; i<N-1; i++)
  A[i][N-1] = N;
B[N-1] = x;
for(j=0; j<N; j++)
  A[N-1][j] = N;
```

(d)

Figure 7.8: (a) P_N, (b) Program with Outer-most Loops Peeled, (c) P^p_N, and (d) $\mathsf{Q}_{N-1}; \mathsf{Peel}(\mathsf{P}_N)$

Fig. 7.8(a) to the program in Fig. 7.8(b) however left the nested loop non-peeled. This is unsatisfactory for the purposes of our inductive reasoning. We would like the inner loop to also be peeled like the outer loops. Fig. 7.8(c) shows how this can be done where the peel of the inner loop is still within the non-peeled part of the outer loop that contains it. Once again, using reasoning similar to that used in Lemma 5.6, programs in Fig. 7.8(b) and Fig. 7.8(c) are semantically equivalent. Notice that there are two kinds of peels in the program shown in Fig. 7.8(c), marked using blue and red colored boxes. The peel in the blue colored box appears within the non-peeled part of an enclosing loop, while the peels in red colored boxes do not appear within any enclosing loop. This motivates us to ask: *How should these peels be moved to the end of the program while preserving the semantics of the original program?*

Fig. 7.8(d) shows how this can be done. There are two important points to note here: (i) the peel of the inner loop shown in the blue box appears within a loop when we move it to the end of the program, (ii) the read-after-write dependence between the peeled statement `x = x + N;` and the statement `B[i] = x;` in the non-peeled part of the second loop in Fig. 7.8(c) has resulted in the right hand side of the assignment `B[i] = x;` to change to `B[i] = x + N;`. If we were to use the same notation as used in the previous example shown in Fig. 7.3, then the part of the program in Fig. 7.8(d) before the boxed statements would constitute Q_{N-1} and the remainder of the program would constitute the peel of the original program (with nested loops). The above example motivates us to under take a careful study of how to construct peels of programs with nested loops and how to construct Q_{N-1} for such programs.

Generating Peels for Nested Loops

Consider a pair of abstract nested loops, L_1 and L_2, as shown in Fig. 7.9. Suppose that `B1` and `B3` are loop-free code fragments in the body of L_1 that precede and succeed the nested loop L_2. Suppose further that the loop body, `B2`, of L_2 is loop-free.

The peel of the abstract nested loop L_1 is as shown in Fig. 7.10. The first loop in the peel includes the last iteration of L_2 in each of the $N-1$ iterations of L_1, that was missed in Q_{N-1}. The subsequent code includes the last iteration of L_1 that was missed in Q_{N-1}.

Formally, we use the notation L_1`(N)` to denote a loop L_1 that has no nested loops

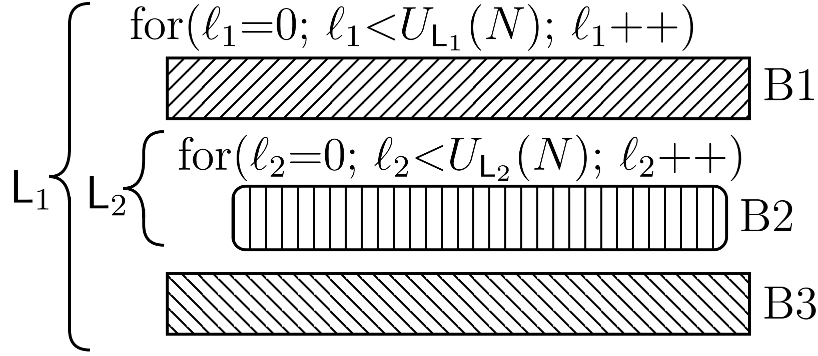


Figure 7.9: A Generic Nested Loop

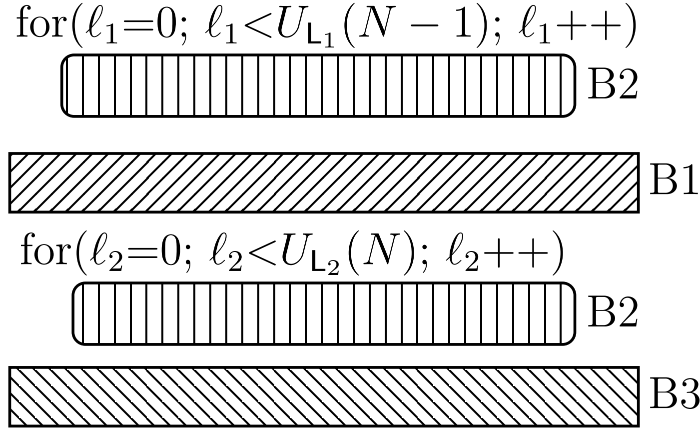


Figure 7.10: Peel of the Nested Loop

within it, and its loop counter, say ℓ_1, increases from 0 to an upper bound that is given by an expression in N. Similarly, we use `L`$_1$`(N, L`$_2$`(N))` to denote a loop L_1 that has another loop L_2 nested within it. The loop counter ℓ_1 of L_1 increases from 0 to an upper bound expression in N, while the loop counter ℓ_2 of L_2 increases from 0 to an upper bound expression in ℓ_1 and N. Using this notation, `L`$_1$`(N, L`$_2$`(N, L`$_3$`(N)))` represents three nested loops, and so on. Notice that the upper bound expression for a nested loop can depend not only on N but also on the loop counters of other loops nesting it. For notational clarity, we define a macro `LPeel(L`$_i$`, a, b)` as follows:

Definition 7.1 `LPeel(L`$_i$`, a, b)` *denotes the peel of loop* L_i *consisting of all iterations of* L_i *where the value of* ℓ_i *ranges from* `a` *to* `b-1`*, both inclusive.*

Note that if `b-a` is a constant, `LPeel(L`$_i$`, a, b)` corresponds to the concatenation of `(b-a)` peels of L_i. If L_i is a non-nested loop and `b-a` is a constant, then `LPeel(L`$_i$`, a, b)` is loop-free.

We will now try to see how we can implement the transformation from Fig. 7.8(a) to Fig. 7.8(c) for a nested loop L_1(N, L_2(N)). The first step is to truncate all loops to use $N-1$ instead of N in the upper bound expressions. Using the notation introduced above, this gives the loop L_1(N-1, L_2(N-1)). Note that all uses of N other than in loop upper bound expressions stay unchanged as we go from L_1(N, L_2(N)) to L_1(N-1, L_2(N-1)). We now ask: *Which are the loop iterations of* L_1(N, L_2(N)) *that have been missed (or skipped) in going to* L_1(N-1, L_2(N-1))*?* Let the upper bound expression of L_1 in L_1(N, L_2(N)) be $U_{\mathsf{L}_1}(N)$, and that of L_2 be $U_{\mathsf{L}_2}(\ell_1, N)$. It is not hard to see that in every iteration ℓ_1 of L_1, where $0 \leq \ell_1 < U_{\mathsf{L}_1}(N-1)$, the iterations corresponding to $\ell_2 \in \{U_{\mathsf{L}_2}(\ell_1, N-1), \ldots, U_{\mathsf{L}_2}(\ell_1, N)-1\}$ have been missed. In addition, all iterations of L_1 corresponding to $\ell_1 \in \{U_{\mathsf{L}_1}(N-1), \ldots, U_{\mathsf{L}_1}(N)-1\}$ have also been missed. This implies that the "peel" of L_1(N, L_2(N)) must include all the above missed iterations. This peel therefore is the program fragment shown in Fig. 7.11.

$$
\begin{array}{l}
\texttt{for(}\ell_1\texttt{=0; }\ell_1\texttt{<}U_{\mathsf{L}_1}\texttt{(N-1); }\ell_1\texttt{++)} \\
\quad \texttt{LPeel(}\mathsf{L}_2\texttt{, }U_{\mathsf{L}_2}\texttt{(}\ell_1\texttt{,N-1), }U_{\mathsf{L}_2}\texttt{(}\ell_1\texttt{,N))} \\
\texttt{LPeel(}\mathsf{L}_1\texttt{, }U_{\mathsf{L}_1}\texttt{(N-1), }U_{\mathsf{L}_1}\texttt{(N))}
\end{array}
$$

Figure 7.11: Peel of L_1(N, L_2(N))

Notice that if U_{L_2}(ℓ_1,N) - U_{L_2}(ℓ_1,N-1) is a constant (as is the case if U_{L_2}(ℓ_1,N) is any linear function of ℓ_1 and N), then the peel does not have any loop with nesting depth 2. Hence, the maximum nesting depth of loops in the peel is strictly less than that in L_1(N, L_2(N)), yielding a peel that is "simpler" than the original program. This argument can be easily generalized to loops with arbitrarily large nesting depths. The peel of L_1(N, L_2(N, L_3(N))) is as shown in Fig. 7.12.

Consider the program in Fig. 7.8(a). Suppose we wish to compute the peel of this program containing sequentially composed loops L_1 and L_2, with L_3 nested within L_2. In this case, the upper bounds of all three loops in the program are $U_{\mathsf{L}_1}(N) = U_{\mathsf{L}_2}(N) = U_{\mathsf{L}_3}(N) = N$. The peel is shown using boxed statements in Fig. 7.8(d). It consists of two sequentially composed non-nested loops. The first loop takes into account the missed iterations of the inner loop L_3 (a single iteration in this example, shown in blue colored box) that are executed in P_N but are missed in Q_{N-1}. The second loop takes into account

```
for(ℓ1=0; ℓ1<U_L1(N-1); ℓ1++) {
   for(ℓ2=0; ℓ2<U_L2(ℓ1,N-1); ℓ2++)
      LPeel(L3, U_L3(ℓ1,ℓ2,N-1), U_L3(ℓ1,ℓ2,N))
   LPeel(L2, U_L2(ℓ1,N-1), U_L2(ℓ1,N))
}
LPeel(L1, U_L1(N-1), U_L1(N))
```

Figure 7.12: Peel of L_1(N, L_2(N, L_3(N)))

the iterations of the outer loop L_2 that are executed in P_N but are missed in Q_{N-1}.

The ideas presented above generalize easily to programs with nested loops having arbitrary nesting depths. Recall that Definition 5.2 in Section 5.4.4 defines $\mathsf{Peel}(\mathsf{P}_N)$ for programs with non-nested loops. We now extend the definition of $\mathsf{Peel}(\mathsf{P}_N)$ to programs with (possibly multiple, and sequentially composed) nested loops. For convenience, we refer to the body of a loop L_i as $\texttt{Body}(\mathsf{L}_i)$. The following recursive definition formally defines the program $\mathsf{Peel}(\mathsf{P}_N)$.

Definition 7.2 *For a loop-free program P_N, $\mathsf{Peel}(\mathsf{P}_N)$ is an empty program. For a program P_N consisting of sequentially composed loops L_1, L_2, ..., L_m, where each L_i, $1 \leq i \leq m$, may have a loop nested within it, $\mathsf{Peel}(\mathsf{P}_N)$ is computed as follows:*

$$\begin{aligned}
\mathsf{Peel}(\mathsf{P}_N) &:= \mathsf{Peel}(\mathsf{L}_1);\ \mathsf{Peel}(\mathsf{L}_2);\ \ldots;\ \mathsf{Peel}(\mathsf{L}_m); \\
\mathsf{Peel}(\mathsf{L}_i) &:= \texttt{for}(\ell_i = 0;\ \ell_i < U_{\mathsf{L}_i}(N-1);\ \ell_i++)\ \{\ \mathsf{Peel}(\mathsf{Body}(\mathsf{L}_i));\ \} \\
&\quad\ \texttt{LPeel}(\mathsf{L}_i, U_{\mathsf{L}_i}(N-1), U_{\mathsf{L}_i}(N));
\end{aligned}$$

Generating the Program Q_{N-1}

Recall from Section 5.4.2 that when peeled statements are moved to the end of a program certain read-after-write and write-after-write dependencies in the original program can be violated in general. If such dependencies involve affected variables, these can sometimes be taken care of by introducing additional statements in the peel as discussed in Section 6.2. In the current section, we describe how to take care of such violated dependencies by modifying some statements in the non-peeled part of the program without disturbing

the peel.

To understand this better, consider the example shown in 7.8(c). The value of variable `x` computed by the statement `x = x + N;` in the peel of the first loop is used by the statement `B[i] = x;` within the non-peeled part of the second loop. This creates a read-after-write dependence which would get violated if we move the peel to the end of the program. Suppose we use `x + N`, i.e. right hand side expression of the assignment `x = x + N;`, in place of `x` in the statement `B[i] = x;` in the non-peeled part of the second loop. Then the peeled statement `x = x + N;` can be safely moved to the end of the program while having compensated for the violated read-after-write dependence. Note that the same idea was also used in the previous example (Fig. 7.3). Below, we discuss how this idea can be generalized.

Recall from Section 5.3.2 that if P_N has no nested loops, then P_N^p is the program obtained by peeling each loop in the program P_N and by placing the peel immediately after the loop with appropriately reduced iteration bound. For clarity of exposition, we call this transformation "peeling loops in-place". If P_N has nested loops, we extend the definition of P_N^p to denote the program obtained by recursively peeling each loop in-place. Specifically, we first peel in-place each loop at nesting depth 1 in P_N. Next, we peel in-place each loop at nesting depth 2 in the non-peeled part of each loop at nesting depth 1 in the resulting program. We continue this iteratively all the way until the inner-most nested loops have been peeled in-place. Since peeling a loop in-place preserves the semantics of the program, Lemma 5.6 continues to hold for programs with this extended definition of P_N^p.

Our goal now is to see if the peeled statements in P_N^p can be moved to the end of the program without affecting its semantics, like what we did in Section 6.2. Towards this end, we now construct the dependence graph (as described in Section. 5.3.3) for this program to identify all read-after-write and write-after-write dependencies. We now identify if there are data dependencies where a variable/array element is in the *def* set of a peeled statement, say S_1, and is also in the *use* set of a statement, say S_2, that is not present in any peel. Every such data dependence is potentially violated when we move peels to the end of the program. We try to compensate for such violations for read-after-write dependencies by substituting the expression in the right hand side of the assignment statement at S_1 for the variable/array element of interest in statement S_2. Violations of

write-after-write dependencies between statements S_1 and S_2 cannot be compensated for just by expression substitution, and hence, we do not handle such cases currently.

To keep the exposition simple, we preclude violations of read-after-write and write-after-write dependencies arising from moving of peels corresponding to nested loops by imposing some restrictions on nested loops. In order to formalize these requirements, we introduce some notation. Given an execution trace of P_N, the *iteration index* of a statement S in the trace gives the value of loop counter ℓ_i for each loop L_i, such that S appears in the loop body of L_i. As an example, consider the nested loop L_1 shown in Fig. 7.9. While executing a statement in block B1 during the first iteration of loop L_1 the iteration index is $(\ell_1 \mapsto 1)$ and while executing a statement in block B2 during the last iteration of loop L_1 and the first iteration of L_2 the iteration index is $(\ell_1 \mapsto N-1, \ell_2 \mapsto 1)$. Assuming loop L_2 is nested within loop L_1 the restrictions on nested loops can now be stated as follows:

1. A scalar variable/array element written in a statement S_1 with iteration index $(\ell_1 \mapsto x_1, \ell_2 \mapsto x_2)$ where $0 \le x_1 < U_{\mathsf{L}_1}(N-1)$ and $U_{\mathsf{L}_2}(N-1) \le x_2 < U_{\mathsf{L}_2}(N)$ must not be used/written in any statement S_2 with iteration index $(\ell_1 \mapsto x_1')$ or $(\ell_1 \mapsto x_1', \ell_2 \mapsto x_2')$ where $x_1 \le x_1' < U_{\mathsf{L}_1}(N-1)$ and $0 \le x_2' < U_{\mathsf{L}_2}(N-1)$.

2. A scalar variable/array element used/written in a statement S_1 with iteration index $(\ell_1 \mapsto x_1, \ell_2 \mapsto x_2)$ where $0 \le x_1 < U_{\mathsf{L}_1}(N-1)$ and $U_{\mathsf{L}_2}(N-1) \le x_2 < U_{\mathsf{L}_2}(N)$, must not be subsequently written in any statement S_2 with iteration index $(\ell_1 \mapsto x_1')$ or $(\ell_1 \mapsto x_1', \ell_2 \mapsto x_2')$ where $x_1 \le x_1' < U_{\mathsf{L}_1}(N-1)$ and $0 \le x_2' < U_{\mathsf{L}_2}(N-1)$.

The restrictions above pertain to the abstract nested loops L_1 and L_2 shown in Fig. 7.9. These restrictions can be easily extended to any pair of loops L_i, L_j such that L_j is nested within L_i at nesting depth > 2. It is important to note that these restrictions are imposed primarily to simplify the exposition and the correctness proofs. It is possible to use our technique even with some relaxations of these restrictions, for example, by resorting to expression substitution, loop summarization and over-approximation. We present some of these optimizations later along with the description of the algorithm for generating Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$. The following lemma asserts an important property of programs with loops that satisfy the above mentioned restrictions.

Lemma 7.1 *Consider the program* P_N *given by* L_1`(N, `L_2`(N))`*, i.e.* L_2 *is nested within* L_1*. Suppose there are no sequentially composed loops at any nesting depth in* P_N *and suppose* L_1*,* L_2 *satisfy conditions 1 and 2. Then the set of read-after-write and write-after-write dependencies between statements in* P_{N-1}; $\mathsf{Peel}(\mathsf{P}_N)$ *are exactly the same as those between corresponding statements in* P_N*.*

Proof. Since conditions 1 and 2 are satisfied, the following hold:

- There exists no read-after-write or write-after-write data dependence in P_N where a variable/array element is written in a statement in the peel of L_2 and is subsequently read/written in a statement in the non-peeled part of L_1 or L_2. This vacuously ensures that no read-after-write or write-after-write dependencies originally present in P_N are violated in P_{N-1}; $\mathsf{Peel}(\mathsf{P}_N)$.

- There exists no write-after-read or write-after-write data dependence in P_N where a variable/array element is used/written in a statement in the peel of L_2 and it is subsequently written in a statement in the non-peeled part of L_1 or L_2. This vacuously ensures that no read-after-write or write-after-write dependencies that were not originally present in P_N are introduced in P_{N-1}; $\mathsf{Peel}(\mathsf{P}_N)$.

The two cases considered above together show that the set of read-after-write and write-after-write dependencies in P_{N-1}; $\mathsf{Peel}(\mathsf{P}_N)$ are exactly the same as those in P_N. □

For programs with loops having nesting depth > 2 the above lemma can be applied to each pair of loops L_i, L_j such that L_j is nested within L_i. This effectively allows us to generalize this lemma to programs with nested loops having arbitrary nesting depth.

Theorem 7.1 *If a program* P_N *has no sequentially composed loops at any nesting depth, then* P_{N-1}; $\mathsf{Peel}(\mathsf{P}_N)$ *is semantically equivalent to* P_N*. Hence,* P_{N-1} *serves as* Q_{N-1}*.*

Proof. Follows immediately from Lemma 7.1. □

We now consider what happens when the program P_N has sequentially composed loops at some nesting depth. Let P_N^p denote the program with in-place peeled loops. Let $\mathsf{P}_N^\star$ denote the program obtained by removing the peels of all loops, including nested loops from P_N^p. It is not hard to see that the program P_{N-1}; $\mathsf{Peel}(\mathsf{P}_N)$ may not be semantically

equivalent to P_N, in general. This can happen because of read-after-write and write-after-write dependencies between a statement in the in-place peel of loop L_1 and a statement in the non-peeled part of loop L_2, where L_1 precedes L_2 in the composition order. In such cases, we need to construct a program Q_{N-1} such that Q_{N-1}; $\mathsf{Peel}(\mathsf{P}_N)$ is semantically equivalent to P_N.

We now give a formal definition for the program Q_{N-1}.

Definition 7.3 *Let* L_1 *and* L_2 *be a pair of sequentially composed loops at the same nesting depth in* P_N^p *(and hence, in* $\mathsf{P}_N^\star$*). Let* L_1 *precede* L_2 *in the composition order. Suppose there is a variable/array element* vA *in the def set of a statement* S_1 *in* $\mathsf{Peel}(\mathsf{L}_1)$ *that is also in the use set of statement* S_2 *in the non-peeled part of* L_2*. Let* E *be an expression that gives the value of* vA *after executing* $\mathsf{Peel}(\mathsf{L}_1)$ *in terms of the values of variables before* $\mathsf{Peel}(\mathsf{L}_1)$ *is executed. Then,* Q_{N-1} *is the program obtained from* $\mathsf{P}_N^\star$ *by substituting the use of each such variable/array element* vA *in* S_2 *with the corresponding expression* E*.*

Identifying the expression E mentioned in the definition above may not always be easy. Specifically, when the peel of a loop L_i has another loop L_j within it that computes the value of vA, then identifying the expression E that gives the value of vA after executing $\mathsf{Peel}(\mathsf{L}_i)$ in terms of the values of variables and arrays before $\mathsf{Peel}(\mathsf{L}_i)$ is executed may be difficult. In such cases, to simplify the identification of the expression E mentioned in the definition above, the effect of loops L_j in peels can be summarized, whenever possible, to compute a closed form for L_j while generating the program Q_{N-1}. Such loops L_j occurring in the peel of loops L_i can also be summarized in $\mathsf{Peel}(\mathsf{P}_N)$ to simplify the inductive step of the reasoning. We now present the algorithm for computing Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$, which discusses some of these cases in detail.

Algorithm to Compute Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$

Generalizing the above intuition, Algorithm 16 presents function GenQandPeel for computing Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$ for a given program P_N that has loops L_1, L_2, ..., L_m in the sequential composition order, where each loop L_i may have loops nested within it. Due to the grammar of our programs (refer Section 3.1), our loops are well nested.

At a high-level, computing Q_{N-1} consists of peeling each loop in-place and then propagating these peels across subsequent loops. We call the routine PeelAllLoopsInPlace, on line 2, to compute the program P_N^p that has each loop peeled in-place, including

Algorithm 16 GenQandPeel(P_N: program)

1: Let sequentially composed loops in P_N be in the program order L_1, L_2, ..., L_m;
2: P_N^p := PeelAllLoopsInPlace(P_N);
3: Q_{N-1} := RemoveInPlacePeels(P_N^p); ▷ At this point Q_{N-1} is same as $\mathsf{P}_N^\star$
4: **for** each loop $\mathsf{L}_i \in$ TopLevelLoops(P_N) **do**
5:     $\mathsf{Peel}(\mathsf{L}_i)$:= GenPeel(L_i); ▷ Procedure of GenPeel is given below
6: **for** each pair of loops $\mathsf{L}_i, \mathsf{L}_j \in \mathsf{P}_N^p$ for some i, j s.t. $1 \leq i < j \leq m$ **do**
7:     Let S_1, S_2 be statements in $\mathsf{Peel}(\mathsf{L}_i)$ and the non-peeled part of L_j respectively;
8:     **if** $\exists vA.\ vA \in def(S_1) \cap use(S_2)$ **then**
9:         **if** $\mathsf{Peel}(\mathsf{L}_i)$ has loops nested within it **then**
10:             $\mathsf{Peel}(\mathsf{L}_i)$:= TransformPeel($\mathsf{Peel}(\mathsf{L}_i)$);
        ▷ Routine TransformPeel attempts to summarize/over-approximate loops
11:         **if** $\mathsf{Peel}(\mathsf{L}_i)$ is loop-free **then**
12:             E := ComputeSymbolicExpr(vA, $\mathsf{Peel}(\mathsf{L}_i)$);
13:             Q_{N-1} := Substitute the use of vA at S_2 in loop $\mathsf{L}_j \in \mathsf{Q}_{N-1}$ with E;
14:         **else** **abort**;
15: $\mathsf{Peel}(\mathsf{P}_N)$:= $\mathsf{Peel}(\mathsf{L}_1)$; $\mathsf{Peel}(\mathsf{L}_2)$; ...; $\mathsf{Peel}(\mathsf{L}_m)$;
16: **return** $\langle \mathsf{Q}_{N-1}, \mathsf{Peel}(\mathsf{P}_N) \rangle$;

17: **procedure** GenPeel(L_i: loop)
18:     **if** L_i has loops nested within it **then**
19:         Let loops nested in L_i be in sequential composition order L_{i_1}, L_{i_2}, ..., L_{i_n};
20:         $\mathsf{Peel}(\mathsf{L}_i)$:= `for(i=0; i<`$U_{\mathsf{L}_i}(N-1)$`; i++) {` GenPeel(L_{i_1}); GenPeel(L_{i_2}); ...; GenPeel(L_{i_n}); `}` `LPeel(`L_i`, `$U_{\mathsf{L}_i}(N-1)$`, `$U_{\mathsf{L}_i}(N)$`);`
21:     **else**
22:         $\mathsf{Peel}(\mathsf{L}_i)$:= `LPeel(`L_i`, `$U_{\mathsf{L}_i}(N-1)$`, `$U_{\mathsf{L}_i}(N)$`);`
23:     **return** $\mathsf{Peel}(\mathsf{L}_i)$;

nested loops. Next, on line 3, we use the routine RemoveInPlacePeels to initialize Q_{N-1} with a program where the in-place peels of all loops, including nested loops, are removed. At this point, the program Q_{N-1} is the same as the program $\mathsf{P}_N^\star$.

We now require the peel of each loop in the program P_N for the subsequent part

of the computation of Q_{N-1}. The loop on line 4 iterates over the top level sequentially composed loops L_i, $1 \leq i \leq m$, in P_N and invokes the routine GenPeel(L_i) to generate $\mathsf{Peel}(\mathsf{L}_i)$ on line 5. The routine GenPeel, presented in lines 17 – 23, recursively computes the peel of L_i. In doing so, it also ends up computing the peels of loops nested within L_i.

Recall that $\mathsf{Peel}(\mathsf{L}_i)$ consists of missed loop iterations of L_i in the program P_N. We identify the missed iterations of each loop in the program P_N from the upper bound expression UB. The upper bound of each loop L_k at nesting depth k, denoted by U_{L_k} is in terms of the loop counters $\ell_1 \leq \ell_2 \leq \ldots \leq \ell_{k-1}$ of outer nested loops and the program parameter N. We need to peel $U_{\mathsf{L}_k}(\ell_1, \ell_2, \ldots, \ell_{k-1}, N) - U_{\mathsf{L}_k}(\ell_1, \ell_2, \ldots, \ell_{k-1}, N-1)$ number of iterations from each loop, where $\ell_1 \leq \ell_2 \leq \ldots \leq \ell_{k-1}$ are counters of the outer nesting loops. As previously discussed, whenever this difference is a constant value, we are guaranteed that the loop nesting depth reduces by one. For non-nested loops L_i, its peel is just the last few iterations of L_i. It is computed on line 22 using the macro `LPeel` and the upper bound expression U_{L_i}. It may so happen that there are multiple sequentially composed loops L_{i_j}, $1 \leq j \leq n$ nested within loop L_i and not just a single loop. For loops L_i that have other loops $\mathsf{L}_{i_1}, \mathsf{L}_{i_2}, \ldots, \mathsf{L}_{i_n}$ nested within it, we recursively build the peel of L_i on line 20 as per Definition 7.2. We again use the routine GenPeel to compute $\mathsf{Peel}(\mathsf{L}_{i_j})$ of each loop L_{i_j} nested within L_i. The peeled iterations of $\mathsf{L}_{i_1}, \mathsf{L}_{i_2}, \ldots, \mathsf{L}_{i_n}$ were missed in the first $U_{\mathsf{L}_i}(N-1)$ iterations of L_i in P_N. Hence, these are placed within a loop in the peel of L_i where the loop counter goes from 0 to $U_{\mathsf{L}_i}(N-1)$ as shown in line 20. The last few iterations of L_i that were executed in P_N but missed in Q_{N-1} are also placed in $\mathsf{Peel}(\mathsf{L}_i)$ using the macro `LPeel` and the upper bound expression U_{L_i} (line 20).

Since all the peels of all loops are moved at the end of the program Q_{N-1}, we need to repair the expressions appearing in the loops in Q_{N-1}. The repairs are applied by the loop on line 6. The loop iterates over each pair of sequentially composed loops L_i and L_j at the same nesting depth in P_N^p, such that L_i precedes L_j in the composition order. We look for statements S_1 in $\mathsf{Peel}(\mathsf{L}_i)$ that update the value of a variable/array element vA such that it is subsequently used in a statement S_2 in the non-peeled part of L_j (checked on line 8). We identify an expression E that gives the value of a variable/array element vA at S_2 in terms of the values of variables and array elements prior to executing $\mathsf{Peel}(\mathsf{L}_i)$ on line 12. In the repair step, on line 13, we substitute the uses of such variables and array elements vA at statement S_2 in $\mathsf{L}_j \in \mathsf{Q}_{N-1}$ with the computed expression E.

Note that the loop in lines 6 – 14 implements the substitution represented by the arrow in the second column of Fig. 7.3. This is necessary in order to move the peel of a loop to the end of the program. If the variables/arrays modified in the peel of a loop are not used later, then the loop condition on line 8 is not satisfied. In such cases, the peel can be trivially moved. $\mathsf{Peel}(\mathsf{L}_i)$ itself may have a loop that computes the value of vA (checked on line 9). This makes it significantly more challenging to identify the expression E to be substituted in Q_{N-1}. We use several optimizations to transform the loops in $\mathsf{Peel}(\mathsf{L}_i)$ before trying to identify such an expression. If the modified values in the peel can be summarized as closed form expressions, then we can replace the loop in the peel with its summary. For example, consider the loop `for (`ℓ_1`=0;` ℓ_1`<N;` ℓ_1`++)` { `S = S + 1;` } that occurs within the peel of a loop. This loop is summarized as `S = S + N;` before it can be moved across subsequent code. In several cases, loops in the peel can also be substituted with their conservative over-approximation. The routine TransformPeel, invoked on line 10, carries out these transformations when there are loops in $\mathsf{Peel}(\mathsf{L}_i)$. We have implemented these optimizations in our tool and are able to verify several benchmarks with sequentially composed loops that have loops nested within them. It may not always be possible to transform a peel that has loops and move it across subsequent loops. For example, if a loop uses array elements as index to other arrays then it can be difficult to identify the expression E to be used for substitution in Q_{N-1}. In such cases, the procedure aborts on line 14. However, such scenarios occur less often, and hence, they hardly impact the effectiveness of our technique. We have observed that the optimizations mentioned here suffice for a large class of programs seen in practice.

Finally, the peels of all top level loops are stitched together on line 15. The computed programs Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$ are returned on line 16.

Theorem 7.2 *Let Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$ be generated by the application of function* GenQandPeel *from Algorithm 16 on program P_N. If P_N and Q_{N-1}; $\mathsf{Peel}(\mathsf{P}_N)$ are executed starting from the same state σ, then both programs terminate in the same state.*

Proof. By Lemma 5.6, if P_N and P_N^p are executed from the same state σ, then each variable/array element vA has the same value on termination of both programs.

We first consider programs that have no sequentially composed loops at any nesting depth. Let L_i`(N)` be a loop with the highest nesting depth $k = 1$ (i.e. it does not have any

loops nested within it). Then, it is easy to see that decomposing L_i`(N)` into L_i`(N-1)` and $\mathsf{Peel}(\mathsf{L}_i)$ preserves the semantics of the loop. Now let loop L_i`(N)` having the highest nesting depth $k > 1$ be decomposed into L_i`(N-1)` and $\mathsf{Peel}(\mathsf{L}_i)$. Conditions 1 and 2 ensure that the set of read-after-write and write-after-write dependencies between statements in the decomposition L_i`(N-1);` $\mathsf{Peel}(\mathsf{L}_i)$ are exactly the same as those between the corresponding statements in L_i`(N)` (refer proof of Lemma 7.1). Hence, if a nested loop L_i`(N)` and its decomposition L_i`(N-1);`$\mathsf{Peel}(\mathsf{L}_i)$ are executed from the same state σ, then both of them terminate in the same state.

We now consider programs with sequentially composed loops. Let L_i and L_j a pair of sequentially composed loops in P_N^p such that L_i precedes L_j. Let S_1 be a statement in $\mathsf{Peel}(\mathsf{L}_i)$, and S_2 be a statement in the non-peeled part of the loop L_j. There cannot be a write-after-write data dependence from S_1 to S_2, since renaming ensures that each loop updates its own version of scalar variables and arrays. Hence, there can only be a read-after-write data dependence, if at all, from the statement S_1 in $\mathsf{Peel}(\mathsf{L}_i)$ to the statement S_2 in the non-peeled part of L_j. The substitution performed on line 13 compensates for such read-after-write dependencies, ensuring that $\mathsf{Peel}(\mathsf{L}_i)$ can be moved across subsequent loops L_j without affecting the program semantics. This ensures that if P_N and Q_{N-1}; $\mathsf{Peel}(\mathsf{P}_N)$ are executed starting from the same state σ, then both programs terminate in the same state. □

Lemma 7.2 *Suppose the following conditions hold;*

1. *Program P_N satisfies our syntactic restrictions (see Section 3.1 of Chapter 3).*
2. *The upper bound expressions of all loops are linear expressions in N and in the loop counters of outer nesting loops.*

Then, the max nesting depth of loops in $\mathsf{Peel}(\mathsf{P}_N)$ is strictly less than that in P_N.

Proof. Let $U_{\mathsf{L}_k}(\ell_1, \ldots, \ell_{k-1}, N)$ be the upper bound expression of a loop L_k at nesting depth k. Suppose $U_{\mathsf{L}_k} = c_1.\ell_1 + \cdots c_{k-1}.\ell_{k-1} + C.N + D$, where $c_1, \ldots c_{k-1}, C$ and D are constants. Then $U_{\mathsf{L}_k}(\ell_1, \ldots, \ell_{k-1}, N) - U_{\mathsf{L}_k}(\ell_1, \ldots \ell_{k-1}, N-1) = C$, i.e. a constant. Now, recalling the discussion in Section 7.3.1, we see that `LPeel(`L_k`,` $U_k(\ell_1, \ldots, \ell_{k-1}, N-1)$`,` $U_k(\ell_1, \ldots, \ell_{k-1}, N)$`)` simply results in concatenating a constant number of peels of the loop

L_k. Hence, the maximum nesting depth of loops in `LPeel(`L_k`,` $U_k(\ell_1, \ldots, \ell_{k-1}, N-1)$`,` $U_k(\ell_1, \ldots, \ell_{k-1}, N)$`)` is strictly less than the maximum nesting depth of loops in L_k.

Suppose loop L with nested loops (having maximum nesting depth t) is passed as the argument of the recursive function GenPeel (see Algorithm 16). In line 20 of function GenPeel, we recursively compute the peels of all loops from nesting depth 2 and above within L. Let L_k be a loop at nesting depth k, where $2 \leq k \leq t$. Clearly, L_k can have at most $t-k$ nested levels of loops within it. When `LPeel` is invoked on such a loop, the maximum nesting depth of loops in the peel generated for L_k can be at most $k-1$, due to the premise that the upper bound expressions of all loops are linear expressions in N and in the loop counters of outer nesting loops. From line 20 of function GenPeel, we also know that this `LPeel` can itself appear at nesting depth $t-k$ of the overall peel $\mathsf{Peel}(\mathsf{L})$. Hence, the maximum nesting depth of loops in $\mathsf{Peel}(\mathsf{L})$ can be $k-1+t-k$, i.e. $t-1$. This is strictly less than the maximum nesting depth of loops in L. □

Corollary 7.1 *If* P_N *has no nested loops, then* $\mathsf{Peel}(\mathsf{P}_N)$ *is loop-free.*

Proof. Follows from Lemma 7.2. □

7.3.2 Generating $\varphi'(N-1)$ and $\Delta\varphi'(N)$

Suppose $\varphi(N)$ is of the form $\bigwedge_{i=0}^{N-1} \rho_i$ (resp. $\bigvee_{i=0}^{N-1} \rho_i$), where ρ_i is a formula on the ith elements of one or more arrays, and scalars used in P_N. We infer $\varphi'(N-1)$ to be $\bigwedge_{i=0}^{N-2} \rho_i$ (resp. $\bigvee_{i=0}^{N-2} \rho_i$) and $\Delta\varphi'(N)$ to be ρ_{N-1} (assuming variables/array elements in ρ_{N-1} are not modified by Q_{N-1}). Note that all uses of N in ρ_i are retained as is (i.e. not changed to $N-1$) in $\varphi'(N-1)$. In general, when deriving $\varphi'(N-1)$, we do not replace any use of N in $\varphi(N)$ by $N-1$ unless it is the limit of an iterated conjunct/disjunct as discussed above. Specifically, if $\varphi(N)$ does not contain an iterated conjunct/disjunct as above, then we consider $\varphi'(N-1)$ to be the same as $\varphi(N)$ and $\Delta\varphi'(N)$ to be True. Thus, our generation of $\varphi'(N-1)$ and $\Delta\varphi'(N)$ differs from that of the full-program induction technique. As discussed earlier, this makes it possible to reason about a much larger class of pre-conditions than that admissible by the full-program induction technique.

Lemma 7.3 *The difference pre-condition* $\Delta\varphi'(N)$ *is such that (i)* $\varphi(N) \Rightarrow (\varphi'(N-1) \odot \Delta\varphi'(N))$*, where* $\odot$ *is operator* $\wedge$ *if* $\varphi(N)$ *is an iterated conjunct, and* $\odot$ *is operator* $\vee$ *if* $\varphi(N)$ *is an iterated disjunct, and (ii)* Q_{N-1} *does not modify variables/arrays in* $\Delta\varphi'(N)$.

Proof. Follows naturally from the construction of $\varphi'(N-1)$ and $\Delta\varphi'(N)$. $\square$

7.3.3 Inferring Inductive Relational Invariants

Once we have P_{N-1}, Q_{N-1}, $\varphi(N-1)$ and $\varphi'(N-1)$, we infer relational invariants. We introduce the concept of *difference invariants* that specializes the relation between the values of corresponding variables/arrays in Q_{N-1} and P_{N-1} by considering only the difference of their values. We assume that programs Q_{N-1} and P_{N-1} operate on disjoint copies of variables and arrays (this can be easily achieved by adding a suffix to the names of variables/arrays in Q_{N-1}). We construct the cross-product [ZP08] of programs Q_{N-1} and P_{N-1}, and infer difference invariants at key control points in the cross-product program. A cross-product between two programs Q_{N-1} and P_{N-1} is a program $\mathsf{Q}_{N-1} \times \mathsf{P}_{N-1}$ in which the corresponding statements from Q_{N-1} and P_{N-1} are executed synchronously in lockstep. Recall that the programs Q_{N-1} and P_{N-1} are structurally similar, even though they may not be semantically equivalent. Specifically, these programs are guaranteed to have synchronized iterations of corresponding loops (since both programs are obtained from P_N by restricting the upper bounds of all loops to use $N-1$ instead of N). However, the conditional statements within the loop body may not always be synchronized. Thus, whenever we can infer that the corresponding conditions $c' \in \mathsf{Q}_{N-1}$ and $c \in \mathsf{P}_{N-1}$ are equivalent, we synchronize the branches of the conditional statement. Otherwise, we consider all four possibilities of the branch conditions namely, $\langle c, c' \rangle$, $\langle c, \neg c' \rangle$, $\langle \neg c, c' \rangle$, and $\langle \neg c, \neg c' \rangle$. It can be seen that the net effect of the cross-product is executing the programs Q_{N-1} and P_{N-1} one after the other, since they operate on a disjoint set of variables.

We run a data-flow analysis pass over the constructed cross-product to infer difference invariants at loop head, loop exit and at each branch condition. The only data-flow values of interest are differences between corresponding variables in Q_{N-1} and P_{N-1}. Indeed, since structure and variables of Q_{N-1} and P_{N-1} are similar, we can create the correspondence map between the variables. We start the difference invariant generation by considering relations between corresponding variables/array elements appearing in pre-conditions of the two programs. We apply static analysis that can track equality expressions (including disjunctions over equality expressions) over variables as we traverse the program. These equality expressions are our difference invariants.

We observed in our experiments the most of the inferred equality expressions are

simple expressions of N (at most quadratic in N). This not totally surprising and similar observations have been independently made in other contexts, for example, translation validation [GRB20], equivalence checking [CPSA19] and proving relational properties of programs such as non-interference, secure information flow and continuity [BCK11]. Note that the difference invariants may not always be equalities. We can easily extend our analysis to learn inequalities using interval domains in static analysis. We can also use a library of expressions to infer difference invariants using a guess-and-check framework. Moreover, guessing the predicates involved in the difference invariants can be easy, since in many cases, the expressions may be independent of the program constructs. For example, the equality expression $\mathtt{v}' = \mathtt{v}$, where $\mathtt{v}' \in \mathsf{Q}_{N-1}$ and $\mathtt{v} \in \mathsf{P}_{N-1}$, does not depend on any other variable from the two programs. Specifically, when the variable $v \in \mathsf{P}_N$ is not identified as affected (see Section 5.3.4) then the difference invariants are indeed equality expressions $\mathtt{v}' = \mathtt{v}$. It is worth nothing that though we restrict ourselves to difference invariants, other kinds of relational invariants that can aid in the computation of $\psi'(N-1)$ from $\psi(N-1)$ by relating the corresponding variables in Q_{N-1} and P_{N-1} are also permitted in our technique.

Example 7.1 Consider the programs Q_{N-1} and P_{N-1} from the third and the fourth column resp. of Fig. 7.3. The difference invariant at the head of the first loop of $\mathsf{Q}_{N-1} \times P_{N-1}$ is $D(V_\mathsf{Q}, V_\mathsf{P}, N-1) \equiv \big(\mathtt{x}' - \mathtt{x} = \mathtt{i} \times (2 \times \mathtt{N} - 1) \wedge \forall \mathtt{i} \in [0, \mathtt{N}-1), \mathtt{a}'[\mathtt{i}] - \mathtt{a}[\mathtt{i}] = 1\big)$, where $\mathtt{x}', \mathtt{a}' \in V_\mathsf{Q}$ and $\mathtt{x}, \mathtt{a} \in V_\mathsf{P}$. Given this difference invariant for the first loop, we easily get the difference invariant $\mathtt{x}' - \mathtt{x} = (\mathtt{N}-1) \times (2 \times \mathtt{N} - 1)$ at the exit point when the first loop terminates. For the second loop, we compute the difference invariant $D(V_\mathsf{Q}, V_\mathsf{P}, N - 1) \equiv \big(\forall \mathtt{j} \in [0, \mathtt{N}-1),\ \mathtt{b}'[\mathtt{j}] - \mathtt{b}[\mathtt{j}] = (\mathtt{x}' - \mathtt{x}) + \mathtt{N}^2 = (\mathtt{N}-1)\ \times\ (2 \times \mathtt{N} - 1) + \mathtt{N}^2\big)$, where $\mathtt{x}', \mathtt{b}' \in V_\mathsf{Q}$ and $\mathtt{x}, \mathtt{b} \in V_\mathsf{P}$. □

Note that difference invariants and its computation are agnostic of the given post-condition $\psi(N)$. Hence, our technique does not need to re-run this analysis for proving a different post-condition for the same program P_N. Let $\psi'(N-1)$ (resp. $\psi(N)$) be the post-condition and $\xi'(N-1)$ (resp. $\xi(N)$) be a formula that strengthens this post-condition upon executing the program Q_{N-1} (resp. P_N) starting from a state that satisfies the pre-condition $\varphi'(N-1)$ (resp. $\varphi(N)$). Then the following lemma relates the pre- and post-conditions of Q_{N-1} and P_N via the difference invariants $D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$.

Lemma 7.4 *If* $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N) \wedge \xi(N)\}$ *holds, then* $\{\varphi'(N-1)\}\ \mathsf{Q}_{N-1}\ \{\psi'(N-1) \wedge \xi'(N-1)\}$ *holds, where* $\psi'(N-1) \equiv \exists V_{\mathsf{P}}\big(\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)$ *and* $\xi'(N-1) \equiv \exists V_{\mathsf{P}}\big(\xi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)$

Proof. Follows from the construction of the difference invariants $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$. □

7.3.4 The Relational Full-Program Induction Algorithm

We present our method Diffy for verification of programs using relational full-program induction in Algorithm 17. It takes a Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ as input, where $\varphi(N)$ and $\psi(N)$ are pre- and post-condition formulas. We check the base in line 1 to verify the Hoare triple for $N = 1$. If this check fails, we report a counterexample. Subsequently, we compute Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$ as described in Section 7.3.1 using the function GenQandPeel from Algorithm 16. On line 5, we compute the formulas $\varphi'(N-1)$ and $\Delta\varphi'(N)$ as described in Section 7.3.2. For automation, we analyze the quantifiers appearing in $\varphi(N)$ and modify the quantifier ranges such that the conditions in Section 7.3.2 hold. We infer difference invariants $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ on line 6 using the method described in Section 7.3.3, wherein V_{Q} and V_{P} are sets of variables and arrays from Q_{N-1} and P_{N-1} respectively. On line 7, we compute $\psi'(N-1)$ by eliminating the variables and arrays of P_{N-1} in the set V_{P} from $\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$. On line 8, we check the inductive step of our analysis. If the inductive step succeeds, then we conclude that the assertion holds and return True on line 9. If that is not the case then, we try to iteratively strengthen both the pre- and post-condition of $\mathsf{Peel}(\mathsf{P}_N)$ simultaneously by invoking Strengthen (line 11).

The function Strengthen first initializes the formula $\chi(N)$ with $\psi(N)$ and the formulas $\xi(N)$ and $\xi'(N-1)$ to True. To strengthen the pre-condition of $\mathsf{Peel}(\mathsf{P}_N)$, we infer a formula $\chi'(N-1)$ using Dijkstra's weakest pre-condition computation of $\chi(N)$ over the $\mathsf{Peel}(\mathsf{P}_N)$ in line 18. It may happen that we are unable to infer such a formula. In such a case, if the program $\mathsf{Peel}(\mathsf{P}_N)$ has loops then we recursively invoke Diffy on line 21 to further simplify the program. Otherwise, we abandon the verification effort (line 23). We use quantifier elimination to infer $\chi(N-1)$ from $\chi'(N-1)$ and $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1))$ on line 24.

The inferred pre-conditions $\chi(N)$ and $\chi'(N-1)$ are accumulated in $\xi(N)$ and $\xi'(N-$

Algorithm 17 DIFFY($\{\varphi(N)\}$ P_N $\{\psi(N)\}$)

1: **if** $\{\varphi(1)\}$ P_1 $\{\psi(1)\}$ fails **then** ▷ Base case for N=1
2: **print** "Counterexample found!";
3: **return** False;
4: $\langle \mathsf{Q}_{N-1}, \mathsf{Peel}(\mathsf{P}_N)\rangle$:= GENQANDPEEL(P_N);
5: $\langle \varphi'(N-1), \Delta\varphi'(N)\rangle$:= FORMULADIFF($\varphi(N)$); ▷ $\varphi(N) \Rightarrow \varphi'(N-1) \wedge \Delta\varphi'(N)$
6: $D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$:= INFERDIFFINVS($\mathsf{Q}_{N-1}, \mathsf{P}_{N-1}, \varphi'(N-1), \varphi(N-1)$);
7: $\psi'(N-1)$:= QE($V_\mathsf{P}, \psi(N-1) \wedge D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$);
8: **if** $\{\psi'(N-1) \wedge \Delta\varphi'(N)\}$ $\mathsf{Peel}(\mathsf{P}_N)$ $\{\psi(N)\}$ **then**
9: **return** True; ▷ Verification Successful
10: **else**
11: b := STRENGTHEN($\mathsf{P}_N, \mathsf{Peel}(\mathsf{P}_N), \varphi(N), \psi(N), \psi'(N-1), \Delta\varphi'(N), D(V_\mathsf{Q}, V_\mathsf{P}, N)$);
12: **return** b;
13: **procedure** STRENGTHEN(P_N, $\mathsf{Peel}(\mathsf{P}_N)$, $\varphi(N)$, $\psi(N)$, $\psi'(N-1)$, $\Delta\varphi'(N)$, $D(V_\mathsf{Q}, V_\mathsf{P}, N)$)
14: $\chi(N) := \psi(N)$;
15: $\xi(N)$:= True;
16: $\xi'(N-1)$:= True;
17: **repeat**
18: $\chi'(N-1)$:= WP($\chi(N), \mathsf{Peel}(\mathsf{P}_N)$); ▷ Dijkstra's WP for loop free code
19: **if** $\chi'(N-1) = \emptyset$ **then**
20: **if** $\mathsf{Peel}(\mathsf{P}_N)$ has a loop **then**
21: **return** DIFFY($\{\xi'(N-1) \wedge \Delta\varphi'(N) \wedge \psi'(N-1)\}$ $\mathsf{Peel}(\mathsf{P}_N)$ $\{\xi(N) \wedge \psi(N)\}$);
22: **else**
23: **return** False; ▷ Unable to prove
24: $\chi(N)$:= QE($V_\mathsf{Q}, \chi'(N) \wedge D(V_\mathsf{Q}, V_\mathsf{P}, N)$);
25: $\xi(N) := \xi(N) \wedge \chi(N)$;
26: $\xi'(N-1) := \xi'(N-1) \wedge \chi'(N-1)$;

```
27:        if {φ(1)} P_1 {ξ(1)} fails then
28:            return False;                                         ▷ Unable to prove
29:        if {ξ'(N − 1) ∧ Δφ'(N) ∧ ψ'(N − 1)} Peel(P_N) {ξ(N) ∧ ψ(N)} holds then
30:            return True;                                  ▷ Verification Successful
31:    until timeout;
32:    return False;
```

1), which strengthen the post-conditions of P_N and Q_{N-1} respectively in lines 25 - 26. We again check the base case for the inferred formulas in $\xi(N)$ on line 27. If the check fails we report a failure to verify the post-condition on line 28. If the base case succeeds, we proceed to the inductive step (line 29). When the inductive step succeeds, we conclude that the assertion is verified and return True on line 30. Otherwise, the method tries to infer more pre-conditions while iterating in the loop. In doing so, if the algorithm exceeds the timeout value, it indicates an inconclusive result by returning False on line 32.

We now demonstrate the working of the algorithm on an example.

Example 7.2 Consider the example in Fig. 7.3. The given pre-condition is $\varphi(N) \equiv \mathsf{True}$ and the post-condition is $\psi(N) \equiv \forall \mathtt{j} \in [\mathtt{0}, \mathtt{N}),\ \mathtt{b}[\mathtt{j}] = \mathtt{j} + \mathtt{N}^3)$. On line 5, the algorithm computes $\varphi'(N-1)$ and $\Delta\varphi'(N-1)$ to be True. The difference invariant $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ is the formula computed in Example 7.1 from Section 7.3.3. Using the difference invariant, $\psi'(N-1) \equiv (\forall \mathtt{j} \in [\mathtt{0}, \mathtt{N}-\mathtt{1}),\ \mathtt{b}'[\mathtt{j}] = \mathtt{j} + (\mathtt{N}-\mathtt{1})^3 + (\mathtt{N}-\mathtt{1}) \times (\mathtt{2} \times \mathtt{N} - \mathtt{1}) + \mathtt{N}^2 = \mathtt{j} + \mathtt{N}^3)$ is computed on line 7. The algorithm then invokes function Strengthen on line 11 that infers the formulas $\chi'(N-1) \equiv (\mathtt{x}' = (\mathtt{N}-\mathtt{1})^3)$ on line 18 and $\chi(N) \equiv (\mathtt{x} = \mathtt{N}^3)$ on line 24. The formulas $\chi(N)$ and $\chi'(N-1)$ are accumulated in $\xi(N)$ and $\xi'(N-1)$ resp. on line 25 and 26. The formula $\xi'(N-1)$ strengthens the pre-condition of $\mathsf{Peel}(\mathsf{P}_N)$ and $\xi(N)$ strengthens its post-condition. Verification succeeds after this strengthening iteration. □

The following theorem guarantees the soundness of our technique.

Theorem 7.3 *Suppose there exist formulas $\xi'(N)$ and $\xi(N)$ and an integer $M > 0$ such that the following hold*

1. $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N) \wedge \xi(N)\}$ *holds for* $0 < N \leq M$, *for some* $M > 0$.

2. $\xi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1) \Rightarrow \xi'(N-1)$ *for all* $N > 1$.

3. $\{\xi'(N-1) \wedge \Delta\varphi'(N) \ \wedge \ \psi'(N-1)\}$ $\mathsf{Peel}(\mathsf{P}_N)$ $\{\xi(N) \wedge \psi(N)\}$ *holds for all* $N > M$, *where* $\psi'(N-1) \equiv \exists V_{\mathsf{P}}\big(\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)$.

Then, $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ *holds for all* $N > 0$.

Proof. Condition 1, the base case, ensures that $\{\varphi(N)\}$ P_N $\{\psi(N) \wedge \xi(N)\}$ holds, for $0 < N \leq M$ for some $M > 0$. Using Lemma 7.3, we decompose the pre-condition $\varphi(N)$. The resultant Hoare triple is $\{\varphi'(N-1) \wedge \Delta\varphi'(N)\}$ P_N $\{\psi(N) \wedge \xi(N)\}$. Using Lemma 7.2, we decompose the program P_N into Q_{N-1}; $\mathsf{Peel}(\mathsf{P}_N)$, resulting in the Hoare triple $\{\varphi'(N-1) \wedge \Delta\varphi'(N)\}$ Q_{N-1}; $\mathsf{Peel}(\mathsf{P}_N)$ $\{\psi(N) \wedge \xi(N)\}$. Using $\psi'(N-1) \wedge \xi'(N-1)$ as the mid-condition by virtue of condition 2, we generate the Hoare triples $\{\varphi'(N-1)\}$ Q_{N-1} $\{\psi'(N-1) \wedge \xi'(N-1)\}$ and $\{\psi'(N-1) \wedge \xi'(N-1) \wedge \Delta\varphi'(N)\}$ $\mathsf{Peel}(\mathsf{P}_N)$ $\{\psi(N) \wedge \xi(N)\}$ for all $N > M$. The first Hoare triple follows from the induction hypothesis and condition 2. Condition 3 guarantees that the second Hoare triple holds. This concludes the proof. □

Lemma 7.5 *If the function* DIFFY *returns* False, *then we have found a valid counterexample. When it returns* True, $\{\varphi_N\}$ P_N $\{\psi_N\}$ *holds for all* $N \geq 1$.

Proof. Verifying the given Hoare triple $\{\varphi_N\}$ P_N $\{\psi_N\}$ for all $N \geq 1$ requires establishing the conditions 1, 2 and 3 mentioned in Theorem 7.3. The function DIFFY ensures condition 1 through the check on line 1. If this check fails, DIFFY returns False along with a valid counterexample that violates the base-case. The call to the function FORMULADIFF in line 5 decomposes the pre-condition to compute a difference pre-condition that satisfies the conditions mentioned in Lemma 7.3. The computation of difference invariants $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N)$ via the call to function INFERDIFFINVS in line 6, the computation of $\psi'(N-1)$ in line 7 and $\xi'(N-1)$ in lines 24 – 26 ensure that condition 2 holds. The checks in lines 8 and 29 ensure condition 3 Theorem 7.3. Hence, $\{\varphi_N\}$ P_N $\{\psi_N\}$ holds for all $N \geq 1$ when DIFFY returns True. □

Relative Completeness of Relational Full-Program Induction in Special Cases

Suppose that the program P_N has only non-nested loops then the difference program $\mathsf{Peel}(\mathsf{P}_N)$ is loop-free. Further, suppose that the variables/arrays in $\varphi(N)$ and those computed in P_N are not identified as affected (refer Section 5.3.4). In such cases, the programs Q_{N-1} and P_{N-1} are equal. The post-conditions of Q_{N-1} and P_{N-1} are also the same i.e.

$\psi'(N-1) \equiv \psi(N-1)$. Further, the difference invariants $D(V_{\mathsf{Q}}, V_{\mathsf{P}})$ are restricted to the template $\mathtt{vA}' = \mathtt{vA}$, where $\mathtt{vA}' \in V_{\mathsf{Q}}$ and $\mathtt{vA} \in V_{\mathsf{P}}$ denote a variable/array element. The strengthening step also gets simplified since $\xi'(N-1) \equiv \xi(N-1)$. And generating these strengthening predicates using Dijkstra's weakest pre-condition computation is always possible for loop-free programs $\mathsf{Peel}(\mathsf{P}_N)$. As a result, even when the given pre- and post-conditions have quantifiers with arbitrary nesting depth, the base-case and the inductive step can be proved (by the underlying SMT solver). The following theorem states this relative completeness result.

Theorem 7.4 *Relational Full-Program Induction is sound and relatively complete for the class of Hoare triples* $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ *satisfying the following conditions.*

- *program* P_N *has only non-nested loops*
- *variables and arrays in* $\varphi(N)$ *and those computed in* P_N *are not identified as affected*

Relative completeness is with respect to a proof system for the underlying logic in which $\varphi(N)$, $\psi(N)$ *and the semantics of the statements in the program* P_N *are expressed.*

Proof. Follows from the fact that the base-case and the inductive step need to be proved by the underlying solver on a loop-free difference program $\mathsf{Peel}(\mathsf{P}_N)$ and that each strengthening iteration involves computing Dijkstra's weakest pre-condition on loop-free programs. □

7.4 Experimental Evaluation

In this section, we experimentally evaluate the effectiveness and efficacy of the relational full-program induction technique described previously on a large set of array-manipulating benchmarks.

7.4.1 Implementation

We have instantiated the relational full-program induction technique in a prototype tool called DIFFY. It is written in `C++` and is built using the LLVM(v6.0.0) [LA04] compiler. We use the SMT solver Z3(v4.8.7) [MB08] for proving Hoare triples of loop-free programs. DIFFY and the supporting data to replicate the experiments are openly available at [CGU21a].

Program Category		Diffy			Vajra		VeriAbs		VIAP		
		S	U	TO	S	U	S	TO	S	U	TO
Safe C1	110	110	0	0	110	0	96	14	16	1	93
Safe C2	24	21	0	3	0	24	5	19	4	0	20
Safe C3	23	20	3	0	0	23	9	14	0	23	0
Total	157	151	3	3	110	47	110	47	20	24	113
Unsafe C1	99	98	1	0	98	1	84	15	98	0	1
Unsafe C2	24	24	0	0	17	7	19	5	22	0	2
Unsafe C3	23	20	3	0	0	23	22	1	0	23	0
Total	146	142	4	0	115	31	125	21	120	23	3

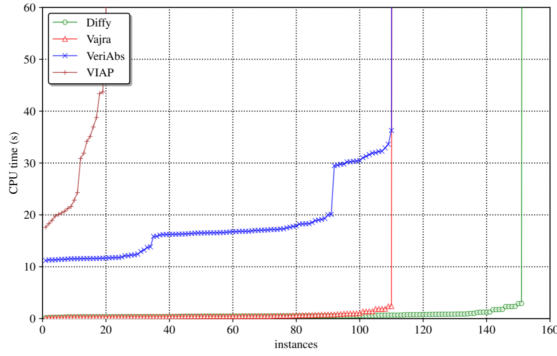
Table 7.1: Summary of the Experimental Results. S is Successful Result. U is Inconclusive Result. TO is Timeout.

7.4.2 Experimental Setup

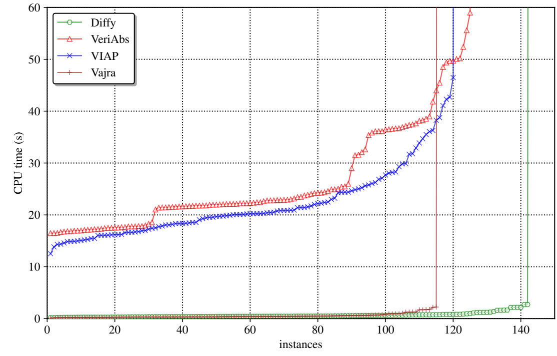
All experiments were performed on a machine with Intel i7-6500U CPU, 16GB RAM, running at 2.5 GHz, and Ubuntu 18.04.5 LTS operating system. We have compared the results obtained from Diffy with Vajra(v1.0) [CGU20b, CGU20a, CGU22], VIAP(v1.1) [RL18] and VeriAbs(v1.4.1-12) [ACC+20]. We choose Vajra which also employs inductive reasoning for proving array programs and verify the benchmarks in its test-suite. We compared with VeriAbs as it is the winner of the arrays sub-category in SV-COMP 2020 [Bey20] and 2021 [Bey21]. VeriAbs applies a sequence of techniques from its portfolio to verify array programs. We compared with VIAP which was the winner in arrays sub-category in SV-COMP 2019 [Bey19]. VIAP also employs a sequence of tactics, implemented for proving a variety of array programs. Diffy does not use multiple techniques, however we choose to compare it with these portfolio verifiers to show that it performs well on a class of programs and can be a part of their portfolio. All tools take C programs in the SV-COMP format as input. Timeout of 60 seconds was set for each tool. A summary of the results is presented in Table 7.1.

7.4.3 Benchmarks

We have evaluated Diffy on a set of 303 array benchmarks, comprising of the entire test-suite of [CGU20a, CGU22], enhanced with challenging benchmarks to test the efficacy of our approach. These benchmarks take a symbolic parameter N which specifies the

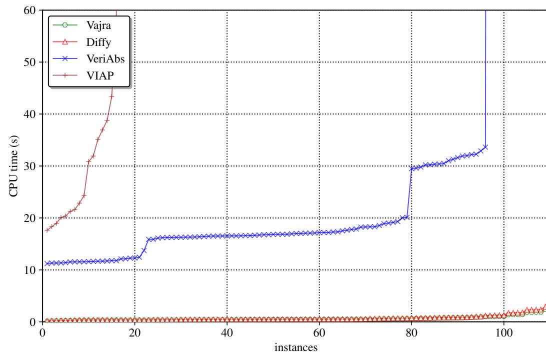


(a)

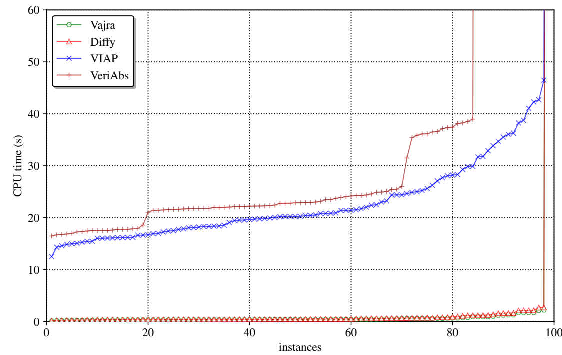


(b)

Figure 7.13: Cactus Plots (a) All Safe Benchmarks (b) All Unsafe Benchmarks



(a)



(b)

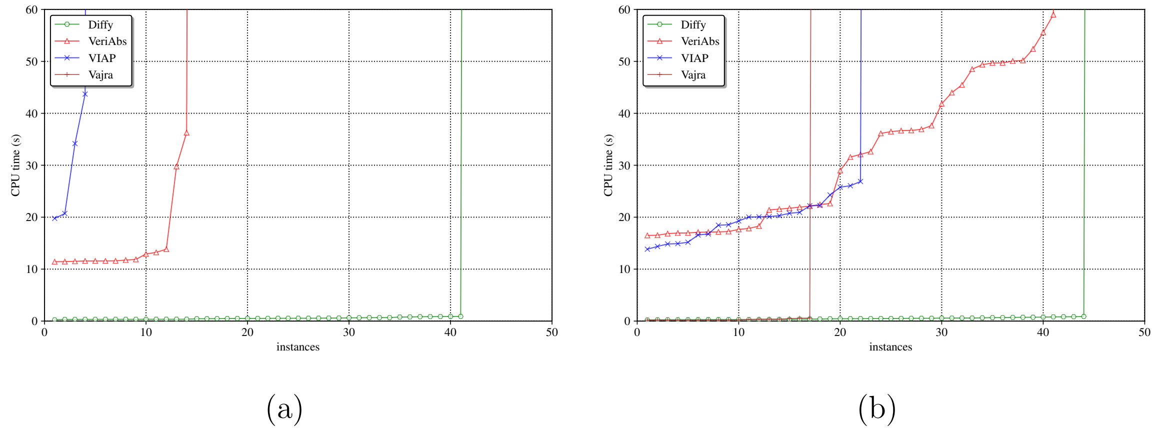
Figure 7.14: Cactus Plots (a) Safe C1 Benchmarks (b) Unsafe C1 Benchmarks

size of each array. Assertions are (in-)equalities over array elements, scalars and (non-)linear polynomial terms over N. We have divided both the safe and unsafe benchmarks in three categories. Benchmarks in C1 category have standard array operations such as min, max, init, copy, compare as well as benchmarks that compute polynomials. In these benchmarks, branch conditions are not affected by the value of N, operations such as modulo and nested loops are not present. There are 110 safe and 99 unsafe programs in the C1 category in our test-suite. In C2 category, the branch conditions are affected by change in the program parameter N and operations such as modulo are used in these benchmarks. These benchmarks do not have nested loops in them. There are 24 safe and unsafe benchmarks in the C2 category. Benchmarks in category C3 are programs with atleast one nested loop in them. There are 23 safe and unsafe programs in category C3 in our test-suite. The test-suite has a total of 157 safe and 146 unsafe programs.

7.4.4 Results and Analysis

Of the 157 safe benchmarks, Diffy verified 151 safe benchmarks, compared to 110 verified by Vajra as well as VeriAbs and 20 verified by VIAP. Diffy was unable to verify 6 safe benchmarks. In 3 of these 6 benchmarks, the SMT solver timed out while trying to prove the induction step since the formulated query had a modulus operation and in the other 3 benchmarks it was unable to compute the predicates needed to prove the assertions. In 6 benchmarks difference invariants with quadratic terms were needed, and the rest required difference invariants with only linear terms. Vajra was unable to verify 47 programs from categories C2 and C3. These are programs with nested loops, branch conditions affected by N, and benchmarks where it could not compute the difference program. The sequence of techniques employed by VeriAbs, ran out of time on 47 programs while trying to prove the given assertion. VeriAbs proved 2 benchmarks in category C2 and 3 benchmarks in category C3 where Diffy was inconclusive or timed out. VeriAbs spends considerable amount of time on different techniques in its portfolio before it resorts to Vajra, and hence, it could not verify 14 programs that Vajra was able to prove efficiently. VIAP was inconclusive on 24 programs which had nested loops or constructs that could not be handled by the tool. It ran out of time on 113 benchmarks as the initial tactics in its sequence took up the allotted time but could not verify the benchmarks. Diffy was able to verify all programs that VIAP and Vajra were able to verify within the specified time limit.

The cactus plot in Fig. 7.13(a) shows the performance of each tool on all safe benchmarks. Diffy was able to prove most of the programs within 3 seconds. The cactus plot in Fig. 7.14(a) shows the performance of each tool on safe benchmarks in C1 category. Vajra and Diffy perform equally well in the C1 category. This is due to the fact that both tools perform efficient inductive reasoning. Diffy outperforms VeriAbs and VIAP in this category. The cactus plot in Fig. 7.15(a) shows the performance of each tool on safe benchmarks in the combined categories C2 and C3, that are difficult for Vajra as most of these programs are not within its scope. Diffy outperforms all other tools in categories C2 and C3. VeriAbs was an order of magnitude slower on programs it was able to verify, as compared to Diffy. VeriAbs spends significant amount of time in trying techniques from its portfolio, including Vajra, before one of them succeeds in verifying the assertion or takes up the entire time allotted to it. VIAP took 70 seconds



(a) (b)

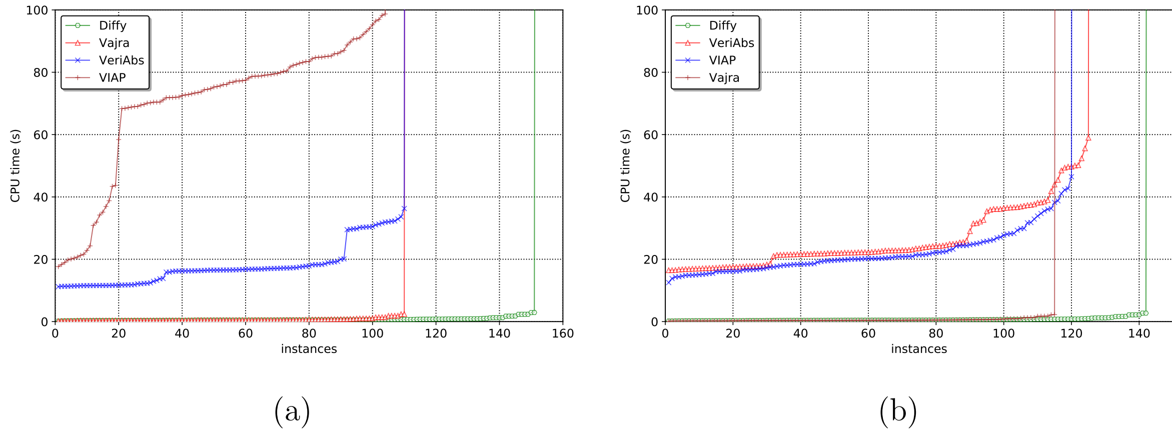
Figure 7.15: Cactus Plots (a) Safe C2 & C3 Benchmarks (b) Unsafe C2 & C3 Benchmarks

more on an average as compared to DIFFY to verify the given benchmark. VIAP also spends a large portion of time in trying different tactics implemented in the tool and solving the recurrence relations in programs.

The relational full-program induction technique reports property violations when the base case of the analysis fails for small fixed values of N. While the focus of our work is on proving assertions, we report results on unsafe versions of the safe benchmarks from our test-suite. DIFFY was able to detect a property violation in 142 unsafe programs and was inconclusive on 4 benchmarks. For unsafe benchmarks, DIFFY does not require computation of difference invariants since the analysis concludes after the base-case gets violated. VAJRA detected violations in 115 programs and was inconclusive on 31 programs. VERIABS reported 125 programs as unsafe and ran out of time on 21 programs. VIAP reported property violation in 120 programs, was inconclusive on 23 programs and timed out on 3 programs.

The cactus plot in Fig. 7.13(b) shows the performance of each tool on all unsafe benchmarks. DIFFY was able to detect a violation faster than all other tools and on a larger number of benchmarks from the test-suite. Fig. 7.14(b) and Fig. 7.15(b) give a finer glimpse of the performance of these tools on the categories that we have defined. In the C1 category, DIFFY and VAJRA have comparable performance and DIFFY disproves the same number of benchmarks as VAJRA and VIAP. In C2 and C3 categories, DIFFY is able to detect property violations in a larger number of benchmarks as compared to other tools and took relatively less time.

To observe any changes in the performance of these, we also ran all the benchmarks with an increased time out of 100 seconds. The plots for this experiment on safe and



(a) (b)

Figure 7.16: Cactus Plots. TO=100s. (a) Safe Benchmarks (b) Unsafe Benchmarks

unsafe benchmarks are shown in Fig. 7.16. Performance remains unchanged for DIFFY, VAJRA and VERIABS on both safe and unsafe benchmarks, and of VIAP on unsafe benchmarks. VIAP was able to additionally verify 89 safe programs in categories C1 and C2 with the increased time limit.

7.4.5 Limitations

There are several scenarios under which the technique described in this chapter may remain inconclusive. Currently, relational full-program induction can only verify nested loops of that satisfy specific constraints mentioned in Section 7.3.1. We are unable to verify programs that do not satisfy these conditions. Fig. 6.12 shows a concrete example that violates the necessary conditions for computing programs Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$.

The inductive reasoning may remain inconclusive when we are unable to compute difference invariants for a program. Programs with deeply nested loops may require difference invariants with high-degree polynomial terms. The program that have complicated conditional statements may require difference invariants with multiple disjunctive clauses. Computing such difference invariants may be at times quite challenging.

The difference program $\mathsf{Peel}(\mathsf{P}_N)$ consists of the peeled iterations of all the loops in P_N (that are missed in P_{N-1}). Hence, relational full-program needs to know the symbolic upper bound on the value of the loop counter to be able to compute the number of iterations to be peeled from the program. The heuristics used for weakest pre-condition computation may either fail or return a pre-condition that causes violation of the base-case. The solver may be unable to prove the verification conditions within the stipulated

time frame.

Our prototype implementation also has a few limitations. We currently support assignment statements with expressions containing $\{+, -, \times, \div\}$ operators. We support a single program parameter and peel only the last iterations of loops. Despite these limitations, the experiments show that relational full-program induction performs remarkably well on a large suite of benchmarks.

7.5 Verifying Existentially Quantified Properties

In this section, we revisit our entire verification approach while eliminating the restriction that the pre- and post-conditions must be universally quantified formulas or quantifier-free formulas of a specific form. We show that the relational full-program induction technique extends naturally to verifying programs with existentially quantified pre- and post-conditions. We now look at how the existential quantified pre- and post-conditions are handled in each component of our technique. An existentially quantified pre-condition $\varphi(N)$ has the form $\bigvee_{i=0}^{N-1} \rho_i$, where ρ_i is a formula on the ith elements of one or more arrays, and scalars used in P_N. As previously stated, we infer $\varphi'(N-1)$ to be $\bigvee_{i=0}^{N-2} \rho_i$ and $\Delta\varphi'(N)$ to be ρ_{N-1} while ensuring that variables/array elements in ρ_{N-1} are not modified by Q_{N-1}. Further, the computation of difference invariants now considers the relations between corresponding variables/array elements appearing in the existentially quantified pre-conditions of programs Q_{N-1} and P_{N-1} to begin with. At last, the generation of Q_{N-1} and $\mathsf{Peel}(\mathsf{P}_N)$ from a given program P_N is independent of the given pre- and post-condition formulas, and hence, remains unaffected. We demonstrate the relational full-program induction technique in detail on programs that have existentially quantified pre- and/or post-conditions with a couple of examples illustrated in Figs. 7.17 and 7.18.

Consider the Hoare triple shown in Fig. 7.17(a). The pre-condition in the Hoare triple states that there exists an index `i` in array `A` where the value of the array element is greater than twice the value of `N`. The program has a couple of sequentially composed loops that update arrays `B` and `C`. The first loop assigns to `B[j]` an expression that subtracts `N` from `A[j]`. Subsequently, the second loop assigns to `C[k]` an expression that subtracts `N` from `B[k]`. The post-condition asserts that there exists an element in array `C` that has a non-negative value greater than 0.

```
// assume(∃i∈[0,N) A[i] > 2×N)
1. for (int j=0; j<N; j=j+1)
2.   B[j] = A[j] - N;
3. for (int k=0; k<N; k=k+1)
4.   C[k] = B[k] - N;
// assert(∃i∈[0,N) C[i] > 0)
```

(a)

```
// assume(∃i∈[0,1) A[i] > 2×1)
1. for (int j=0; j<1; j=j+1)
2.   B[j] = A[j] - 1;
3. for (int k=0; k<1; k=k+1)
4.   C[k] = B[k] - 1;
// assert(∃i∈[0,1) C[i] > 0)
```

(b)

```
// assume(∃i∈[0,N-1) A[i] > 2×N)

1. for (int j=0; j<N-1; j=j+1)
2.   B[j] = A[j] - N;
3. for (int k=0; k<N-1; k=k+1)
4.   C[k] = B[k] - N;

// assert(∃i∈[0,N-1) C[i] > 0)
```

(c)

```
// assume(A[N-1] > 2×N)    // Δφ'(N)
// assert(∃i∈[0,N-1) C[i] > 0) // ψ'(N-1)

1. B[N-1] = A[N-1] - N;

2. C[N-1] = B[N-1] - N;

// assert(∃i∈[0,N) C[i] > 0) // ψ(N)
```

(d)

Figure 7.17: (a) Hoare Triple with Existentially Quantified Pre- and Post-conditions, (b) Base-case, (c) Hoare Triple on Q_{N-1} and (d) Inductive Step

The Hoare triple shown in Fig. 7.17(b) checks the base-case of our technique where `N` is substituted with the constant value 1. This check is easily proved by the back-end SMT solver Z3. Next, we peel the loops in the program and propagate them across subsequent loops. Observe that the values of array elements computed by the statements within loops do not have any read-after-write dependence on the peeled iterations of prior loops. Hence, no substitutions are needed while computing Q_{N-1}. However, note that due to the use of the program parameter `N` in the loop body, the arrays in the program are indeed identified as affected. Hence, Q_{N-1} and P_{N-1} are not semantically equivalent programs.

The pre-condition of Q_{N-1} computed by us is $\varphi'(N-1) := \exists i \in [0, N-1)\ \big(A'[i] > 2 \times N\big)$, whereas the pre-condition of P_{N-1} is $\varphi(N-1) := \exists i \in [0, N-1)\ \big(A[i] > 2 \times (N-1)\big)$. Notice that $\varphi'(N-1)$ and $\varphi(N-1)$ are not equivalent. We now relate the corresponding

variables in Q_{N-1} and P_{N-1}. The difference invariant computed by our technique is $D(V_\mathsf{Q}, V_\mathsf{P}, N-1) := \forall i \in [0, N-1)\ \big(A'[i] - A[i] = 2 \wedge B'[i] - B[i] = 1 \wedge C'[i] - C[i] = 0\big)$ where $\mathtt{A}', \mathtt{B}', \mathtt{C}' \in V_\mathsf{Q}$ and $\mathtt{A}, \mathtt{B}, \mathtt{C} \in V_\mathsf{P}$. Using this difference invariant, we compute the post-condition of Q_{N-1} as $\psi'(N) := \exists i \in [0, N-1)\ \big(C'[i] > 0\big)$. The resultant Hoare triple $\{\varphi'(N-1)\}\ \mathsf{Q}_{N-1}\ \{\psi'(N-1)\}$ computed by our technique is shown in Fig. 7.17(c).

Next, we compute the difference pre-condition as $\Delta\varphi'(N) := \big(A'[N-1] > 2 \times N\big)$. The difference program $\mathsf{Peel}(\mathsf{P}_N)$ is just the peeled iterations of both the loops. These quantities are computed for the inductive step shown in Fig. 7.17(d). The back-end SMT solver now proves the inductive step, and we conclude that the given post-condition holds for all values of $N > 0$.

Now we consider the Hoare triple shown in Fig. 7.18(a) where the pre-condition is universally quantified and the post-condition is existentially quantified. The pre-condition states that the value of each element in array `A` is equal to `N`. The program has a couple of sequentially composed loops that update the scalar `sum` and array `B`. Before the first loop starts, the scalar variable `S` is initialized to 0. The first loop in the program computes a recurrence in variable `sum`, accumulating the content of array `A`. Subsequently, in each iteration of the second loop, `B[k]` is assigned the expression that adds the value of the loop counter `k` to `sum`. The post-condition asserts that there exists an element in array `B` that has the value given by a non-linear expression in `N` and the quantified variable.

The Hoare triple shown in Fig. 7.18(b) checks the base-case of the technique where `N` is substituted with the constant value 1. It is proved by the back-end SMT solver Z3. Notice that the value of array `B` computed by the statement within the second loop has a read-after-write dependence on the peeled iteration of the first loop. Further, due to the dependence, array `B` is identified as affected. Due to the use of the program parameter `N` in the predicate on `A` in the pre-condition, array `A` is also identified as affected. Naturally, Q_{N-1} and P_{N-1} are not semantically equivalent programs in this case due to the above mentioned dependence. The program Q_{N-1} computed after peeling each loop and propagating the peels across subsequent loops, by substituting the right hand side expression from the peel of the first loop in the statement in the second loop, is shown in Fig. 7.18(c).

The pre-condition of Q_{N-1} is $\varphi'(N-1) := \forall i \in [0, N-1)\ \big(A'[i] = N\big)$, whereas the pre-condition of P_{N-1} is $\varphi(N-1) := \forall i \in [0, N-1)\ \big(A[i] = N-1\big)$. The difference

```
// assume(∀i∈[0,N) A[i] = N)
1. sum = 0;
2. for (int j=0; j<N; j=j+1)
3.  sum = sum + A[j];
4. for(int k=0; k<N; k++)
5.  B[k] = sum + k;
// assert(∃i∈[0,N) B[i]=i+N×N)
```

(a)

```
// assume(∀i∈[0,N) A[i] = 1)
1. sum = 0;
2. for (int j=0; j<1; j=j+1)
3.  sum = sum + A[j];
4. for(int k=0; k<1; k++)
5.  B[k] = sum + k;
// assert(∃i∈[0,N) B[i]=i+1×1)
```

(b)

```
// assume(∀i∈[0,N-1) A[i] = N)

1. sum = 0;
2. for (int j=0; j<N-1; j=j+1)
3.  sum = sum + A[j];
4. for(int k=0; k<N-1; k++)
5.  B[k] = sum + A[N-1] + k;

// assert(∃i∈[0,N-1) B[i]=i+N×N)
```

(c)

```
// assume(A[N-1] = N)
// assert(∃i∈[0,N-1) B[i]=i+N×N)
// assume(sum = N×(N-1))

1. sum = sum + A[N-1];

2. B[N-1] = sum + N-1;

// assert(∃i∈[0,N) B[i]=i+N×N)
// assert(sum = N×N)
```

(d)

Figure 7.18: (a) Hoare Triple with Universally Quantified $\varphi(N)$ and Existentially Quantified $\psi(N)$, (b) Base-case, (c) Hoare Triple on Q_{N-1} and (d) Inductive Step after Strengthening

invariant computed by our technique is $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1) := \forall i \in [0, N-1) \big(A'[i] - A[i] = 1 \wedge sum' = sum \wedge B'[i] - B[i] = 2 \times N - 1\big)$ where $\mathtt{sum}', \mathtt{A}', \mathtt{B}' \in V_{\mathsf{Q}}$ and $\mathtt{sum}, \mathtt{A}, \mathtt{B} \in V_{\mathsf{P}}$. Using this difference invariant, we compute the post-condition of Q_{N-1} as $\psi'(N-1) := \exists i \in [0, N-1) \big(B'[i] = i + N \times N\big)$ starting from the pre-condition $\varphi'(N-1)$. Note that the post-condition $\psi'(N-1)$ is not the same as the post-condition $\psi(N-1)$ of P_{N-1}. The difference pre-condition computed by the technique is $\Delta\varphi'(N) := \big(A'[N-1] = N\big)$ and the difference program $\mathsf{Peel}(\mathsf{P}_N)$ is just the peeled iterations of both the loops. The inductive step that uses these quantities is shown in Fig. 7.17(d). The formula $sum = N \times (N-1)$ (denoted $\xi'(N-1)$ in our description) strengthens the pre-condition and $sum = N \times N$

(denoted $\xi(N)$ in our description) strengthens the post-condition. The formulas on the variable `sum` are computed during the iterative strengthening of pre- and post-condition by using Dijkstra's weakest pre-condition computation. The back-end SMT solver now successfully proves the inductive step. Hence, we conclude that the given post-condition holds for all values of $N > 0$.

7.6 Comparison with Related Techniques

The relational full-program induction technique described in this chapter is closely related to several efforts that apply inductive reasoning to verify properties of array programs. This work subsumes the full-program induction technique in [CGU20a, CGU22] that works by inducting on the entire program via a program parameter N. We propose a principled method for computation and use of difference invariants, instead of computing difference programs which is more challenging. An approach to construct safety proofs by automatically synthesizing squeezing functions that shrink program traces is proposed in [ISIRS20]. Such functions are not easy to synthesize, whereas difference invariants are relatively easy to infer. The technique in [KN22] attempts to deductively verify loops and localize errors without using inductive invariants for a very restricted class of programs. Detailed experimental evaluation is not reported and their system is not available for experimentation. The method may be aimed specifically at verifying selected algorithms. In [CGU17], the post-condition is inductively established by identifying a tiling relation between the loop counter and array indices used in the program. Relational full-program induction can verify programs from [CGU17], when supplied with the *tiling* relation. The technique in [SB12] identifies recurrent program fragments for induction using the loop counter. They require restrictive data dependencies, called *commutativity of statements*, to move peeled iterations across subsequent loops. Unfortunately, these restrictions are not satisfied by a large class of programs in practice, where relational full-program induction succeeds.

Computing differences of program expressions has been studied for incremental computation of expensive expressions [PK82, LST98], optimizing programs with arrays [LSLR05], and checking data-structure invariants [SB07]. These differences are not always well suited for verifying properties. In contrast, difference invariants enable the inductive step of the

reasoning in relational full-program induction. Further, computation of the difference programs is challenging in many cases for the full-program induction technique, however, difference invariants succeed in enabling the verification of such programs [CGU21b].

In [GGK20], trace logic that implicitly captures inductive loop invariants is described. They use theorem provers to introduce and prove lemmas at arbitrary time points in the program, whereas we infer and prove lemmas at key control points during the inductive step using SMT solvers. VIAP [RL18] translates the program to an quantified first-order logic formula using the scheme proposed in [Lin16]. The tool uses a portfolio of tactics to simplify and prove the generated formulas. Dedicated solvers for recurrences are used in these methods whereas the relational full-program induction technique adapts handles even recurrences via inductive reasoning.

Several techniques generate invariants for array programs. QUIC3 [GSV18], FreqHorn [FPMG19] and [BMR13] infer universally quantified invariants over arrays for Constrained Horn Clauses (CHCs) among other tools. Template-based techniques [GMT08, SG09, BHMR07] search for inductive quantified invariants by instantiating parameters of a fixed set of templates. We generate relational invariants, which are often easier to infer compared to inductive quantified invariants for each loop. For many benchmarks, the relational invariants of a highly restricted form suffice, for example, when a program does not refer to any affected variables/arrays.

Counterexample-guided abstraction refinement using prophecy variables for programs with arrays is proposed in [MIG+21]. VeriAbs [ACC+20] uses a portfolio of techniques, specifically to identify loops that can be soundly abstracted by a bounded number of iterations. Vaphor [MG16] transforms array programs to array-free Horn formulas to track bounded number of array cells. Booster [AGS14] combines lazy abstraction based interpolation [ABG+12a] and acceleration [BIK10, JSS14] for array programs. Abstractions in [CCL11, DDA10, GRS05, HP08, JM07, LR15, MA15] implicitly or explicitly partition the range array indices to infer and prove facts on array segments. In contrast, the relational full-program induction technique does not rely on abstractions.

7.7 Conclusion

We presented a novel generalization of the full-program induction technique that combines generating relations between two version of a program and inductive reasoning. These invariants relate corresponding variables and arrays from two versions of a program. These relations facilitate inductive reasoning by assisting in the inductive step. Invariants that refer to only the differences between the values of variables and arrays from the two versions are easy to infer and prove. We presented an instantiation of the technique in a prototype tool DIFFY. We experimentally showed that DIFFY out-performs the tools that have won the Arrays sub-category in SV-COMP 2019, 2020 and 2021. Since SV-COMP 2022, the verification tool DIFFY is a part of the portfolio of reasoning techniques in VERIABS and assists in proving the challenging examples from the ReachSafety-Arrays sub-category.

Chapter 8

Conclusions & Prospects

In this thesis, we presented three novel inductive verification techniques that adapt induction in different ways to prove a sub-class of quantified and quantifier-free properties of programs that manipulate arrays of parametric sizes. We described the instantiation of these techniques in prototype tools and demonstrated their effectiveness against related state-of-the-art verification techniques.

We first presented a *theory of tiling* that decomposes reasoning about an array into reasoning about automatically identified tiles in the array in programs that use complex index expressions to access arrays. While the generation of tiles is difficult in general, we showed that simple heuristics are often quite effective in automatically generating tiles that work well on programs seen in practice. We described an implementation of the technique in the tool Tiler. We presented an experimental evaluation of Tiler on a large suite of benchmarks that manipulate arrays. Tiler outperforms other verification tools that can prove universally quantified properties of array programs on a suite of benchmarks.

Next, we presented the *full-program induction* technique that obviates the need for loop-specific invariants during verification and is orthogonal to tile-wise reasoning. The technique performs induction over the entire program via parameter N by automatically computing the difference programs and difference pre-conditions. We presented different techniques for computing difference programs. Full-program induction can be recursively applied to simplify verification until the difference program is loop-free. We also presented techniques for computing the difference pre-conditions as well as for simultaneously strengthening the pre- and post-conditions. We showed that the generalizations of

the full-program induction and its sub-components expand the scope of the full-program induction technique to a larger class of programs. We proved the correctness of the full-program induction technique and presented the progress guarantees. We also presented various problem settings in which full-program induction can be applied. We described a prototype implementation of full-program induction in the tool VAJRA. We showed via experiments that VAJRA performs remarkably well vis-a-vis state-of-the-art tools for verifying array programs. The verification tool VERIABS from an industry research group that participates regularly in SV-COMP has integrated VAJRA into its verification strategy since 2020 to boost its capabilities in proving programs from the ReachSafety-Arrays sub-category.

Finally, building on the basic principle of full-program induction, we presented another technique called *relational full-program induction*. Significantly, it applies to programs with nested loops that have hitherto been beyond the reach of most automated verification techniques that analyze array programs. Relational full-program induction successfully overcomes the challenges associated with the computation of difference programs. It combines inductive reasoning with reasoning based on relational invariants on corresponding variables and arrays in two slightly different versions of a program. These relations facilitate inductive reasoning by assisting the formulation of the inductive step. We studied the restriction of such relations to differences between the values of variables/arrays in the two versions of a program and showed that it suffices for a large class of verification problems. Such relations, termed *difference invariants*, are comparatively easy to infer and use. We presented an instantiation of the technique in a prototype tool called DIFFY. We experimentally showed that DIFFY out-performs the tools that have won the ReachSafety-Arrays sub-category in SV-COMP 2019, 2020 and 2021. Since 2022 DIFFY is an integral part of the portfolio of verification strategies in VERIABS, an verification tool from industry that regularly participates in SV-COMP, to prove programs from the ReachSafety-Arrays sub-category.

The inductive reasoning techniques presented in this thesis offer several advantages over techniques based on loop-specific invariant generation. All the three techniques are compositional and property-driven. The use of induction makes them scalable, precise and efficient in practice. These techniques are well-suited to be part of a portfolio of verification techniques and indeed our techniques are a useful part of industrial verification tools.

8.1 Future Prospects

The techniques presented in this thesis have laid the foundation for exploration of several interesting lines of work. In this section, we discuss some of these future possibilities. We divide these into two categories: (i) ideas to extend the techniques described in this thesis for program verification and (ii) useful adaptations of our techniques in domains beyond program verification.

Enhancements to Our Techniques

Full-program induction verifies a given program P_N by recursively computing the difference program $\partial\mathsf{P}_N$ until the base case and the inductive step are proved. We have previously demonstrated that the differencing operation ∂ may not always generate a difference program $\partial\mathsf{P}_N$ that is "simpler" to verify than the given program P_N. For instance, even though rare, it may happen that even after k recursive attempts the kth difference program $\partial^k\mathsf{P}_N$ computed by full-program induction is not simpler to verify than P_N. Due to this recursive nature of full-program induction, we proposed a metric, based on the syntactic changes in the difference program, to check progress after each recursive application of full-program induction. An alternative method, that comes with guarantee of reduction in verification complexity, can be explored for computing difference programs. For example, techniques in automatically integrating expressions and program fragments [HPR89] can be explored to compute a database of difference programs. Such a database would consists of two categories of programs. The base category, consisting of programs that can be verified without further application of inductive reasoning. This includes programs that can be proved using an SMT solver after compilation into an SMT-LIB formula, such as, loop-free programs, loopy programs with finite loop bounds and programs whose loops can be summarized/accelerated using known techniques. The inductive category, consisting of programs that can be verified by a fixed number (say k) of applications of full-program induction. When N is 1, a program P_1 can be viewed as the sequential composition of an initial program fragment P_0 and the difference program $\partial\mathsf{P}(1)$. When N is 2, P_2 is $\mathsf{P}(0)$; $\partial\mathsf{P}(1)$; $\partial\mathsf{P}(2)$. The program P_N can be represented as $\mathsf{P}(0)$; `for(i=0; i<`N`; i++)` $\partial\mathsf{P}(\mathtt{i})$. The computation in P_N can thus be viewed as an integral of the difference programs $\partial\mathsf{P}(i)$. This allows us to build a calculus for rea-

soning with programs when we know their differences from the database. The guarantee here is that if a difference program is found in such a database, we are guaranteed that, within a fixed number of recursive invocations, full-program induction would be able to successfully decide if the given post-condition holds or not. This would help in simplifying the computation of difference programs required for proving properties of programs using full-program induction.

Another interesting line of work that is worth investigating is the use of synthesis-based techniques for automatically computing difference programs and difference pre-conditions. These can also be specialized for programs from interesting domains. At the same time, investigations in using synthesis techniques for automatic generation of relational invariants is also equally appealing and can be considered in the same breath. While there is a lot of literature on synthesis techniques that can automatically generate invariants for a given program, very few techniques have attempted generation of relational invariants between different programs/versions. Specifically, the computation of relational invariants using techniques inspired from translation validation literature, syntax-guided synthesis techniques and invariant generation methodologies can be explored. It would also be interesting to search for other novel ways of adapting the relational invariants in an inductive framework for the purposes of proving properties of programs.

Explorations in techniques that aid the transformations of predicates on the variables/arrays of a program (say Q_{N-1}) to the variables/arrays of another (say P_{N-1}) and vice-versa may be beneficial. In the relational full-program induction technique, we used Dijkstra's weakest pre-condition computation to infer $\xi'(N-1)$ for strengthening the pre-condition of the difference program. Subsequently, we have used quantifier elimination techniques to compute the post-condition $\xi(N-1)$ of the program P_{N-1} using difference invariants $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ that are restricted to equalities. This can be further extended to the case where the difference invariants are not necessarily equality predicates. In such cases, the idea is to use abduction techniques for computing a formula $\xi(N-1)$ given the formulas $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ and $\xi'(N-1)$ such that $\xi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1) \Rightarrow \xi'(N-1)$.

Potential Applications Beyond Verification

Ideas used in full-program induction, especially the computation of difference programs, have potential applications in several program transformations and compiler optimiza-

tions such as incrementalization. To see this consider applications that generate a stream of data during their operation. The generated data is subsequently processed by programs that compute the result of applying a function over the input data. In such computations, modification to the input must be processed instantaneously, instead of batch processing, so that its effect on the output is immediately visible. A program is called *offline* if the entire data is required at the beginning of the computation to compute the result a function. In contrast, a *online* program does not need the entire data at the beginning of the computation and performs partial computation on a data point as and when it is provided as input to the program. Incrementalization is a program transformation technique proposed in the literature that aims to achieve this. Incrementalization transforms a program in a way that executing it on the modified input is comparatively cheaper than executing the original program. This optimization can be formally described as follows. Suppose $\mathcal{I}_N$ represents the input to a program P_N, that can be decomposed into $\mathcal{I}_{N-1}$ and $\Diamond\mathcal{I}_N$. Then, incrementalization generates an incrementalized program $\Diamond\mathsf{P}_N$ such that $\mathsf{P}_N(\mathcal{I}_{N-1} + \Diamond\mathcal{I}_N)$ is semantically equivalent to $\mathsf{P}_{N-1}(\mathcal{I}_{N-1})$; $\Diamond\mathsf{P}_N(\Diamond\mathcal{I}_N)$ and the cost of computing $\Diamond\mathsf{P}_N$ on the new piece of input $\Diamond\mathcal{I}_N$ and combining it with the output of $\mathsf{P}_{N-1}(\mathcal{I}_{N-1})$ is significantly cheaper than executing P_N on the entire input $\mathcal{I}_N$. Every time the input is modified, i.e. it is appended with $\Diamond\mathcal{I}_N$, then the incrementalized program $\Diamond\mathsf{P}_N$ is executed. Notice that the computation of $\Diamond\mathsf{P}_N$ can be viewed as the decomposition of P_N into P_{N-1}; $\Diamond\mathsf{P}_N$, which resembles our difference computation. We strongly believe that the computation of difference programs can be adapted to automatically transform offline batch processing programs into their online versions. While the goal of computing the difference program in our technique is to simplify verification, it would be interesting to see how difference computation can be adapted for the purposes of generating the incrementalized program $\Diamond\mathsf{P}_N$ in this optimization.

The difference computation may have a worthwhile impact on the way in which loops in a program are fused together by compilers. Loop fusion (also called loop jamming) is a program transformation technique implemented in compilers that can potentially reduce the number of loops in the input program. Programs transformed using loop fusion offer advantages such as (i) they are easier to parallelize, (ii) they can make better reuse of data, and (iii) they have an increased scope for the application of other program optimizations. Past work on loop fusion is limited due to the restrictions imposed by the

applicable program analysis techniques to return fruitful results. Further, the techniques in the literature do not take in to consideration the given post-condition that the resultant optimized program must satisfy. Hence, such technique take a very conservative approach in fusing loops. We take a very aggressive approach while generating the difference programs during full-program induction. Our approach may be useful to generate a significant more optimized version of the program with multiple sequentially composed loops fused in a way that the given post-condition is satisfied after the transformation.

Application of inductive reasoning to verify programs with other data structures can also be considered. For example, separation logic has been used for verifying programs that manipulate heaps, lists, trees, and graphs. Adapting the full-program induction technique and its variants to verify heap-manipulating programs makes an interesting direction that can be pursued.

Finally, automated support for applying the full-program induction technique to verify array-manipulating programs that store integers, matrices, polynomials, vectors and other types of data is another interesting line of work that can be pursued. Arrays storing matrices as elements, called *tensors*, are extensively used in machine learning and cryptography domains. We believe that our induction-based techniques can be adapted for efficiently verifying APIs in libraries that manipulate tensors.

Bibliography

[ABG+12a] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. Lazy abstraction with interpolants for arrays. In *Proc. of LPAR*, pages 46–61, 2012. 19, 68, 187, 240

[ABG+12b] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. SAFARI: SMT-based abstraction for arrays with interpolants. In *Proc. of CAV*, pages 679–685, 2012. 19

[ACC+20] Mohammad Afzal, Supratik Chakraborty, Avriti Chauhan, Bharti Chimdyalwar, Priyanka Darke, Ashutosh Gupta, Shrawan Kumar, Charles Babu M, Divyesh Unadkat, and R. Venkatesh. VeriAbs: Verification by abstraction and test generation (competition contribution). In *Proc. of TACAS*, pages 383–387, 2020. 18, 72, 140, 144, 182, 189, 190, 193, 195, 230, 240

[AGS14] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. Booster: An acceleration-based verification framework for array programs. In *Proc. of ATVA*, pages 18–23, 2014. 19, 41, 60, 62, 68, 72, 140, 182, 186, 187, 240

[ARG+21] Omar M Alhawi, Herbert Rocha, Mikhail R Gadelha, Lucas C Cordeiro, and Eddie Batista. Verification and refutation of C programs based on k-induction and invariant inference. *STTT*, 23(2):115–135, 2021. 24, 187

[Bas04] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT*, pages 7–16, 2004. 27

[Bas12] C Bastoul. Clay: the chunky loop alteration wizardry. Technical report, LRI, Paris-Sud University, France, 2012. 27

[Bas13] C Bastoul. Code generation in the polyhedral model. In *PACT*, pages 7–16, 2013. 27

[BBK+08] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *CC*, pages 132–146, 2008. 27

[BC00] Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *Proc. of FMCAD*, pages 409–426, 2000. 24, 187

[BCG+03] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In *LCPC*, pages 209–225, 2003. 27

[BCK11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *Proc. of FM*, pages 200–214, 2011. 201, 224

[BCS20] Denis Bueno, Arlen Cox, and Karem A Sakallah. EUFicient reachability in software with arrays. In *Proc. of FMCAD*, pages 57–66, 2020. 18

[BDW15] Dirk Beyer, Matthias Dangl, and Philipp Wendler. Boosting k-induction with continuously-refined invariants. In *Proc. of CAV*, pages 622–640, 2015. 24, 187

[BEG+19] Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovács, and Matteo Maffei. Verifying relational properties using trace logic. In *Proc. of FMCAD*, pages 170–178, 2019. 23

[Bey19] Dirk Beyer. Competition on software verification (SV-COMP). 2019. 230

[Bey20] Dirk Beyer. Competition on software verification (SV-COMP). 2020. 230

[Bey21] Dirk Beyer. Competition on software verification (SV-COMP). 2021. 230

[BGDR10] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *PACT*, pages 343–352, 2010. 27

[BGR12] Roberto Bruttomesso, Silvio Ghilardi, and Silvio Ranise. Quantifier-free interpolation of a theory of arrays. *LMCS*, 8(2), 2012. 19

[BHI+09] Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konečnỳ, and Tomáš Vojnar. Automatic verification of integer array programs. In *Proc. of CAV*, pages 157–172, 2009. 21

[BHMR07] Dirk Beyer, Thomas A Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *Proc. of VMCAI*, pages 378–394, 2007. 21, 75, 187, 240

[BHRS08] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, pages 101–113, 2008. 28

[BIK10] Marius Bozga, Radu Iosif, and Filip Konecný. Fast acceleration of ultimately periodic relations. In *Proc. of CAV*, pages 227–242, 2010. 19, 68, 187, 240

[Bin92] David W Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proc. of ICSM*, pages 41–50, 1992. 26, 190

[BJ15] Nikolaj Bjørner and Mikolás Janota. Playing with quantified satisfaction. In *Proc. of LPAR*, pages 15–27, 2015. 68

[BJKS15] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. Safety verification and refutation by k-invariants and k-induction. In *Proc. of SAS*, pages 145–161, 2015. 24, 187

[BMR13] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. On solving universally quantified horn clauses. In *Proc. of SAS*, pages 105–125, 2013. 22, 240

[BP12] Cédric Bastoul and Louis-Noël Pouchet. Candl: the chunky analyzer for dependences in loops. Technical report, LRI, Paris-Sud University, France, 2012. 27

[BPCB10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *CC*, pages 283–303, 2010. 27

[Bra07] Aaron R Bradley. *Safety analysis of systems*. PhD thesis, Stanford University, USA, 2007. 37

[Bra11] Aaron R Bradley. SAT-based model checking without unrolling. In *Proc. of VMCAI*, pages 70–87, 2011. 20, 24

[Bra12] Aaron R Bradley. Understanding IC3. In *Proc. of SAT*, pages 1–14, 2012. 20

[BWZ94] Olaf Bachmann, Paul S Wang, and Eugene V Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *SAC*, pages 242–249, 1994. 28

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *FMSD*, 19(1):7–34, 2001. 75

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL*, pages 238–252, 1977. 19

[CCH08] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, 2008. 28

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proc. of POPL*, pages 105–118, 2011. 19, 49, 68, 75, 189, 240

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV*, pages 154–169, 2000. 20

[CGU17] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Verifying array manipulating programs by tiling. In *Proc. of SAS*, pages 428–449, 2017. vi, 39, 72, 140, 186, 187, 188, 239

[CGU20a] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Verifying array manipulating programs with full-program induction. In *Proc. of TACAS*, pages 22–39, 2020. v, 71, 137, 182, 193, 195, 203, 230, 239

[CGU20b] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Verifying array manipulating programs with full-program induction - Artifacts TACAS 2020, 10.6084/m9.figshare.11875428.v1 figshare 2020. v, vi, 137, 180, 193, 195, 230

[CGU21a] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Diffy: Inductive reasoning of array programs using difference invariants, 10.6084/m9.figshare.14509467 figshare 2021. v, 189, 229

[CGU21b] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Diffy: Inductive reasoning of array programs using difference invariants. In *Proc. of CAV*, pages 911–935, 2021. v, 176, 177, 187, 189, 194, 240

[CGU22] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Full-Program Induction: Verifying array programs sans loop invariants. *STTT*, 24(5):843–888, 2022. v, 71, 137, 193, 195, 203, 230, 239

[CJRS13] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *Proc. of CADE*, pages 392–406, 2013. 24, 187

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proc. of TACAS*, pages 168–176, 2004. 41, 43, 60

[CL98] Philippe Clauss and Vincent Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI signal processing systems for signal, image and video technology*, 19(2):179–194, 1998. 28

[Cou03] Patrick Cousot. Verification by abstract interpretation. In *Proc. of VTP*, pages 243–268, 2003. 19

[CPSA19] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proc. of PLDI*, pages 1027–1040, 2019. 224

[CS01] Michael A Colóon and Henny B Sipma. Synthesis of linear ranking functions. In *Proc. of TACAS*, pages 67–81, 2001. 169

[CS02] Michael A Colón and Henny B Sipma. Practical methods for proving program termination. In *Proc. of CAV*, pages 442–454, 2002. 169

[Dat09] Kaushik Datta. *Auto-tuning stencil codes for cache-based multicore platforms.* PhD thesis, University of California, Berkeley, USA, 2009. 68

[DB17] Manjeet Dahiya and Sorav Bansal. Black-box equivalence checking across compiler optimizations. In *Proc of APLAS*, pages 127–147, 2017. 201

[DDA10] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid Updates: Beyond strong vs. weak updates. In *Proc. of ESOP*, pages 246–266, 2010. 19, 68, 189, 240

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical computer science*, 17(3):279–301, 1982. 171

[DGG00] Dennis Dams, Rob Gerth, and Orna Grumberg. A heuristic for the automatic generation of ranking functions. In *Proc. of WAV*, pages 1–8, 2000. 169

[DHKR11] Alastair F Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In *Proc. of SAS*, pages 351–368, 2011. 24, 187

[DKR10] Alastair F Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *Proc. of TACAS*, pages 280–295, 2010. 24, 187

[DM97] David Déharbe and Anamaria Martins Moreira. Using induction and BDDs to model check invariants. In *Proc. of AHDV*, pages 203–213. 1997. 24, 187

[DMRS03] Leonardo De Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In *Proc. of CAV*, pages 14–26, 2003. 24, 187

[Eng01] Robert A van Engelen. Efficient symbolic analysis for optimizing compilers. In *CC*, pages 118–132, 2001. 28

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007. 22, 43, 46, 59, 75

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *ENTCS*, 89(4):543–560, 2003. 24, 187

[Fea88] Paul Feautrier. Parametric integer programming. *RAIRO-Operations Research*, 22(3):243–268, 1988. 28

[Fea91] Paul Feautrier. Dataflow analysis of scalar and array references. *IJPP*, 20(1):23–53, 1991. 27

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proc. of FME*, pages 500–517, 2001. 75

[FL11] Paul Feautrier and Christian Lengauer. Polyhedron model. *Encyclopedia of Parallel Computing*, 1:1581–1592, 2011. 27

[FMV14] Carlo A Furia, Bertrand Meyer, and Sergey Velder. Loop invariants: Analysis, classification, and examples. *CSUR*, 46(3):1–51, 2014. 23

[FOW87] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987. 106

[FPMG19] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified invariants via syntax-guided-synthesis. In *Proc. of CAV*, pages 259–277, 2019. 22, 72, 75, 140, 182, 186, 240

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *ACM SIGPLAN Notices*, pages 191–202, 2002. 20

[GFM+15] Juan P Galeotti, Carlo A Furia, Eva May, Gordon Fraser, and Andreas Zeller. Inferring loop invariants by mutation, dynamic analysis, and static checking. *TSE*, 41(10):1019–1037, 2015. 22

[GGK20] Pamina Georgiou, Bernhard Gleiss, and Laura Kovács. Trace logic for inductive loop reasoning. In *Proc. of FMCAD*, pages 255–263, 2020. 23, 186, 187, 240

[GIC17] Mikhail YR Gadelha, Hussama I Ismail, and Lucas C Cordeiro. Handling loops in bounded model checking of C programs via k-induction. *STTT*, 19(1):97–114, 2017. 24, 187

[GLD09] Daniel Große, Hoang M Le, and Rolf Drechsler. Induction-based formal verification of SystemC TLM designs. In *Proc. of WMTV*, pages 101–106, 2009. 24, 187

[GMT08] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proc. of POPL*, pages 235–246, 2008. 20, 68, 75, 187, 240

[GR09] Ashutosh Gupta and Andrey Rybalchenko. InvGen: An efficient invariant generator. In *Proc. of CAV*, pages 634–640, 2009. 21, 43

[GR10] Silvio Ghilardi and Silvio Ranise. MCMT: A model checker modulo theories. In *Proc. of AR*, pages 22–29, 2010. 19

[GRB20] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. Counterexample-guided correlation algorithm for translation validation. In *Proc. of OOPSLA*, volume 4, pages 1–29, 2020. 201, 224

[GRS05] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *Proc. of POPL*, pages 338–350, 2005. 19, 68, 75, 189, 240

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Proc. of CAV*, pages 72–83, 1997. 20

[GSV18] Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Quantifiers on demand. In *Proc. of ATVA*, pages 248–266, 2018. 20, 22, 72, 186, 240

[GZA+11] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *IMPACT*, volume 2011, page 1, 2011. 27

[HB08] Reiner Hähnle and Richard Bubel. A Hoare-style calculus with explicit state updates. *FMCSE*, pages 49–60, 2008. 52

[HCE+15] Arvind Haran, Montgomery Carter, Michael Emmi, Akash Lal, Shaz Qadeer, and Zvonimir Rakamaric. SMACK+Corral: A modular verifier - (competition contribution). In *Proc. of TACAS*, pages 451–454, 2015. 62

[HHKR10] Thomas A Henzinger, Thibaud Hottelier, Laura Kovács, and Andrey Rybalchenko. Aligators for arrays (tool paper). In *Proc. of LPAR*, pages 348–356, 2010. 22, 72, 186

[HIV08] Peter Habermehl, Radu Iosif, and Tomáš Vojnar. A logic of singly indexed arrays. In *Proc. of LPAR*, pages 558–573, 2008. 21

[HKV11] Kryštof Hoder, Laura Kovács, and Andrei Voronkov. Invariant generation in vampire. In *Proc. of TACAS*, pages 60–64, 2011. 22

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580, 1969. 3

[HP08] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *Proc. of PLDI*, pages 339–348, 2008. 19, 68, 75, 189, 240

[HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *TOPLAS*, 11(3):345–387, 1989. 25, 26, 190, 245

[HR92] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proc. of ICSE*, pages 392–411, 1992. 106

[HT08] George Hagen and Cesare Tinelli. Scaling up the formal verification of lustre programs with SMT-based techniques. In *Proc. of FMCAD*, pages 1–9, 2008. 24, 187

[ISIRS20] Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Putting the squeeze on array programs: Loop verification via inductive rank reduction. In *Proc. of VMCAI*, pages 112–135, 2020. 24, 187, 188, 239

[JK11] Youngjoon Jo and Milind Kulkarni. Enhancing locality for recursive traversals of recursive structures. In *Proc. of OOPSLA*, pages 463–482, 2011. 49, 69

[JKD+16] Anushri Jana, Uday P. Khedker, Advaita Datar, R. Venkatesh, and Niyas C. Scaling bounded model checking by transforming programs with arrays. In *Proc. of LOPSTR*, pages 275–292, 2016. 68

[JM07] Ranjit Jhala and Kenneth L. McMillan. Array abstractions from proofs. In *Proc. of CAV*, pages 193–206, 2007. 19, 68, 75, 187, 189, 240

[JSP+11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *Proc. of NFM*, pages 41–55, 2011. 52, 189

[JSS14] Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. Abstract acceleration of general linear loops. In *Proc. of POPL*, pages 529–540, 2014. 19, 68, 187, 240

[KA01] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. 2001. 106

[KBGM15] Anvesh Komuravelli, Nikolaj Bjorner, Arie Gurfinkel, and Kenneth L McMillan. Compositional verification of procedural programs using Horn clauses over integers and arrays. In *Proc. of FMCAD*, pages 89–96, 2015. 72, 186

[KBI+17] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *JACM*, 64(1):1–33, 2017. 20

[KGC14] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-Based model checking for recursive programs. In *Proc. of CAV*, pages 17–34, 2014. 18, 62, 67

[KMW67] Richard M Karp, Raymond E Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *JACM*, 14(3):563–590, 1967. 27

[KN22] DA Kondratyev and VA Nepomniaschy. Automation of c program deductive verification without using loop invariants. *PCS*, 48(5):331–346, 2022. 239

[KS98] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *Proc. of POPL*, pages 107–120, 1998. 87, 207

[KS16] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition.* Springer, 2016. 37

[KT11] Temesghen Kahsai and Cesare Tinelli. PKind: A parallel k-induction based model checker. In *Proc. of PDMC*, pages 55–62, 2011. 24, 187

[Kuc78] David L Kuck. *Structure of Computers and Computations.* John Wiley & Sons, Inc., USA, 1978. 98

[KV09] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proc. of FASE*, pages 470–485, 2009. 22

[KVGG19] Hari Govind Vediramana Krishnan, Yakir Vizel, Vijay Ganesh, and Arie Gurfinkel. Interpolating strong induction. In *Proc. of CAV*, pages 367–385, 2019. 24, 187

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of CGO*, pages 75–86, 2004. 59, 143, 180, 197, 229

[Lam74] Leslie Lamport. The parallel execution of do loops. *CACM*, 17(2):83–93, 1974. 27

[LB04] Shuvendu K Lahiri and Randal E Bryant. Constructing quantified invariants via predicate abstraction. In *Proc. of VMCAI*, pages 267–281, 2004. 20

[Len93] Christian Lengauer. Loop parallelization in the polytope model. In *International Conference on Concurrency Theory*, pages 398–416, 1993. 27

[Les82] Pierre Lescanne. Some properties of decomposition ordering, a simplification ordering to prove termination of rewriting systems. *RAIRO. Informatique théorique*, 16(4):331–347, 1982. 171

[LHKR12] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Proc. of CAV*, pages 712–717, 2012. 26, 190

[Lin16] Fangzhen Lin. A formalization of programs in first-order logic with a discrete linear order. *Artificail Intelligence*, 235:1–25, 2016. 23, 187, 196, 240

[LMS+04] Richard Lethin, Peter Mattson, Eric Schweitz, Allen Leung, Vass Litvinov, Michael Engle, and Charlie Garrett. R-stream 3.0: Technologies for high level embedded application mapping. Technical report, 2004. 27

[LQL12] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In *Proc. of CAV*, pages 427–443, 2012. 41

[LR15] Jiangchao Liu and Xavier Rival. Abstraction of arrays based on non contiguous partitions. In *Proc. of VMCAI*, pages 282–299, 2015. 20, 68, 75, 189, 240

[LSLR05] Yanhong A Liu, Scott D Stoller, Ning Li, and Tom Rothamel. Optimizing aggregate array computations in loops. *TOPLAS*, 27(1):91–125, 2005. 26, 190, 239

[LST98] Yanhong A Liu, Scott D Stoller, and Tim Teitelbaum. Static caching for incremental computation. *TOPLAS*, 20(3):546–585, 1998. 26, 190, 239

[LVH10] Shuvendu K Lahiri, Kapil Vaswani, and C AR Hoare. Differential static analysis: Opportunities, applications, and challenges. In *Proc. of WFSER*, pages 201–204, 2010. 25, 26, 190

[MA15] David Monniaux and Francesco Alberti. A simple abstraction of arrays and maps by program translation. In *Proc. of SAS*, pages 217–234, 2015. 19, 68, 187, 240

[MB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS*, pages 337–340, 2008. 18, 38, 52, 60, 67, 68, 75, 143, 181, 197, 202, 229

[MBS+22] Benoit Meister, Muthu Baskaran, Maxime Schmitt, Klint Qinami, James Gilles, Adithya Dattatri, Peter Couperus, Ryan Senanayake, Jonathan Springer, and Richard Lethin. R-stream 3.22. 3-doe. Technical report, 2022. 27

[McM08] Kenneth L McMillan. Quantified invariant generation using an interpolating saturation prover. In *Proc. of TACAS*, pages 413–427, 2008. 21, 22

[MG16] David Monniaux and Laure Gonnord. Cell Morphing: From array programs to array-free horn clauses. In *Proc. of SAS*, pages 361–382, 2016. 18, 41, 60, 62, 67, 68, 72, 140, 182, 186, 240

[MGJ+19] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proc. of SOSP*, pages 370–384, 2019. 22

[MIG+21] Makai Mann, Ahmed Irfan, Alberto Griggio, Oded Padon, and Clark Barrett. Counterexample-guided prophecy for model checking modulo the theory of arrays. In *Proc. of TACAS*, 2021. 17, 189, 240

[MR22] Canberk Morelli and Jan Reineke. Warping cache simulation of polyhedral programs. In *PLDI*, pages 316–331, 2022. 27

[Muc97] Steven Muchnick. *Advanced Compiler Design Implementation.* Morgan kaufmann, 1997. 8, 9, 48, 69

[MVB15] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *ASPLOS*, pages 429–443, 2015. 27

[Nec00] George C Necula. Translation validation for an optimizing compiler. In *Proc of PLDI*, pages 83–94, 2000. 201

[NKWF12] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to discover polynomial and array invariants. In *Proc. of ICSE*, pages 683–693, 2012. 22

[NO79] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *TOPLAS*, 1(2):245–257, 1979. 37

[PBCC08] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part ii, multidimensional time. *ACM SIGPLAN Notices*, 43(6):90–100, 2008. 27

[PBCV07] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *CGO*, pages 144–156, 2007. 27

[PCB+06] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. Graphite: Polyhedral analyses and optimizations for gcc. In *GCC Developers Summit*, volume 6, pages 90–91, 2006. 27

[PK82] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *TOPLAS*, 4(3):402–454, 1982. 25, 190, 239

[Pou10] LN Pouchet. Pocc, the polyhedral compiler collection, version 1.3, 2010. 28

[PR04] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proc. of VMCAI*, pages 239–251, 2004. 169, 172, 173

[Pug91a] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991. 28

[Pug91b] William Pugh. Uniform techniques for loop optimization. In *ICS*, pages 341–352, 1991. 27

[QRW00] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000. 27

[RAL+13] Bin Ren, Gagan Agrawal, James R. Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. SIMD parallelization of applications that traverse irregular data structures. In *Proc. of CGO*, pages 20:1–20:10, 2013. 69

[RHK13] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for Horn-clause verification. In *Proc. of CAV*, pages 347–363, 2013. 18, 67

[RK15] Andrew Reynolds and Viktor Kuncak. Induction for SMT solvers. In *Proc. of VMCAI*, pages 80–98, 2015. 24, 187

[RL18] Pritom Rajkhowa and Fangzhen Lin. Extending VIAP to handle array programs. In *Proc. of VSTTE*, pages 38–49, 2018. 23, 29, 72, 140, 182, 186, 187, 193, 196, 230, 240

[RWZ88] Barry K Rosen, Mark N Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proc. of POPL*, pages 12–27, 1988. 86, 207

[SB07] Ajeet Shankar and Rastislav Bodik. DITTO: Automatic incrementalization of data structure invariant checks (in Java). *ACM SIGPLAN Notices*, 42(6):310–319, 2007. 26, 190, 239

[SB11] Fabio Somenzi and Aaron R Bradley. IC3: where monolithic and incremental meet. In *Proc. of FMCAD*, pages 3–8, 2011. 20

[SB12] Mohamed Nassim Seghir and Martin Brain. Simplifying the verification of quantified array assertions via code transformation. In *Proc. of LOPSTR*, pages 194–212, 2012. 2, 25, 187, 188, 193, 203, 239

[SG09] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. *ACM Sigplan Notices*, 44(6):223–234, 2009. 20, 75, 187, 240

[SGH+13] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori. A data driven approach for algebraic loop invariants. In *Proc. of ESOP*, pages 574–592, 2013. 22

[SGSV19] Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Property directed self composition. In *Proc. of CAV*, pages 161–179, 2019. 20

[Sho78] Robert E Shostak. An algorithm for reasoning about equality. *CACM*, 21(7):583–585, 1978. 37

[SLLM06] Eric Schweitz, Richard Lethin, Allen Leung, and Benoit Meister. R-stream: A parametric high level compiler. *HPEC*, 2006. 27

[SSCA13] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-driven equivalence checking. In *Proc of OOPSLA*, pages 391–406, 2013. 201

[SSK17] Kirshanthan Sundararajah, Laith Sakka, and Milind Kulkarni. Locality transformations for nested recursive iteration spaces. In *Proc. of ASPLOS*, pages 281–295, 2017. 49

[SSK22] Kirshanthan Sundararajah, Charitha Saumya, and Milind Kulkarni. Unirec: a unimodular-like framework for nested recursions and loops. *PACMPL*, 6(OOPSLA2):1264–1290, 2022. 27

[SSM04] Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. Non-linear loop invariant generation using Gröbner bases. In *Proc. of POPL*, pages 318–329, 2004. 21

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proc. of FMCAD*, pages 127–144, 2000. 2, 24, 71, 187

[TCE+10] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GROW*, 2010. 27

[TKA22] Hugo Thievenaz, Keiji Kimura, and Christophe Alias. Lightweight array contraction by trace-based polyhedral analysis. In *C3PO*, 2022. 27

[Tow76] Ross Albert Towle. *Control and Data Dependence for Program Transformations.* PhD thesis, University of Illinois at Urbana-Champaign, USA, 1976. 98

[TW16] Nishant Totla and Thomas Wies. Complete instantiation-based interpolation. *AR*, 57(1):37–65, 2016. 22

[Unaa] Divyesh Unadkat. DIFFY CAV 2021 Artifact. https://github.com/divyeshunadkat/diffy-artifact. Accessed: 2022-06-30. v

[Unab] Divyesh Unadkat. TILER SAS 2017 VM. https://bit.ly/3zNBYkn. Accessed: 2022-06-30. vi, 39, 59

[Unac] Divyesh Unadkat. VAJRA TACAS 2020 Artifact. https://github.com/divyeshunadkat/VAJRA. Accessed: 2022-06-30. vi

[UTS17] Hiroshi Unno, Sho Torii, and Hiroki Sakamoto. Automating induction for solving Horn clauses. In *Proc. of CAV*, pages 571–591, 2017. 24, 187

[VCJC+13] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–23, 2013. 28

[Ver10] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *ICMS*, pages 299–302, 2010. 28

[YBH21] Emily Yu, Armin Biere, and Keijo Heljanko. Progress in certifying hardware model checking results. In *Proc. of CAV*, pages 363–386, 2021. 24, 187

[YHR92] Wuu Yang, Susan Horwitz, and Thomas Reps. A program integration algorithm that accommodates semantics-preserving transformations. *TOSEM*, 1(3):310–354, 1992. 27

[YU13] Anand Yeolekar and Divyesh Unadkat. Assertion checking using dynamic inference. In *Proc. of HVC*, pages 199–213, 2013. 22

[YUA+13] Anand Yeolekar, Divyesh Unadkat, Vivek Agarwal, Shrawan Kumar, and R Venkatesh. Scaling model checking for test generation using dynamic inference. In *Proc. of ICST*, pages 184–191, 2013. 22

[ZBY+22] Jie Zhao, Cédric Bastoul, Yanzhi Yi, Jiahui Hu, Wang Nie, Renwei Zhang, Zhen Geng, Chong Li, Thibaut Tachon, and Zhiliang Gan. Parallelizing neural network models effectively on gpu by implementing reductions atomically. In *PACT*, volume 22, 2022. 27

[ZLL+22] Jianwei Zheng, Yu Liu, Xuejiao Liu, Luhong Liang, Deming Chen, and Kwang-Ting Cheng. Reaap: A reconfigurable and algorithm-oriented array processor with compiler-architecture co-design. *IEEE Transactions on Computers*, 2022. 27

[ZP08] Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *Proc of FM*, pages 35–51, 2008. 201, 223

[ZPFG02] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A translation validator for optimizing compilers. *ENTCS*, 65(2):2–18, 2002. 201

[ZYR+14] Lingming Zhang, Guowei Yang, Neha Rungta, Suzette Person, and Sarfraz Khurshid. Feedback-driven dynamic invariant discovery. In *Proc. of ISSTA*, pages 362–372, 2014. 22

Acknowledgments

"If I am able to see further than others, it is by standing on the shoulders of giants"

- Sir Issac Newton

My heart is filled with gratitude and appreciation for the wonderful faculty of IIT Bombay who have supported me during my PhD. First and foremost, I would like to express sincere gratitude to my supervisor *Prof. Supratik Chakraborty*. His able leadership and guidance enabled this work. His passion for research, enthusiasm for challenging problems, ability to simplify complex concepts, sharp focus and a never give up attitude is the motivation behind all our contributions. He has been one of the few people to have stood by me during the toughest of times. His ability to understand student mindset and provide the right amount of motivation and guidance has helped me through the course of my PhD. I am forever indebted to him.

I am thankful to my co-supervisor *Prof. Ashutosh Gupta* for his guidance. Under his supervision, I excelled at building the verification tools presented in the thesis. His passion for systems such as LLVM compiler framework and Z3 SMT solver is quite infectious. On countless occasions his belief in my abilities has helped me overcome tooling challenges. He is adapt at easing tense moments with situational awareness and timely advice. With him I also have enjoyed many visits to Starbucks in Mumbai and walks at the TIFR seashore.

I express my sincere gratitude to all the faculty members of CSE department. I am extremely grateful to the members of my research progress committee *Prof. Uday Khedker* and *Prof. Bharat Adsul*. Their invaluable feedback has helped me improve the work. I would like to thank *Prof. Shankara Narayanan Krishna*, *Prof. S Akshay* and *Prof. Nutan Limaye* for being a constant source of encouragement. I would like to thank *Prof. Sharat Chandran* and *Prof. Mythili Vutukuru* for organizing the 3 minute thesis talk and poster presentation competition *RISC* in the CSE department.

I acknowledge the administrative staff of IIT Bombay for their assistance during the course of my studies. Particularly, Vijay Ambre has been helpful at several occasions. I thank Chandrakant Talekar and Killedar Kakasaheb from the CFDVS lab for their resourcefulness.

I am thankful to TCS Research for giving me the opportunity to pursue Ph.D. Their support is gratefully acknowledged. I would like to thank all my colleagues from TCS Innovation Labs Pune.

I acknowledge all my friends who supported and encouraged me during my Ph.D. journey. Especially all members of the CFDVS lab who make it an amazing place to hang out and learn. I will cherish the wonderful memories from my days in the lab for the rest of my life.

Last and most importantly, I express heartfelt gratitude to my family for their unconditional and unwavering support through this endeavor. Their love and care saw me through the toughest times of my life.

A big thanks again to all those who have been involved with different aspects of this work and my life. It is an honor for me to have interacted with you. I am sincerely thankful to you for bringing out the best in me.

Date: 2022 **Unadkat Divyesh Pankaj**