Automatic Data Dependence Analysis by Deductive Verification

Automatische Datenabhängigkeitsanalyse durch deduktive Verifikation
Zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)
Genehmigte Dissertation im Fachbereich Informatik von Asmae Heydari Tabar aus Teheran, Iran
Tag der Einreichung: 1.03.2024, Tag der Prüfung: 24.04.2024

1. Gutachten: Prof. Dr. Reiner Hähnle
2. Gutachten: Prof. Dr. Philipp Rümmer
Darmstadt, Technische Universität Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department
Software Engineering Group

Automatic Data Dependence Analysis by Deductive Verification
Automatische Datenabhängigkeitsanalyse durch deduktive Verifikation

Accepted doctoral thesis in the department of Computer Science by Asmae Heydari Tabar

Date of submission: 1.03.2024
Date of thesis defense: 24.04.2024

Darmstadt, Technische Universität Darmstadt

To Mahsa Zhina Amini.

Erklärungen laut Promotionsordnung

§ 8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§ 9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation – abgesehen von den in ihr ausdrücklich genannten Hilfen – selbstständig verfasst wurde und dass die „Grundsätze zur Sicherung guter wissenschaftlicher Praxis an der Technischen Universität Darmstadt“ und die „Leitlinien zum Umgang mit digitalen Forschungsdaten an der TU Darmstadt“ in den jeweils aktuellen Versionen bei der Verfassung der Dissertation beachtet wurden.

§ 9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 1.03.2024

A. Heydari Tabar

Acknowledgments

I am in debt of gratitude to Prof. Dr. Reiner Hähnle for giving me the opportunity to work with him and providing a friendly work environment where everyone can grow while rooting for each other. I appreciate him always being available and providing guidance while never losing sight of the big picture. Working at the Software Engineering group is a luxury where people's well-being and happiness always come first.

I am grateful to Prof. Dr. Philipp Rümmer for agreeing to be the second reviewer of my thesis and for his previous guidance regarding my academic career. I also appreciate the examination committee members, Prof. Dr. Christian Bischof, Prof. Dr. Georgia Chalvatzaki, and Prof. Dr. Dr. Mira Mezini, for their time and effort.

This work would not have been possible without the everyday help of Dr. Richard Bubel. I appreciate his kindness, patience, and humbleness. It is a privilege to learn from him.

I am thankful for my former colleagues, whom I always look up to, Dr. Dominic Steinhöfel and Dr. Eduard Kamburjan, who were always supportive, kind, and humorous. The same goes for my current colleagues (in alphabetic order): Stefan Dillmann, Daniel Drodt, Lukas Grätz, Anna Schmitt, Marco Scaletta, and Dr. Adele Veschetti. I also appreciate our kind and patient secretary, Claudia Roßmann.

Special thanks to Dr. Richard Bubel and Daniel Drodt for providing constructive and detailed feedback on my thesis.

Through working at the Software Engineering group, I had the privilege of meeting great scientists who took my work seriously and tried to promote it in different ways they could. Particularly, I am heart-warmed by the support of Prof. Dr. Ina Schaefer, Prof. Dr. Marieke Huisman, and Prof. Dr. Einar Broch Johnsen.

I would like to thank my parents for their tireless encouragement throughout my life to pursue a career in science. This is for them.

I was lucky to meet my boyfriend, Tobias Hamann, during and through doing my Ph.D. I appreciate his presence in my life every day. I could not have undertaken this journey without his unconditional love and support.

During my Ph.D., funding was provided by the Software Factory 4.0 project, Technical University of Darmstadt, and the DEEP-SEA project, for which I am grateful.

Abstract

In the realm of High-Performance Computing (HPC), the parallelization of programs holds significant importance. However, the correctness of parallelization hinges on the reliable exclusion of certain data dependences, such as read-after-write dependences, where a read access follows a write access on a given memory location. It is imperative that data dependence analyses are not only correct but also as precise as possible to seize every opportunity for parallelization.

While various static, dynamic, and hybrid analysis approaches have been proposed within the HPC community, none have been based on program logic and deductive verification, despite the significant advantages this approach offers, including soundness, precision, and modularity.

In this thesis, we present an automatic, sound, and highly precise approach to generate data dependences based on deductive verification. We define a program logic based on precise semantics for data dependences. As loops are usually the main source of parallelization in HPC applications, we equip our approach with an automatic loop invariant generation technique in the same program logic. To achieve full automation, we incorporate predicate abstraction tailored to the needs of data dependence analysis. To retain as much precision as possible, we generalize logic-based symbolic execution to compute abstract data dependence predicates.

We provide a prototype demonstrating that fully automatic data dependence analysis based on deductive verification is feasible and is a promising alternative to the dependence analyses commonly used in HPC. Implementing our approach for Java atop a deductive verification tool, we conducted evaluations demonstrating its ability to analyze data dependences highly precisely for representative code extracted from HPC applications.

Zusammenfassung

Im Bereich des High-Performance Computing (HPC) kommt der Parallelisierung von Programmen eine große Bedeutung zu. Die Richtigkeit der Parallelisierung hängt jedoch vom zuverlässigen Ausschluss bestimmter Datenabhängigkeiten ab, beispielsweise von Read-After-Write-Abhängigkeiten, bei denen ein Lesezugriff auf einen Schreibzugriff auf einen bestimmten Speicherort folgt. Es ist zwingend erforderlich, dass Datenabhängigkeitsanalysen nicht nur korrekt, sondern auch so präzise wie möglich sind, um jede Gelegenheit zur Parallelisierung zu nutzen.

Während in der HPC-Community verschiedene statische, dynamische und hybride Analyseansätze vorgeschlagen wurden, basierte keiner auf Programmlogik und deduktiver Verifizierung, trotz der erheblichen Vorteile, die dieser Ansatz bietet, einschließlich Korrektheit, Präzision und Modularität.

In dieser Arbeit stellen wir einen automatischen, korrekten und hochpräzisen Ansatz zur Generierung von Datenabhängigkeiten basierend auf deduktiver Verifizierung vor. Wir definieren eine Programmlogik basierend auf einer präzisen Semantik für Datenabhängigkeiten. Da Schleifen in der Regel die Hauptquelle der Parallelisierung in HPC-Anwendungen sind, statten wir unseren Ansatz mit einer automatischen Technik zur Erzeugung von Schleifeninvarianten in derselben Programmlogik aus. Um eine vollständige Automatisierung zu erreichen, integrieren wir eine Prädikatenabstraktion, die auf die Anforderungen der Datenabhängigkeitsanalyse zugeschnitten ist. Um so viel Präzision wie möglich beizubehalten, verallgemeinern wir die logikbasierte symbolische Ausführung, um abstrakte Datenabhängigkeitsprädikate zu berechnen.

Der vorgestellte Ansatz wurde protptypisch implementiert und zeigt, dass eine vollautomatische Datenabhängigkeitsanalyse auf Basis deduktiver Verifizierung machbar ist und eine vielversprechende Alternative zu den im HPC üblicherweise verwendeten Abhängigkeitsanalysen darstellt. Wir implementierten unseren Ansatz für Java auf einem deduktiven Verifizierungstool und führten Evaluierungen durch, die seine Fähigkeit demonstrierten, hochpräzise Datenabhängigkeiten für repräsentativen Code zu analysieren, der aus HPC-Anwendungen extrahiert wurde.

Contents

Acknowledgments vii

Abstract ix

Zusammenfassung xi

1. Introduction 1
- 1.1. State of the Art . . . 2
- 1.2. Approach . . . 2
- 1.3. Contributions . . . 3
- 1.4. Overview of Publications . . . 4
- 1.5. Structure of The Thesis . . . 4

2. Preliminaries 7
- 2.1. Java Dynamic Logic . . . 7
  - 2.1.1. Syntax . . . 7
  - 2.1.2. Semantics . . . 11
  - 2.1.3. Reasoning . . . 12
  - 2.1.4. Update Application and Simplification Rules . . . 14
  - 2.1.5. Symbolic Execution Rules . . . 15
  - 2.1.6. Memory Locations and Heap . . . 17
  - 2.1.7. Soundness and Completeness of the Calculus . . . 19
- 2.2. KeY . . . 20
  - 2.2.1. Prover Core . . . 20
  - 2.2.2. Reasoning about Programs . . . 21
- 2.3. Data Dependence Analysis . . . 21
- 2.4. Loop Invariant Generation with Predicate Abstraction . . . 24
  - 2.4.1. Predicate Abstraction . . . 25
  - 2.4.2. Predicate Refinement . . . 26
  - 2.4.3. Loop Invariant . . . 27

3. Data Dependence-Aware Program Logic **29**
3.1. Semantics of Read and Write Memory Accesses . . . 30
3.2. Memory Access Updates . . . 31
3.3. Specification of Data Dependence Properties . . . 33
3.4. Reasoning about Data Dependence Properties . . . 34
3.4.1. Modified Calculus Rules . . . 34
3.4.2. Update Simplification Rules . . . 35
3.4.3. Axiomatization . . . 37

4. Automatic Loop Invariant Generation for Data Dependence Analysis **41**
4.1. Reconciling Predicate Abstraction and Symbolic Execution . . . 42
4.2. Data Dependence Loop Invariant Generation with Predicate Abstraction . . . 48
4.2.1. Predicate Abstraction . . . 49
4.2.2. Predicate Refinement . . . 51
4.3. Reasoning . . . 54
4.3.1. Verification of the Data Dependence Loop Invariant . . . 54
4.3.2. Renamed Memory Access Update Application . . . 54
4.3.3. Subsumption Relations . . . 54
4.3.4. Embedding Predicate Abstraction . . . 55

5. Automatic Loop Invariant Generation for Inter-Iteration Data Dependence Analysis **61**
5.1. Inter- vs. Intra-Iteration Loop Data Dependences . . . 62
5.2. Specification of Intra-Iteration Data Dependence Properties . . . 62
5.2.1. Syntax and Semantics . . . 62
5.3. Loop Invariant Generation . . . 68
5.3.1. Symbolic Execution . . . 68
5.3.2. Predicate Abstraction . . . 70
5.4. Reasoning . . . 71
5.4.1. Update Application and Simplification Rules . . . 71
5.4.2. Subsumption Relations . . . 73
5.4.3. Embedding Predicate Abstraction . . . 75

6. Nested Loop Invariant Generation **77**
6.1. Generation Algorithm . . . 78
6.1.1. Computation of the Inner Loop Invariant . . . 79
6.1.2. Using the Inner Loop Invariant . . . 80

6.2. Anonymization of Memory Access Updates . . . 81
6.2.1. Syntax and Semantics . . . 81
6.2.2. Update Application Rules . . . 82
6.3. Anonymization of a Sequence of Memory Access Updates . . . 83
6.3.1. Syntax and Semantics . . . 84
6.3.2. Update Application Rules . . . 86

7. Multi-Dimensional Arrays **89**
7.1. Syntax and Semantics . . . 89
7.2. Calculus Rules . . . 92
7.3. Proof Search Strategy . . . 96

8. Experimental Results **97**
8.1. Single Loops . . . 97
8.1.1. Test Cases . . . 98
8.1.2. Evaluation . . . 109
8.2. Nested Loops . . . 110
8.2.1. Test Cases . . . 112
8.2.2. Evaluation . . . 117
8.3. Threats to Validity . . . 117

9. Related Work **119**
9.1. Data Dependence Profilers . . . 119
9.1.1. Static . . . 119
9.1.2. Dynamic . . . 121
9.1.3. Hybrid . . . 121
9.2. Loop Invariant Generation . . . 122
9.3. Others . . . 123

10. Conclusion and Future Work **125**

Bibliography **129**

A. Semantics **139**
A.1. Semantics of Dependence Predicates . . . 139
A.2. Semantics of History Dependence Predicates . . . 139
A.3. Semantics of noWaR . . . 140
A.4. Semantics of Inter-Iteration History and Access Predicates . . . 141

B. Sequent Calculus Rules **143**
B.1. Axiomatization of Data Dependences . . . 143
B.1.1. Rules for noWaR . . . 143
B.1.2. Rules for noWaW . . . 144
B.1.3. Rules for noR . . . 144
B.1.4. Rules for noW . . . 145
B.2. Shift Write Rule . . . 145
B.3. Renamed Memory Access Update Application . . . 146
B.4. Subsumption Rules for History Data Dependence Predicates . . . 147
B.5. Relation of wPred with History Data Dependence Predicates . . . 148
B.5.1. Relation of wPred and noWHist . . . 148
B.5.2. Relation of wPred and noRHist . . . 148
B.5.3. Relation of wPred and noWaRHist . . . 149
B.5.4. Relation of wPred and noWaWHist . . . 150
B.6. Update Application on Inter-Iteration Data Dependence Predicates . . . 151

C. Proof of Sequent Calculus Rules **153**
C.1. Proof of Soundness and Completeness of writeAccessAppOnNoRaW . . . 153
C.2. Proof of Soundness and Completeness of readAccessAppOnNoRaW . . . 154
C.3. Proof of Theorem 4.1.1 . . . 156
C.4. Proof of Theorem 4.1.2 . . . 157
C.5. Proof of Soundness and Completeness of renamedReadAppOnNoRHist . . . 158
C.6. Proof of Theorem 5.3.1 . . . 159
C.7. Proof of Soundness and Completeness of matrixRangeMinusSingleton . . . 162

List of Figures

2.1. Grammar of Java$_{HPC}$ . . . 8
2.2. Grammar of JavaDL . . . 10
2.3. Excerpt of JavaDL semantics . . . 13
2.4. A selection of update application and simplification rules . . . 15
2.5. Example of update application . . . 16
2.6. Different data dependence types . . . 22
2.7. Child loops of Listing 2.3 . . . 23
2.8. Sign analysis domains. . . . 25

3.1. Excerpts of JavaDL$_{Dep}$ Semantics . . . 32
3.2. Selection of memory access update rules . . . 36
3.3. Axiomatization of data dependence predicates . . . 38
3.4. Example of a formal verification proof of a `noRaW` property . . . 40

4.1. Order between memory locations . . . 49
4.2. Order between data dependence predicates . . . 50
4.3. Selection of rules for renamed memory access update application . . . 55
4.4. Subsumption relations between history data dependence predicates . . . 55
4.5. Relation of access and history predicates . . . 56
4.6. Relation of `rPred` and `noRHist` rules . . . 56
4.7. Relation of `rPred` and `noWHist` rules . . . 57
4.8. Relation of `rPred` and `noRaWHist` rules . . . 59

5.1. Program semantics of JavaDL$_{\widehat{Dep}}$ . . . 64
5.2. Semantics of selected inter-iteration history predicates . . . 67
5.3. Predicate abstraction lattice for inter-iteration predicates . . . 71
5.4. Selected rules for application update on inter-iteration predicates . . . 72
5.5. Renamed update application on inter-iteration predicates . . . 74
5.6. Subsumption relations between inter-iteration predicates . . . 74

6.1. Anonymized memory access update application rules . . . 83

6.2. Anonymized sequence of memory access updates application 88

7.1. Two-dimensional rectangular shaped array 91
7.2. Different Java arrays 91
7.3. Matrix range minus singleton 94
7.4. Matrix range minus matrix range 95

8.1. Loop invariants for Listing 8.1 98
8.2. Loop invariants for Listing 8.2 99
8.3. Loop invariants for Listing 8.3 100
8.4. Loop invariants for Listing 8.4 101
8.5. Loop invariants for Listing 8.5 102
8.6. Loop invariants for Listing 8.6 103
8.7. Loop invariants for Listing 8.7 104
8.8. Loop invariants for Listing 8.8 105
8.9. Loop invariants for Listing 8.9 106
8.10. Loop invariants for Listing 8.10 107
8.11. Loop invariants for Listing 8.11 108
8.12. Loop invariants for Listing 8.12 110
8.13. Loop invariants for Listing 8.13 112
8.14. Loop invariants for Listing 8.14 113
8.15. Loop invariants for Listing 8.15 114
8.16. Loop invariants for Listing 8.16 115
8.17. Loop invariants for Listing 8.17 116
8.18. Loop invariants for Listing 8.18 116

List of Tables

8.1. Data dependence loop invariant generation for single loops 111
8.2. Inter-iteration data dependence loop invariant generation for single loops . 111
8.3. Data dependence loop invariant generation results for nested loops 117

List of Algorithms

1. Data Dependence Loop Invariant Generation Algorithm 52
2. Dependence Predicate Weakening Heuristics 53

3. Data Dependence Loop Invariant Generation Algorithm for Nested Loops . . 78

List of Definitions

Definition 2.1.1. Programs Prg . . . 8
Definition 2.1.2. Signature Σ . . . 9
Definition 2.1.3. Update . . . 9
Definition 2.1.4. Syntax of JavaDL . . . 10
Definition 2.1.5. Semantics of JavaDL . . . 11
Definition 2.1.6. Variable assignment . . . 12
Definition 2.1.7. Modification of state . . . 12
Definition 2.1.8. Evaluation Function *val* . . . 12
Definition 2.1.9. Validity . . . 12
Definition 2.1.10. Location . . . 17
Definition 2.1.11. Soundness and completeness of a rule . . . 19
Definition 2.3.1. Data dependence . . . 21
Definition 2.4.1. Abstract domain . . . 26
Definition 2.4.2. Partial order set . . . 26
Definition 2.4.3. Abstract language . . . 26

Definition 3.1.1. Domains, JavaDL$_{Dep}$ states . . . 30
Definition 3.2.1. Memory access update . . . 31
Definition 3.2.2. Sequential memory access updates . . . 31
Definition 3.2.3. Semantics of JavaDL$_{Dep}$ . . . 33
Definition 3.3.1. Data Dependence Predicate . . . 33

Definition 4.1.1. Shift state update rule schema [49] . . . 43
Definition 4.1.2. Renamed memory access update predicate . . . 44
Definition 4.1.3. Memory access predicate . . . 46
Definition 4.1.4. Rule schema shiftRead . . . 46
Definition 4.1.5. History data dependence predicate . . . 47
Definition 4.2.1. Abstract domain . . . 49
Definition 4.2.2. Data Dependence abstract language . . . 50
Definition 4.2.3. Data dependence loop invariant . . . 50

Definition 5.2.1. Domain, JavaDL$_{\widehat{Dep}}$ State 63
Definition 5.2.2. Projection 65
Definition 5.2.3. Inter-iteration data dependence predicates: semantics . . . 65
Definition 5.2.4. Marker update semantics 67

Definition 6.2.1. Sequence of anonymized memory accesses updates 81
Definition 6.2.2. Anonymized memory access update 81
Definition 6.2.3. Length of anonymized sequence of memory accesses updates 81
Definition 6.2.4. Renamed anonymized memory access update 82
Definition 6.3.1. Sequence of sequence of anonymized memory accesses updates 84
Definition 6.3.2. Extended projection 84
Definition 6.3.3. Anonymized sequence of sequences of memory access update 84
Definition 6.3.4. Length of anonymized sequence of sequence of memory accesses updates 85
Definition 6.3.5. Renamed anonymized sequence of sequences of memory access update 85

Definition 7.1.1. Infinite union 89
Definition 7.1.2. Matrix range 90
Definition 7.1.3. Well-formed matrix predicate 91

List of Theorems, Corollaries, and Propositions

Proposition 2.1.1. Soundness[32] 19
Proposition 2.1.2. Relative Completeness[32] 19

Theorem 3.4.1. Soundness and Completeness of JavaDL Calculus 39
Corollary 3.4.1. Soundness of JavaDL Calculus 39

Theorem 4.1.1. Soundness and completeness of dualityAccAndRenamedAcc 45
Theorem 4.1.2. Soundness and completeness of shiftRead 46

Theorem 5.3.1. Soundness and completeness of shiftNextUpdate 69
Theorem 5.4.1. Soundness and Completeness of JavaDL$_{\widehat{Dep}}$ Calculus 75
Corollary 5.4.1. Soundness of JavaDL$_{\widehat{Dep}}$ Calculus 75

1. Introduction

Widespread usage of multi-core processors calls for a need for parallel programs to benefit from this capacity to the full extent. As most legacy software are sequential programs, the need to identify parallelization opportunities in sequential programs is raised in the HPC community.

The process of transforming a sequential program to its parallel equivalent can be performed manually by developers who possess the domain knowledge, automatically with auto-parallelizers, or semi-automatically with the help of parallelization recommendation systems. Either way, the performance gained by parallelization is usually worthless (especially in safety-critical applications) if the result is incorrect.

Soundness of the program parallelization process depends on preserving the program semantics. To preserve the semantics, parallelization process must not violate the data dependences.[1,2] Therefore, one of the core steps in the parallelization process is identifying data dependences. Informally, two code statements depend on each other, whenever they access the exact memory location and at least one of them writes.

Identifying and even testing data dependences is undecidable as it generalizes the reachability problem. Restricting the domain and performing exact memory-based dependence analysis, it can be reduced to an NP-complete problem [1] and solved by approximation. State-of-the-art approaches approximate the solution differently but all suffer from over- and/or under-approximation. In addition, they lack a rigorous, formal definition of data dependences, which makes it impossible to formally argue about their correctness.

For the reasons stated, we see room for a more *precise static approach* to data dependence analysis with *verifiable* results.

[1] Not to be confused with much simpler *data flow* properties that are commonly analyzed by static checking.
[2] In the HPC community the term *dependence* (pl. *dependences*) is used rather than *dependency / dependencies*. We follow their convention.

1.1. State of the Art

State-of-the-art data dependence profilers are classified into static, dynamic, and hybrid approaches.

Static data dependence analysis approaches [1–15] profile data dependences based on the information available at compile time. They produce sound results but suffer from over-approximation. Since the value of pointers and array indices usually can not be resolved at compile time, these techniques are conservative and tend to over-approximates data dependences, which means that they can report data dependences that do not occur at run time. The presence of data dependences either cancels parallelization or declares the need for restructuring before parallelization. Therefore, over-approximating data dependences entails missing parallelization opportunities or expensive parallelization.

On the other hand, dynamic data dependence analysis approaches [16–22] rely on runtime information. Although they do not suffer from over-approximation, they have the major drawback of under-approximation. These analyses are optimistic as they rely on actual program runs. This can cause missing data dependences that do not belong to the paths traversed in the current execution, resulting in parallelization where it leads to incorrect results. In addition, these techniques have a high runtime overhead as the whole program must be executed several times under realistic load.

To benefit from the soundness of static approaches while reducing their over-approximation by using the runtime information, hybrid data dependence analysis approaches [23–26] emerged. They solved the soundness problem by limiting their application to a specific loop shape and function. Although they have reduced the over-approximation and high overhead problems compared to the dynamic approaches, their result is still unverified and can not guarantee sound parallelization.

1.2. Approach

The approach presented in this thesis makes automatic data dependence analysis based on deductive [27] verification of imperative programs with loops over arrays possible. The code to be analyzed, as well as data dependence relations, are formalized in a program logic. Automated deduction is used to conjecture and formally verify data dependences.

We define a program logic to model data dependence with formal semantics. The logic permits to specify and verify dependences with full precision for any *loop-free* and *non-recursive* program. The soundness of our approach is provable as it is based on a program logic. We use *predicate abstraction* [28, 29] to generate data dependence loop invariants with high precision. Due to path coverage and value sensitivity, full precision is

achievable in principle.

Deductive verification of functional properties requires human interaction and non-trivial formal specifications [27]. However, it can be a *fully automatic* technique if properties and specifications are either sufficiently generic and/or can be inferred automatically [30, 31]. We show that this applies to data dependence analysis, too. In addition, since deductive verification systems are procedure-modular [27], there is no need to analyze a whole program. Analysis can focus on the typically compact and computation-intensive parts.

In consequence, our approach establishes deductive verification as a new, viable alternative in the portfolio of dependence analysis tools.

1.3. Contributions

Our main *methodological contributions* are:

- defining formal semantics of data dependences;
- extending a program logic for deductive verification [32] to make formal verification of data dependences possible;
- an automatic and fully precise approach to verify data dependences in loop-free and non-recursive programs;
- an automatic and highly precise approach to express and verify data dependences in programs with loops;
- a predicate abstraction loop invariant generation approach that can synthesize sound data dependence loop invariant;
- distinguishing between data dependences within a loop iteration and across loop iterations; and
- the capability to verify data dependences conjectured by state-of-the-art approaches.

Our *technical contributions* include:

- the concept of data dependence-aware program logic; and
- combination of symbolic execution and predicate abstraction in a program logic and deductive framework that permits to verify and propagate data dependences.

1.4. Overview of Publications

I contributed to the following papers prior to finishing this thesis.

Publications Included in This Thesis

- *A Program Logic for Dependence Analysis* (IFM 2019) [33]: I contributed to this paper by writing the data dependence program logic, implementing and evaluating it. This publication is the base of Chapter 3.
- *Safer Parallelization* (ISoLA 2020) [34]: My contribution is writing the Restructuring for Parallelization section. The connection made between different parallelization patterns and data dependences in different chapters of this thesis is based on this publication.
- *Automatic Loop Invariant Generation for Data Dependence Analysis* (FormaliSE@ICSE 2022) [35]: I contributed to this paper by developing and implementing the loop invariant generation algorithm and the program logic. This publication is the base of Chapter 4.

Other Publications

Following publications are not included in my thesis. My contributions are evaluating the approach and researching the related work.

- *Modeling Non-deterministic C Code with Active Objects* (FSEN 2019) [36]
- *Automated model extraction: From non-deterministic C code to active objects* (SCP 2021) [37]

1.5. Structure of The Thesis

The remainder of this document is organized as follows:

Chapter 2: Preliminaries This chapter gives an overview of the four pillars that this thesis stands on: Java Dynamic Logic (JavaDL), KeY system, data dependence analysis, and loop invariant generation with predicate abstraction.

Chapter 3: Data Dependence-Aware Program Logic This chapter formally defines data dependences and provides a program logic for data dependence analysis of loop-free and non-recursive programs.

Chapter 4: Automatic Loop Invariant Generation for Data Dependence Analysis For analyzing loops, we need loop invariants. This chapter explains how we adapt different techniques to generate data dependence loop invariants.

Chapter 5: Automatic Loop Invariant Generation for Inter-Iteration Data Dependence Analysis We take the approach developed in the previous chapter a step further and focus on the data dependences spanning over different loop iterations.

Chapter 6: Nested Loop Invariant Generation To make our approach widely applicable, we extend the loop invariant generation approach introduced in the previous chapters to support nested loops.

Chapter 7: Multi-Dimensional Arrays As nested loops often iterate over multi-dimensional arrays, and such loops constitute the majority of HPC applications, we extend the underlying program logic with support for multi-dimensional arrays.

Chapter 8: Development in KeY and Experiments We implemented our approach in the KeY system. In this chapter, we go through different test cases to showcase the benefits of our approach over the state-of-the-art.

Chapter 9: Related Work This chapter classifies different research areas that tackle the same or similar problems as this thesis from various aspects.

Chapter 10: Conclusion and Future Work This chapter concludes the thesis and discusses further opportunities for expanding our approach.

2. Preliminaries

This chapter introduces the preliminary concepts needed for this thesis. We begin with the JavaDL program logic (Section 2.1) that is the program logic used in this thesis. JavaDL expresses properties about Java programs. We describe the calculus that is used to reason about the validity of these properties. We follow closely the definition of JavaDL as presented in [32]. Following chapters extend JavaDL to reason about data dependences. The verification framework KeY [32] that is based on JavaDL is introduced in Section 2.2. Approaches developed in this thesis are all realized by implementing them in KeY. In Section 2.3 the concept of data dependence is explained as it is the core focus for verification in this thesis. Section 2.4 explains the generation of loop invariants that is essential for analyzing data dependences in loops.

2.1. Java Dynamic Logic

Java dynamic logic (JavaDL) [32] is an instance of dynamic logic (DL) [38, 39] for sequential deterministic Java programs. Dynamic logic is an extension of first-order logic for reasoning about the behavior of programs.

In this section we give an overview on the syntax, semantics, and calculus of JavaDL. As well as addressing memory handling and formulation of soundness and completeness. The definitions below are adopted from [32] to which we refer for a full account.

2.1.1. Syntax

For ease of presentation we use the Java fragment given by the grammar in Figure 2.1. We call it Java$_{HPC}$. This fragment supports primitive types `int`, `boolean`, array type `int[]` for one-dimensional integer arrays of fixed length (`length`), and array type `int[][]` for two-dimensional integer arrays of fixed length and width. We assume that all rows have the same length (e.g. for `int[][] a`, width is `a[0].length`). We assume the domain of `int` to be integers $\mathbb{Z}$ and that variables of type `int[]` and `int[][]` refer to existing array objects. Note that Java$_{HPC}$ assignments are not expressions.

$\langle stmnt \rangle$ ::= $\langle lhs \rangle$ '=' $\langle exp \rangle$';' | $\langle stmnt \rangle$ $\langle stmnt \rangle$
| 'if' '(' $\langle exp \rangle$ ')' '{' $\langle stmnt \rangle$ '}' 'else' '{' $\langle stmnt \rangle$ '}'
| 'while' '(' $\langle exp \rangle$ ')' '{' $\langle stmnt \rangle$ '}'

$\langle exp \rangle$ ::= $\langle var \rangle$ | $\langle var \rangle$ '[' $\langle exp \rangle$ ']' | '(' $\langle exp \rangle$ ')' | $\langle aexp \rangle$ | $\langle bexp \rangle$

$\langle aexp \rangle$ ::= $\mathbb{Z}$ | $\langle var \rangle$'.length' | '-'$\langle exp \rangle$ | $\langle exp \rangle \circledast \langle exp \rangle$ ($\circledast \in$ {'+', '-', '*', '/'})

$\langle bexp \rangle$ ::= 'true' | 'false' | '!'$\langle exp \rangle$ | $\langle exp \rangle \circledast \langle exp \rangle$ ($\circledast \in$ { '&&', '||' })
| $\langle exp \rangle \circledast \langle exp \rangle$ ($\circledast \in$ {'<', '<=', '==', '>=', '>'})

$\langle lhs \rangle$ ::= $\langle var \rangle$ (not of array type) | $\langle var \rangle$ '[' $\langle exp \rangle$ ']'

$\langle var \rangle$ ::= x $x \in$ PV

Figure 2.1.: Grammar of Java$_{HPC}$ (a fragment of the Java programming language)

We define Java$_{HPC}$ for a streamlined presentation. The implementation covers full Java, includes aliasing, array creation, and etc.

JavaDL addresses aliasing from two different aspects. For example, whether `int[] a` and `int[] b` are aliases of each other, and whether `a[i]` and `a[j]` are aliases of each other.

PV denotes the set of all program variables with infinitely many variables for each type. Variables of type `int[]` cannot occur on the left side of an assignment, except as part of an array access and different variables of type `int[]` denote different array objects. We omit the straightforward typing rules.

```
i = 0;
while (i < a.length - 1) {
    a[i] = a[i+1];
    i=i+1;
}
```

Listing 2.1: Array shift program

Example 2.1.1. An example program in Java$_{HPC}$ with program variables `i` and `a` of types `int` and `int[]`, respectively, is in Listing 2.1. The program shifts the content of array `a` to the left by one.

Definition 2.1.1 (Programs Prg)**.** The set of all sequences of deterministic executable Java$_{HPC}$ statements.

JavaDL extends first-order logic with two modalities over programs $\langle\cdot\rangle\cdot$ ("diamond"), $[\cdot]\cdot$ ("box"), and a syntactic category called updates. A modality takes a $\texttt{prg} \in Prg$, and a JavaDL formula ϕ (possibly also containing modalities) as arguments. Formula $\psi \rightarrow [\texttt{prg}]\phi$ is equivalent to the Hoare triple [40] $\{\psi\}$ `prg` $\{\phi\}$ in case ψ, and ϕ are first-order formulas. If `prg` is executed in any state for which formula ψ holds, and *if* `prg` terminates then formula ϕ holds in the final state. This corresponds to partial correctness. On the other hand, by using the diamond modality the sequence of statements `prg`, *has to* terminate and in the final state ϕ must hold for the formula to be valid. This is called total correctness.

A type hierarchy is extracted from the Java$_{HPC}$ program under verification. In a *type hierarchy* $\mathcal{T} = (\mathsf{Sort}, \preceq)$, Sort is a set of type names with reflexive and transitive subtype relation $\preceq$: Sort $\times$ Sort. The set of types contains at least (i) $\top$, `Any`, `Heap`, `Field`, `LocSet`, `boolean`, `int`, where $\top$ is the unique root of $\mathcal{T}$, `Any` is the supertype of all types except `Heap` and `Field`; (ii) any Java$_{HPC}$ type declared or used in the program under verification (closed w.r.t. supertypes).

Definition 2.1.2 (Signature Σ)**.** A $\mathcal{T}$-typed JavaDL signature $\Sigma^{\mathcal{T}} = (\text{PSym}, \text{FSym}, \text{VSym})$ consists of non-empty sets:

- PSym of predicate symbols $p : T_1 \times \ldots \times T_n, \;\; T_i \in \mathcal{T}$
- FSym of function symbols $f : T_1 \times \ldots \times T_n \rightarrow S, \;\; T_i, S \in \mathcal{T}$
- VSym of first-order variables $v : T, \;\; T \in \mathcal{T}$

Predicate symbols of arity 0 are *propositional variables* and function symbols of arity 0 are *constants*. Predicate and function symbols are partitioned into rigid PSym$_r$/FSym$_r$ and non-rigid PSym$_{nr}$/FSym$_{nr}$ symbols. Interpretation of non-rigid symbols depends on the state, thus they capture side effects of program execution. Logic (first-order) variables are always rigid. JavaDL is closed under all first-order operators, quantifiers, and modalities. The set of program variables PV is the set of all non-rigid constants, which are disjoint from rigid logic (first-order) variables. The latter can be bound by quantifiers and may not occur in programs. Program variables may occur in programs and first-order terms, but cannot be quantified over.

JavaDL includes a syntactic category named *updates*, which represent state changes.

Definition 2.1.3 (Update)**.** An *elementary update* is written as $l := r$ with $l : T \in \mathsf{PV}$ and r a term of type T' with $T' \preceq T$. The meaning is that of an assignment, where the value of term r is assigned to program variable l. An update u can be applied to a term $\{u\}t$, to a formula $\{u\}\phi$ or it can be composed with another update u' into a *sequential update* $u; u'$

$$
\begin{array}{lll}
\langle \mathit{term} \rangle & ::= & v \quad (v \in \text{VSym}) \\
& | & f\ [\text{`('}\ \langle \mathit{term} \rangle \text{`,'}\ \ldots \text{`,'}\ \langle \mathit{term} \rangle\ \text{`)'}]_{opt} \quad (f \in \text{FSym}) \\
& | & \text{`\{'} \langle \mathit{update} \rangle \text{`\}'} \langle \mathit{term} \rangle \\
\\
\langle \mathit{fml} \rangle & ::= & \text{tt} \mid \text{ff} \mid p\ [\text{`('}\ \langle \mathit{term} \rangle \text{`,'}\ \ldots \text{`,'}\ \langle \mathit{term} \rangle\ \text{`)'}]_{opt} \quad (p \in \text{PSym}) \\
& | & \langle \mathit{term} \rangle \doteq \langle \mathit{term} \rangle \qquad \text{(equality)} \\
& | & \neg \langle \mathit{fml} \rangle \mid \langle \mathit{fml} \rangle \circledast \langle \mathit{fml} \rangle\ (\circledast \in \{\wedge, \vee, \rightarrow\}) \qquad \text{(propositional connectives)} \\
& | & \mathcal{Q} v\ \text{`.'}\ \langle \mathit{fml} \rangle \quad (\mathcal{Q} \in \{\exists, \forall\},\ v \in \text{VSym}) \\
& | & \text{`[']} \langle \mathit{stmnt} \rangle \text{`]'} \langle \mathit{fml} \rangle \mid \text{`\{'} \langle \mathit{update} \rangle \text{`\}'} \langle \mathit{fml} \rangle \\
\\
\langle \mathit{update} \rangle & ::= & x\ \text{`:='}\ \langle \mathit{term} \rangle \quad (x \in \text{PV}) \qquad \text{(elementary update)} \\
& | & \langle \mathit{update} \rangle\ \text{`||'}\ \langle \mathit{update} \rangle \qquad \text{(parallel update)} \\
& | & \langle \mathit{update} \rangle\ \text{`;'}\ \langle \mathit{update} \rangle \qquad \text{(sequential update)}
\end{array}
$$

Figure 2.2.: Grammar of JavaDL

or a *parallel update* $u \| u'$. Parallel updates are applied simultaneously, i.e., they do not influence each other; in case a variable occurs more than once on the left-hand side of a parallel update, the syntactically last update wins. Set of all updates defined on signature Σ is Upd^{Σ}.

We give examples of JavaDL formulas and their intuitive meaning:

Example 2.1.2. Let $\mathtt{i}$, $\mathtt{j}$ be program variables, $\mathtt{i}_0$, $\mathtt{j}_0$ *rigid* constants of type `int`.

1. Formula $(\mathtt{i} \doteq \mathtt{i}_0 \wedge \mathtt{j} \doteq \mathtt{j}_0) \rightarrow \langle \texttt{i=i-j; j=i+j; i=j-i;} \rangle (\mathtt{i} \doteq \mathtt{j}_0 \wedge \mathtt{j} \doteq \mathtt{i}_0)$ means: if the program in the diamond is executed in an initial state, where $\mathtt{i}$, and $\mathtt{j}$ have the values $\mathtt{i}_0$, and $\mathtt{j}_0$, then the program terminates and in its final state the program variables have their initial value swapped.

2. Formula $\{\mathtt{i} := \mathtt{i} + 1\}(\mathtt{i} \geq 0)$ is equivalent to formula $\mathtt{i} \geq 0$ except that the value of $\mathtt{i}$ is increased by one.

3. Formula $(\mathtt{i} \doteq \mathtt{i}_0 \wedge \mathtt{j} \doteq \mathtt{j}_0) \rightarrow \{\mathtt{j} := \mathtt{i} \| \mathtt{i} := \mathtt{j}\}(\mathtt{i} \doteq \mathtt{j}_0 \wedge \mathtt{j} \doteq \mathtt{i}_0)$ means the same as the formula in 1., but expresses the effect of the program with a parallel update. Parallel updates are executed simultaneously, this means parallel updates do not influence each other.

Definition 2.1.4 (Syntax of JavaDL)**.** Syntax of JavaDL includes, terms, formulas, and updates. Terms and formulas are defined inductively as in standard typed first-order logic and are given in Figure 2.2[1]. We list only the non-standard cases:

- If ϕ is a JavaDL formula and `prg` a legal program fragment (sequence of statements) then $\langle \texttt{prg} \rangle \phi$ and $[\texttt{prg}]\phi$ are JavaDL formulas.
- If ϕ is a JavaDL formula, t a term of type T and u an update (elementary or parallel), then $\{u\}\phi$ is a JavaDL formula and $\{u\}t$ is a term of type T. We say "u is applied to ϕ (to t)".

2.1.2. Semantics

Classical first-order logic, evaluates a formula (or term) regarding to one interpretation (or model) that gives meaning to rigid symbols. JavaDL, like other modal logics, expresses properties relating execution states, and has a big-step Kripke semantics. Each state s in the set of states $\mathcal{S}$ of a given program can be seen as an interpretation of symbols, specifically, of program variables.

Definition 2.1.5 (Semantics of JavaDL)**.** Given a signature $\Sigma_{\mathcal{T}}$ for type hierarchy $\mathcal{T}$. JavaDL semantics is defined over a *Kripke structure* $\mathcal{K} = (\mathcal{D}, \mathcal{S}, [\![\cdot]\!])$, consisting of a non-empty domain $\mathcal{D}$, a set of states $\mathcal{S}$, and state transition relation $[\![\cdot]\!]$. The set of states $\mathcal{S}$ is infinite and non-empty. State $s \in \mathcal{S}$ assigns meaning to (*rigid* and *non-rigid*) symbols.

- s assigns to each type $T \in \mathcal{T}$ its domain $s(T) = \mathcal{D}^T$ respecting the subtype relation, i.e. $\mathcal{D}^T \subseteq \mathcal{D}^{T'}$ for $T \preceq T'$.
- s assigns to each function symbol $f : T_1 \times \ldots \times T_n \to T$ a function $s(f) : \mathcal{D}^{T_1} \times \ldots \times \mathcal{D}^{T_n} \to \mathcal{D}^T$.
- s assigns to each predicate symbol $p : T_1 \times \ldots \times T_n$ a relation $s(p) : \mathcal{D}^{T_1} \times \ldots \times \mathcal{D}^{T_n}$.
- All $s \in \mathcal{S}$ coincide on their domain and the interpretation of rigid functions and predicates.
- A state transition relation $[\![\cdot]\!] : Prg \times \mathcal{S} \times \mathcal{S}$ that associates each program `prg` with all pairs of states (s, s') such that if `prg` is executed in s then it terminates in s' in accordance to Java's program semantics as defined in the Java Language Specification (JLS) [41].

[1] Terms, formulas, and updates need to be well-typed.

- As JavaDL is deterministic, there is at most one tuple $(s, s') \in [\![\texttt{prg}]\!]$ for each program `prg` and state $s \in \mathcal{S}$.

Definition 2.1.6 (Variable assignment)**.** A *variable assignment* $\beta : \text{VSym} \to \mathcal{D}$ maps each first-order variable v of type T to an element of its domain $\mathcal{D}^T$.

Definition 2.1.7 (Modification of state)**.** We write $s[x \leftarrow d]$ for the state s' coinciding with s, except for the value of program variable x of type T, which is evaluated to $d \in \mathcal{D}^T$.

Definition 2.1.8 (Evaluation Function *val*)**.** Let $\mathcal{K}$ be a Kripke structure, s a state, β a variable assignment. Let f be a function symbol, t, t_i ($i \in \mathbb{N}$) terms, and x, $y \in \mathsf{PV}$. The *evaluation function* $val_{\mathcal{K},s,\beta}$ maps terms to values of their domain, and updates to a pair of states. It is defined in Figure 2.3. The semantics of parallel updates implies that when the same program variable is assigned more than once within a parallel update, then only the textually last assignment is relevant.

The semantics of assignments with side effect-free right hand sides is identical to that of elementary updates. This becomes important in the calculus where updates are used to represent the effect of assignments.

Definition 2.1.9 (Validity)**.** $\mathcal{K}$ is a Kripke structure, s a state, β a variable assignment. Let p be a predicate symbol. We define the validity relation $\models$ for formulas in Figure 2.3 Write $\mathcal{K} \models \phi$ iff $\mathcal{K}, s, \beta \models \phi$ holds for all $s \in \mathcal{S}$ and all β. Formula ϕ is *valid*, written $\models \phi$, iff $\mathcal{K} \models \phi$ holds for all Kripke structures $\mathcal{K}$.

The evaluation function $val_{\mathcal{K},s,\beta} : \mathit{Prg} \to 2^{\mathcal{S}}$ maps programs `prg` to sets of states. Programs are deterministic, so the result is either the empty set, when `prg` does not terminate if started in s, otherwise, a singleton set with the final state reached by `prg` started in s. The semantics for diamond (box) means that after executing `prg`, there exists a state (for all states) property ϕ holds, which is equivalent to total (partial) correctness for deterministic programming languages.

2.1.3. Reasoning

Reasoning about validity of formulas is done using sequent calculus that follows the symbolic execution [42–45] paradigm to handle formulas containing programs. A *sequent* is an abstract data structure with schema $\phi_1, \ldots, \phi_n \Longrightarrow \psi_1, \ldots, \psi_m$ consisting of two sets of formulas and has the same meaning as the formula $\bigwedge_{i=1,\ldots,n} \phi_i \to \bigvee_{i=1,\ldots,m} \psi_i$.

Update

$$
\begin{array}{lll}
val_{\mathcal{K},s,\beta}(x := t) & = s' & \text{with } s' = s[x \leftarrow val_{\mathcal{K},s,\beta}(t)] \\
val_{\mathcal{K},s,\beta}(u_1 || u_2) & = val_{\mathcal{K},s,\beta}(u_2)(val_{\mathcal{K},s,\beta}(u_1)) & \\
val_{\mathcal{K},s,\beta}(u_1; u_2) & = s'' & \text{with } s'' = val_{\mathcal{K},s',\beta}(u_1) \text{ and } s' = val_{\mathcal{K},s,\beta}(u_2) \\
val_{\mathcal{K},s,\beta}(\{u_1\}u_2) & = val_{\mathcal{K},s',\beta}(u_2) & \text{with } val_{\mathcal{K},s,\beta}(u_1) = s'
\end{array}
$$

Terms

$$
\begin{array}{lll}
val_{\mathcal{K},s,\beta}(v) & = \beta(v) & \text{for a first-order variable } v \in \text{VSym} \\
val_{\mathcal{K},s,\beta}(x) & = s(x) & x \in \mathsf{PV} \\
val_{\mathcal{K},s,\beta}(f(t_1,\ldots,t_n)) & = s(f)(val_{\mathcal{K},s,\beta}(t_1),\ldots,val_{\mathcal{K},s,\beta}(t_n)) & \\
val_{\mathcal{K},s,\beta}(\{u\}t) & = val_{\mathcal{K},s',\beta}(t) & \text{with } val_{\mathcal{K},s,\beta}(u) = s'
\end{array}
$$

Formulas

$$
\begin{array}{lll}
\mathcal{K},s,\beta \models q(t_1,\ldots,t_n) & \text{iff } (val_{\mathcal{K},s,\beta}(t_1),\ldots,val_{\mathcal{K},s,\beta}(t_n)) \in s(q) & \\
\mathcal{K},s,\beta \models \neg\phi & \text{iff } \mathcal{K},s,\beta \not\models \phi & \\
\mathcal{K},s,\beta \models \phi \wedge \psi & \text{iff } \mathcal{K},s,\beta \models \phi \text{ and } \mathcal{K},s,\beta \models \psi & (\text{similarly } \vee, \rightarrow, \ldots) \\
\mathcal{K},s,\beta \models \forall x;\phi & \text{iff for all } d \in D^T : \mathcal{K},s,\beta^d_x \models \phi & (\text{similarly } \exists x;\phi) \\
\mathcal{K},s,\beta \models \langle \texttt{prg} \rangle \phi & \text{iff } val_{\mathcal{K},s,\beta}(\texttt{prg}) = \{(s,s')\} \text{ and } \mathcal{K},s',\beta \models \phi & \\
\mathcal{K},s,\beta \models [\texttt{prg}]\phi & \text{iff } val_{\mathcal{K},s,\beta}(\texttt{prg}) \subseteq \{(s,s')\} \text{ and } \mathcal{K},s',\beta \models \phi & \\
\mathcal{K},s,\beta \models \{u\}\phi & \text{iff } \mathcal{K},s',\beta \models \phi \text{ with } val_{\mathcal{K},s,\beta}(u) = s' &
\end{array}
$$

Programs

$$
\begin{array}{ll}
val_{\mathcal{K},s,\beta}(stmnt_1; stmnt_2) & := \{(s,s'') |\ (s,s') \in [\![stmnt_1]\!] \text{ and } (s',s'') \in [\![stmnt_2]\!]\} \\
val_{\mathcal{K},s,\beta}(x = t) & := \{(s,s') \mid s' = s[x \leftarrow val_{\mathcal{K},s,\beta}(t)] \\
val_{\mathcal{K},s,\beta}(\texttt{if}(b)\ \{stmnt_1\}\ \texttt{else}\ \{stmnt_2\}) & \\
\qquad := \{(s,s') \mid (\mathcal{K},s,\beta \models b \text{ and } (s,s') \in [\![stmnt_1]\!]) & \\
\qquad\quad \text{or } (\mathcal{K},s,\beta \not\models b \text{ and } (s,s') \in [\![stmnt_2]\!])\} & \\
val_{\mathcal{K},s,\beta}(\texttt{while}(b)\{stmnt\}) := \{(s,t) & \mid \text{there is a sequence } s = s_0 \ldots s_n = t \text{ with} \\
& \quad \mathcal{K},s_i,\beta \models b,\ (s_i,s_{i+1}) \in [\![stmnt]\!],\ i < n \\
& \quad \text{and } \mathcal{K},t,\beta \not\models b\}
\end{array}
$$

Figure 2.3.: Excerpt of JavaDL semantics

Formulas $\phi_1, \ldots, \phi_n$ are called *antecedent* and $\psi_1, \ldots \psi_m$ *succedent*. A sequent calculus *rule schema* has the form

$$\mathsf{ruleName}\ \frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \quad \cdots \quad \Gamma_k \Longrightarrow \Delta_k}^{\text{premises}}}{\underbrace{\Gamma \Longrightarrow \Delta}_{\text{conclusion}}}\ \text{cond}$$

where Γ, Γ_i, Δ, Δ_i are schematic variables matching formula sets, and $cond$ is a decidable side condition. Sequent calculus rules are applied bottom-to-top to construct a proof.

A *sequent calculus proof* for the validity of a formula ϕ is a tree (i) whose nodes are labeled with a sequent and (ii) for each inner node n with children $n_1, \ldots, n_k$ there is a rule r with k premises such that r's conclusion matches the sequent of n and the sequent at child n_k is equal to the instantiated i-th premise of r. A proof is *closed* if at all leaves an axiom rule was applied, i.e., a rule without premises ($k = 0$).

For rewrite rules we use the following notion

$$lhs \rightsquigarrow rhs$$

where lhs and rhs are both either schematic terms, or schematic formulas, or schematic updates. Rewrite rules are sound if for all Kripke structure $\mathcal{K}$, state s, and variable assignent β evaluation of lhs is equal to evaluation of rhs. An example for a rewrite rule is $t \doteq t \rightsquigarrow$ true.

JavaDL calculus rules can be categorized into three groups. The first group is the first-order logic rules. As dynamic logic is an extension of first-order logic, all the first-order logic rules hold in dynamic logic, too. Update application and simplification rules (Figure 2.4) form the second group, which realizes the effect of updates on other updates, formulas, and terms. The third group is the Symbolic Execution rules for dealing with programs. We skip the first-order logic rules and briefly explain the other two groups.

2.1.4. Update Application and Simplification Rules

A selection of update application and simplification rules is shown in Figure 2.4. Application of an update on a formula results in propagation of update to the subterms below the (rigid) operator. Ultimately, the update can either be simplified away, or it is applied to the target program variable. In applying an update to a modal operator, the program prg must first be eliminated using the Symbolic Execution rules. Only afterwards can the resulting update be applied on ϕ.

Schematic variables

x, y: program variables, t, $t_1, \ldots, t_n$: terms,
ϕ: formulas, p: rigid predicate,
Γ, Δ: context formulas, u, u_1, u_2: updates,

Update-on-Term

$$\{x := t\}x \rightsquigarrow t \qquad\qquad \{x := t\}y \rightsquigarrow y, \text{ for } y \neq x$$

Update-on-Formula

$$\{u\}p(t_1, \ldots, t_n) \rightsquigarrow p(\{u\}t_1, \ldots, \{u\}t_n) \qquad \{u\}(\phi \circ \psi) \rightsquigarrow (\{u\}\phi) \circ (\{u\}\psi)$$
$$\circ \in \{\wedge, \vee, \rightarrow, \ldots\}$$

Update-on-Update

$$\{u\}(x := t) \rightsquigarrow x := \{u\}t \qquad\qquad \{u_1\}\{u_2\}\phi \rightsquigarrow \{u_1 \| \{u_1\}u_2\}\phi$$

Figure 2.4.: A selection of update application and simplification rules

Application of an update on an elementary update is propagated to the term in the right hand side of the update. The left hand side of an elementary update (here x) does not refer to a value, but refers to a location that its value is stored. Therefore, it can not be changed by an update. A cascade of two update applications $\{u_1\}\{u_2\}$ is converted into the application of a single parallel update. Due to the last-win semantics for parallel updates, this is possible by applying the first update to the second, and replacing the sequential composition by parallel composition.

2.1.5. Symbolic Execution Rules

We explain a few rule schemata that deal with programs to explain how the calculus uses Symbolic Execution to eliminate programs and thus reducing a sequent to a pure first-order sequent.

The calculus decomposes complex statements into simpler ones until the first active statement can be atomically executed, e.g., an assignment of a side effect-free expression to a local variable. Rule assignmentLocalVariable symbolically executes such an assignment by representing it in the logic as an *update*. The sequent rule schema

$$\text{assignmentLocalVariable}\ \frac{\Gamma \Longrightarrow \{u\}\{x := t\}[r]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\texttt{x=t;}\ \ r]\phi, \Delta}$$

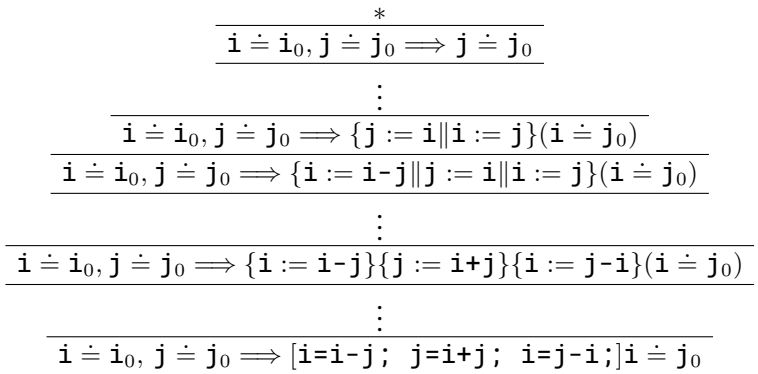


Figure 2.5.: Example of a application of rewrite rules for turning a sequence of update applications into one parallel update

moves an assignment into an update where x is a schema variable matching a local program variable, t a side-effect free expression (without memory accesses) of compatible type and u, r and ϕ are schema variables matching an update, a statement and a formula, respectively. Γ and Δ match sets of formulas. For update simplification, there exist rules that allow any sequence of update applications to be rewritten into a single parallel update.

Example 2.1.3. Figure 2.5 shows a sequent calculus proof for the following sequent

$$\mathtt{i} \doteq \mathtt{i}_0,\ \mathtt{j} \doteq \mathtt{j}_0 \Longrightarrow [\mathtt{i=i-j;\ j=i+j;\ i=j-i;}]\mathtt{i} \doteq \mathtt{j}_0.$$

After three times of applying the rule assignmentLocalVariable assignments inside the program are turned to a sequence of update applications. Applying the rules Update-on-Update and Update-on-Term (Figure 2.4) we end up with a parallel update consisting of three elementary updates. The first and last update are updating i, therefore the first one is discarded. In the following proof we show the use of rewrite rules for turning a sequence of update applications into one parallel update. After applying the rules Update-on-Formula and Update-on-Term (Figure 2.4) we end up with the formula $\mathtt{j} \doteq \mathtt{j}_0$ on the right side. According to antecedent the formula holds and the final sequent can be discharged with close.

Rule conditional symbolically executes a conditional statement. Schema variable b matches a local program variable of type `boolean`. As b is symbolic and is evaluated to either true or false depending on the state, the calculus must consider both possibilities.

The rule splits the proof into two branches, one where it assumes b to be true and the then -branch is executed, the other where b is assumed to be false and Symbolic Execution continues with the else -branch.

$$\text{conditional}\ \frac{\Gamma, \{u\}b \Longrightarrow [stmnt_1\ \texttt{;}\ r]\phi, \Delta \quad \Gamma, \{u\}\neg b \Longrightarrow [stmnt_2\ \texttt{;}\ r]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\texttt{if (}b\texttt{) \{}stmnt_1\texttt{\} else \{}stmnt_2\texttt{\}}\ r]\phi, \Delta}$$

The two following rule schemata permit reasoning about loops.

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \{u\}[\texttt{if (}b\texttt{) \{}stmnt\texttt{; while (}b\texttt{) \{}stmnt\texttt{\}\}}\ r\texttt{;}]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\texttt{while (}b\texttt{) \{}stmnt\texttt{\}}\ r\texttt{;}]\phi, \Delta}$$

Rule schema unwindLoop does not terminate when the loop has no fixed bound on the number of iterations. Hence, in program verification the concept of a *loop invariant* is used to describe the behavior of a loop:

$$\text{loop\_inv}\ \frac{\begin{array}{c}\Gamma \Longrightarrow \{u\}\mathit{LoopInv}, \Delta \\ \Gamma, \{u\}\{v\}(b \wedge \mathit{LoopInv}) \Longrightarrow \{u\}\{v\}[stmnt]\mathit{LoopInv}, \Delta \\ \Gamma, \{u\}\{v\}(\neg b \wedge \mathit{LoopInv}) \Longrightarrow \{u\}\{v\}[r]\phi, \Delta\end{array}}{\Gamma \Longrightarrow \{u\}[\texttt{while (}b\texttt{) \{}stmnt\texttt{\}}\ r\texttt{;}]\phi, \Delta}$$

The loop invariant rule splits the proof into three branches where we have to prove that the loop invariant 1. holds initially, 2. is preserved by the loop body, and, 3. together with the negated loop guard, is sufficiently strong to prove that after executing the remaining program r the postcondition ϕ holds.

The second and third branch execute the loop body for an *arbitrary* iteration and the remaining program after the loop, respectively. Hence, we must "forget" any knowledge about current values embodied in update u and context Γ, Δ, because it might have changed in previous iterations. This is realized by a so-called *anonymizing update* v, which assigns an unknown value to each local program variable and location that possibly has been changed.

2.1.6. Memory Locations and Heap

A JavaDL-specific domain is $\mathcal{D}^{\texttt{LocSet}} = 2^{\texttt{Location}}$ for type `LocSet`, representing the power set of *locations*. Locations are used to abstract away from addresses.

Definition 2.1.10 (Location)**.** The pair (o, f), with term $o \in \mathcal{D}^{\texttt{Object}}$ and term $f \in \mathcal{D}^{\texttt{Field}}$ where type `Field` represents an object's field, is a memory location.

Type `LocSet` models memory regions as location sets. The `LocSet` data type has a constructor for singleton sets of memory locations $\texttt{singleton}(obj, fld)$. The set of memory locations for an array element is $\texttt{singleton}(a, \texttt{arr}(i))$ with $i \in \mathcal{D}^{\texttt{int}}$, and $a \in \mathcal{D}^{T[\,]}$, where injective function $\texttt{arr} : \texttt{int} \to \texttt{Field}$ and $i \in \mathcal{D}^{\texttt{int}}$. Intuitively, location $\texttt{singleton}(a, \texttt{arr}(i))$ for array element $a[i]$ is the memory location where its value is stored. Set of memory locations associated with a sub-array is shown as $\texttt{arrayRange}(a, l, h)$ with $l, h \in \mathcal{D}^{\texttt{int}}$, and $a \in \mathcal{D}^{T[\,]}$. For pretty-printing we use $\underline{a[l..h]}$ to express the set of locations $\{(a, i) \mid l \leq i \leq h\}$, and $\underline{a[i]}$ to express $\texttt{singleton}(a, \overline{\texttt{arr}(i)})$.

There are fixed interpreted function symbols on `LocSet` for standard operations `union`, `intersect`, `setMinus`, etc. Program variables are not modeled as memory locations, as they are not on the heap. The set `allLocs` denotes all possible locations.

Canonical domains are fixed for the primitive types, for instance, type `int` is mapped in all states to $\mathbb{Z}$. Interpretation of rigid function symbols representing arithmetic operations like $+$ is fixed to their canonical semantics, similar for comparison predicates like $\leq, \geq$.

The heap is modeled by type `Heap` which is axiomatized as a theory of arrays with functions `select` and `store`. Function $\texttt{select}(h, o, f)$ looks up the value stored in heap h for field f of object o, while $\texttt{store}(h, o, f, x)$ updates in heap h the value stored in field f of object o with value v. Programs access and modify the heap stored in the global program variable $\texttt{heap} : \texttt{Heap}$. Semantically, an element h of $\mathcal{D}^{\texttt{Heap}}$ is a function $h : \mathcal{D}^{\texttt{Object}} \times \mathcal{D}^{\texttt{Field}} \to \mathcal{D}^{\texttt{Any}}$, mapping memory locations to values. Let s be a state that maps program variable $o : \texttt{Object}$ to some non-null value, x a program variable:

$$val_{\mathcal{K},s,\beta}(o.f = x;) = s' \text{ with} \tag{2.1}$$

$$s'(y) = \begin{cases} s'(\texttt{heap})(u,g) = \begin{cases} s(x)\,, \text{ if } u = s(o) \text{ and } g = s(f) \\ s(\texttt{heap})(u,g)\,, \text{ otherwise} \end{cases} & , y = \texttt{heap} \\ s(y) & , \text{otherwise} \end{cases}$$

The calculus rules for reading from and writing to an object field are:

readAttribute

$$\frac{\Gamma \Longrightarrow o \neq \texttt{null}, \Delta \quad \Gamma, o \neq \texttt{null} \Longrightarrow \{x := \texttt{select}(\texttt{heap}, o, f)\}[r]\phi, \Delta}{\Gamma \Longrightarrow [x = o.f;\, r]\phi, \Delta}$$

writeAttribute

$$\frac{\Gamma \Longrightarrow o \neq \texttt{null}, \Delta \quad \Gamma, o \neq \texttt{null} \Longrightarrow \{\texttt{heap} := \texttt{store}(\texttt{heap}, o, f, se)\}[r]\phi, \Delta}{\Gamma \Longrightarrow [\,o.f = se;\, r]\phi, \Delta}$$

Similarly, the calculus rules for reading from and writing to an array element are:

readArrayElement

$$\frac{\Gamma \Longrightarrow a \neq \texttt{null}, \Delta \quad \Gamma, a \neq \texttt{null} \Longrightarrow \{x := \texttt{select}(\texttt{heap}, a, \texttt{arr}(i))\}[r]\phi, \Delta}{\Gamma \Longrightarrow [x = a[i]; r]\phi, \Delta}$$

writeArrayElement

$$\frac{\Gamma \Longrightarrow a \neq \texttt{null}, \Delta \quad \Gamma, a \neq \texttt{null} \Longrightarrow \{\texttt{heap} := \texttt{store}(\texttt{heap}, a, \texttt{arr}(i), se)\}[r]\phi, \Delta}{\Gamma \Longrightarrow [a[i] = se; r]\phi, \Delta}$$

2.1.7. Soundness and Completeness of the Calculus

The most important property of the JavaDL calculus as any other validity calculus is *soundness*. This property makes only valid formulas derivable. The whole calculus is sound if and only if all its rules are sound.

Proposition 2.1.1 (Soundness[32])**.** If a sequent $\Gamma \Longrightarrow \Delta$ is derivable in the JavaDL calculus, then it is valid, i.e., the formula $\bigwedge_{\phi \in \Gamma} \phi \Longrightarrow \bigvee_{\psi \in \Delta} \psi$ is logically valid.

Intuitively, if the premisses of a rule application are valid sequents, then the conclusion also is valid.

Another property of a program verification calculus is *completeness*, which means all valid sequents should be derivable. However, this is impossible because JavaDL includes first-order arithmetic, which is already inherently incomplete [46]. In addition to this argument, a complete calculus for JavaDL would yield a decision procedure for the Halting Problem, which is undecidable. Thus, a logic like JavaDL cannot ever have a calculus that is both sound and complete. Nevertheless, it is possible to define a notion of relative completeness [47], which intuitively states that the calculus is complete *up to* the inherent incompleteness in its first-order part. A relatively complete calculus contains all the rules that are necessary to prove valid program properties. It only may fail to prove such valid formulas whose proof would require the derivation of a nonprovable first-order property.

Proposition 2.1.2 (Relative Completeness[32])**.** If a sequent $\Gamma \Longrightarrow \Delta$ is valid, i.e., the formula $\bigwedge_{\phi \in \Gamma} \phi \Longrightarrow \bigvee_{\psi \in \Delta} \psi$ is logically valid, then there is a finite set Γ of logically valid first-order formulas such that the sequent $\Gamma \Longrightarrow \Delta$ is derivable in the JavaDL calculus.

Definition 2.1.11 (Soundness and completeness of a rule)**.** A rule

$$\frac{\Gamma_1 \Longrightarrow \Delta_1 \qquad \Gamma_2 \Longrightarrow \Delta_2}{\Gamma \Longrightarrow \Delta}$$

of a sequent calculus is:

- *sound*, if whenever $\Gamma_1 \Longrightarrow \Delta_1$ and $\Gamma_2 \Longrightarrow \Delta_2$ are valid so is $\Gamma \Longrightarrow \Delta$;
- *complete*, if whenever $\Gamma \Longrightarrow \Delta$ is universally valid then also $\Gamma_1 \Longrightarrow \Delta_1$ and $\Gamma_2 \Longrightarrow \Delta_2$ are valid.

For nonbranching rules and rules with side conditions the obvious modifications have to be made.

2.2. KeY

KeY [32] is a multi-purpose formal verification system that uses JavaDL as the program logic. Its main application is the formal verification of Java programs specified by Java Modeling Language (JML) [48] annotations. KeY fully supports the Java Card language [32]. Information flow analysis, test case generation, and a debugging tool are other applications that have been realized in KeY.

2.2.1. Prover Core

The core of KeY is a theorem prover implementing a sequent calculus. Schematic sequent calculus rules in KeY are specified as *taclets*. They contain the declarative, logical content of the rule schemata, and pragmatic information such as in which context and when a rule should be applied by an automated reasoning strategy and how it is to be presented to the user. Taclets usually have one main formula of a sequent in focus that can be manipulated and automated proof search does not implement backtracking. They are optimized for automated proof search in typed first-order logic and logic-based symbolic execution in JavaDL [32].

Taclets provide flexibility for the various application scenarios in KeY, and they dispense with the need to support higher-order quantification in the logic. This makes interaction with the prover easier for humans and other programs. But also, the proximity of modeling languages such as JML and of the language of SMT solvers to typed first-order logic make it simple to import and export formulas from KeY 's program logic. On the one hand, this makes it possible to have JML-annotated Java as an input language of KeY, on the other hand, using SMT solvers as a backend increases the degree of automation [32].

2.2.2. Reasoning about Programs

Given a Java program and its JML specification, KeY first translates it into *proof obligations*. They are formulas that their validity corresponds to correctness of the program with respect to its specification. JavaDL is used for showing this validity. As programs directly appear in JavaDL formulas, the pre-processing step of generating the proof obligations is relatively small.The rest of the verification process is deductive.

KeY performs symbolic execution on the programs, and with help of JavaDL updates it handles the programs in the calculus. Symbolic execution resembles executing the program, using symbolic instead of concrete values for the program variables. Ultimately programs are removed from formulas, and the verification problem is reduced to the problem of proving the logical validity of formulas in first-order predicate logic with built-in theories.

These remaining tasks are usually also handled within KeY itself. Alternatively, KeY allows sending such verification problems to external Satisfiability Modulo Theories (SMT) solvers. These are then used as trusted black boxes that can automatically determine the validity of some first-order formulas, thereby sacrificing some traceability for sometimes better automation and performance than offered by KeY itself [49].

2.3. Data Dependence Analysis

In HPC, one of the most important tasks is to find opportunities for parallelization of sequential programs, for example, of loop bodies. This requires to identify program segments that cannot possibly influence each other's outcome: a prerequisite for being able to run them in parallel without changing semantics. The central notion employed is an analysis of read and write data dependences. For example, a read-after-write data dependence (termed *RaW*) in a piece of code `prg` with respect to a memory location x is defined as follows: for a possible execution sequence e of `prg` started in an arbitrary state, there is a write access to x in e which has a subsequent read access to x in e.

Based on Bernstein [50] and Banerjee [2] definitions of data dependences, we define data dependence on memory locations as following.

Definition 2.3.1 (Data dependence)**.** There is a *data dependence* on memory location loc if there are memory accesses acc_1 and acc_2 such that

- both acc_1 and acc_2 access loc, and at least one of these accesses is a write;
- during sequential execution, acc_1 is executed before acc_2.

Data dependence	Notation	Example
Flow dependence	$RaW(\texttt{x})$	`x = y; ... z = x;`
Anti-dependence	$WaR(\texttt{x})$	`y = x; ... x = z;`
Output dependence	$WaW(\texttt{x})$	`x = y; ... x = z;`

Figure 2.6.: Different data dependence types with examples. `x`, `y`, and , `z` are memory locations.

```
i = 0;
while (i < a.length - 1) {
    a[i] = a[i]+1;
    i++;
}
```

Listing 2.2: Increasing value of array elements

Accordingly, there are three types of data dependence: *read-after-write* (*RaW*, aka flow dependence), *write-after-read* (*WaR*, aka anti-dependence), and *write-after-write* (*WaW*, aka output dependence). To state that there exists a data dependence on memory location loc the notation $RaW(loc)$, etc. is used.

Figure 2.6 summarizes the data dependence types with examples.

Data dependences that span over different loop iteration are called *inter-iteration* data dependences.

Example 2.3.1. In Listing 2.2 there is *WaR* data dependence on elements of array `a` for $1 \leq$ `i` $\leq$ `a.length - 2`. This is an inter-iteration data dependence as, for example, `a[1]` is read in iteration one and written in iteration two.

Importance of Data Dependence Analysis

Pattern based parallelization is a common parallelization approach that uses parallel design patterns [51], which allow making best use of parallel programming interfaces such as OpenMP. When such patterns cannot be implemented directly, it can be necessary to apply code transformations beforehand to suitably reshape the input program [34]. Finding the data dependences in the sequential code is a crucial step in parallelization process. Parallelization in presence of data dependences causes concurrency-related errors such as deadlocks and data races.

```
i = 0; sum = 0;
while (i < a.length) {
    sum += a[i];
    i++;
}
```

Listing 2.3: Sum of array elements

```
i = 0;
sum_1 = 0;

 while (i < a.length/2) {
    sum_1 += a[i];
    i++;
}
```

```
i = a.length/2;
sum_2 = 0;

 while (i < a.length) {
    sum_2 += a[i];
    i++;
}
```

Figure 2.7.: Child loops of Listing 2.3

Here we go through the relation of existence and absence of certain kind of data dependences with the application of different parallel design patterns.

If there is no inter-iteration data dependence between different iterations of a loop, it can be parallelized according to *DoAll* pattern. In DoAll pattern all iterations of a loop can be executed in parallel. For example, in Listing 2.2 all loop iterations can be executed in parallel as the loop has DoAll pattern.

With a *WaR* data dependence on a memory location between different iterations of a loop, it is possible to parallelize the loop with *reduction* pattern. For example there is an inter-iteration *WaR* data dependence on variable sum in Listing 2.3. Consequently, different iterations of the loop are not independent. However, if this data dependence is detected it can be resolved using a temporary variable.

Figure 2.7 shows the result of parallelizing the loop in Listing 2.3 with reduction pattern assuming two threads are available. First, we split the iteration space of the loop into two partition and assign each of them a thread. In each thread sum of the array elements accessed by the corresponding child loop is computed. In the end, the master thread calculates the result `sum = sum_1 + sum_2`.

In cases that the data accessed by the loop (or function) can be partitioned into multiple sections, the loop can be ran in parallel on those sections. This pattern is called *geometric decomposition* and it is applicable when all the child loops (or child functions) can be

```
i = 0; b = 0;
while (i < a.length) {
    b = f(a[i]);
    g(b);
    i++;
}
```

Listing 2.4: Pipeline example

parallelized with DoAll or reduction pattern.

If the loop (or function) body can be divided into a sequence of statements that can execute independently *pipeline* pattern is applicable. If there is a *WaR* or *WaW* data dependence between these statements, pipeline is applicable after restructuring the loop [52]. For example, in Listing 2.4 function `f` is called for each element of array `a` and the result is written on `b`. Then, function `g` is called for variable `b`. Functions `f` and `g` create a pipeline of tasks that can execute in parallel. Meaning that `g(b)` can be executed in parallel to `f` for a new array element.

Therefore, showing absence of inter-iteration data dependences opens the opportunity for applying DoAll, reduction, and geometric decomposition patterns. In addition, showing absence of intra-iteration *RaW* data dependence indicates an opportunity for applying pipeline pattern. In this case, the auto-parallelizer does not need to apply different restructuring techniques and after each of them check if parallelization is possible.

2.4. Loop Invariant Generation with Predicate Abstraction

A loop invariant is a formula describing an over-approximation of all states reachable by repeated execution of a loop's body while the loop condition is true. Using a loop invariant essentially is an inductive argument, proving that the invariant holds for any number of loop iterations and, thus, still holds when the loop terminates [32].

A loop specification is similar to a method contract in that it formalizes an abstraction of the relationship between the state before a method or loop is executed and the state when the method or loop body, respectively, terminates. In that sense a loop invariant is both the precondition and postcondition of the loop body. Yet, in most cases, a useful loop invariant is more difficult to find than a method contract because it relates the initial state with the states after *every* loop iteration [32].

Abstract interpretation [53] is a framework for static analysis of programs. It is defined

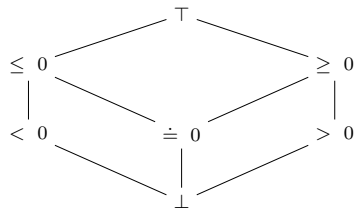


Figure 2.8.: Sign analysis domains.

by fixing its *abstract domain*. Abstract domain is a set of formulas that can describe properties of an abstract state of the program. The set of abstract states has a partial order relation. Abstract domain equipped with this partial order relation forms an *abstract language* that can describe an abstract state of the program. For reasoning, it is easier to work on an abstract domain than using the concrete domain, as for example it ensures termination.

Example 2.4.1. We might be interested in checking that after executing

```
i = 0; while(i < N) i++;
```

the value of `i` > 0 under the assumption that `N` > 0. This can be achieved by using abstract interpretation and abstracting the value of `i` using the abstract sign domain depicted in Figure 2.8. Termination of the abstract interpretation is guaranteed by the finiteness of the lattice.

Predicate abstraction [28] is a well-known abstract interpretation technique in which the abstract domain is constructed from a finite set of predicates over program variables. It approximates the set of reachable states by iterating over the loop and stopping when the fixpoint is reached. The loop invariant is constructed at the fixpoint.

In this section we summarize a version of loop invariant generation by predicate abstraction adapted to deductive verification [29]. It pursues a *bottom-up* approach, i.e. it starts with the code under analysis to construct a loop invariant without the need to rely on external specifications (though these can be used, if available).

2.4.1. Predicate Abstraction

An initial set of predicates $\mathcal{D}_{\mathsf{init}}$ is provided (by the user, heuristics, etc.) that approximates the set of reachable states at loop entry. Predicate abstraction refines this set by iteratively applying the strongest postcondition transformer to the loop body. Each iteration produces

a larger approximation of the set of reachable states. This set is abstracted at each loop entry by calling an automatic theorem prover to show whether a predicate holds in the current state. This process is often called *predicate refinement* 2.4.2.

For a set of predicates $\mathcal{P}$ related to a loop we define the abstract domain that abstracts the (infinite) set of concrete states as following:

Definition 2.4.1 (Abstract domain)**.** The abstract *domain* $\mathcal{D}^{\mathcal{P}}$ consists of the atomic formulas constructed from PSym and PV, i.e. $\mathcal{D}^{\mathcal{P}} = \{p(pv) \mid p \in \text{PSym and } pv \in \text{PV}\}$.

Definition 2.4.2 (Partial order set)**.** Partial order set is a tuple $\langle \mathcal{S}, \sqsubseteq \rangle$ where $\mathcal{S}$ is a set and $\sqsubseteq$: $\mathcal{S} \times \mathcal{S}$ is a partial order relation which is a reflexive, antisymmetric, and transitive [54].

Based on the partial order relation between the predicates of the abstract domain the abstract language is defined.

Definition 2.4.3 (Abstract language)**.** The *abstract language* is a partial order set on the abstract domain $L^{\mathcal{P}} = \langle \mathcal{D}^{\mathcal{P}}, \sqsubseteq \rangle$ [55].

2.4.2. Predicate Refinement

In predicate refinement, the loop body is iteratively executed and abstracted. At each iteration, the loop body is symbolically executed. For that, the loop is unwound once applying the loopUnwind rule. Now information about this iteration is provided in form of *updates* on the modality. Predicate abstraction needs to use this information to abstract the current set of reachable states. It would work better if this information is provided in the form of *formulas*. For reconciling predicate abstraction and symbolic execution, we follow the approach suggested in [56] and we explain it in Section 4.1.

The new sequent resulting from unwinding the loop and turning the updates to formulas is used for abstraction. In this sequent, the program formula is substituted by a predicate from the current set of predicates. The automated theorem prover (here, KeY) tries to prove the predicate. If it is proven, it remains in the set of predicates and is used in the next round of abstraction. Meaning, that it is a candidate for abstracting the loop. Otherwise, it is replaced by a more precise predicate based on the partial order relation. Such sequent is formed for each predicate in the current set of predicates and subsequently the theorem prover is called. This forms a full round of predicate refinement.

The iterative process of predicate refinement continues until fixpoint is reached. The fixpoint can be calculated in different ways. We assume if in two consecutive iterations of predicate abstraction the set of predicates does not change, the fixpoint is reached. Reaching the fixpoint is guaranteed since the abstract domain is finite.

2.4.3. Loop Invariant

Predicate refinement stops when the fixpoint is reached. The loop invariant is a Boolean combination of the predicates left in the predicate set. We follow the approach in [56] that only allows conjunctions of the predicates. Although this is less expressive than supporting arbitrary Boolean combinations, it is much cheaper to compute.

Loop invariant LI is a conjunction of p_i's, where $p_i \in \mathcal{D}_{\mathcal{P}}$. By construction, when the fixpoint is reached all p_i's left in the predicate set are proven to initially hold be preserved by the loop body.

Loops with Branching Statements

When symbolic execution reaches a branching statement, it splits the proof into multiple branches. We need to represent these differing executions on the branches within a single loop invariant. This is achieved by joining proof branches according to the symbolic state merging framework of [57]. In the simplest approach, a fully precise merged sequent is constructed by encoding the differences as implications of the (negated) branch condition (for instance, from an if statement).

Invariants generated for such loop bodies with multiple control flow still are mere conjunctions over predicates, but since the *merged* sequent serves as input of the next iteration of predicate refinement, it is guaranteed that the loop invariants cover the union of all possible control flows.

3. Data Dependence-Aware Program Logic

In this chapter, we extend JavaDL, defined in Section 2.1, to dependence-aware program logic JavaDL$_{Dep}$ that formalizes data dependences. While our presentation uses JavaDL as a logical framework, our approach can be easily transferred to other program logics for imperative languages.

A standard programming language semantics based on traces, i.e. finite or infinite sequences of states (see Section 2.1.2), is insufficient to characterize read and write dependences, as shown by a simple consideration: take two programs consisting of a single assignment $stmnt \equiv \mathsf{x = 42;}$ and $stmnt' \equiv \mathsf{x = y;}$. If $stmnt$ and $stmnt'$ are started in state s with $s(\mathsf{y}) = 42$ both yield exactly the same trace, however, $stmnt'$ has a read access that $stmnt$ has not. *Traces do not record memory access*.

Rather than supplying a special purpose semantic construct, we give a general solution. It is well-known (e.g., [58]) that *non-functional* properties (such as dependences) can often be formally specified with the help of *ghost variables*. These are memory locations not part of the program under verification that record meta properties of program execution (e.g., memory accesses). We could add ghost variables r, w to states that record read and write access at each state change. For example, the state after executing *stmnt'* in s is: $\{\mathsf{x} \mapsto 42, \mathsf{y} \mapsto 42, r \mapsto \{\mathsf{y}\}, w \mapsto \{\mathsf{x}\}\}$. This state can be distinguished from the state that results from executing $stmnt$ in s: $\{\mathsf{x} \mapsto 42, \mathsf{y} \mapsto 42, r \mapsto \{\}, w \mapsto \{\mathsf{x}\}\}$. With this approach we can not record the *order* of memory accesses.

Therefore, in our semantics we extend the notion of state to a pair that includes a function for assigning meaning, and a finite sequence that records the whole *history* of memory accesses in the current execution. The sequence is not accessible in the syntax. Instead, we introduce memory access updates to represent changes in this sequence.

In this chapter, we define the semantics of read and write memory accesses (Section 3.1) and the syntactic representation of them (Section 3.2). After formally defining data dependence properties in Section 3.3, we introduce sequent calculus rules for reasoning about them in Section 3.4.

3.1. Semantics of Read and Write Memory Accesses

Definition 3.1.1 (Domains, JavaDL$_{Dep}$ states)**.** Given a non-empty *domain* $\mathcal{D}$, a JavaDL$_{Dep}$ state $s = (\sigma, Acc)$ is a pair of

- an interpretation σ assigning each
  - type T its domain $\sigma(T) = \mathcal{D}^T \subseteq \mathcal{D},\ \mathcal{D}^T \neq \emptyset$, with the type hierarchy reflected in the subset relation of their domains,
  - function symbol $f : T_1 \times \ldots \times T_n \to T$ to a function $\sigma(f) : \mathcal{D}^{T_1} \times \ldots \times \mathcal{D}^{T_n} \to \mathcal{D}^T$,
  - predicate symbol $p : T_1 \times \ldots \times T_n$ a relation $\sigma(p) \subseteq \mathcal{D}^{T_1} \times \ldots \times \mathcal{D}^{T_n}$;
- a finite sequence $Acc = \langle acc_1 \rangle \circ \cdots \circ \langle acc_n \rangle$ of memory read and write accesses with

$$acc_i \ ::= \ Read(ls) \ \mid \ Write(ls)$$

  where $ls \in \mathcal{D}^{LocSet}$. For acc_i and acc_j with $i < j$, acc_j has happened after acc_i.

We write often $s(f)$ instead of $\sigma(f)$, when $s = (\sigma, Acc)$.

Example 3.1.1. $\langle \mathit{Read}(\{(o, f)\}) \rangle \circ \langle \mathit{Write}(\{(u, g)\}) \rangle$ shows a sequence of length two, with a read access to memory location (o, f) then a write access to location (u, g).

From now on we evaluate programs and formulas relative to states that contain variable Acc that records the trace of memory accesses during execution of a program. We only track heap memory access and *not* access to local program variables. One semantic rule (from Section 2.1.6) in need of modification is heap assignment (2.1). Let $s = (\sigma, Acc)$ be a state that maps program variable o : `Object` to some non-null value, x a program variable:

$$\mathit{val}_{\mathcal{K},s,\beta}(o.f = x;) = s' \text{ with } s' = (\sigma', Acc') \text{ where} \tag{3.1}$$

$$\sigma'(y) = \begin{cases} \sigma'(\mathsf{heap})(u,g) = \begin{cases} \sigma(x)\text{, if } u = \sigma(o) \text{ and } g = \sigma(f) \\ \sigma(\mathsf{heap})(u,g)\text{, otherwise} \end{cases} & , \ y = \mathsf{heap} \\ \sigma(y) & , \text{otherwise} \end{cases}$$

and $Acc' = Acc \circ \langle \mathit{Write}(\{\sigma(o), \sigma(f)\}) \rangle$

The rule now records write access (x is a program variable and the read access to it is not recorded). With the trace of memory accesses stored in Acc it is possible to formally specify dependence properties like read-after-write as follows: Given a state s and a location set ls then there is a read-after-write iff there are i, j with $i < j$, $s(acc_i) = (\mathit{Write}, ls')$ and $s(acc_j) = (\mathit{Read}, ls'')$ such that $ls \cap ls' \cap ls'' \neq \emptyset$.

There is an advantage of maintaining the *whole* access history in each state: it is sufficient to know only the final state to define data dependences. Hence, having the history in the final state makes it sufficient to look at a big-step semantic relation. This is important, because we base logical reasoning about dependences on the big-step program logic introduced in Sect. 2.1.3 and the existing tools.

3.2. Memory Access Updates

For efficiency, Acc which keeps track of read and write memory accesses, is not exposed in the JavaDL$_{Dep}$ syntax. Instead, we only record changes to Acc. JavaDL updates keep track of state changes syntactically. Therefore, we introduce a new kind of updates, called *memory access updates*.

Definition 3.2.1 (Memory access update)**.** The *syntax* of a *memory access update* is

⟨*update*⟩ ::= ... | \R '('⟨*term*⟩')' | \W '('⟨*term*⟩')'

where the term given as argument denotes a location set. Its *semantics* is defined as follows:

$$val_{\mathcal{K},s,\beta}(\backslash \mathsf{R}(loc)) = s' \qquad \text{where } s' = (\sigma, Acc') \text{ with } s = (\sigma, Acc) \text{ and } Acc' = Acc \circ \langle \mathit{Read}(val_{\mathcal{K},s,\beta}(loc)) \rangle$$

$$val_{\mathcal{K},s,\beta}(\backslash \mathsf{W}(loc)) = s' \qquad \text{where } s' = (\sigma, Acc') \text{ with } s = (\sigma, Acc) \text{ and } Acc' = Acc \circ \langle \mathit{Write}(val_{\mathcal{K},s,\beta}(loc)) \rangle$$

To keep the notion succent, we use \M to show \R or \W, and M for showing *Read* or *Write*.

Definition 3.2.2 (Sequential memory access updates)**.** Several memory access updates $\backslash \mathsf{M}_1(loc_1)$; $\backslash \mathsf{M}_2(loc_2)$ can be combined sequentially using the sequential update operator ";". The semantics is defined as following given program state $s = (\sigma, Acc)$

$$val_{\mathcal{K},s,\beta}(\backslash \mathsf{M}_1(loc_1);\ \backslash \mathsf{M}_2(loc_2)) = s'' \text{ where}$$

$$s' = (\sigma, Acc') \text{ and } Acc' = Acc \circ \langle M_1(val_{\mathcal{K},s,\beta}(loc_1)) \rangle$$

and

$$s'' = (\sigma, Acc'') \text{ with } Acc'' = Acc' \circ \langle M_2(val_{\mathcal{K},s,\beta}(loc_2)) \rangle$$

Programs

$$
\begin{aligned}
&\mathit{val}_{\mathcal{K},s,\beta}(x = t) && := \{(s, s') \mid s' = (\sigma', Acc') \text{ with } s = (\sigma, Acc), \text{ where}\\
&&& \quad \sigma' = \sigma[x \leftarrow \mathit{val}_{\mathcal{K},s,\beta}(t)] \text{ and}\\
&&& \quad Acc' = Acc \circ \mathit{acc}_{\mathcal{K},s,\beta}(t)\}\\
&&& \quad (x \in \mathsf{PV})\\
&\mathit{val}_{\mathcal{K},s,\beta}(a[i] = t) && := \{(s, s') \mid s' = (\sigma', Acc') \text{ with } s = (\sigma, Acc), \text{ where}\\
&&& \quad \sigma' = \sigma[a[i] \leftarrow \mathit{val}_{\mathcal{K},s,\beta}(t)] \text{ and}\\
&&& \quad Acc' = Acc \circ \mathit{acc}_{\mathcal{K},s,\beta}(t) \circ \mathit{acc}_{\mathcal{K},s,\beta}(i) \circ \langle \mathit{Write}(\underline{a[i]})\rangle\}\\
&\mathit{val}_{\mathcal{K},s,\beta}(\texttt{if}(b)\ \{stmnt_1\}\ \texttt{else}\ \{stmnt_2\})\\
&&& := \{(s, s'') \mid s' = (\sigma, Acc') \text{ with } s = (\sigma, Acc) \text{ where}\\
&&& \quad \text{and } Acc' = Acc \circ \mathit{acc}_{\mathcal{K},s,\beta}(b),\\
&&& \quad \text{and } s'' = (\sigma, Acc'') \text{ where}\\
&&& \quad ((\mathcal{K}, s', \beta \models b, (s', s'') \in [\![stmnt_1]\!])\\
&&& \quad \text{or } (\mathcal{K}, s', \beta \not\models b \text{ and } (s', s'') \in [\![stmnt_2]\!]))\}\\
&\mathit{val}_{\mathcal{K},s,\beta}(\texttt{while}(b)\ \{stmnt\}) && := \{(s, t) \mid \text{ there is a sequence } s = s_0 \ldots s_n = t \text{ with}\\
&&& \quad \mathcal{K}, s_i, \beta \models b,\ i < n \text{ and } \mathcal{K}, t, \beta \not\models b \text{ and}\\
&&& \quad (s'_i, s_{i+1}) \in [\![stmnt]\!], i < n \text{ with}\\
&&& \quad s'_i = (\sigma_i, Acc_i \circ \mathit{acc}_{\mathcal{K},s'_i,\beta}(b))\}
\end{aligned}
$$

Figure 3.1.: Excerpts of JavaDL$_{Dep}$ Semantics

Definition 3.2.3 (Semantics of JavaDL$_{Dep}$)**.** The semantics of JavaDL$_{Dep}$ is an extension of the semantics of JavaDL. Definition 3.2.1 and Figure 3.1 show this extension. The valuation function $val_{\mathcal{K},s,\beta}$ uses the function

$$acc_{\mathcal{K},s,\beta} : Term \to \mathsf{MemoryAccess}^*$$

to extract all *read* accesses needed to evaluate a given term from left to right.

$$\begin{aligned} acc_{\mathcal{K},s,\beta}(c) &= \emptyset \ (c \text{ rigid constant or a local program variable}) \\ acc_{\mathcal{K},s,\beta}(a[i]) &= \langle Read(\underline{a[i]}) \rangle \\ acc_{\mathcal{K},s,\beta}(f(t_1,\ldots,t_n)) &= acc_{\mathcal{K},s,\beta}(t_1) \circ \ldots \circ acc_{\mathcal{K},s,\beta}(t_n) \\ &\quad (f \text{ is a rigid function symbol}) \end{aligned}$$

In this manuscript, we are only interested in partial correctness, and hence, we restrict ourselves to the box modality (Figure 3.1).

3.3. Specification of Data Dependence Properties

Memory accesses can be tracked through Acc, but how are properties about the content of Acc expressed, given that it is not directly accessible in the syntax? This is achieved with defining *non-rigid data dependence predicates* that allow expressing data dependence properties of memory locations.

Definition 3.3.1 (Data Dependence Predicate)**.** The syntax of *data dependence predicates* is

$$\begin{aligned} \langle fml \rangle ::= \ \ldots \ &| \ \mathsf{noRaW}\ \text{`('}\langle term \rangle\text{`)'} \ | \ \mathsf{noWaR}\ \text{`('}\langle term \rangle\text{`)'} \ | \ \mathsf{noWaW}\ \text{`('}\langle term \rangle\text{`)'} \\ &| \ \mathsf{noR}\ \text{`('}\langle term \rangle\text{`)'} \ | \ \mathsf{noW}\ \text{`('}\langle term \rangle\text{`)'} \end{aligned}$$

where the term given as argument denotes a location set. The semantics of data dependence predicate noRaW specifies that there has not been a read memory access to the location set loc after a write memory access to it. It is defined as follows (the semantics of noWaR and noWaW are analogous; see Appendix A.1):

$$\begin{aligned} s(\mathsf{noRaW}) = \{(ls) \mid\ & s = (\sigma, Acc),\ Acc = \langle M_1(ls_1) \rangle \circ \ \ldots \circ \langle M_n(ls_n) \rangle \\ & \text{for all } i, j = 1 \ldots n \text{ for which } i < j \\ & \text{and } M_i = \mathit{Write} \text{ and } M_j = \mathit{Read} \\ & \text{it holds that } ls \cap ls_i \cap ls_j = \emptyset\} \end{aligned}$$

Predicate noR is specifying an absence of read memory accesses, and its semantics is defined as following (the semantics of noW is analogous and mentioned in Appendix A.1)[1]:

$$s(\mathsf{noR}) = \{(ls) \mid s = (\sigma, Acc),\ Acc = \langle M_1(ls_1)\rangle \circ \ \ldots \circ \langle M_n(ls_n)\rangle, M_i \in \{\mathit{Read}, \mathit{Write}\} \\ \text{and for all } i = 1 \ldots n \text{ for which } M_i = \mathit{Read} \text{ it holds that } ls \cap ls_i = \emptyset\}$$

Data dependence predicates take a location set loc as argument and evaluate to true iff in the memory access history of the current state for *no* location in loc there is a read-after-write, write-after-read, write-after-write, read or write access, respectively. They allow us to express properties such as the following which says: if there was no read-after-write before execution of program p to the memory location (o, f) then there is no read-after-write after execution of p.

$$\mathsf{noRaW}(\{(\mathsf{o}, \mathsf{f})\}) \rightarrow \langle \mathsf{p} \rangle \mathsf{noRaW}(\{(\mathsf{o}, \mathsf{f})\})$$

Data dependence predicates are non-rigid, because their value not only depends on the value of their argument, but also on the memory accesses recorded in Acc.

3.4. Reasoning about Data Dependence Properties

For reasoning about the data dependence properties some of the existing JavaDL sequent calculus rules need to be modified to incorporate memory access updates. Update simplification rules are needed to establish the effect of different kinds of updates on each other. Moreover, the effect of memory access updates on data dependence predicates should be formalized in terms of sequent calculus rules.

3.4.1. Modified Calculus Rules

With memory access updates we can design assignment rules that reflect the semantics defined in Section 3.1. We modify rules from Section 2.1.6 for reading from memory location (o, f) and assigning a side effect-free expression se over local variables to it, as

[1] Although predicate symbols noR and noW do not represent data dependence properties they are crucial for verifying data dependence properties. Therefore, we call them data dependence predicates for uniformity.

following:

readAttribute

$$\frac{\Gamma \Longrightarrow o \neq \texttt{null}, \Delta \qquad \Gamma, o \neq \texttt{null} \Longrightarrow \{x := \texttt{select}(\texttt{heap}, o, f) \parallel \backslash\texttt{R}(\{(o,f)\})\}[r]\phi, \Delta}{\Gamma \Longrightarrow [x = o.f;\ r]\phi, \Delta}$$

writeAttribute

$$\frac{\Gamma \Longrightarrow o \neq \texttt{null}, \Delta \qquad \Gamma, o \neq \texttt{null} \Longrightarrow \{\texttt{heap} := \texttt{store}(\texttt{heap}, o, f, se) \parallel \backslash\texttt{W}(\{(o,f)\})\}[\pi\omega]\phi, \Delta}{\Gamma \Longrightarrow [\pi\ o.f = se;\ \omega]\phi, \Delta}$$

Rules readAttribute and writeAttribute realize the idea of associating a read memory access update with each select operation and a write memory access update with each store operation.

Similarly, rules for accessing array elements are modified:

readArrayElement

$$\frac{\Gamma \Longrightarrow o \neq \texttt{null}, \Delta \qquad \Gamma, o \neq \texttt{null} \Longrightarrow \{x := \texttt{select}(\texttt{heap}, a, \texttt{arr}(i)) \parallel \backslash\texttt{R}(\underline{a[i]})\}[r]\phi, \Delta}{\Gamma \Longrightarrow [x = a[i];\ r]\phi, \Delta}$$

writeArrayElement

$$\frac{\Gamma \Longrightarrow o \neq \texttt{null}, \Delta \qquad \Gamma, o \neq \texttt{null} \Longrightarrow \{\texttt{heap} := \texttt{store}(\texttt{heap}, a, \texttt{arr}(i), se) \parallel \backslash\texttt{W}(\underline{a[i]})\}[\pi\omega]\phi, \Delta}{\Gamma \Longrightarrow [\pi\ a[i] = se;\ \omega]\phi, \Delta}$$

3.4.2. Update Simplification Rules

In Figure 3.2 a selection of memory update simplification rules are shown. In this figure update su represents elementary or parallel *state* updates with no memory access updates. Update $\backslash\texttt{M}$ is a read ($\backslash\texttt{R}$), write ($\backslash\texttt{W}$), or a sequence of memory access updates.

Rule reorderUpdate that allows to swap state and memory access updates is responsible for establishing the normal form. This rule is sound, because state updates cannot change the value of Acc. All rules are designed to maintain the order of memory access updates as required.

Rule accessOnStateUpdate and accessOnAccess are adaptations of Update-on-Update rule category from Figure 2.4. In rule accessOnStateUpdate the cascade of two update

Schematic variables
su = Elementary or parallel *state* updates
$\mathtt{\backslash M}$ = Read ($\mathtt{\backslash R}$), write ($\mathtt{\backslash W}$), or a sequence of memory access updates
Update Simplification Rules

reorderUpdate
$\mathtt{\backslash M}(loc) \parallel su \rightsquigarrow su \parallel \mathtt{\backslash M}(loc)$

accessOnStateUpdate
$\{\mathtt{\backslash M}(loc)\}\{su\}\phi \rightsquigarrow \{\mathtt{\backslash M}(loc) \parallel \{\mathtt{\backslash M}(loc)\}su\}\phi$

accessOnAccess
$\{\mathtt{\backslash M}_1(loc_1)\}\{\mathtt{\backslash M}_2(loc_2)\}\phi \rightsquigarrow \{\mathtt{\backslash M}_1(loc_1);\ \mathtt{\backslash M}_2(\{\mathtt{\backslash M}_1(loc_1)\}loc_2)\}\phi$

applyStateUpdateOnAccess
$\{su\}\mathtt{\backslash M}(loc) \rightsquigarrow \mathtt{\backslash M}(\{su\}loc)$

applyAccessOnElementary
$\{\mathtt{\backslash M}(loc)\}x := t \rightsquigarrow x := \{\mathtt{\backslash M}(loc)\}t$

Figure 3.2.: Selection of memory access update application and simplification rules

applications $\{\mathtt{\backslash M}(loc)\}\{su\}$ is converted into the application of a single parallel update. Due to the last-win semantics for parallel updates, this is possible by applying the first update to the second, and replacing the sequential composition by parallel composition.

Similarly, in Rule accessOnAccess the cascade of two memory access update applications $\{\mathtt{\backslash M}_1(loc_1)\}\{\mathtt{\backslash M}_2(loc_2)\}$ is converted into the application of a single parallel update. However, instead of $\{\mathtt{\backslash M}_1(loc_1) \parallel \mathtt{\backslash M}_2(\{\mathtt{\backslash M}_1(loc_1)\}loc_2)\}\phi$ (parallel memory access updates) we end up with $\{\mathtt{\backslash M}_1(loc_1)\ ;\ \mathtt{\backslash M}_2(\{\mathtt{\backslash M}_1(loc_1)\}loc_2)\}$ (sequential memory access updates) as unlike state updates, memory access updates can only be sequential.

Application of state update on a memory accesses update is depicted in the rule applyStateUpdateOnAccess. Here the update is moved inward and applied on the location set loc. As loc can be a complex term including memory access updates and data dependence predicates, the memory access update can affect its evaluation.

For applying a memory access update on an elementary update the rule applyAccessOnElementary is used, which is adaptation of the rule $\{u\}(v := t) \rightsquigarrow v := \{u\}t$ in Figure 2.4 with u being a memory access update.

Symbolic execution accumulates updates in front of the modality. To reason efficiently, we designed update simplification rules that allow to establish a normal form:

$$u_1 \parallel \ldots \parallel u_n \parallel m_1\ ;\ \ldots\ ;\ m_o,$$

where:

- u_i are state updates as introduced in Section 2.1.1, and
- m_j, m_k are combined memory access updates with the property that if $j < k$ then m_j is symbolically executed before m_k.

3.4.3. Axiomatization

The axiomatization of data dependence predicates in the calculus is given by rules for the application of memory access updates to data dependence predicates. A selection of these rules is provided in Figure 3.3. After the symbolic execution rules of the calculus have eliminated the program and updates have been normalized, what is left, are proof obligations of the form $\Gamma \Longrightarrow \{su \parallel mau\}\phi, \Delta$. In this sequent su is an elementary or a parallel *state* update with no memory access update, and mau is sequence of memory access updates. The memory access updates are applied one by one from right to left. The order of application of the memory access updates is opposite of their chronological order. The reason is that we are interested in a memory access update if it happens *after* another specific memory access update and forms a data dependence. For evaluating the data dependence predicates Update-on-Formula rules are used. Some of the specific update application rules required for noRaW are explained below.

- stateUpdateAppOnNoRaW: A state update su cannot change the content of Acc. Consequently, the extension of predicate noRaW remains unchanged and the update can be propagated inwards as it might affect the evaluation of loc.
- writeAccessAppOnNoRaW: Similar to application of a state update, a write access applied to noRaW can also be propagated inwards. The justification is that we process the memory access sequence from the end, i.e. when applying the update we look at the final entry and in this case a write cannot invalidate the no read-after-write property.
- readAccessAppOnNoRaW: In case of a read access $\backslash\mathsf{R}(loc_1)$, for $\mathsf{noRaW}(loc_2)$ to hold, one must prove that no write to any location in loc_2 happened before. First, the read memory access update $\backslash\mathsf{R}(loc_1)$ is propagated inwards. For the common locations of loc_1 and loc_2 there is a read from a location for which we want to show the no read-after-write property. This property holds only if no write memory access happened before. For all other locations noRaW still must be shown.

stateUpdateAppOnNoRaW

$$\{su\}\mathsf{noRaW}(loc) \rightsquigarrow \mathsf{noRaW}(\{su\}loc)$$

writeAccessAppOnNoRaW $condI$

$$\{\setminus \mathsf{W}(loc_1)\}\mathsf{noRaW}(loc_2) \rightsquigarrow \mathsf{noRaW}(\{\setminus \mathsf{W}(loc_1)\}loc_2)$$

readAccessAppOnNoRaW $condI$

$$\{\setminus \mathsf{R}(loc_1)\}\mathsf{noRaW}(loc_2) \rightsquigarrow \mathsf{noW}(loc_1 \cap (\{\setminus \mathsf{R}(loc_1)\}loc_2)) \wedge \mathsf{noRaW}((\{\setminus \mathsf{R}(loc_1)\}loc_2) \setminus loc_1)$$

knownNoRaW

$$\frac{\Gamma, \mathsf{noRaW}(loc_1) \Longrightarrow \text{if } (loc_2 \subseteq loc_1) \text{ then } (\text{true}) \text{ else } (\mathsf{noRaW}(loc_2 \setminus loc_1)), \Delta}{\Gamma, \mathsf{noRaW}(loc_1) \Longrightarrow \mathsf{noRaW}(loc_2), \Delta}$$

knownNoR $\dfrac{}{\Gamma, \mathsf{noR}(loc) \Longrightarrow \mathsf{noRaW}(loc), \Delta}$

knownNoW $\dfrac{}{\Gamma, \mathsf{noW}(loc) \Longrightarrow \mathsf{noRaW}(loc), \Delta}$

$condI$: loc_2 does not contain data dependence predicates

Figure 3.3.: A selection of rules for axiomatization of data dependence predicates. loc, loc_1, and loc_2 are of type `LocSet`. For the rest of the rules, see Appendix B.1

.

- knownNoRaW: For showing that noRaW(loc_2) holds it is sufficient to know that noRaW(loc_1) holds and loc_2 is a subset of loc_1, otherwise, one only needs to show the property for those locations in loc_2 not contained in loc_1.
- knownNoR and knownNoW: These rules exploit that if there is no read (write) to any location in loc then there is also no read-after-write to any location in loc.

Note. For soundness, it is important to propagate $\backslash\mathsf{R}(loc_1)$ inwards. In practice loc_2 often does not contain data dependence predicates,[2] such that $\{\backslash\mathsf{R}(loc_1)\}loc_2$ is automatically simplified to loc_2 by the prover.

Except the rules mentioned above and a few related ones, only a few calculus rules of [32] needed to be modified. These were minor technical changes which we mention gradually.

There are similar calculus rules for application of state and memory access updates on other data dependence predicates in Appendix B.1.

Theorem 3.4.1 (Soundness and Completeness of JavaDL Calculus)**.** The calculus rules of JavaDL program logic are sound and complete.

Corollary 3.4.1 (Soundness of JavaDL Calculus)**.** Since all the rules in JavaDL are sound, its calculus is sound.

As examples, we provide the proof of soundness and completeness of the rules writeAccessAppOnNoRaW, and readAccessAppOnNoRaW introduced in Figure 3.3 in Appendices C.1 and C.2, respectively.

Example 3.4.1. Figure 3.4 shows the outline of a sequent calculus proof for the sequent:

$$\mathtt{a} \neq \mathtt{null},\ \mathtt{b} \neq \mathtt{null},\ \mathtt{a} \neq \mathtt{b},\ \mathsf{noW}(\underline{\mathtt{a[0]}}) \Longrightarrow [\mathtt{a[0]=v;\ z=b[0];}]\mathsf{noRaW}(\underline{\mathtt{a[0]}}).$$

It expresses that if we start in a state with no write access to a[0], and where a and b refer to different existing array objects, then after executing a[0]=v; z=b[0]; there is no read-after-write access to a[0]. Program variables v and z are local program variables and the accesses to them are not recorded. First, the program is symbolically executed and update normalization takes place. Subsequent update simplification eliminates state update su, as it does not affect the data dependence property. This results in the sequent:

$$\mathtt{a} \neq \mathtt{null},\ \mathtt{b} \neq \mathtt{null},\ \mathtt{a} \neq \mathtt{b},\ \mathsf{noW}(\underline{\mathtt{a[0]}}) \Longrightarrow \{\backslash\mathsf{W}(\underline{\mathtt{a[0]}});\backslash\mathsf{R}(\underline{\mathtt{b[0]}})\}\mathsf{noRaW}(\underline{\mathtt{a[0]}})$$

[2] This can be checked syntactically.

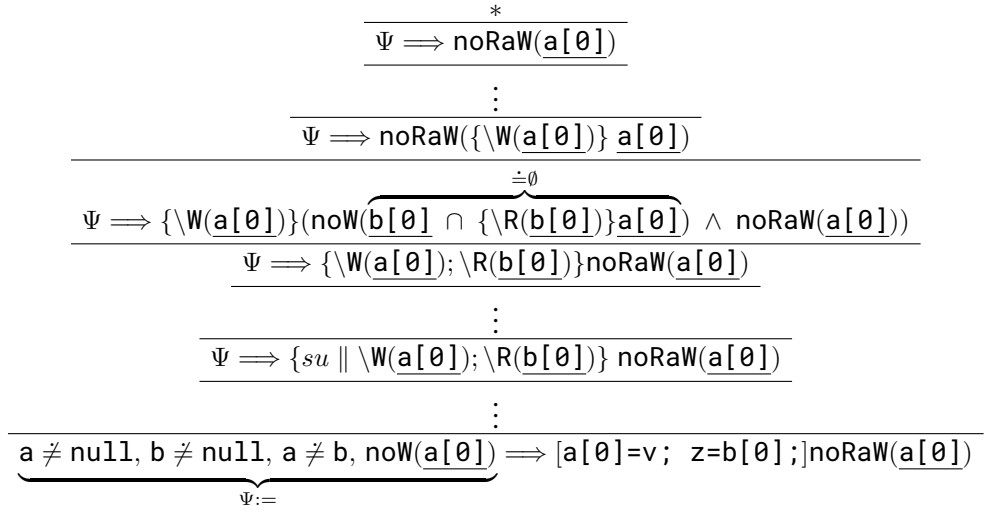


Figure 3.4.: Example of a formal verification proof of a noRaW property

Now rule readAccessAppOnNoRaW is applied in a variant for combined memory access updates. The intersection of the two location sets inside the noW is provably empty and $\mathsf{noW}(\emptyset) \rightsquigarrow$ true. The read memory access $\backslash\mathsf{R}(\underline{\mathtt{b[0]}})$ can be removed, because its location set is disjoint from $\underline{\mathtt{a[0]}}$. [3] Now write memory access $\backslash\mathsf{W}(\underline{\mathtt{a[0]}})$ can be moved inwards with applying writeAccessAppOnNoRaW rule. After further simplification we apply noWnoRaW rule (Ψ' contains $\mathsf{noW}(\underline{\mathtt{a[0]}})$ to close the proof.

[3] This is easily justified formally by using $\{\backslash\mathsf{W}(\underline{\mathtt{a[0]}})\}\{\backslash\mathsf{R}(\underline{\mathtt{b[0]}})\}$ instead of $\{\backslash\mathsf{W}(\underline{\mathtt{a[0]}}); \backslash\mathsf{R}(\underline{\mathtt{b[0]}})\}$ and the original rule readAccessAppOnNoRaW, but would result in a longer, more technical proof.

4. Automatic Loop Invariant Generation for Data Dependence Analysis

To analyze and verify data dependences in programs with loops we need to synthesize data dependences that hold in each loop iteration. Traditional loop invariants specify (or rather: over-approximate) the set of states reachable both at the beginning and at the end of each loop iteration. In addition, we need to specify dependences that occur during an arbitrary loop iteration, including dependences between different iterations (Chapter 5). Consequently, we need to design a *dependence-aware* loop invariant generation technique.

For automatic loop specification generation we focus on the data dependence analysis and not on the termination. In our application area HPC, loops are for-loops whose termination can easily be established.

In addition, we focus on dependences among array types, because these are by far the most relevant data structure in HPC. If there is a parallelization opportunity at all, then it occurs typically in terms of array segments that can be processed in parallel. Array intervals of the form $a[i..j]$ for $0 \leq i \leq j <$ `a.length` are a natural data abstraction. They induce a lattice via the "contains more memory locations" order (Figure 4.1). Also the dependence relations form a lattice with "no dependence" as the top element (Figure 4.2). The existence of lattice abstractions for dependences and program data is one reason why *predicate abstraction* [28] is an appropriate loop invariant generation technique.

The loop invariant generation process works on a language of abstraction predicates based on the mentioned lattices (Section 4.2.1), viewed as an abstract domain on which to express data dependence loop invariants. Predicate abstraction normally works by *abstract interpretation* [53] of the code to generate invariant candidates [29]. For retaining full precision concerning the analyzed program, we use fully precise, logic-based symbolic execution [32, Chapter 3]. To achieve this, we use deductive verification based on an extended symbolic execution calculus that can generate abstract dependence predicates (Section 4.2.2). The main technical difficulty we need to solve is to combine formula-based predicate abstraction and state-based symbolic execution in a deductive framework.

The loop invariant generation process itself follows a fixpoint iteration approach: a loop body is symbolically executed by calling the deductive verifier to produce a data depen-

dence invariant candidate. Then the same loop body is executed once more, assuming the invariant holds. If the invariant is re-established at the end, then it is valid. Otherwise, the differing data dependence invariant candidates (at the beginning and at the end of one iteration, respectively) are combined into the least common abstraction and a further iteration is performed. The process ends, once a fixpoint is reached.

Some complications need to be considered: first, symbolic execution of a loop body in general results in more than one possible control flow, so it branches into many different symbolic execution paths. This is handled with a symbolic path merging rule [57]. Second, bounds in the lattice in Figure 4.1 are symbolic and not necessarily known. There is an unknown number of sequence of arrays $\mathsf{a}[i..j] \subseteq \cdots \subseteq \mathsf{a}[i'..j']$ with symbolic bounds such that $i > i'$ and $j < j'$. As usual in abstract interpretation, termination is enforced by widening steps that are heuristically triggered.

We focus on the generation of the *part of the loop invariant* concerned with data dependences and call it *data dependence loop invariant*. Additionally, we need to ensure that array accesses are within their bounds. For this we rely on existing predicate abstraction techniques for deductive verification [56].

The loop invariants we generate are conjunctions of predicates that capture atomic data dependences between memory areas. For example, the predicate $\mathsf{noR}(loc)$ specifies that *no* location in the memory location set *loc* is read. To simplify the presentation, we consider only location sets that describe contiguous memory ranges of arrays. The extension to general memory locations is straightforward.

The memory locations associated with an array form a lattice using the subset relation on their extension. An excerpt is shown in Figure 4.1.

In Section 4.1, we develop foundations for adapting predicate abstraction in symbolic execution. We use this foundation to produce data dependence loop invariant automatically in Section 4.2. We end the chapter with sequent calculus rules for embedding predicate abstraction in JavaDL$_{Dep}$ (Section 4.3).

4.1. Reconciling Predicate Abstraction and Symbolic Execution

For generating loop invariants we use the predicate abstraction technique explained in Section 2.4. It starts with an initial set of predicates describing the initial state of the loop. This set is iteratively refined till reaching a fixpoint.

To refine a given set of data dependence predicates D we combine predicate abstraction with symbolic execution. This allows us to refine D while at the same time, symbolically executing the loop body. After unwinding the loop (applying the loopUnwind rule), the

sequent to be proven has the following shape:

$$\Gamma, \ldots \Longrightarrow \{u\}[\texttt{while (}b\texttt{) \{ }body\texttt{ \}}]\phi, \Delta$$

Our forward symbolic execution calculus, including unwinding, accumulates state updates *in front of* modalities. However, for technical reasons, predicate abstraction is easier to implement, if state changes are expressed as equations corresponding to the strongest postcondition, not as weakest preconditions that result from update application. Hence, we need to rewrite updates accordingly using state shifting rules. A state update shifting technique that turns state updates into formulas usable in predicate abstraction was discussed in [49]. It works by:

1. translating updates into equations, which are then added to the antecedent, and
2. transforming the remaining formulas into their strongest postcondition.

To retain access to pre-update values all program variables occurring on the left-hand side of an elementary update must be renamed.

Definition 4.1.1 (Shift state update rule schema [49])**.**

$$\mathsf{shiftStateUpdate}\ \frac{\{u'\}\Gamma, upd \Longrightarrow \phi, \{u'\}\Delta}{\Gamma \Longrightarrow \{u\}\phi, \Delta}\ ,$$

where $upd = \bigwedge_{i\in\{1,\ldots,n\}} a_i \doteq \{u'\}\{u\}a_i$ with:

- $u' = a_1 := a'_1 || \ldots || a_n := a'_n$
- $a'_i \in \mathsf{FSym}_r$ fresh constants for all $i \in \{1, \ldots, n\}$
- $\{a_1, \ldots, a_n\} = updated(u)$ with $updated : \mathit{Upd}^{\Sigma} \to 2^{\mathsf{PV}}$

$$updated(u) = \begin{cases} a & u = a := t \\ updated(u_1) \cup updated(u_2) & u = u_1 || u_2 \\ updated(u_2) & u = \{u_1\}u_2 \end{cases}$$

Update u' substitutes each updated program variable a_i with a fresh constant a'_i representing the old (pre-update) value of a_i. Formula upd links the old instances with the updated ones. Weiß [49] showed that $\{u'\}\Gamma, upd$ is the strongest postcondition of Γ under u.

The idea behind this rule is, instead of transforming ϕ to its weakest precondition under update u, to transform the rest of sequent into its strongest postcondition. This is

done by *advancing* the symbolic state of the sequent. The technique is known as *priming*. Formula $upd(u, u')$ establishes the relation between the original update u and its renamed ("primed") version u'. The effect of this rule is that, instead of computing the pre-state of formula ϕ locally, we advance the symbolic state of the sequent.

Example 4.1.1. Assuming array `a` is not null, after applying rule schema assignmentLocalVariable to Listing 2.1 we obtain the sequent:

$$\texttt{a} \neq \texttt{null} \Longrightarrow \{\texttt{i} := \texttt{0}\}$$
$$[\texttt{while (i < a.length - 1) \{a[i] = a[i+1]; i++;\}}]\phi$$

After applying shiftStateUpdate we have:

$$\texttt{a} \neq \texttt{null}, \texttt{i} \doteq \{\texttt{i} := \texttt{i}'\}(\{\texttt{i} := \texttt{0}\}\texttt{i}) \Longrightarrow$$
$$[\texttt{while (i < a.length - 1) \{a[i] = a[i+1]; i++;\}}]\phi$$

After unwinding the loop and simplifying the antecedent, assuming we are in the branch, where the `while` condition holds, the sequent above becomes:

$$\texttt{a} \neq \texttt{null}, \texttt{i} \doteq \texttt{0} \Longrightarrow \{\texttt{i} := \texttt{i+1} \| \texttt{a[0]} := \texttt{a[1]} \| \backslash\texttt{R}(\underline{\texttt{a[1]}}); \backslash\texttt{W}(\underline{\texttt{a[0]}})\}$$
$$[\texttt{while (i < a.length - 1) \{a[i] = a[i+1]; i++;\}}]\phi$$

Applying the shiftUpdate rule again and simplifying gives:

$$\texttt{a} \neq \texttt{null}, \texttt{i}' \doteq \texttt{0}, \texttt{i}' < \texttt{a.length - 1}, \texttt{i} \doteq \texttt{1}, \texttt{a[0]} \doteq \texttt{a[1]} \Longrightarrow$$
$$\{\backslash\texttt{R}(\underline{\texttt{a[1]}}); \backslash\texttt{W}(\underline{\texttt{a[0]}})\}$$
$$[\texttt{while(i < a.length-1) \{a[i] = a[i+1]; i++;\}}]\phi$$

We use shiftStateUpdate rule for turning state updates into formulas. To address memory access updates we need further rules. To this end we define two *dependence-aware* shift rules corresponding to read and write access, respectively.

Read and write memory accesses are not stored explicitly as values of a program variable, but in terms of memory access updates and non-rigid dependence predicates, so we need a technique to rewrite formulas containing dependence predicates into their strongest postcondition. Technically, we achieve this by introducing dual access predicates that contain the renamed location sets.

Definition 4.1.2 (Renamed memory access update predicate)**.** The *syntax* of a *renamed memory access update* is

$\langle update \rangle$ ::= ... | \R′ '('$\langle term \rangle$')' | \W′ '('$\langle term \rangle$')'

where the term given as argument denotes a location set. Their *semantics* is defined as follows:

$$val_{\mathcal{K},s,\beta}(\backslash \mathsf{R}'(loc)) = s' \text{ where}$$

$$s' = \begin{cases} (\sigma, acc_{seq}), & \text{if } s = (\sigma, acc_{seq} \circ \langle Read(val_{\mathcal{K},s,\beta}(loc))\rangle) \\ sChoice(s, val_{\mathcal{K},s,\beta}(loc)), & \text{otherwise} \end{cases}$$

$$val_{\mathcal{K},s,\beta}(\backslash \mathsf{W}'(loc)) = s' \text{ where}$$

$$s' = \begin{cases} (\sigma, acc_{seq}), & \text{if } s = (\sigma, acc_{seq} \circ \langle Write(val_{\mathcal{K},s,\beta}(loc))\rangle) \\ sChoice(s, val_{\mathcal{K},s,\beta}(loc)), & \text{otherwise} \end{cases}$$

We assume function $sChoice(s, val_{\mathcal{K},s,\beta}(loc))$ that takes state s and location set $val_{\mathcal{K},s,\beta}(loc)$ and returns some state. This way, we ensure that the evaluation function is a total and well-defined function.

Intuitively the renamed memory access update removes the last recorded memory access of a given state and acts as an 'inverse' to the corresponding memory access update. The semantics is the inverse of the memory access updates. This corresponds to the duality between weakest precondition and strongest postcondition. For instance, rule dualityAccAndRenamedAcc, with a side condition that loc does not contain data dependence predicates, uses this duality:

dualityAccAndRenamedAcc $\quad \{\backslash \mathsf{R}'(loc)\}\{\backslash \mathsf{R}(loc)\}\phi \rightsquigarrow \phi$

Theorem 4.1.1 (Soundness and completeness of dualityAccAndRenamedAcc)**.** Rule dualityAccAndRenamedAcc is sound and complete.

Proof of Theorem 4.1.1 is shown in Appendix C.3.

With multiple sequential memory access updates $\{\backslash \mathsf{M}_1(loc_1); \backslash \mathsf{M}_2(loc_2); \ldots; \backslash \mathsf{M}_n(loc_n)\}\phi$, shifting process starts from the chronologically latest memory access update $\backslash \mathsf{M}_n(loc_n)$ towards the first one $\backslash \mathsf{M}_1(loc_1)$. Finally, to link the renamed memory accesses update with the original memory access updates, we define *memory access predicates*, which play the role of formula *upd* in shiftStateUpdate.

Definition 4.1.3 (Memory access predicate)**.** The syntax of a *memory access predicate* is

$\langle fml \rangle$ ::= ... | `rPred` '('$\langle term \rangle$, $\langle term \rangle$')' | `wPred` '('$\langle term \rangle$, $\langle term \rangle$')'

where the term given as first argument denotes a location set and the second term an integer. Its semantics specifies that the i-th most recent memory access of the given kind was limited to locations ls and is defined as follows (the semantics of `wPred` is analogous):

$$\begin{aligned} s(\texttt{rPred}) = \{(ls, i) \mid s = (\sigma, Acc),\ 0 \leq i < n,\ & \\ Acc = \langle M_1(ls_1)\rangle \circ \ldots \circ \langle M_{n-i}(ls_{n-i})\rangle \circ \ldots \circ \langle M_n(ls_n)\rangle,& \\ \text{and } M_{n-i} = \mathit{Read} \text{ and } ls_{n-i} = ls\} & \end{aligned}$$

Using the renamed memory access updates and memory access predicates, we can shift memory access updates. The rule for shifting a renamed memory access update for read is as follows:

Definition 4.1.4 (Rule schema shiftRead)**.**

$$\frac{\{\backslash \mathsf{R}'(loc)\}\Gamma, \texttt{rPred}(loc, 0) \Longrightarrow \phi, \{\backslash \mathsf{R}'(loc)\}\Delta}{\Gamma \Longrightarrow \{\backslash \mathsf{R}(loc)\}\phi, \Delta} \quad \text{cond}$$

With the side condition cond that *loc* does not contain data dependence predicates.

As before, we transform all formulas of a sequent into their strongest postcondition. Technically, we achieve this by applying the renamed memory access update $\backslash \mathsf{R}'(loc)$ to all formulas *and* adding the formula $\texttt{rPred}(loc, 0)$. The latter says that in the new state of the sequent, the most recent memory access, which is labeled by 0, was a read access on loc. The rule schema shiftWrite is completely analogous, see Appendix B.2.

After shifting and update simplifications, the resulting sequent expresses now any symbolic state in terms of formulas instead of updates. This enables predicate abstraction.

Theorem 4.1.2 (Soundness and completeness of shiftRead)**.** Let Γ, Δ be sets of formula, a formula ϕ, and the location set $loc \in \mathcal{D}^{\texttt{LocSet}}$. If and only if

$$\models \{\backslash \mathsf{R}'(loc)\}\Gamma, \texttt{rPred}(loc, 0) \Longrightarrow \phi, \{\backslash \mathsf{R}'(loc)\}\Delta$$

and loc does not contain data dependence predicates, then the following holds:

$$\Gamma \Longrightarrow \{\backslash \mathsf{R}(loc)\}\phi, \Delta.$$

Theorem 4.1.2 is proven in Appendix C.4.

Example 4.1.2. Continuing Example 4.1.1, in a state with no data dependence on array a (noDep(a[0..a.length-1])) we obtain the sequent:

$$\texttt{a} \neq \texttt{null},\ \texttt{i}' \doteq \texttt{0},\ \texttt{i}' < \texttt{a.length - 1},\ \texttt{i} \doteq \texttt{1},\ \texttt{a[0]} \doteq \texttt{a[1]},$$
$$\texttt{noDep(a[0..a.length-1])} \Longrightarrow$$
$$\{\backslash\texttt{R(a[1])};\backslash\texttt{W(a[0])}\}\ [\texttt{while (i < a.length - 1) \{a[i] = a[i+1]; i++;\}}]\phi$$

Applying shiftRead, *then* shiftWrite and update simplification rules:

$$\texttt{a} \neq \texttt{null},\ \texttt{i}' \doteq \texttt{0},\ \texttt{i}' < \texttt{a.length - 1},\ \texttt{i} \doteq \texttt{1},\ \texttt{a[0]} \doteq \texttt{a[1]},$$
$$\texttt{noR(a[0..a.length-1]} \setminus \texttt{a[1])},\ \texttt{noW(a[1..a.length-1])},$$
$$\texttt{rPred(a[1]}, 1), \texttt{wPred(a[0]}, 0) \Longrightarrow$$
$$[\texttt{while (i < a.length - 1) \{a[i] = a[i+1]; i++;\}}]\phi$$

Memory access updates (\R(a[1]), \W(a[0])) in the first sequent are turned into antecedent formulas rPred(a[1], 1), wPred(a[0], 0). Application of the renamed memory access update increased the second argument of rPred(loc, lb) by one (we will explain this in Section 4.3) and excluded locations, where a read and write access took place, from the initial dependence predicate noDep(a[0..a.length-1]). The rewrite rules are discussed later Section 4.3.

Example 4.1.2 exhibits that sequents are still missing information. After applying shiftRead and before applying shiftWrite to memory access \W(a[0]), the information that noW(a[0..a.length-1]) held is missing. As a remedy we introduce *history predicates* that record historical information before a memory access update.

Definition 4.1.5 (History data dependence predicate)**.** The *syntax* of the *history data dependence predicates* is

⟨*fml*⟩ ::= noRHist '('⟨*term*⟩, ⟨*term*⟩')' | noWHist '('⟨*term*⟩, ⟨*term*⟩')' | ...

where the first argument is a location set and second an integer. The *semantics* is as follows:

$$\begin{aligned} s(\texttt{noRHist}) = \{(ls, i) \mid\ & s = (\sigma, Acc),\ 0 \leq i < n, \\ & Acc = \langle M_1(ls_1)\rangle \circ \ldots \circ \langle M_{n-i}(ls_{n-i})\rangle \circ \ldots \circ \langle M_n(ls_n)\rangle, \\ & \text{for all } 0 < j \leq n - i \text{ with} \\ & M_j = \textit{Read} \text{ it holds that } ls \cap ls_j = \emptyset\} \end{aligned}$$

(analogously for noWHist in Appendix A.2) and

$$s(\texttt{noRaWHist}) = \{(ls, i) \mid s = (\sigma, Acc), 0 \leq i < n,$$
$$Acc = \langle M_1(ls_1)\rangle \circ \ldots \circ \langle M_{n-i}(ls_{n-i})\rangle \circ \ldots \circ \langle M_n(ls_n)\rangle,$$
$$\text{and for all } 0 < j < k \leq n - i \text{ with}$$
$$M_j = \textit{Write} \text{ and } M_k = \textit{Read}$$
$$\text{it holds that } ls \cap ls_j \cap ls_k = \emptyset\}$$

(analogously for noWaRHist, noWaWHist, see Appendix A.2).

History predicates are introduced by applying a renamed memory access update on a dependence predicate. This enables us to keep the necessary precision in the shifted sequent. They are sufficient for our purpose, but do not result in a complete calculus. For that, *exact* history information at an arbitrary position in the past is needed.The nature of data dependence analysis that is concerned about the *flow* of memory accesses demands such information.

Example 4.1.3. With history predicates the final sequent of Example 4.1.2 becomes:

$$\texttt{a} \neq \texttt{null},\ \texttt{i}' \doteq 0,\ \texttt{i}' < \texttt{a.length - 1},\ \texttt{i} \doteq 1,\ \texttt{a[0]} \doteq \texttt{a[1]},$$
$$\texttt{noR}(\underline{\texttt{a[0..a.length-1]}} \setminus \underline{\texttt{a[1]}}),\ \texttt{noW}(\underline{\texttt{a[1..a.length-1]}}),$$
$$\texttt{rPred}(\underline{\texttt{a[1]}}, 1),\ \texttt{wPred}(\underline{\texttt{a[0]}}, 0),$$
$$\texttt{noRHist}(\underline{\texttt{a[0..a.length-1]}}, 2), \texttt{noWHist}(\underline{\texttt{a[0..a.length-1]}}, 1) \Longrightarrow$$
$$[\texttt{while(i < a.length - 1) \{ a[i] = a[i+1]; i++;\}}]\phi$$

By turning the updates over a modality to formulas in the antecedent, we can use this information to refine the set of predicates.

4.2. Data Dependence Loop Invariant Generation with Predicate Abstraction

We propose a version of loop invariant generation by predicate abstraction [29] adapted to deductive verification and dependence analysis. It is a suitable technique, because (i) we work in a specific domain (data dependence) that suggests effective heuristics,

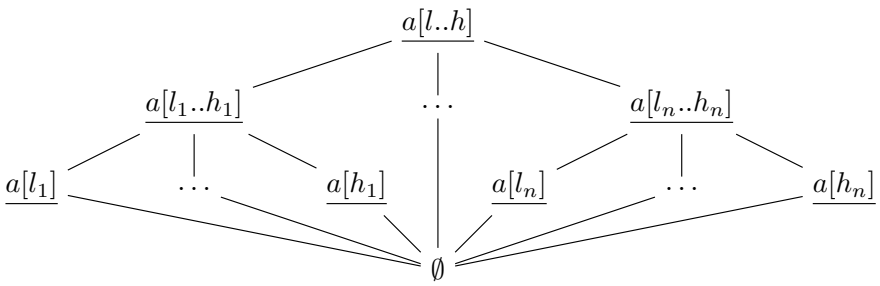


Figure 4.1.: Order between memory locations corresponding to array ranges and elements, where for all $1 \leq i \leq n$ it holds that $l \leq l_i \leq h_i \leq h$

(ii) data dependences are specified by predicates (Definition 3.3.1), (iii) there are abstract domains that provide appropriate approximation. We pursue a *bottom-up* approach, i.e. we start with the code under analysis to construct a loop invariant without the need to rely on external specifications (though these can be used, if available).

4.2.1. Predicate Abstraction

The set of all memory location *terms* is $\mathcal{D}^{\mathsf{LocSet}}$. The location terms occur as arguments of dependence predicates in loop invariants: Dependence properties are expressed in our logic with unary non-rigid predicates $\mathrm{PSym}^{Dep} = \{\mathsf{noRaW}, \mathsf{noWaR}, \mathsf{noWaW}, \mathsf{noR}, \mathsf{noW}\}$. These predicates take a set of memory locations as argument and specify that there are no flow, anti-, or output dependences and no read or write accesses to any of the memory locations in their argument, respectively. Again, the predicates PSym^{Dep} induce a lattice based on the partial order of read and write accesses they exclude. For example, if there is no read access on location set *loc*, there can not be any flow dependence (*RaW*) or anti-dependence (*WaR*) on any $loc' \subseteq loc$. Put differently, $\mathsf{noR}(loc)$ implies $\mathsf{noRaW}(loc')$ and $\mathsf{noWaR}(loc')$. An excerpt of the dependence predicate lattice is in Figure 4.2.

It is worth noting that predicates noAaR, and noAaW which mean no access after a read access and no access after write access are virtual predicates. They are not important in reasoning and are skipped in the refinement process.

In predicate abstraction the atomic formulas are equipped with an abstraction relation based on a partial order. For this we use the product of the dependence predicate lattice (for the predicate symbols) and the location set lattice (for their arguments). As both lattices are partially ordered, their product lattice is also partially ordered.

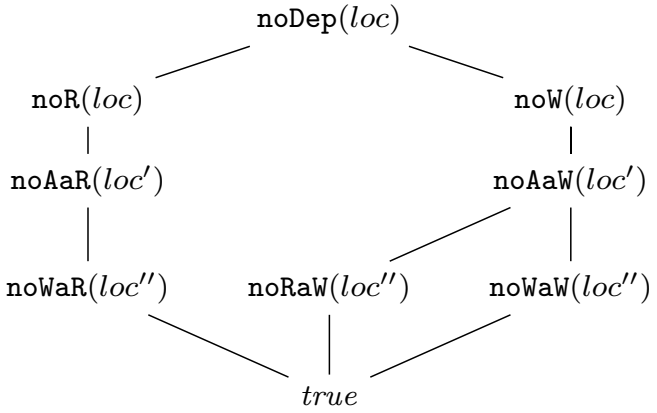


Figure 4.2.: Order between dependence predicates with locations sets $loc'' \subseteq loc' \subseteq loc$ induced by the lattice in Figure 4.1

Definition 4.2.1 (Abstract domain)**.** The abstract *domain* D^{Dep} consists of the atomic formulas constructed from PSym^{Dep} and $\mathcal{D}^{\texttt{LocSet}}$, i.e. $D^{Dep} = \{dp(loc) \mid dp \in \text{PSym}^{Dep}$ and loc is of type `LocSet`$\}$.

Definition 4.2.2 (Data Dependence abstract language)**.** The *abstract language* $L_{Dep} = \langle D^{Dep}, \sqsubseteq^{Dep} \rangle$ is the abstract domain D^{Dep} equipped with the partial order $\sqsubseteq^{Dep}$ that is induced by logical consequence.

Using our defined abstract language, the data dependence loop invariant for a loop over array a, is defined as follows:

Definition 4.2.3 (Data dependence loop invariant)**.** A *data dependence loop invariant* LI^{Dep} has the form $\bigwedge_i \ dp_i$, where $dp_i \in D^{Dep}$.

Concerning the functional properties required for sufficiently strong loop invariants, recall that we focus on dependence properties over arrays. In consequence, we restrict functional invariant generation to arithmetic constraints over array bounds and loop counters. This aspect is taken from the literature [29, 56].

Example 4.2.1. The precise data dependence loop invariant of the `while` loop in Listing 2.1 is

$$\texttt{noR}(\underline{\texttt{a[0]}} \cup \underline{\texttt{a[i+1..a.length-1]}}) \wedge$$
$$\texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge$$
$$\texttt{noRaW}(\underline{\texttt{a[0..a.length-1]}}) \ \wedge \ \texttt{noWaW}(\underline{\texttt{a[0..a.length-1]}})$$

The functional part of the loop invariant is $0 \leq \texttt{i} \leq \texttt{a.length} - 1$ which implies $\texttt{i} \doteq \texttt{a.length} - 1$ after the final iteration when the guard is false. The conjunction of both invariants constitutes the overall loop invariant.

The quality of the generated loop invariants depends on the level of abstraction embodied in the initial set of predicates provided for predicate abstraction. In the following section we describe how the current set of dependence predicates D^{Dep} is initialized and refined through symbolic execution.

4.2.2. Predicate Refinement

The initial set of data dependence predicates starts at the top element of the domain (Figure 4.2) and assumes there were no read or write accesses on the arrays in the code until then. For the example in Listing 2.1 the initial data dependence predicate set is

$$D_{\text{init}} = \{\texttt{noDep}(\underline{\texttt{a[0..a.length-1]}})\} \subseteq D^{Dep} \ .$$

To refine a given set of data dependence predicates D we use logic modeling tool box introduced in Section 4.1. This allows us to refine D while symbolically executing the loop body. After unwinding the loop, the sequent to be proven has the following shape:

$$\Gamma, \ldots \Longrightarrow \{u\}[\texttt{while (}b\texttt{) \{}body\texttt{\}}]\phi, \Delta$$

Update u contains the state changes (including memory accesses) accumulated during symbolic execution of the previous loop iteration. By applying sequent calculus rules shiftStateUpdate, shiftRead and shiftWrite, the concise information stored in u is represented in the form of formulas. This formulas are used to refine the set of predicates.

Algorithm 1 shows data dependence loop invariant generation. If D_{old} and D_{ref} differ then the antecedent of the current sequent is replaced by the latter and one further loop iteration is symbolically executed. The refinement cycle continues until the invariant candidate reaches a fixpoint. This means the dependence predicate set stays unchanged after some number of refinement steps.

The returned conjunction of predicates is a data dependence loop invariant. It is guaranteed to be correct, but not necessarily inductive, as it might fail to imply the postcondition.

Refinement is performed in function *refine* after each symbolic execution of a loop iteration and subsequent shifting of memory access updates. The task is to refine a set of memory access predicates $D \subseteq D^{Dep}$ until it becomes provable in the current sequent context. Each $dp \in D$ implied by the current sequent (without the program formula) stays

function *loopInvariantGenerator*
  input : Sequent $seq : pre \implies [\texttt{loop}]post$
  output : Loop Invariant
  D_{old}, $D_{ref} \leftarrow D_{\mathsf{init}}$, $\emptyset$;
  while $D_{old} \neq D_{ref}$ **do**
    $seq \leftarrow$ Apply rule unwindLoop on seq;
    $seq \leftarrow$ Symbolic execution on seq;
    $seq \leftarrow$ Merge branches below seq;
    /* seq has now the form $pre' \implies \{u\}[\texttt{loop}]post$ */
    $seq \leftarrow$ apply shift update rules on u in seq;
    D_{ref}, $D_{old} \leftarrow$ *refine*(seq, D_{old}), D_{ref};
    $seq \leftarrow \big(\bigwedge D_{ref} \implies [\texttt{loop}]post\big)$;
  end
  return $\bigwedge D_{ref}$;
function *refine*
  inputs : Sequent $seq : pre \implies [\texttt{loop}]post$
    Dependence invariant candidate D
  output : Refined candidate D_{ref}
  $D_{ref} \leftarrow \emptyset$;
  foreach $dp \in D$ **do**
    isProveable $\leftarrow$ *prove*$(pre \implies dp)$;
    if *isProveable* **then**
      $D_{ref} \leftarrow D_{ref} \cup \{dp\}$;
    else
      $D_{ref} \leftarrow D_{ref} \cup$ *refine*$(seq,$ *weaken*$(dp, seq))$;
    end
  end
  return D_{ref};

Algorithm 1: Data Dependence Loop Invariant Generation Algorithm

in D. Otherwise, dp is replaced by a weaker version. Weakening conforms to the lattices in Figures 4.1–4.2 and is described in Algorithm 2. Once we find a weaker predicate implied by the current sequent, it is added to D and we do not descend further in the lattice. If no provable weaker version is found, dp is replaced with the bottom element $true$, amounting to its removal from D.

Termination is ensured provided that after a finite number of weakening steps the bottom element (*true*) of the lattice in Figure 4.2 is reached for any $\mathsf{noX}(loc) \in D$. We achieve this by ensuring that the number of used location sets in LS, and hence, D_{Dep} is finite. For example, if $\mathsf{noR}(loc)$ is not implied by the current sequent, we try out $\mathsf{noRaW}(loc)$ and $\mathsf{noWaR}(loc)$, as well as $\mathsf{noR}(loc')$ for proper subsets $loc' \subset loc$. The loc'

function *weaken*

inputs : Data dependence predicate noX($\underline{a[low..high]}$)
Sequent $seq : pre \implies [\texttt{loop}]post$

output : Set of weaker predicates $weakened$

$weakened \leftarrow \emptyset$;

$index \leftarrow$ index from `loop`;

$step \leftarrow$ increment from `loop`;

if noX *is in* {noR, noW} **then**

  $weakened \leftarrow weakened \cup$ noRaW($\underline{a[low..high]}$) ;

  $weakened \leftarrow weakened \cup$ noWaR($\underline{a[low..high]}$) ;

end

if noX *is* noW **then**

  $weakened \leftarrow weakened \cup$ noWaW($\underline{a[low..high]}$) ;

end

if *first refinement iteration ;* `// done only once`

then

  $weakened \leftarrow$
  $weakened \cup$ noX($\underline{a[low]}$) $\cup$ noX($\underline{a[low + step..high - step]}$) $\cup$ noX($\underline{a[high]}$);

end

if $low < index < high$ **then**

  $weakened \leftarrow weakened \cup$ noX($\underline{a[low..index]}$);

  $weakened \leftarrow weakened \cup$ noX($\underline{a[index..high]}$);

end

return $weakened$

Algorithm 2: Dependence Predicate Weakening Heuristics

are determined heuristically based on trigger occurrences in the current sequent: If a predicate with location set $\underline{a[low..high]}$ needs to be weakened, we split the location set into $\underline{a[low]}$, $\underline{a[low + step..high - step]}$, and $\underline{a[high]}$, where we use the increment for the loop counter variable as value for $step$. To ensure termination we perform this refinement only once, otherwise splitting might not terminate, because low and $high$ are symbolic. The second split along the loop $index$ reflects the heuristics that location sets before the current loop index are accessed differently than locations after the loop index. For example, in the program of Listing 2.1 all array elements up to index $i + 1$ have been read, while noR($\underline{a[i + 2..a.length - 1]}$) holds.

4.3. Reasoning

For reasoning, we extend JavaDL$_{Dep}$ calculus to support the newly introduced updates and predicates.

4.3.1. Verification of the Data Dependence Loop Invariant

We described a data dependence loop invariant generation process. If we obtain a conjectured set D of data dependence predicates for a given piece of code `prg`, then it is easily possible to verify (or disprove) it with the logic developed here.

It is sufficient to encode D as a data dependence loop invariant, i.e. as a conjunction of dependence predicates and prove the sequent $pre,\ D,\ I \Longrightarrow [\texttt{prg}](D \wedge I)$, where pre contains possible assumptions on initial values of `prg` and I is the functional invariant containing boundary values.

In the following, we mention a selection of sequent calculus rules developed for realizing the logic behind data dependence loop invariant generation process. These rules reflect the chronological order of memory access updates.

4.3.2. Renamed Memory Access Update Application

Intuitively, application of a renamed memory access update *ages* memory access and history data dependence predicates. Rules renamedReadAppOnNoR, renamedReadAppOnReadPred and renamedReadAppOnNoRHist in Figure 4.3 are examples of the rules reflecting this effect. Rules for application of renamed read memory access update on the rest of memory access and history data dependence predicates are analogous, see Appendix B.3. Rules for application of renamed write memory access update are analogous and therefore omitted.

4.3.3. Subsumption Relations

In addition to subsumption relations depicted in Figure 4.2 and realized by rules like knownNoR, knownNoRaW and etc. (Subsection 3.4.3), there are subsumption relations between history data dependence predicates. Figure 4.4 shows examples of such rules. Intuitive meaning of them is as following:

- Rule schema noRImpliesNoRHist states that if predicate noR(loc) holds after n memory accesses of Acc, it also holds after lb memory accesses in it where $lb < n$.

renamedReadAppOnNoR

$$\{\backslash \mathsf{R}'(rLoc)\}\mathsf{noR}(loc) \rightsquigarrow \mathtt{noRHist}(\{\backslash \mathsf{R}'(rLoc)\}loc, 1)$$

renamedReadAppOnReadPred

$$\{\backslash \mathsf{R}'(rLoc)\}\mathtt{rPred}(loc, lb) \rightsquigarrow \mathtt{rPred}(\{\backslash \mathsf{R}'(rLoc)\}loc, \{\backslash \mathsf{R}'(rLoc)\}lb + 1)$$

renamedReadAppOnNoRHist

$$\{\backslash \mathsf{R}'(rLoc)\}\mathtt{noRHist}(loc, lb) \rightsquigarrow \mathtt{noRHist}(\{\backslash \mathsf{R}'(rLoc)\}loc, \{\backslash \mathsf{R}'(rLoc)\}lb + 1)$$

Figure 4.3.: Selection of rules for renamed memory access update application

noRImpliesNoRHist $\quad \mathsf{noR}(loc) \rightsquigarrow \mathtt{noRHist}(loc, lb)$

noRHistOnBothSides

$$\frac{\Gamma, \mathtt{noRHist}(loc_1, lb_1), lb_1 \leq lb_2 \Longrightarrow \mathtt{noRHist}(loc_2 \setminus loc_1, lb_2), \Delta \quad \Gamma, \mathtt{noRHist}(loc_1, lb_1), lb_1 > lb_2 \Longrightarrow \mathtt{noRHist}(loc_2, lb_2), \Delta}{\Gamma, \mathtt{noRHist}(loc_1, lb_1) \Longrightarrow \mathtt{noRHist}(loc_2, lb_2), \Delta}$$

Figure 4.4.: Excerpts of subsumption relations between history data dependence predicates

- In the rule noRHistOnBothSides, having $lb_1 \leq lb_2$ and consequently $n - lb_1 \geq n - lb_2$ infers that $\mathtt{noRHist}(loc_1, lb_1)$ takes into account a larger sequent of memory accesses therefore it has more information than $\mathtt{noRHist}(loc_2, lb_2)$ regarding the read memory accesses on $loc_1 \cap loc_2$.

Subsumption relations between other predicates are analogous and can be found in Appendix B.4.

4.3.4. Embedding Predicate Abstraction

Intuitively, the formula $\mathtt{rPred}(loc, lb)$ holds in a state

$$s = (\sigma, \langle M_1(loc_1)\rangle \ldots \langle M_{n-lb}(loc_{n-lb})\rangle \ldots \langle M_n(loc_n)\rangle)$$

if and only if $M_{n-lb} = \mathit{Read}$ and $loc_{n-lb} = loc$. In other words, the formula $\mathtt{rPred}(loc, lb)$ expresses that the lb-th most recent memory access is a read access on exactly the locations

$$\underbrace{\langle M_1(loc_1)\rangle \circ \ldots \circ \langle M_{n-(lb+1)}\rangle}_{\texttt{noRHist}(loc,lb+1)} \circ \overbrace{\langle M_{n-lb}(loc_{n-lb})\rangle}^{\texttt{rPred}(loc,lb)} \circ \langle M_{n-(lb-1)}(loc_{n-(lb-1)})\rangle \circ \ldots \circ \langle M_n(loc_n)\rangle$$

Figure 4.5.: Subsequences of accesses referred to by $\texttt{rPred}(loc, lb)$ and $\texttt{noRHist}(loc, lb+1)$ $(0 \leq lb < n)$

readPredANDnoRHistSameLabel

$$\frac{\Gamma, \texttt{rPred}(loc_1, lb), \texttt{noRHist}(loc_2, lb), loc_1 \cap loc_2 \doteq \emptyset \Longrightarrow \phi, \Delta}{\Gamma, \texttt{rPred}(loc_1, lb), \texttt{noRHist}(loc_2, lb) \Longrightarrow \phi, \Delta}$$

readPredANDnoRHistRightSameLabel

$$\frac{\Gamma, \texttt{rPred}(loc_1, lb) \Longrightarrow loc_1 \cap loc_2 \doteq \emptyset \wedge \texttt{noRHist}(loc_2, lb+1), \Delta}{\Gamma, \texttt{rPred}(loc_1, lb) \Longrightarrow \texttt{noRHist}(loc_2, lb), \Delta}$$

readPredAFTERnoRHist

$$\frac{\Gamma, \texttt{rPred}(loc_1, lb), \texttt{noRHist}(loc_2, lb+1), \texttt{noRHist}(loc_2 \setminus loc_1, lb) \Longrightarrow \phi, \Delta}{\Gamma, \texttt{rPred}(loc_1, lb), \texttt{noRHist}(loc_2, lb+1) \Longrightarrow \phi, \Delta}$$

Figure 4.6.: Relation of rPred and noRHist rules

in loc. Similarly, the formula $\texttt{noRHist}(loc, lb+1)$ holds in state s if there has been no read access on any location in loc up to and including $M_{n-(lb+1)}$. Figure 4.5 visualizes the subsequences of memory accesses that determine the validity of formulas.

Meaningful relations between access predicates and history data dependence predicates can be established only when they have the same label or the difference is 1. Otherwise, a rule has to consider an arbitrary number of memory accesses between the two points described by an access predicate and a history data dependence predicate.

Relation of rPred and noRHist

Figure 4.6 shows the rules needed for establishing the logical relation between rPred and noRHist.

readPredANDnoWHistRightSameLabel

$$\frac{\Gamma, \texttt{rPred}(loc_1, lb) \Longrightarrow \texttt{noWHist}(loc_2, lb+1), \Delta}{\Gamma, \texttt{rPred}(loc_1, lb) \Longrightarrow \texttt{noWHist}(loc_2, lb), \Delta}$$

readPredAFTERnoWHist

$$\frac{\Gamma, \texttt{rPred}(loc_1, lb), \texttt{noWHist}(loc_2, lb+1), \texttt{noWHist}(loc_2, lb) \Longrightarrow \phi, \Delta}{\Gamma, \texttt{rPred}(loc_1, lb), \texttt{noWHist}(loc_2, lb+1) \Longrightarrow \phi, \Delta}$$

Figure 4.7.: Relation of rPred and noWHist rules

In state s, predicate $\texttt{noRHist}(loc_2, lb)$ indicates that there has not been a read memory access on loc_2 from M_1 until and including M_{n-lb}, while $\texttt{rPred}(loc_1, lb)$ states that M_{n-lb} is a read memory access. If $\texttt{noRHist}(loc_2, lb)$ holds this access should not be on loc_2 or its subsets ($loc_1 \cap loc_2 \doteq \emptyset$). Rule readPredANDnoRHistSameLabel reflects this fact.

For proving $\texttt{noRHist}(loc_2, lb)$ in addition to M_{n-lb}, none of the accesses before it should a read on loc_2 or its subsets. This information is aggregated in $\texttt{noRHist}(loc_2, lb+1)$. Rule readPredANDnoRHistRightSameLabel concerns this matter.

Rule schema readPredAFTERnoRHist states that having $\texttt{rPred}(loc_1, lb+1)$, predicate $\texttt{noRHist}(loc_2, lb)$ would hold if it has not been violated by the corresponding memory access of $\texttt{rPred}(loc_1, lb+1)$.

Rules for establishing the relation of wPred and noWHist are analogous, and can be seen in Appendix B.5.1.

Relation of rPred and noWHist

Rules for establishing the relation of rPred and noWHist are shown in Figure 4.7. Rule schema readPredANDnoWHistRightSameLabel states that two different memory accesses can not happen at the same time. If M_{n-lb} is a read, for proving $\texttt{noWHist}(loc_2, lb)$ we only need to show that there has not been a write access on loc_2 *up to* M_{n-lb}. Rule schema readPredAFTERnoWHist shows that knowing there has not been a write access on loc_2 up to M_{n-lb}, and M_{n-lb} is read we can deduce that $\texttt{noWHist}(loc_2, lb)$.

Rules for establishing the relation of wPred and noRHist are analogous and included in Appendix B.5.2.

Relation of `rPred` and `noRaWHist`

Rules for establishing the relation of `rPred` and `noRaWHist` are shown in Figure 4.8. We explain the intuitive meaning of these rules below:

- Rule readPredANDnoRaWHistSameLabel: Knowing that at $n - lb$ in Acc there is a read access on loc_1 and at the same time `noRaWHist` holds for loc_2, we can deduce that `noWHist` holds for $loc_1 \cap loc_2$ *up to* $n - lb$.

- Rule readPredANDnoRAWHistoryRightSameLabel: To show `noRaWHist` while having a read access at the same time, for memory locations that are not accessed we need need to show `noRaWHist` *up to* this access. For the $loc_1 \cap loc_2$ we only need to show that `noWHist` holds right before the read access.

- Rule readPredBEFOREnoRaWHist: If `noRaWHist` holds at a point $(n - lb)$ in Acc and right before this point $((n - lb) + 1)$ there has been a read, we can deduce that before this access $((n - lb) + 1 + 1)$ predicate `noWHist` held on $loc_1 \cap loc_2$.

- Rule readPredBEFOREnoRaWHistRight: To show that `noRaWHist` holds at a point $(n - lb)$ in Acc while knowing that right before this point $((n - lb) + 1)$ there has been a read, we need to show that before this access $((n - lb) + 1 + 1)$ predicate `noWHist` held on $loc_1 \cap loc_2$.

Rules for establishing the relation of `wPred` with `noWaRHist` and `noWaWHist` are analogous. These rules can be seen in Appendices B.5.3 and B.5.4, respectively.

The Curious Case of `wPred`

There is a special rule for relating two `wPred` to each other. They must indicate write accesses to the same location sets if they have the same label. The reason is that the labels are unique. The intersection between their accessed location sets must be checked in the case of different labels. If this intersection is non-empty, then `noWaWHist` does not hold for it from when the most recent write access has happened. Rule schema writePredANDwritePred shows this fact.

readPredANDnoRaWHistSameLabel

$$\frac{\Gamma, \texttt{rPred}(loc_1, lb), \texttt{noRaWHist}(loc_2, lb), \texttt{noWHist}(loc_1 \cap loc_2, lb+1) \Longrightarrow \phi, \Delta}{\Gamma, \texttt{rPred}(loc_1, lb), \texttt{noRaWHist}(loc_2, lb) \Longrightarrow \phi, \Delta}$$

readPredANDnoRaWHistRightSameLabel

$$\frac{\Gamma, \texttt{rPred}(loc_1, lb) \Longrightarrow \texttt{noRaWHist}(loc_2 \setminus loc_1, lb+1) \wedge \texttt{noWHist}(loc_1 \cap loc_2, lb+1), \Delta}{\Gamma, \texttt{rPred}(loc_1, lb) \Longrightarrow \texttt{noRaWHist}(loc_2, lb), \Delta}$$

readPredBEFOREnoRaWHist

$$\frac{\Gamma, \texttt{rPred}(loc_1, lb+1), \texttt{noRaWHist}(loc_2, lb), \ \texttt{noWHist}(loc_1 \cap loc_2, lb+2) \Longrightarrow \phi, \Delta}{\Gamma, \texttt{rPred}(loc_1, lb+1), \texttt{noRaWHist}(loc_2, lb) \Longrightarrow \phi, \Delta}$$

readPredBEFOREnoRaWHistRight

$$\frac{\Gamma, \texttt{rPred}(loc_1, lb+1) \Longrightarrow \texttt{noRaWHist}(loc_2 \setminus loc_1, lb) \wedge \texttt{noWHist}(loc_1 \cap loc_2, lb+2), \Delta}{\Gamma, \texttt{rPred}(loc_1, lb+1) \Longrightarrow \texttt{noRaWHist}(loc_2, lb), \Delta}$$

Figure 4.8.: Relation of rPred and noRaWHist rules

writePredANDwritePred

$$\frac{\begin{array}{l}\Gamma, \mathtt{wPred}(loc_1, lb_1), \mathtt{wPred}(loc_2, lb_2), lb_1 \doteq lb_2, loc_1 \doteq loc_2 \Longrightarrow \phi, \Delta \\ \\ \Gamma, \mathtt{wPred}(loc_1, lb_1), \mathtt{wPred}(loc_2, lb_2), lb_1 \not\doteq lb_2, \\ \quad loc_1 \cap loc_2 \not\doteq \emptyset \ \rightarrow \\ \forall\, \mathtt{LocSet}\ loc \in\ loc_1 \cap loc_2;\ \neg \mathtt{noWaWHist}(loc, \min(lb_1, lb_2)) \Longrightarrow \phi, \Delta\end{array}}{\Gamma, \mathtt{wPred}(loc_1, lb_1), \mathtt{wPred}(loc_2, lb_2) \Longrightarrow \phi, \Delta}$$

Theorem 3.4.1 and Corollary 3.4.1 still hold after adding the new calculus rules introduced in this section. As an example, we provide the proof of soundness and completeness of rule renamedReadAppOnNoRHist in Appendix C.5.

5. Automatic Loop Invariant Generation for Inter-Iteration Data Dependence Analysis

So far we generate and verify logical formulas expressing the absence of data dependences on memory location sets for a given piece of code. For example, in case of noRaW(ls), there is *no* execution that contains first a write on a location in $l \in ls$ that is later followed by a read on l.

This is a rather strong property that might preclude parallelization in cases, where it is actually warranted. Assume noRaW(ls) does *not* hold for a given loop body over ls. As long as the *RaW* happens *within* one loop iteration, the loop is still parallelizable. Only when a write happens to a location $l \in ls$ in one iteration and a read from l is a subsequent iteration, there is a problem. Data dependences spanning over more than one iteration are called *inter-iteration*.Those that are confined to the same iteration are called *intra-iteration* dependence.

Our approach (Chapter 4) can be extended to permit intra-iteration dependences, but to exclude critical inter-iteration dependences. It is achieved by relaxing noX dependence predicates, so they distinguish between different loop iterations and tolerate intra-iteration dependences.

The loop invariant generation algorithm for JavaDL$_{Dep}$ can now be reused for JavaDL$_{\widehat{Dep}}$, an extension of JavaDL$_{Dep}$ for reasoning about inter-iteration data dependences. It is merely necessary to (i) replace the unwindLoop rule by a version that also marks the start of a new iteration, (ii) a shifting rule for the new kind of updates, as well as (iii) axiomatization and simplification rules for the new data inter-dependence predicates.

In Section 5.1 we briefly compare inter- and intra-iteration data dependences. We explain our approach in specifying intra-iteration data dependences in Section 5.2. Updating the data dependence loop invariant generation algorithm with necessary changes is the subject of Section 5.3. Developing sequent calculus rules for reasoning about intra-iteration data dependences is covered in Section 5.4.

5.1. Inter- vs. Intra-Iteration Loop Data Dependences

The data dependence analysis approach introduced in the previous chapters can analyze the data dependences whether they are within one loop iteration or spread across different iterations. In pattern based parallelization, depending on the parallelization pattern, some data dependences are more important than the others. If the pattern parallelizes the loop body (e.g., Pipeline pattern), the data dependence profiler must extract data dependences inside the body. On the other hand, the conventional way of parallelizing loops is to divide the iteration space and run the sub-loops in parallel (e.g., DoAll and Loop Splitting patterns). In this case, there is no need to profile data dependences occurring inside the loop body, as it is executed sequentially. In such applications only the data dependences between different loop iterations matter. Data dependences spanning over more than one iteration are called *inter-iteration*, *cross-iteration*, or *loop-carried* dependence.

5.2. Specification of Intra-Iteration Data Dependence Properties

The granularity of tracking data dependences of JavaDL$_{Dep}$ as presented in Chapters 3 and 4 is too fine-grained for certain loop parallelization patterns, because it tracks *all* data dependences. For example, a simple loop parallelization that assigns each loop iteration its own process requires that there are no data dependences between *two different* loop iterations, but is oblivious to data dependences *within* the same iteration. The approach sketched in Chapters 3 and 4 is not able to distinguish between intra- and inter-iteration dependences and prevents taking advantage of such a parallelization opportunity.

Therefore we extend JavaDL$_{Dep}$ by adding support for *inter-iteration* data dependence predicates that permit dependences stretching between different loop iterations.

We also adapt the loop invariant generation algorithm (Algorithm 1) to generate corresponding loop invariants.

5.2.1. Syntax and Semantics

We extend the logic JavaDL$_{Dep}$ to JavaDL$_{\widehat{Dep}}$ for reasoning about inter-iteration dependences. To specify and reason about inter-iteration dependences, JavaDL$_{\widehat{Dep}}$ provides three additional non-rigid predicates: $\widehat{\texttt{noRaW}}$, $\widehat{\texttt{noWaR}}$ taking four arguments of type `LocSet`, and $\widehat{\texttt{noWaW}}$ with three arguments of type `LocSet`. Their semantics is like the dependence predicates without $\widehat{\cdot}$, except that they are only sensitive to data dependences that stretch over different loop iterations. The extra arguments express history contexts and will be explained in Definition 5.2.3.

```
1         sum = 0; i = 0;
2         while (i < a.length) {
3             a[i] = b[i] + c[i];
4             sum += a[i];
5             i=i+1;
6         }
```

Listing 5.1: Program `arraySum`: component-wise array addition

Example 5.2.1. Consider the program `arraySum` in Listing 5.1, which adds the content of arrays `b` and `c` component-wise and stores the result in array `a`. In addition, it computes the sum over all elements stored in `a`. Formula $\texttt{noRaW}(\underline{\texttt{a[0..a.length]}})$ does not hold after execution of `arraySum`, as array element `a[i]` is written in line 3 and read in line 4. In contrast, $\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..a.length]}}, \texttt{empty}, \texttt{empty}, \texttt{empty})$ expresses that there are no read-after-write dependences *across* loop iterations and it holds after execution of the program.

As different iterations need to be distinguished, the memory access sequence of JavaDL$_{\widehat{Dep}}$ states must provide more structure than JavaDL$_{Dep}$. Instead of a sequence of memory accesses, JavaDL$_{\widehat{Dep}}$ uses a sequence of sequences of memory accesses $\widehat{Acc}$. Each element in $\widehat{Acc}$ (i.e. a sequence of memory accesses) consists of the memory accesses of a single loop iteration.

Definition 5.2.1 (Domain, JavaDL$_{\widehat{Dep}}$ State)**.** Given a non-empty *domain* $\mathcal{D}$, a JavaDL$_{\widehat{Dep}}$ state $s = (\sigma, \widehat{Acc})$ is a pair of

- an interpretation σ as in JavaDL$_{Dep}$ (Def. 3.1.1) and
- a finite sequence $\widehat{Acc} = \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_n$ of sequences of memory read and write accesses $\widehat{acc}_i ::= acc_{i,1} \circ \ldots \circ acc_{i,m}$, where $acc_{i,j} ::= \langle Read(ls) \rangle \mid \langle Write(ls) \rangle$, $ls \in \mathcal{D}^{LocSet}$, and $\bigcirc$ is the concatenation between elements of $\widehat{Acc}$ (using big circles saves brackets and helps readability).

The semantics is defined in Figure 5.1.

Executing an assignment $x = t$ from side-effect free[1] expression t to a local variable x in a state $s = (\sigma, \widehat{Acc})$ terminates in a state s' which is almost identical to state s except

[1] Side-effect free has its usual meaning: not changing the state. Although memory access updates change the state, we consider them side-effect free.

$$
\begin{aligned}
val_{\mathcal{K},s,\beta}(x = t) \quad := \{(s,s') \mid \;& s' = (\sigma', \widehat{Acc}') \text{ with } s = (\sigma, \widehat{Acc}), \sigma' = \sigma[x \leftarrow val_{\mathcal{K},s,\beta}(t)] \text{ and} \\
& \widehat{Acc}' = \widehat{Acc} \circ \langle acc_{\mathcal{K},s,\beta}(t) \rangle = \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_n \circ acc_{\mathcal{K},s,\beta}(t) \\
& = \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_{n-1} \bigcirc (acc_{n,1} \circ \cdots \circ acc_{n,m} \circ acc_{\mathcal{K},s,\beta}(t))\} \\
& (x \in \mathsf{PV})
\end{aligned}
$$

$$
\begin{aligned}
val_{\mathcal{K},s,\beta}(a[k] = t) \quad := \{(s,s') \mid \;& s' = (\sigma', \widehat{Acc}') \text{ with } s = (\sigma, \widehat{Acc}), \sigma' = \sigma[a[k] \leftarrow val_{\mathcal{K},s,\beta}(t)] \\
& \text{and} \\
& \widehat{Acc}' = \widehat{Acc} \circ acc_{\mathcal{K},s,\beta}(t) \circ acc_{\mathcal{K},s,\beta}(k) \circ \langle Write(\underline{a[k]}) \rangle\}
\end{aligned}
$$

$$
\begin{aligned}
& val_{\mathcal{K},s,\beta}(\texttt{if}(b)\ \{stmnt_1\}\ \texttt{else}\ \{stmnt_2\}) \\
& \quad := \{(s,s'') \mid \; s' = (\sigma, \widehat{Acc}') \text{ with } s = (\sigma, \widehat{Acc}) \text{ and } \widehat{Acc}' = \widehat{Acc} \circ acc_{\mathcal{K},s,\beta}(b), \\
& \qquad \text{and } s'' = (\sigma, \widehat{Acc}'') \text{ where} \\
& \qquad ((\mathcal{K}, s', \beta \models b, (s', s'') \in [\![stmnt_1]\!]) \\
& \qquad \vee (\mathcal{K}, s', \beta \not\models b \text{ and } (s', s'') \in [\![stmnt_2]\!]))\}
\end{aligned}
$$

$$
\begin{aligned}
& val_{\mathcal{K},s,\beta}(\texttt{while}(b)\{stmnt\}) \\
& \quad := \{(s,t) \mid \text{there is a sequence } s = s_0\ s'_0\ \cdots\ s_n\ = t \text{ with} \\
& \qquad \mathcal{K}, s_i, \beta \models b,\ i < n \text{ and } \mathcal{K}, t, \beta \not\models b \text{ and} \\
& \qquad (s'_i, s_{i+1}) \in [\![stmnt]\!] \text{ with } s'_i = (\sigma_i, \widehat{Acc}_i \bigcirc acc_{\mathcal{K},s'_i,\beta}(b))
\end{aligned}
$$

$$
(\widehat{Acc} = \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_n \text{ and } \widehat{acc}_i = acc_{i,1} \circ \cdots \circ acc_{i,m_i},\ i \in \{1 \ldots n\})
$$

Figure 5.1.: Program semantics of JavaDL$_{\widehat{Dep}}$

for the value of variable x *and* the possible extension of read memory accesses resulting from evaluating t. As an assignment inside a loop body does neither start nor end the current loop iteration, the reads are appended to the sequence of memory accesses of the *current* iteration (i.e., the last element of $\widehat{Acc}$). Similarly for $val_{\mathcal{K},s,\beta}(a[k] = t)$ and $val_{\mathcal{K},s,\beta}(\texttt{if}(b)\ \{stmnt_1\}\ \texttt{else}\ \{stmnt_2\})$. In contrast, a loop *ends* the current iteration and *starts* a new one with each evaluation of its guard. This is reflected in the semantics by appending a *new* sequence to $\widehat{Acc}_i$ at start of each loop iteration i. A minor observation is that on loop exit the final evaluation of the loop guard (evaluating to `false`) starts a *new* iteration and thus ends the current one.

It remains to define the semantics for the inter-iteration dependence predicates, which will enable us to specify and reason about inter-iteration data dependences. To streamline the definition of their semantics, we first introduce two projection functions that aggregate the locations of all read/write accesses for a given sequence of memory accesses:

Definition 5.2.2 (Projection)**.** For $\widehat{acc} = acc_1 \circ \ldots \circ acc_n$ the projection function for read accesses $\downarrow_R$ defined as

$$\downarrow_R \widehat{acc} := \{ls \mid ls = \bigcup_{i=1}^{n} ls_i \text{ where } acc_i = \langle Read(ls_i)\rangle\}$$

collects all memory locations read in loop iteration $\widehat{acc}$. The projection function for write accesses $\downarrow_W$ is defined analogously:

$$\downarrow_W \widehat{acc} := \{ls \mid ls = \bigcup_{i=1}^{n} ls_i \text{ where } acc_i = \langle Write(ls_i)\rangle\}.$$

Example 5.2.2. Let $\widehat{acc} := \langle Read(\underline{\texttt{a[0]}})\rangle \circ \langle Write(\underline{\texttt{a[1]}})\rangle \circ \langle Read(\underline{\texttt{a[2]}})\rangle$ be a sequence of memory accesses. Then $\downarrow_R \widehat{acc} = \{\underline{\texttt{a[0]}}, \underline{\texttt{a[2]}}\}$ and $\downarrow_W \widehat{acc} = \{\underline{\texttt{a[1]}}\}$.

Definition 5.2.3 (Inter-iteration data dependence predicates: semantics)**.** Let *ls*, *rLs*, *wLs*, *futRLs* and *futWLs* denote location sets in $\mathcal{D}^{\mathsf{LocSet}}$. The *semantics* of predicate symbols $\widehat{\mathsf{noRaW}}$ (similar for predicate $\widehat{\mathsf{noWaR}}$, see Appendix A.3) and $\widehat{\mathsf{noWaW}}$ is defined as follows:

$$\begin{aligned} s(\widehat{\mathsf{noRaW}}) = \{(ls, rLs, wLs, futRLs) \mid\ & s = (\sigma, \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_n), \\ & rLs, wLs, futRLs \in \mathcal{D}^{\mathsf{LocSet}} \text{ such that } ls \cap (wLs \cup \downarrow_W \widehat{acc}_n) \cap futRLs = \texttt{empty} \\ & \text{and} \\ & \text{if } n > 1:\ (ls, \texttt{empty}, \texttt{empty}, futRLs \cup rLs \cup \downarrow_R \widehat{acc}_n) \in s'(\widehat{\mathsf{noRaW}}) \\ & \qquad \text{with } s' = (\sigma, \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_{n-1})\} \end{aligned}$$

$$s(\widehat{\texttt{noWaW}}) = \{(ls, \mathit{wLs}, \mathit{futWLs}) \mid s = (\sigma, \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_n,), \mathit{wLs} = \downarrow_W \widehat{acc}_n,$$
$$\mathit{wLs} \text{ and } \mathit{futWLs} \in \mathcal{D}^{\texttt{LocSet}} \text{ such that } ls \cap \mathit{wLs} \cap \mathit{futWLs} = \texttt{empty} \text{ and}$$
$$\text{if } n > 1 : \ (ls, \downarrow_W \widehat{acc}_n, \mathit{futWLs} \cup \mathit{wLs}) \in s'(\widehat{\texttt{noWaW}})$$
$$\text{with } s' = (\sigma, \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_{n-1})\}$$

The second and third argument of $\widehat{\texttt{noRaW}}$ are the read and write accesses encountered so far in the current iteration, the fourth argument accumulates the read accesses in the already processed iterations in the recursive step. When $ls \cap (\mathit{wLs} \cup \downarrow_W \widehat{acc}_n) \cap \mathit{futRLs} = \texttt{empty}$ in already processed iterations there is no read access on the location sets that are written in the current iteration. At the end of processing each iteration, arguments *rLs* and *wLs* are reset to `empty` and by $\mathit{futRLs} \cup \mathit{rLs} \cup \downarrow_R \widehat{acc}_n$ we accumulates all the location sets that have been subjected to a read access during this iteration in argument *futRLs*.

Example 5.2.3. Consider the states

$$s_1 = (\sigma_1, \langle \mathit{Read}(\underline{a[0]})\rangle \bigcirc \langle \mathit{Write}(\underline{a[0]})\rangle \bigcirc \langle\rangle) \text{ and}$$
$$s_2 = (\sigma_2, \langle \mathit{Write}(\underline{a[0]})\rangle \bigcirc \langle \mathit{Read}(\underline{a[0]})\rangle \bigcirc \langle\rangle)$$

both of which just completed an iteration. To determine that there is no read-after-write on array element `a[0]`, i.e. formula $\widehat{\texttt{noRaW}}(\underline{\texttt{a[0]}}, \texttt{empty}, \texttt{empty}, \texttt{empty})$ is satisfied in s_i, we establish $(\underline{\texttt{a[0]}}, \texttt{empty}, \texttt{empty}, \texttt{empty}) \in s_i$.[2] As the final memory access sequence is empty, the conditions before the recursive step in the semantics are obviously satisfied. For the first recursive step, we check

$$(\underline{a[0]}, \texttt{empty}, \texttt{empty}, \texttt{empty}) \in s_1'(\widehat{\texttt{noRaW}})$$

with $s_1' = (\sigma_1, \langle \mathit{Read}(\underline{a[0]})\rangle \bigcirc \langle \mathit{Write}(\underline{a[0]})\rangle)$. This is straightforward and it remains to check

$$(\underline{a[0]}, \texttt{empty}, \texttt{empty}, \texttt{empty}) \in s_1''(\widehat{\texttt{noRaW}})$$

with $s_1'' = (\sigma_1, \langle \mathit{Read}(\underline{a[0]})\rangle)$. This holds as $\underline{a[0]} \cap (\texttt{empty} \cup \texttt{empty}) \cap \texttt{empty} = \texttt{empty}$.

For state s_2 we have to check instead in the final recursive step that

$$(\underline{a[0]}, \texttt{empty}, \texttt{empty}, \underline{a[0]}) \in s_2''(\widehat{\texttt{noRaW}})$$

with $s_2'' = (\sigma_2, \langle \mathit{Write}(\underline{a[0]})\rangle)$. This does not hold as $\underline{a[0]} \cap (\texttt{empty} \cup \underline{a[0]}) \cap \underline{a[0]} = \underline{a[0]} \neq \texttt{empty}$.

[2] In the rewrite rules we also need other arguments than `empty`.

Inter-Iteration History Data Dependence Predicates

$$
\begin{aligned}
s(\widehat{\texttt{noRHist}}) \quad &= \{(loc, label) \mid s = (\sigma, \widehat{Acc}),\ \widehat{Acc} = \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_n, \\
&\qquad 0 \leq label < n, \text{ and for all } 0 < i \leq n - label \\
&\qquad \text{it holds that } \downarrow_R \widehat{acc}_i \cap loc = \texttt{empty}\} \\
s(\widehat{\texttt{noRaWHist}}) \quad &= \{(loc, label) \mid s = (\sigma, \widehat{Acc}),\ \widehat{Acc} = \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_n, \\
&\qquad 0 \leq label < n, \text{ and for all } 0 < i < j \leq n - label \\
&\qquad \text{it holds that } \downarrow_R \widehat{acc}_j \cap \downarrow_W \widehat{acc}_i \cap loc = \texttt{empty}\} \\
s(\widehat{\texttt{rPred}}) \quad &= \{(loc, label) \mid s = (\sigma, \widehat{Acc}),\ \widehat{Acc} = \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_n, \\
&\qquad 0 \leq label < n, \\
&\qquad loc = \downarrow_R \widehat{acc}_{n-label}\}
\end{aligned}
$$

$\widehat{\texttt{noWHist}}, \widehat{\texttt{noWaRHist}}, \widehat{\texttt{noWaWHist}}, \widehat{\texttt{wPred}}$ are defined analogously in Appendix A.4

Figure 5.2.: Semantics of selected inter-iteration history data dependence predicates

In updates we need to represent that a new iteration starts (i.e., the previous iteration ends) syntactically to group memory access updates. This is done with a "marker" update $\backslash\texttt{next}$, between which the memory access updates for a single iteration are grouped.

Definition 5.2.4 (Marker update semantics)**.** The semantics of marker update $\backslash\texttt{next}$ is defined as

$$val_{\mathcal{K},s,\beta}(\backslash\texttt{next}) = s' \text{with } s' = (\sigma, \widehat{Acc} \bigcirc \langle\varepsilon\rangle) \text{ and } s = (\sigma, \widehat{Acc})$$

and thus starts a new iteration by appending a new empty sequence of memory accesses[3] to $\widehat{Acc}$.

During loop invariant generation (not for using loop invariants), we need predicates to express up to which iteration a given data dependence property holds. We need to

[3]Note that $\langle\varepsilon\rangle$ is not an empty sequent but rather a sequent that contains an empty sequent.

define inter-iteration versions of the history predicates (see Section 4.1) as well as the corresponding read and write predicates. These predicates are $\widehat{\texttt{noRaWHist}}$, $\widehat{\texttt{noWaRHist}}$, $\widehat{\texttt{noWaWHist}}$, $\widehat{\texttt{noRHist}}$, $\widehat{\texttt{noWHist}}$, as well as $\widehat{\texttt{rPred}}$ and $\widehat{\texttt{wPred}}$. All have the type signature $\texttt{LocSet} \times \texttt{int}$. For example, $\widehat{\texttt{noRaWHist}}(ls, i)$ expresses that no read-after-write access occurs on the locations in ls up to the i-th preceding *iteration*.[4] The semantics is summarized in Figure 5.2 together with that of the $\backslash\texttt{next}$ update and $\backslash\texttt{next}'$ update, which is needed to shift $\bigcirc$ (see Section 5.4).

5.3. Loop Invariant Generation

For analyzing and verifying inter-iteration data dependences in programs with loops we need to synthesize inter-iteration data dependences that hold in each loop iteration.

The setting is same as in Chapter 4: (i) analyzing terminating programs as it is the case in Parallelizable HPC programs, (ii) concentrating on inter-iteration data dependences among array types as the most relevant data structure in HPC.

We use the same approach for loop invariant generation as in Chapter 4. Let a set of abstract predicates D contain formulas that express inter-iteration data dependence properties known to hold in the state before executing the loop. Now the loop is unwound once. Afterwards we check which of the formulas in D hold in the state reached. The provable ones are kept in D, the others are replaced with weakened versions (determined using an abstraction lattice) that can be proven. Then the process restarts with unwinding the loop once more, until the set D is stable. Using a finite lattice and appropriate weakening operations, this happens after finitely many rounds.

5.3.1. Symbolic Execution

Instead of the usual unwindLoop, we define the following rule:

markedUnwindLoop

$$\frac{\Gamma \Longrightarrow \{u\}\{\backslash\texttt{next}\}[\texttt{if (}b\texttt{) \{}s\texttt{; while (}b\texttt{) \{}s\texttt{\}\}}\ r\texttt{;}]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\texttt{while (}b\texttt{) \{}s\texttt{\}}\ r\texttt{;}]\phi, \Delta}$$

Rule markedUnwindLoop marks the beginning of each loop iteration with $\backslash\texttt{next}$ update. Memory access updates that belong to the same iteration appear between two $\backslash\texttt{next}$.

[4] Unlike the second argument in history data dependence predicates (Definition 4.1.5) that was showing the i-th preceding *memory access update*.

Forward symbolic execution calculus, including markedUnwindLoop, accumulates state updates *in front of* modalities. As discussed before in Section4.1, predicate abstraction is easier to implement, if state changes are expressed as equations corresponding to the strongest postcondition, not as weakest preconditions that result from update application. Hence, we need to rewrite updates accordingly using state shifting rules.

We need to be able to shift the \next update, hence, we replace the previous shift rules (shiftRead and shiftWrite) with

shiftNextUpdate

$$\frac{\overbrace{\{su' || \ldots; \backslash\mathsf{W}'(wls_1); \ldots; \backslash\mathsf{R}'(rls_1); \backslash\texttt{next}'\}}^{u}\Gamma,\ \widehat{\texttt{rPred}}(\bigcup_{i=1\ldots n} rls_i, 0),\ \widehat{\texttt{wPred}}(\bigcup_{j=1\ldots m} wls_j, 0)) \Longrightarrow [\texttt{while}\ (b)\ \{s\}\ r;]\phi, \{u\}\Delta}{\Gamma \Longrightarrow \{su||\backslash\texttt{next}; \backslash\mathsf{R}(rls_1); \ldots; \backslash\mathsf{W}(wls_1); \ldots\}[\texttt{while}\ (b)\ \{s\}\ r;]\phi, \Delta}$$

where su is an state update that does not contain memory access updates, $\backslash\mathsf{R}(rls_1), \ldots, \backslash\mathsf{W}(wls_1), \ldots$ is a sequence of n read and m write updates ($n, m \in \mathbb{N}_0$). The while-statement implies that rule shiftNextUpdate can only be applied once a loop iteration is complete, but the next one has not yet started. Shifting the preceding updates is equivalent to replacing all formulas in the sequent by their strongest postcondition. This is achieved by applying a similar update consisting of the corresponding renaming updates on each of the sequent formulas.

Predicates $\widehat{\texttt{rPred}}(\bigcup_{i=1\ldots n} rls_i))$ and $\widehat{\texttt{wPred}}(\bigcup_{j=1\ldots m} wls_j))$ keep the information about the read and write memory accesses in the current iteration of the post state.

The implicit assumption in shiftNextUpdate are that:

- update $\{su||\backslash\texttt{next}; \backslash\mathsf{R}(rls_1); \ldots; \backslash\mathsf{W}(wls_1); \ldots\}$ only contains one \next update; and
- loop body s does not contain a nested loop (we investigate nested loops in the next chapter).

Theorem 5.3.1 (Soundness and completeness of shiftNextUpdate)**.** Let Γ, Δ be sets of formulas, a formula ϕ, and location sets $rls_i, wls_j \in \mathcal{D}^{\mathsf{LocSet}}$ for $i = 1 \ldots n$ and $j = 1 \ldots m$. If and only if

$$\overbrace{\{\ldots; \backslash\mathsf{W}'(wls_1); \ldots; \backslash\mathsf{R}'(rls_1); \backslash\texttt{next}'\}}^{u}\Gamma,\ \widehat{\texttt{rPred}}(\bigcup_{i=1\ldots n} rls_i, 0),\ \widehat{\texttt{wPred}}(\bigcup_{j=1\ldots m} wls_j, 0)) \Longrightarrow [\texttt{while}\ (b)\ \{s\}\ r;]\phi, \{u\}\Delta$$

and rls_i and wls_j do not contain data dependence predicates, then the following holds:

$$\Gamma \Longrightarrow \{\backslash\texttt{next}; \backslash\texttt{R}(rls_1); \ldots, \backslash\texttt{W}(wls_1); \ldots\}[\texttt{while (}b\texttt{) \{}s\texttt{\} }r\texttt{;}]\phi, \Delta.$$

Theorem 5.3.1 is proven in Appendix C.6.

Example 5.3.1. We prove the following sequent, the loop body s is the one of program `arraySum` in Listing 5.1.

$$\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..a.length]}}, \emptyset, \emptyset, \emptyset) \Longrightarrow$$
$$\{\texttt{i} := 0 || \texttt{sum} := 0\}[\texttt{while (i<a.length) \{}s\texttt{\} }]\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..a.length]}}, \emptyset, \emptyset, \emptyset)$$

First, unwind the loop by applying rule markedUnwindLoop, continue on the branch where the loop guard is true, and symbolically execute the loop iteration until we are left with the sequent

$$\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..a.length]}}, \emptyset, \emptyset, \emptyset) \Longrightarrow$$
$$\{u || \backslash\texttt{next}; \backslash\texttt{R}(\underline{\texttt{b[0]}}); \backslash\texttt{R}(\underline{\texttt{c[0]}}); \backslash\texttt{W}(\underline{\texttt{a[0]}}); \backslash\texttt{R}(\underline{\texttt{a[0]}})\}$$
$$[\texttt{while (i<a.length) \{}s\texttt{\} }]\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..a.length]}}, \emptyset, \emptyset, \emptyset)$$

where update u contains the state updates that are not of interest here. We shift the updates and obtain the following sequent (u' denotes the renamed update corresponding to u):

$$\{u' || \backslash\texttt{R}'(\underline{\texttt{a[0]}}); \backslash\texttt{W}'(\underline{\texttt{a[0]}}); \backslash\texttt{R}'(\underline{\texttt{c[0]}}); \backslash\texttt{R}'(\underline{\texttt{b[0]}}); \backslash\texttt{next}'\}\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..a.length]}}, \emptyset, \emptyset, \emptyset),$$
$$\widehat{\texttt{rPred}}(\underline{\texttt{a[0]}}, 0), \ldots \Longrightarrow [\texttt{while (i<a.length) \{}s\texttt{\}}]\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..a.length]}}, \emptyset, \emptyset, \emptyset)$$

After applying the update simplification rules we obtain a first-order formula over inter-interation predicates that can be used in the loop invariant generation algorithm.

We discuss the new update application and simplification rules that are needed here in Subsection 5.4.1.

5.3.2. Predicate Abstraction

Weakening of abstract predicates is based on a lattice induced by predicates $\text{PSym}^{\widehat{Dep}}$ considering the partial order relation between them. These subsumption relations are shown in Figure 5.3. For example, if there is a no read-after-write data dependence on memory location set l'', there cannot be such a dependence on any memory locations l''' with $l''' \subseteq l''$. Neither inside a loop iteration nor between different iterations like $\widehat{\texttt{noRaW}}$.

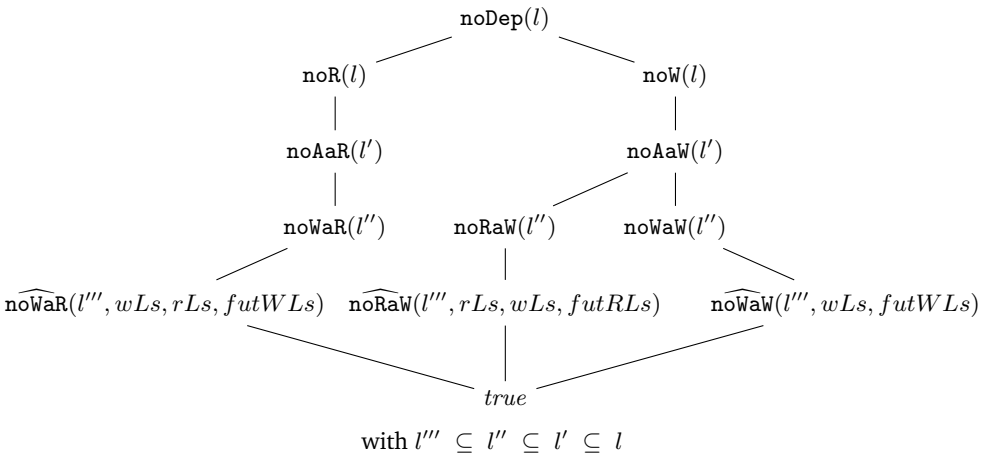


Figure 5.3.: Partial view on predicate abstraction lattice for inter-iteration data dependence predicates

The calculus rules realizing the lattice are discussed in Subsection 5.4.2. Predicate refinement step stays as before.

The loop invariant is the conjunction of predicates remained in D after reaching the fixpoint.

5.4. Reasoning

As a result of introducing new predicates ($\widehat{\texttt{noRaW}}$, $\widehat{\texttt{noWaR}}$, and $\widehat{\texttt{noWaW}}$) and updates (\next and \next′), we need to define corresponding sequent calculus rules.

5.4.1. Update Application and Simplification Rules

The most important update application and simplification rules of JavaDL$_{\widehat{Dep}}$ are shown in Figure 5.4. We explain their intuitive meaning below.

- readAccessOnInterIterationNoRaW: The read access happening in the current iteration is accumulated in the *rLs* argument of $\widehat{\texttt{noRaW}}$.
- writeAccessOnInterIterationNoRaW: The write access happening in the current iteration is accumulated in the *wLs* argument of $\widehat{\texttt{noRaW}}$.

readAccessOnInterIterationNoRaW $cond$

$$\{\backslash\texttt{next};\ \backslash\mathsf{R}(loc_1)\}\widehat{\mathsf{noRaW}}(loc_2, rLs, wLs, futRLs) \rightsquigarrow$$
$$\{\backslash\texttt{next}\}\widehat{\mathsf{noRaW}}(\{\backslash\mathsf{R}(loc_1)\}loc_2, (\{\backslash\mathsf{R}(loc_1)\}rLs) \cup loc_1, \{\backslash\mathsf{R}(loc_1)\}wLs, \{\backslash\mathsf{R}(loc_1)\}futRLs)$$

writeAccessOnInterIterationNoRaW $cond$

$$\{\backslash\texttt{next};\backslash\mathsf{W}(loc_1)\}\widehat{\mathsf{noRaW}}(loc_2, rLs, wLs, futRLs) \rightsquigarrow$$
$$\{\backslash\texttt{next}\}\widehat{\mathsf{noRaW}}(\{\backslash\mathsf{W}(loc_1)\}loc_2, \{\backslash\mathsf{W}(loc_1)\}rLs, (\{\backslash\mathsf{W}(loc_1)\}wLs) \cup loc_1, \{\backslash\mathsf{W}(loc_1)\}futRLs)$$

checkNoRaWForIteration $cond$

$$\{u||\backslash\texttt{next}\}\widehat{\mathsf{noRaW}}(loc, rLs, wLs, futRLs) \rightsquigarrow$$
$$(\{u||\backslash\texttt{next}\}(loc \cap wLs \cap futRLs = \texttt{empty})) \wedge$$
$$\{u\}\widehat{\mathsf{noRaW}}(\{\backslash\texttt{next}\}loc, \texttt{empty}, \texttt{empty}, \{\backslash\texttt{next}\}futRLs \cup \{\backslash\texttt{next}\}rLs)$$

checkNoRaWForLastIteration $cond$

$$\{\backslash\texttt{next}\}\widehat{\mathsf{noRaW}}(loc, rLs, wLs, futRLs) \rightsquigarrow \{\backslash\texttt{next}\}(loc \cap wLs \cap futRLs = \texttt{empty})$$

$cond$: loc, loc_1, loc_2, rLs, wLs, $futRLs$, and lb do not contain data dependence predicates

Figure 5.4.: Selected rules for application update on inter-iteration predicates

- checkNoRaWForIteration: After all the accesses in the current iteration are accumulated in the corresponding argument of $\widehat{\texttt{noRaW}}$, it is time for {\next} to be applied. We need to make sure that location sets written in the current iteration *wLs*, and location sets that are read up to the current iteration (excluding the current iteration) *futRLs* are disjoint from the targeted location set *loc* (emptiness check). Otherwise, a read-after-write has happened. Location sets read in this iteration are added to the *futRLs* argument so that they can be used in the next iteration of loop invariant generation process. Arguments *rLs* and *wLs* are reset to empty in preparation for accumulating accesses of the next iteration.
- checkNoRaWForLastIteration: This rule is similar to the checkNoRaWForIteration rule, except that it is only looking at the last iteration. Therefore, \next is the only update to be applied. This rule only performs the emptiness check.

Analogous rules can be found in Appendix B.6.

Figure 5.5 shows rules for application of renamed updates on data dependence predicates. Here, we briefly mention their intuitive meaning:

- renamedReadAccessOnInterIterationNoRaW: The renamed read access happening in the current iteration is subtracted from the *rLs* argument of $\widehat{\texttt{noRaW}}$.
- renamedWriteAccessOnInterIterationNoRaW: The renamed write access happening in the current iteration is subtracted from the *wLs* argument of $\widehat{\texttt{noRaW}}$.
- interIterationNoRaWToHistory: Application $\texttt{\next}'$, *ages* the $\widehat{\texttt{noRaW}}$ for one iteration and turns it to history predicate.
- renamedNextOnNoRaWHist: Application $\texttt{\next}'$, *ages* the $\widehat{\texttt{noRaWHist}}$ for one iteration.

5.4.2. Subsumption Relations

Subsumption relations between data dependence predicates are shown in Figure 5.3. For example, if there is a no read-after-write data dependence on memory location set l', there cannot be such a dependence on any memory locations l'' with $l'' \subseteq l'$. Neither inside a loop iteration nor between different iterations like $\widehat{\texttt{noRaW}}$.

There are subsumption relations between relaxed data dependence predicates, they imply their history versions. Similarly, there are subsumption relations between relaxed history data dependence predicates. Rule schemas realizing these relations are shown in Figure 5.6.

renamedReadAccessOnInterIterationNoRaW $cond$

$\{\backslash\mathtt{next}';\ \backslash\mathsf{R}'(loc_1)\}\widehat{\mathsf{noRaW}}(loc_2, rLs, wLs, futRLs) \rightsquigarrow$

$\{\backslash\mathtt{next}'\}\widehat{\mathsf{noRaW}}(\{\backslash\mathsf{R}'(loc_1)\}loc_2, (\{\backslash\mathsf{R}'(loc_1)\}rLs) \setminus loc_1, \{\backslash\mathsf{R}'(loc_1)\}wLs, \{\backslash\mathsf{R}'(loc_1)\}futRLs)$

renamedWriteAccessOnInterIterationNoRaW $cond$

$\{\backslash\mathtt{next}', \backslash\mathsf{W}'(loc_1)\}\widehat{\mathsf{noRaW}}(loc_2, rLs, wLs, futRLs) \rightsquigarrow$

$\{\backslash\mathtt{next}'\}\widehat{\mathsf{noRaW}}(\{\backslash\mathsf{W}'(loc_1)\}loc_2, \{\backslash\mathsf{W}'(loc_1)\}rLs, (\{\backslash\mathsf{W}'(loc_1)\}wLs) \setminus loc_1, \{\backslash\mathsf{W}'(loc_1)\}futRLs)$

interIterationNoRaWToHistory $cond$

$\{u||\backslash\mathtt{next}'\}\widehat{\mathsf{noRaW}}(loc, rLs, wLs, futRLs) \rightsquigarrow \{u\}\widehat{\mathsf{noRaWHist}}(\{\backslash\mathtt{next}'\}loc, 1)$

renamedNextOnNoRaWHist $cond$

$\{u||\backslash\mathtt{next}'\}\widehat{\mathsf{noRaWHist}}(loc, lb) \rightsquigarrow \{u\}\widehat{\mathsf{noRaWHist}}(\{\backslash\mathtt{next}'\}loc, \{\backslash\mathtt{next}'\}lb + 1)$

$cond$: loc, loc_1, loc_2, rLs, wLs, $futRLs$, and lb do not contain data dependence predicates

Figure 5.5.: Selected rules for application of renamed update on inter-iteration predicates

noRSubsumuption $\mathsf{noR}(loc) \rightsquigarrow \widehat{\mathsf{noRHist}}(loc, lb)$

interNoRaWSubsumption $\widehat{\mathsf{noRaW}}(loc) \rightsquigarrow \widehat{\mathsf{noRaWHist}}(loc, lb)$

interNoRaWLocSetSubsumption

$$\frac{\Gamma, \widehat{\mathsf{noRaWHist}}(loc, lb_1), lb_1 \leq lb_2 \Longrightarrow \widehat{\mathsf{noRaWHist}}(loc \setminus loc, lb_2), \Delta \quad \Gamma, \widehat{\mathsf{noRaWHist}}(loc, lb_1), lb_1 > lb_2 \Longrightarrow \widehat{\mathsf{noRaWHist}}(loc, lb_2), \Delta}{\Gamma, \widehat{\mathsf{noRaWHist}}(loc, lb_1) \Longrightarrow \widehat{\mathsf{noRaWHist}}(loc, lb_2), \Delta}$$

Figure 5.6.: Subsumption relations between inter-iteration data dependence predicates

5.4.3. Embedding Predicate Abstraction

Part of the sequent calculus rules needed for embedding the predicate abstraction in symbolic execution, are rules regarding $\widehat{\texttt{rPred}}$ and $\widehat{\texttt{wPred}}$.

Shifting utilizes the predicates $\widehat{\texttt{rPred}}$ and $\widehat{\texttt{wPred}}$ to capture knowledge about the post-state. This information is used by our calculus with rules like:

$$\text{interReadPredANDNoRAtHistSameLabel}\quad \frac{\Gamma, \widehat{\texttt{rPred}}(loc_1, lb), \widehat{\texttt{noRHist}}(loc_2, lb), loc_1 \cap loc_2 \doteq \emptyset \Longrightarrow \phi, \Delta}{\Gamma, \widehat{\texttt{rPred}}(loc_1, lb), \widehat{\texttt{noRHist}}(loc_2, lb) \Longrightarrow \phi, \Delta}$$

or

$$\text{interReadPredANDNoRAtHistRightSameLabel}\quad \frac{\Gamma, \widehat{\texttt{rPred}}(loc_1, lb) \Longrightarrow loc_1 \cap loc_2 \doteq \emptyset \wedge \widehat{\texttt{noRHist}}(loc_2, lb+1), \Delta}{\Gamma, \widehat{\texttt{rPred}}(loc_1, lb) \Longrightarrow \widehat{\texttt{noRHist}}(loc_2, lb), \Delta}$$

Rule interReadPredANDNoRAtHistSameLabel derives from the assumptions that there was no read on location set loc_1 and in the same iteration locations in loc_1 have been read, and the fact that loc_1 and loc_2 do not share any memory locations.

Rule interReadPredANDNoRAtHistRightSameLabel simplifies the proof obligation that there is no read on loc_2 at the same iteration that loc_1 is read, by splitting it into two conjuncts: The first conjunct expresses that loc_1 and loc_2 do not share a common memory location and the second conjunct requires then only to prove that there was no read up-to (and including) the preceding iteration ($lb + 1$).

Theorem 5.4.1 (Soundness and Completeness of JavaDL$_{\widehat{Dep}}$ Calculus)**.** Calculus rules of JavaDL$_{\widehat{Dep}}$ program logic are sound and complete.

Corollary 5.4.1 (Soundness of JavaDL$_{\widehat{Dep}}$ Calculus)**.** Since all the rules in JavaDL$_{\widehat{Dep}}$ are sound, its calculus is sound.

6. Nested Loop Invariant Generation

The data dependence loop invariant generation technique developed in Chapter 4 only supports simple loops. In this chapter, we extend the data dependence program logic to generate loop invariants for nested loops.

In our application area HPC nested loops are executed over multi-dimensional arrays. We show how to support programs that contain doubly nested loops over two-dimensional arrays. The generalization to more deeply nested loops on N-dimensional ($N > 2$) arrays is a straightforward extension and omitted from the presentation.

We outline the steps required to generate data dependence loop invariants: The first step is to provide an initial set of abstract predicates. Following the technique described in Chapter 4, we initialize the data dependence predicate set with the top element in the abstraction lattice in Figure 4.2. In the second step the nested loop for which the loop invariant is generated is unwound and symbolically executed by applying rule unwindLoop (see Subsection 2.1.5). Symbolic execution of the outer loop eventually reaches the inner loop. To symbolically execute the inner loop (and continue with the statements after the loop), unwinding the inner loop is not a viable option.

Instead we restart the loop invariant generation process for the inner loop to compute a loop invariant $\mathsf{LI}_{\text{inner}}$ that describes its data dependences invariants $\mathsf{LI}^{\mathsf{Dep}}_{\text{inner}}$ and functional properties as precisely as possible. The inner data dependence loop invariant is generated using the data dependence loop invariant generation algorithm developed in Chapter 4. Afterwards, we use the computed loop invariant $\mathsf{LI}_{\text{inner}}$ to approximate the effect of the inner loop by use of the third premise of rule loop_inv (see Subsection 2.1.5) for the inner loop. This eliminates the inner loop, and symbolic execution of the outer loop body continues with the remaining statements.

In this chapter, we develop an algorithm for generating data dependence invariants for nested loops in Section 6.1. We introduce an anonymizing update and its surrounding calculus in Section 6.2. In Section 6.3, we go a step further and introduce a specialized anonymizing update for inter-iteration data dependences.

6.1. Generation Algorithm

Computation of the outer and inner loop invariant are intertwined: Each time we reach the inner loop during the computation of the outer loop invariant, we generate a fresh loop invariant for the inner loop. This takes into account the changes resulting from unwinding the outer loop as a precondition in generating the inner loop invariant. It provides significant precision, in particular, for loops that are *not perfectly* nested. [1]

A copy of the current set of predicates describing the outer loop is used as the initial set of predicates for the generation of inner loop invariant. Otherwise, the computation of loop invariant follows the data dependence loop invariant generation Algorithm 1.

We explain Algorithm 3 along the example in Listing 6.1.

function *invariantGenerator*
  input : Sequent $seq : pre \implies [\texttt{loop}]\ post$, Data Dependence Predicate set D_{init}
  output : Loop Invariant
  $D_{old},\ D_{ref} \leftarrow D_{\mathsf{init}},\ \emptyset$;
  while $D_{old} \neq D_{ref}$ **do**
    $seq \leftarrow$ Apply rule unwindLoop on seq;
    $seq \leftarrow$ Symbolic execution on seq;
    if *nested* **then**
      $seq_{inner} \leftarrow pre \implies \{u\}[\texttt{innerLoop}]\ true$;
      $\mathsf{LI}_{inner} \leftarrow$ *invariantGenerator*(seq_{inner}, D_{ref});
      $seq_{inner} \leftarrow$ Apply rule loop_inv – usecase on seq_{inner} for LI_{inner};
      $seq \leftarrow$ Replace $innerLoop$ with LI_{inner};
    else
      $seq \leftarrow$ Merge branches below seq;
      /* seq has now the form $pre' \implies \{u\}[\texttt{loop}]post$ */
      $seq \leftarrow$ Apply shift update rules on u in seq;
      $D_{ref},\ D_{old} \leftarrow$ *refine*$(seq, D_{old}),\ D_{ref}$;
      $seq \leftarrow \left(\bigwedge D_{ref} \implies [\texttt{loop}]\ post\right)$;
    end
  end
  return $\bigwedge D_{ref}$;

Algorithm 3: Data Dependence Loop Invariant Generation Algorithm for Nested Loops

[1] Commonly, loops with no statements between the outer and inner loop are called *perfectly nested*.

```
1               i = 0;  j = 0;
2               while (i < N) {
3                   j = 0;
4                   while(j < M) {
5                       a[i][j] = a[i][j] + 1;
6                       j++;
7                   }
8                   i++;
9               }
10
```

Listing 6.1: Increment Elements of 2D Array

6.1.1. Computation of the Inner Loop Invariant

To compute the inner loop invariant the data dependence loop invariant generation algorithm is run on the inner loop for each iteration of the outer loop. We do not use the postcondition during the loop invariant generation process, so we simply take the formula $true$ as postcondition.

Example 6.1.1. Listing 6.1, shows an imperfect nest of loops with the assumption that the array object is not null and no access happens outside of the array bounds. After the first unwind of the outer loop (line 2) and symbolically executing its body, we reach the inner loop (line 4). The sequent used as input for the algorithm to generate the inner loop invariant is:

$$\texttt{a.length} \geq \texttt{N - 1}, \texttt{a[0].length} \geq \texttt{M - 1}, \texttt{a} \not\doteq \texttt{null},$$
$$\texttt{i} \doteq \texttt{0}, \texttt{j} \doteq \texttt{0}, \texttt{noDep}(\underline{\texttt{a[0..a.length-1][0..a[0].length-1]}}), \texttt{i} < \texttt{N} \Longrightarrow$$
$$[\texttt{while (j<M) \{ a[i][j] = a[i][j]+1; j++;\}}]\ \text{true}.$$

The generated loop invariant for the inner loop is:

$$\begin{aligned}
&\texttt{noR}(\underline{\texttt{a[i..a.length-1][j..a[0].length-1]}}) \wedge \\
&\texttt{noW}(\underline{\texttt{a[i..a.length-1][j..a[0].length-1]}}) \wedge \\
&\texttt{noRaW}(\underline{\texttt{a[0..a.length-1][0..a[0].length-1]}}) \wedge \\
&\texttt{noWaW}(\underline{\texttt{a[0..a.length-1][0..a[0].length-1]}}) \wedge \\
&\texttt{i} \doteq \texttt{0} \wedge \texttt{i} < \texttt{N} \wedge \texttt{0} \leq \texttt{j} \leq \texttt{M} \wedge \\
&\texttt{a} \not\doteq \texttt{null} \wedge \texttt{a.length} \geq \texttt{N - 1} \wedge \texttt{a[0].length} \geq \texttt{M - 1}.
\end{aligned}$$

6.1.2. Using the Inner Loop Invariant

After computation of the inner loop invariant LI_{inner} , we return to the generation of the outer loop invariant where symbolic execution stopped at the beginning of the inner loop. Using $\mathsf{LI}_{\text{inner}}$, we apply the loop invariant rule loop_inv (see Section 2.1.3) on the inner loop. The rule splits the proof in three branches. The first two premises ensure correctness of the loop invariant, which we can safely ignore, because our approach guarantees the generation of correct loop invariants.[2] We continue on the branch opened by the third premise, called *use case*, where symbolic execution commences with the first statement after the inner loop.

Example 6.1.2. Continuing Example 6.1.1, the use case branch of rule loop_inv is

$$\begin{aligned}\{u\}\{v\}\big(&\texttt{noR}(\underline{\texttt{a[i..a.length-1][j..a[0].length-1]}}) \land \\ &\texttt{noW}(\underline{\texttt{a[i..a.length-1][j..a[0].length-1]}}) \land \\ &\texttt{noRaW}(\underline{\texttt{a[0..a.length-1][0..a[0].length-1]}}) \land \\ &\texttt{noWaW}(\underline{\texttt{a[0..a.length-1][0..a[0].length-1]}}) \land \\ &\texttt{i} \doteq \texttt{0} \land \texttt{i} < \texttt{N} \land \texttt{a} \neq \texttt{null} \land \texttt{a.length} \geq \texttt{N - 1} \land \texttt{a[0].length} \geq \texttt{M - 1} \land \\ &\texttt{j} \doteq \texttt{M}\big) \Longrightarrow \{u\}\{v\}[\texttt{i++;}]\ \text{true}\end{aligned}$$

where v is an anonymizing update.
Conjunction of $0 \leq \texttt{j} \leq \texttt{M}$ from the inner loop invariant, and $\texttt{j} \geq \texttt{M}$ negation of the inner loop guard results in $\texttt{j} \doteq \texttt{M}$.

As mentioned in Section 2.1.3, application of the loop invariant rule requires to use an anonymizing update v to "forget" the part of the pre-state that might have been changed in the loop. Anonymizing the values of program variables and object fields possibly modified by the loop is done by assigning them a fixed but unknown value. Technically, this is achieved by introducing fresh Skolem constants (for details see [32]).

In the data dependence program logic, however, simply anonymizing the values of memory locations is insufficient, because the loop potentially also performs memory accesses. Hence, the anonymizing update v has to anonymize these accesses as well. Anonymizing memory accesses differs from anonymizing the memory state, because accesses are not recorded as a value of a variable, but represented by memory access updates. The idea is to introduce an anonymizing memory access updates that represents an unknown sequence of arbitrary read or write memory accesses (including possibly no

[2] Even otherwise, they could be ignored, because they are checked once the outer loop invariant is available.

access). In the next two sections we lay out the theoretical foundations for anonymizing memory access updates.

6.2. Anonymization of Memory Access Updates

We introduce an anonymizing memory access update that represents an unknown sequence of arbitrary read or write memory accesses on an unknown set of memory locations.

6.2.1. Syntax and Semantics

For defining the semantics of anonymized memory access updates, we need a helper function:

Definition 6.2.1 (Sequence of anonymized memory accesses updates)**.** The function *anonAcc* : $\mathbf{Z} \to \mathsf{MemoryAccess}^*$ gives a sequence of (possibly empty) unknown memory accesses for the specified length.

We can now define syntax and semantics of anonymized memory access updates.

Definition 6.2.2 (Anonymized memory access update)**.** An *anonymized memory access update* is defined by extension of the update grammar rule:

⟨*update*⟩ ::= ... | \anonAcc‘(’⟨*term*⟩‘)’

where the argument term is of type `int`. Its *semantics* is

$$\mathit{val}_{\mathcal{K},s,\beta}(\backslash\mathsf{anonAcc}(id)) = s' \quad \text{where } s' = (\sigma, Acc') \text{ with } s = (\sigma, Acc) \text{ and } Acc' = Acc \circ \mathit{anonAcc}(\mathit{val}_{\mathcal{K},s,\beta}(id))$$

To use anonymizing memory updates in the calculus (see next section), we need to be able to find out the length of an anonymized sequence of memory access updates.

Definition 6.2.3 (Length of anonymized sequence of memory accesses updates)**.** The length of a sequence of anonymized memory access updates is given by `anonLength` : `int` → `int`. Let id denote a term of type `int`, then

$$\mathit{val}_{\mathcal{K},s,\beta}(\mathsf{anonLength}(id)) = n \text{ where } \mathit{anonAcc}(\mathit{val}_{\mathcal{K},s,\beta}(id)) = acc_1 \circ \cdots \circ acc_n$$

Example 6.2.1. For the sequent in Example 6.1.2 the anonymizing update v is specified as

$$v := v' \mid\mid \backslash\texttt{anonAcc}(\texttt{id})$$

where v' takes care of anonymizing the program variables and array content (for details see [32]) and `id` is a fresh Skolem constant of type `int`.

As before, to enable shifting ("priming"), we need an update to compute the strongest postcondition in presence of an unknown sequence of memory accesses.

Definition 6.2.4 (Renamed anonymized memory access update)**.** The *syntax* of a *renamed anonymized memory access* update is:

⟨*update*⟩ ::= ... | \anonAcc′'('⟨*term*⟩')'

where the argument term is of type `int` and the *semantics* is as follows:

$$val_{\mathcal{K},s,\beta}(\backslash\texttt{anonAcc}'(id)) = s' \text{ where } s' = (\sigma, Acc') \text{ with } s = (\sigma, acc_1 \circ \cdots \circ acc_n) \text{ and}$$

$$Acc' = \left\{ \begin{array}{ll} acc_1 \circ \cdots \circ acc_{n-val_{\mathcal{K},s,\beta}(\texttt{anonLength}(id))}, & \text{if } val_{\mathcal{K},s,\beta}(\texttt{anonLength}(id)) < n \\ \varepsilon & \text{, otherwise} \end{array} \right\}$$

Example 6.2.2. Our approach generates the following data dependence invariant for the outer loop in Listing 6.1

$$\begin{aligned} &\texttt{noR}(\underline{\texttt{a[i..a.length-1][0..a[0].length-1]}}) \wedge \\ &\texttt{noW}(\underline{\texttt{a[i..a.length-1][0..a[0].length-1]}}) \wedge \\ &\texttt{noRaW}(\underline{\texttt{a[0..a.length-1][0..a[0].length-1]}}) \wedge \\ &\texttt{noWaW}(\underline{\texttt{a[0..a.length-1][0..a[0].length-1]}}) \wedge \\ &\texttt{0} \leq \texttt{i} \leq \texttt{N} \wedge \texttt{0} \leq \texttt{j} \leq \texttt{M} \wedge \\ &\texttt{a} \not\doteq \texttt{null} \wedge \texttt{a.length} \geq \texttt{N - 1} \wedge \texttt{a[0].length} \geq \texttt{M - 1}. \end{aligned}$$

6.2.2. Update Application Rules

A selection of the rules for the new update kinds $\backslash\texttt{anonAcc}(id)$ and $\backslash\texttt{anonAcc}'(id)$ is shown in Figure 6.1. We explain them briefly below.

- Rule shiftAnonAcc, like previous shifting rules, achieves shifting for anonymized memory access updates, i.e. it replaces all formulas in the sequent by their strongest postcondition. In contrast to the previous shifting rules, it does not add information on the memory locations accessed in the anonymized update, because these are unknown.

shiftAnonAcc $\dfrac{\{\backslash\mathsf{anonAcc}'(id)\}\Gamma \Longrightarrow \phi, \{\backslash\mathsf{anonAcc}'(id)\}\Delta}{\Gamma \Longrightarrow \{\backslash\mathsf{anonAcc}(id)\}\phi, \Delta}$

anonAccApp

$\{\backslash\mathsf{anonAcc}(id)\}\mathtt{noRHist}(loc, lb) \rightsquigarrow$
$\mathtt{noRHist}(\{\backslash\mathsf{anonAcc}(id)\}loc, \{\backslash\mathsf{anonAcc}(id)\}i - \mathsf{anonLength}(id))$

renamedAnonAccApp

$\{\backslash\mathsf{anonAcc}'(id)\}\mathtt{noRHist}(loc, lb) \rightsquigarrow$
$\mathtt{noRHist}(\{\backslash\mathsf{anonAcc}'(id)\}loc, \{\backslash\mathsf{anonAcc}'(id)\}i + \mathsf{anonLength}(id))$

Figure 6.1.: A selection of calculus rules for (renamed) anonymized memory access update

- Rule anonAccApp applies the anonymized memory access update $\backslash\mathsf{anonAcc}(id)$ to a formula $\mathtt{noRHist}(loc, lb)$, which expresses that there was no read on any location in ls, except possibly for the preceding i accesses. Applying the update modifies the number of preceding accesses to the shorter sequence of accesses in the pre-state. This means instead of going back i steps, we only need to go back $lb - \mathsf{anonLength}(id)$ steps.

- On the other hand, we have rule renamedAnonAccApp for a renaming update $\backslash\mathsf{anonAcc}'(id)$, but instead of decreasing the number of steps going back, we need to increase them by the number of the memory accesses represented by the anonymized update.

6.3. Anonymization of a Sequence of Memory Access Updates

The anonymizing memory access update $\backslash\mathsf{anonAcc}(id)$ is insufficient for analyzing inter-iteration data dependences as it abstracts away from location sets. Hence, we need to go a step further and introduce an anonymizing memory access update that is sufficiently equipped for analyzing inter-iteration data dependences with high precision.

6.3.1. Syntax and Semantics

Similar to anonymization of memory access updates, for defining the semantics of anonymized *sequence* of memory access updates, we need a helper function:

Definition 6.3.1 (Sequence of sequence of anonymized memory accesses updates)**.** The function $\widehat{anonAcc} : \mathbf{Z} \rightarrow \mathsf{SequenceOfMemoryAccess}^*$ gives a sequence of sequence of (possibly empty) memory accesses for the specified length.

We extend Definition 5.2.2 to contain all the location sets that are read in multiple array iterations.

Definition 6.3.2 (Extended projection)**.** For $itrSeq = \widehat{acc}_i \bigcirc \ldots \bigcirc \widehat{acc}_j$ the extended projection function is defined as

$$\downarrow_R(itrSeq) := \bigcup_{k=i}^{j} \bigcup_{l=1}^{n_k} \{ls_{(k,l)} \mid \widehat{acc}_k = acc_{(k,1)} \circ \ldots \circ acc_{(k,n_k)} \text{ and } acc_{(k,l)} = \langle Read(ls_{(k,l)}) \rangle\}$$

collects all memory locations read in loop iterations i to j. We extend the write projection function analogously:

$$\downarrow_W(itrSeq) := \bigcup_{k=i}^{j} \bigcup_{l=1}^{n_k} \{ls_{(k,l)} \mid \widehat{acc}_k = acc_{(k,1)} \circ \ldots \circ acc_{(k,n_k)} \text{ and } acc_{(k,l)} = \langle Write(ls_{(k,l)}) \rangle\}.$$

Using the anonymized sequence of memory access updates and the extended projection functions, we can define syntax and semantics of anonymized sequence of sequences of memory access updates.

Definition 6.3.3 (Anonymized sequence of sequences of memory access update)**.** An *anonymized sequence of sequences of memory access update* is defined by extension of the update grammar rule:

$\langle update \rangle$ ::= ...| $\widehat{\backslash \mathsf{anonAcc}}$ '('$\langle term \rangle$, $\langle term \rangle$, $\langle term \rangle$')'

where the first two arguments are of type `LocSet` and the third term is of type `int`. Its *semantics* is as follows:

$$val_{\mathcal{K},s,\beta}(\widehat{\backslash \mathsf{anonAcc}}(rLocs, wLocs, id)) = s' \quad \text{where } s' = (\sigma, \widehat{Acc}') \text{ with } s = (\sigma, \widehat{Acc}),$$
$$\widehat{Acc}' = \widehat{Acc} \bigcirc \widehat{anonAcc}(val_{\mathcal{K},s,\beta}(id)),$$
$$rLocs = \downarrow_R \widehat{anonAcc}(val_{\mathcal{K},s,\beta}(id)), \text{ and}$$
$$wLocs = \downarrow_W \widehat{anonAcc}(val_{\mathcal{K},s,\beta}(id)).$$

To use $\widehat{\backslash \mathsf{anonAcc}}$ in the calculus (see next section), we need to be able to find out the length of an anonymized sequence of sequence of memory access updates.

Definition 6.3.4 (Length of anonymized sequence of sequence of memory accesses updates)**.** The length of anonymized sequence of sequence memory accesses updates is given by $\widehat{\mathsf{anonLength}} : \mathtt{int} \to \mathtt{int}$. Let id denote a term of type `int`, then

$$val_{\mathcal{K},s,\beta}(\widehat{\mathsf{anonLength}}(id)) = n \text{ where } val_{\mathcal{K},s,\beta}(\widehat{\backslash \mathsf{anonAcc}}(id)) = \widehat{acc}_1 \bigcirc \ldots \bigcirc \widehat{acc}_n$$

Example 6.3.1. For the sequent in Example 6.1.2 the anonymizing update v is specified as

$$v := v' \mid\mid \widehat{\backslash \mathsf{anonAcc}}(\mathtt{id})$$

where `id` is a fresh Skolem constant of type `int` and v' takes care of anonymizing the program variables and array content (for details see [32]).

As before, to enable shifting ("priming"), we need an update to compute the strongest postcondition in presence of an unknown sequence of sequences of memory accesses.

Definition 6.3.5 (Renamed anonymized sequence of sequences of memory access update)**.** The *syntax* of a *renamed anonymized memory access* update is:

$\langle update \rangle$ ::= ... | $\widehat{\backslash \mathsf{anonAcc}}'$ '('$\langle term \rangle$, $\langle term \rangle$, $\langle term \rangle$')'

where the first two arguments are terms of type `LocSet` the last argument term is of type `int` and the *semantics* is as follows:

$$val_{\mathcal{K},s,\beta}(\widehat{\backslash \mathsf{anonAcc}}'(rLocs, wLocs, id)) = s' = (\sigma, \widehat{Acc}') \text{ with } s = (\sigma, \widehat{acc}_1 \bigcirc \ldots \bigcirc \widehat{acc}_n),$$
$$\text{and } \widehat{Acc}' = \begin{cases} \widehat{acc}_1 \bigcirc \ldots \bigcirc \widehat{acc}_{n - val_{\mathcal{K},s,\beta}(\widehat{\mathsf{anonLength}}(id))}, & \text{if } val_{\mathcal{K},s,\beta}(\widehat{\mathsf{anonLength}}(id)) < n \\ \varepsilon & \text{, otherwise} \end{cases}$$
$$rLocs = \downarrow_R val_{\mathcal{K},s,\beta}(\widehat{\backslash \mathsf{anonAcc}}'(id)), \text{ and}$$
$$wLocs = \downarrow_W val_{\mathcal{K},s,\beta}(\widehat{\backslash \mathsf{anonAcc}}'(id)).$$

Example 6.3.2. Our approach generates the following inter-iteration data dependence invariant for Listing 6.1 under the assumption that the array object is not null and no access happens outside of the array bounds.

$$\begin{aligned}
&\mathsf{noR}(\underline{\texttt{a[i..a.length-1][0..a[0].length-1]}}) \wedge \\
&\mathsf{noW}(\underline{\texttt{a[i..a.length-1][0..a[0].length-1]}}) \wedge \\
&\widehat{\mathsf{noRaW}}(\underline{\texttt{a[0..a.length-1][0..a[0].length-1]}}) \wedge \\
&\widehat{\mathsf{noWaR}}(\underline{\texttt{a[0..a.length-1][0..a[0].length-1]}}) \wedge \\
&\widehat{\mathsf{noWaW}}(\underline{\texttt{a[0..a.length-1][0..a[0].length-1]}}) \wedge \\
&\texttt{0} \leq \texttt{i} \leq \texttt{N} \wedge \texttt{a} \neq \texttt{null} \wedge \texttt{a.length} \geq \texttt{N - 1} \wedge \texttt{a[0].length} \geq \texttt{M - 1}.
\end{aligned}$$

6.3.2. Update Application Rules

A selection of the rules for the new update kinds $\widehat{\setminus\mathsf{anonAcc}}(rLocs, wLocs, id)$ and $\widehat{\setminus\mathsf{anonAcc}'}(rLocs, wLocs, id)$ is shown in Figure 6.2. We explain them briefly below.

- Rule shiftItrAnonAcc, like other shifting rules, realizes shifting for a anonymized sequence of memory access updates, i.e. it replaces all formulas in the sequent by their strongest postcondition. Similar to shiftAnonAcc, it does not add information on the memory locations accessed in the sequence of anonymized updates, as these are unknown.
- Rule itrAnonAccApp applies anonymized sequence of memory access updates $\widehat{\setminus\mathsf{anonAcc}}(rLocs, wLocs, id)$ to formula $\widehat{\mathsf{noRaW}}(ls, \mathit{rLs}, \mathit{wLs}, \mathit{futRLs})$, which expresses that there is no read-after-write data dependence on any location in *ls*, at the current iteration location sets *rLs* and *wLs* are read and written, respectively. Processed loop iterations up to now have read *futRLs*. The update has to be applied on all arguments to keep the soundness. Location sets *rLocs* and *wLocs* are added to the location sets *rLs* and *wLs* that are read and written in the current iteration of the loop.
- Rule itrAnonAccAppOnHist applies the anonymized sequence of memory access updates to a formula $\widehat{\mathsf{noRHist}}(loc, lb)$, which expresses that there was no read on any location in loc, except possibly for the preceding lb iterations. Applying the update modifies the number of preceding iterations to a shorter sequence in the pre-state. This means instead of going back lb iterations, we only need to go back $lb + \mathsf{anonLength}(id)$ iterations.

- Rule renamedItrAnonAccApp acts opposite of rule itrAnonAccApp by eliminating *rLocs* and *wLocs* from location sets *rLs* and *wLs*.
- Rule renamedItrAnonAccAppOnHist has an opposite effect of itrAnonAccAppOnHist, instead of decreasing the number of steps going back we need to increase them by the number of iterations represented by $\texttt{anonLength}(id)$.

Still the Theorem 3.4.1 and subsequently Corollary 3.4.1 hold.

shiftItrAnonAcc

$$\frac{\{\widehat{\mathsf{anonAcc}'}(rLocs, wLocs, id)\}\Gamma \Longrightarrow \phi, \{\widehat{\mathsf{anonAcc}'}(rLocs, wLocs, id)\}\Delta}{\Gamma \Longrightarrow \{\widehat{\mathsf{anonAcc}}(rLocs, wLocs, id)\}\phi, \Delta}$$

itrAnonAccApp

$$\{\widehat{\mathsf{anonAcc}}(id)\}\widehat{\mathsf{noRaW}}(ls, rLs, wLs,\ futRLs) \rightsquigarrow$$
$$\widehat{\mathsf{noRaW}}(\{\widehat{\mathsf{anonAcc}}(rLocs, wLocs, id)\}ls, (\{\widehat{\mathsf{anonAcc}}(rLocs, wLocs, id)\}rLs)\ \cup\ rLocs,$$
$$(\{\widehat{\mathsf{anonAcc}}(rLocs, wLocs, id)\}wLs)\ \cup\ wLocs,\ futRLs)$$

itrAnonAccAppOnHist

$$\{\widehat{\mathsf{anonAcc}}(id)\}\widehat{\mathtt{noRHist}}(loc, lb) \rightsquigarrow$$
$$\widehat{\mathtt{noRHist}}(\{\widehat{\mathsf{anonAcc}}(rLocs, wLocs, id)\}loc,$$
$$(\{\widehat{\mathsf{anonAcc}}(rLocs, wLocs, id)\}lb) - \widehat{\mathsf{anonLength}}(id))$$

renamedItrAnonAccApp

$$\{\widehat{\mathsf{anonAcc}'}(rLocs, wLocs, id)\}\widehat{\mathsf{noRaW}}(ls, rLs, wLs,\ futRLs) \rightsquigarrow$$
$$\widehat{\mathsf{noRaW}}(\{\widehat{\mathsf{anonAcc}'}(rLocs, wLocs, id)\}ls, (\{\widehat{\mathsf{anonAcc}'}(rLocs, wLocs, id)\}rLs)\ \setminus\ rLocs,$$
$$(\{\widehat{\mathsf{anonAcc}'}(rLocs, wLocs, id)\}wLs\ \setminus\ wLocs,\ futRLs)$$

renamedItrAnonAccAppOnHist

$$\{\widehat{\mathsf{anonAcc}'}(id)\}\widehat{\mathtt{noRHist}}(loc, lb) \rightsquigarrow$$
$$\widehat{\mathtt{noRHist}}(\{\widehat{\mathsf{anonAcc}'}(rLocs, wLocs, id)\}loc,$$
$$(\{\widehat{\mathsf{anonAcc}'}(rLocs, wLocs, id)\}lb) + \widehat{\mathsf{anonLength}}(id))$$

Figure 6.2.: A selection of calculus rules for (renamed) anonymized sequence of memory access updates application

7. Multi-Dimensional Arrays

On the theoretical side there is no major difference between one- and multi-dimensional arrays. In Java, multi-dimensional arrays are defined as an array of arrays. The general shape can be complex (see Figure 7.2a). Our application area HPC (and many others) use mostly two dimensional arrays to describe matrices, hence, it proves advantageous to provide means to represent the location set of such matrix-like arrays for this more specific form.

Section 7.1 extends the theory of location sets by a constructor that describes rectangular array shapes. This allows us to achieve a higher degree of automation when reasoning over matrix arrays in comparison to describing them using general location set functions. In Section 7.2 we develop sequent calculus rules to realize the extension to theory of location sets theory. In the end, in Section 7.3, we briefly mention how the prover chooses which rule has to be applied, as this is one of the challenging aspects of the implementation.

7.1. Syntax and Semantics

We define the constructor for rectangular shaped array as a definitional extension of infinite union $\texttt{infiniteUnion}\{x\}$ [32].

The function symbol $\texttt{infiniteUnion}\{T\ x\}(t)$ is defined as a variable binding function that binds variable x in t. Intuitively, when t is of type `LocSet` it represents the mathematical set $\bigcup_x t$. We define it formally based on definitions in [32].

Definition 7.1.1 (Infinite union)**.** Infinite union is a function with the following signature:

$$\texttt{infiniteUnion}\{T\ x\} : \texttt{LocSet} \to \texttt{LocSet}$$

where $T\ x$ is a variable of type T. The semantics is defined as follows:

$$\mathit{val}_{\mathcal{K},s,\beta}(\texttt{infiniteUnion}\{T\ x\}(t)) = \bigcup_{a \in D^T} \mathit{val}_{\mathcal{K},s,\beta^a_x}(t).$$

Example 7.1.1. Infinite union is sufficiently expressive to describe a variety of location sets. For example, we can define $\texttt{arrayRange}(\texttt{a}, \texttt{low}, \texttt{high})$ for an array $\texttt{a}$ as $\texttt{infiniteUnion}\{\texttt{int x}\}(\text{if } (\texttt{low} \leq \texttt{x} \wedge \texttt{x} \leq \texttt{high}) \text{ then } (\underline{\texttt{a[x]}}) \text{ else } (\texttt{empty}))$

The infinite union can be used to describe the memory locations for an arbitrary multi-dimensional array of any shape and in particular also for rectangular shaped arrays. But the expressive power, comes with high automation costs.

Hence, we introduce a function symbol (constructor) similar to `arrayRange` [32] to represent memory locations of a two-dimensional array restricted to rectangular form.

For a rectangular shaped two-dimensional array `a` location sets take the form of finite unions of contiguous segments $\underline{a[i..j][k..l]}$.These must be generalized to finite unions of N-dimensional rectangles: $\underline{a[i_1..j_1]} \ldots \underline{[i_N..j_N]}$.

We only focus on two-dimensional rectangular shaped arrays, here. The extension to the N-dimensional rectangular shaped arrays is straightforward.

Definition 7.1.2 (Matrix range)**.** Matrix range is a function with the following signature:

$$\texttt{matrixRange} : \texttt{Heap} \times \texttt{Object} \times \texttt{int} \times \texttt{int} \times \texttt{int} \times \texttt{int} \to \texttt{LocSet}.$$

$\texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH)$ is the union of all the array ranges it is referencing:

$$\begin{aligned}
&\texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH) \;:= \\
&\quad \texttt{infiniteUnion}\{\texttt{int } rowI;\}(\\
&\qquad \text{if } (rowI \geq rowL \;\wedge\; rowI \leq rowH) \\
&\qquad \text{then } (\texttt{arrayRange}(\texttt{select}(heap, matrix, \texttt{arr}(rowI)), colL, colH)) \\
&\qquad \text{else } (\texttt{empty}))
\end{aligned}$$

In $\texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH)$, if the lower bound of rows $rowL$ (columns $colL$) is less than or equal to the upper bound of rows $rowH$ (columns $colH$) then it is describing memory locations of a two-dimensional array. Otherwise, it is the empty location set.

Example 7.1.2. In Figure 7.1, $\texttt{matrixRange}(\texttt{heap}, \texttt{a}, \texttt{i}, \texttt{j}, \texttt{k}, \texttt{l})$ is a rectangular shaped sub-array of the two-dimensional array `a[0..N][0..M]` with $\texttt{0} \leq \texttt{i} \leq \texttt{j} \leq \texttt{N}$ and $\texttt{0} \leq \texttt{k} \leq \texttt{l} \leq \texttt{M}$.

Multi-dimensional Java arrays are modeled as a one-dimensional arrays with references to other one-dimensional array. A multi-dimensional array does not need to have arrays

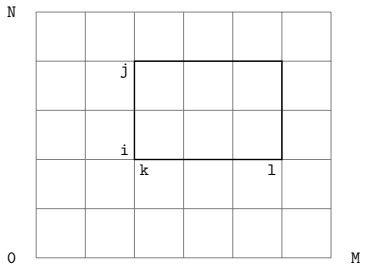
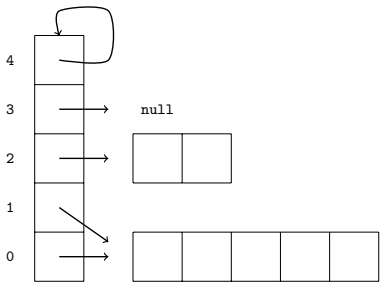
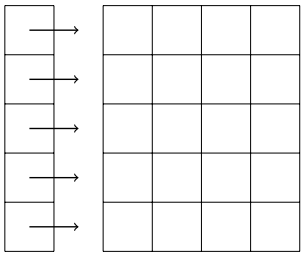


Figure 7.1.: Two-dimensional rectangular shaped array `a[0..N][0..M]` and rectangular shaped sub-array of it described by `matrixRange`$(\texttt{heap}, \texttt{a}, \texttt{i}, \texttt{j}, \texttt{k}, \texttt{l})$



(a) A two-dimensional Java array



(b) A two-dimensional array constructed by `matrixRange`

Figure 7.2.: Different Java arrays

of the same length at each level [41]. The referenced arrays can be the same objects, or `null`. Figure 7.2a shows an example of a two-dimensional Java array. In Figure 7.2a, array elements at index 0 and 2 are referencing to two different one-dimensional arrays with different length. Elements 0 and 1 both are referencing to the same array (aliasing). Array element 3 is not referencing to any array and element 4 is self-referencing. Figure 7.2b shows a rectangular shaped two-dimensional array, constructed by `matrixRange`.

As mentioned before in HPC and many other application scenarios two-dimensional arrays are used to describe matrices and could be described by our `matrixRange` function. But as Java arrays in general are more complex, for instance they can have a non-rectangular shape (7.2a), we need to specify explicitly if an array adheres to the rectangular shape so that we can use `matrixRange`. For example, for having a rectangular all the rows should have the same length `a[0].length`. Therefor, we introduce predicate symbol `wellFormedMatrix`, defined as follows:

Definition 7.1.3 (Well-formed matrix predicate)**.** The syntax of *well-formed matrix* predi-

cate is symbol with the following syntax:

$$\texttt{wellFormedMatrix} : \texttt{Heap} \times \texttt{Object}.$$

The predicate $\texttt{wellFormedMatrix}(heap,\ matrix)$ is defined as follows:

- $matrix \neq \texttt{null}$,
- $\forall\, row;\ \texttt{select}(heap, matrix, \texttt{arr}(row)) \neq \texttt{null}$,
- $\forall\, row1;\ \forall\, row2;$
  $\texttt{length}(\texttt{select}(heap, matrix, \texttt{arr}(row1))) =$
  $\texttt{length}(\texttt{select}(heap, matrix, \texttt{arr}(row2)))$,
- $\forall\, row;\ matrix \neq \texttt{select}(heap, matrix, \texttt{arr}(row))$,
- $\forall\, row1;\ \forall\, row2;\ (row1 \neq row2) \rightarrow$
  $\texttt{select}(heap, matrix, \texttt{arr}(row1)) \neq \texttt{select}(heap, matrix, \texttt{arr}(row2))$,
- $\forall\, matrix2;\ (matrix \neq matrix2) \rightarrow \forall\, row;\ \forall\, row2;$
  $\texttt{select}(heap, matrix, \texttt{arr}(row)) \neq \texttt{select}(heap, matrix2, \texttt{arr}(row2))$.

The first three properties are describing the rectangular shape of a matrix and the rest help to exclude sharing while proving.

The assumptions about the rectangular shape of the matrix make the automation of simplification rules easier.

7.2. Calculus Rules

We define the standard set operations $(\cup, \cap, \setminus, \subset)$ for `matrixRange` as well. These operations are realized through calculus rules. As a result of introducing predicate symbol `matrixRange`, we need to define corresponding sequent calculus rules. There are axiomatic rules for replacing `matrixRange` and `wellFormedMatrix` with their respective definitions. Besides these rules, there are several derived rules that are more amenable for proof automation. In the following we explain some of these rules.

matrixRangeMinusSingleton

$$\texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH) \setminus \texttt{singleton}(o, \texttt{arr}(sCol))$$
$$\rightsquigarrow$$
$$\text{if } (matrix \doteq o \ \vee \ sCol < colL \ \vee \ colH < sCol)$$
$$\text{then } (\texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH))$$
$$\text{else } (\text{if } (\exists k;\ k \geq rowL \ \wedge \ k \leq rowH \ \wedge \ o = \texttt{select}(heap, matrix, \texttt{arr}(k)))$$
$$\text{then } (\texttt{matrixRange}(heap, matrix, rowL, sRow \ \texttt{-} \ 1, colL, colH) \cup$$
$$\texttt{matrixRange}(heap, matrix, sRow \ \texttt{+} \ 1, rowH, colL, colH) \cup$$
$$\texttt{matrixRange}(heap, matrix, sRow, sRow, colL, sCol \ \texttt{-} \ 1) \cup$$
$$\texttt{matrixRange}(heap, matrix, sRow, sRow, sCol \ \texttt{+} \ 1, colH)))$$
$$\text{else } (\texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH))$$

if $\texttt{wellFormedMatrix}(matrix, heap)$ holds, and
sRow is a fresh constant of type `int` such that
$$\big(\exists k;\ k \geq rowL \ \wedge \ k \leq rowH \ \wedge \ o = \texttt{select}(heap, matrix, \texttt{arr}(k))\big) \rightarrow$$
$$sRow \geq rowL \ \wedge \ sRow \leq rowH \ \wedge \ o = \texttt{select}(heap, matrix, \texttt{arr}(sRow))$$

Rule matrixRangeMinusSingleton formalizes how to reduce the set difference operation on a `matrixRange` location set and a `singleton` location set (`singleton`$(o, \texttt{arr}(sCol))$) under the assumption that the *matrix* term is a two-dimensional array that satisfies the well-formedness conditions.

If the singleton location set can be shown to be outside of the matrix range, then the result is the whole matrix range. This is, for instance, the case when the column index *sCol* is outside of the column range of the matrix range, or also for the special case where o refers to the *matrix* array. Otherwise, the singleton location set is part of the matrix range. In this case, there is a row k in the *matrix* that overlaps with the singleton location set, the side condition states that this row is *sRow*. Due to well-formedness of *matrix*, *sRow* is unique. If there is no such row k, then *sRow* is unspecified. The rule decomposes the matrix range to four different matrix range location sets (possibly empty) to exclude the singleton location set. These matrix ranges are shown in Figure 7.3 that are covering the whole matrix range except the singleton location set. The result is union of these location sets.

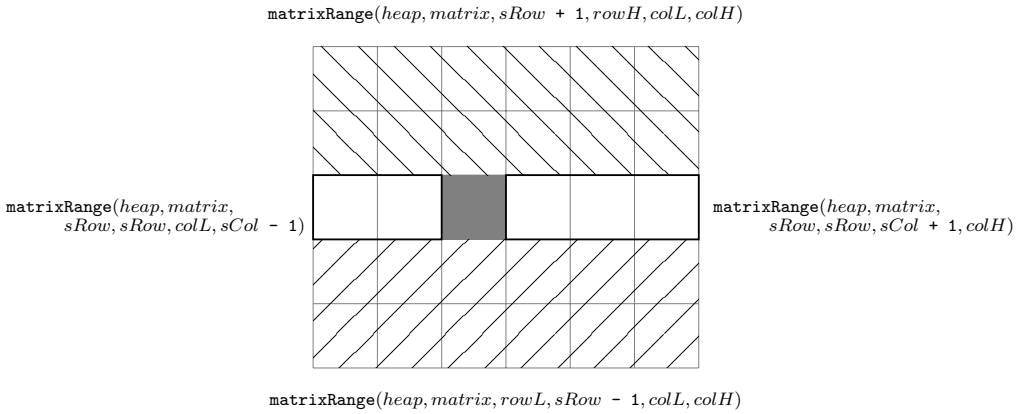


Figure 7.3.: Matrix range minus singleton

matrixRangeMinusMatrixRange

$$\begin{aligned}
&\texttt{matrixRange}(heap, a, aRowL, aRowH, aColL, aColH) \setminus \\
&\quad \texttt{matrixRange}(heap, b, bRowL, bRowH, bColL, bColH) \\
&\rightsquigarrow \\
&\text{if } (a \neq b \ \vee \ bRowH < aRowL \ \vee \ bRowL > aRowH \ \vee \\
&\qquad bColL > aColH \ \vee \ bColH < aColL) \\
&\text{then } (\texttt{matrixRange}(heap, a, aRowL, aRowH, aColL, aColH)) \\
&\text{else } (\texttt{matrixRange}(heap, a, aRowL, bRowL \texttt{ - } 1, aColL, aColH) \cup \\
&\qquad \texttt{matrixRange}(heap, a, bRowH \texttt{ + } 1, aRowH, aColL, aColH) \cup \\
&\qquad \texttt{matrixRange}(heap, a, aRowL, aRowH, aColL, bColL \texttt{ - } 1) \cup \\
&\qquad \texttt{matrixRange}(heap, a, aRowL, aRowH, bColH \texttt{ + } 1, aColH)
\end{aligned}$$

if $\texttt{wellFormedMatrix}(a, heap)$ holds.

Rule matrixRangeMinusMatrixRange formalizes reducing the set difference operation on two `matrixRange` location sets (a and b). The rule is under the assumption that the two-dimensional array a, which is the first argument of the set difference operation, is well-formed. If the matrix range b does not have an overlap with a the set difference results in the whole matrix range a. Similarly, if a and b are different objects they can not have shared location sets, since a is well-formed. Otherwise, a and b are the same object and they are overlapping in some location sets. Similar to matrixRangeMinusSingleton,

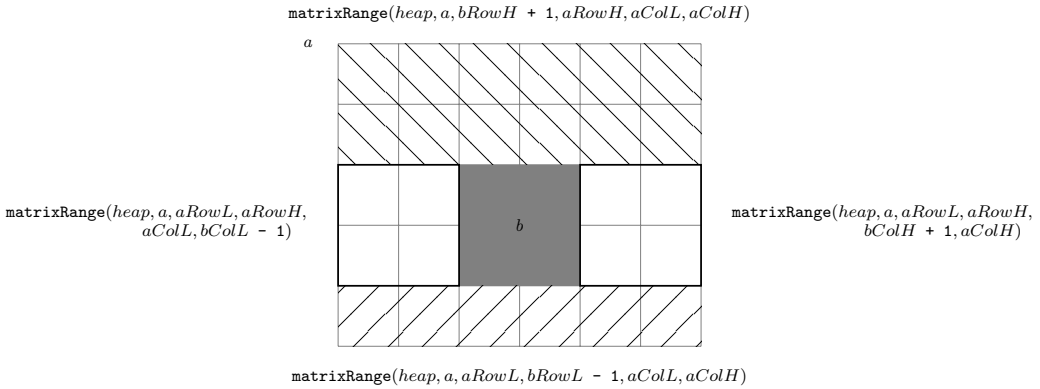


Figure 7.4.: Matrix range minus matrix range

the result of set difference operation is union of four different matrix ranges (possibly empty) depicted in Figure 7.4.

matrixRangeMinusMatrixRangeSpecialCase

$$\texttt{matrixRange}(heap, matrix, a1RowL, a1RowH, a1ColL, a1ColH) \setminus \texttt{matrixRange}(heap, matrix, a2RowL, a2RowH, a2ColL, a2ColH)$$

$$\rightsquigarrow$$

if $(a2RowL \leq a1RowL \ \wedge \ a2RowH \geq a1RowH \ \wedge \ a2ColL \leq a1ColL \ \wedge \ a2ColH \geq a1ColH)$

then $(\texttt{empty})$

else $(\texttt{matrixRange}(heap, matrix, a1RowL, a1RowH, a1ColL, a1ColH) \setminus \texttt{matrixRange}(heap, matrix, a2RowL, a2RowH, a2ColL, a2ColH))$

if $\texttt{wellFormedMatrix}(matrix, heap)$ holds.

Rule matrixRangeMinusMatrixRangeSpecialCase formalizes the set difference operation for two `matrixRange` of the same array object. The assumption is that two-dimensional array *matrix* is well-formed. If the second argument is covering a bigger range than the first argument, the result is an empty location set. Otherwise, the set difference operation is carried on as usual.

Still the Theorem 3.4.1 and subsequently Corollary 3.4.1 hold. As an example we provide the proof for rule matrixRangeMinusSingleton in Appendix C.7.

7.3. Proof Search Strategy

At different states in the proof, there is a subset of rules that are applicable, and one of them must be *chosen*. A *proof search strategy* implements how the choice takes place.

Proof search strategy in KeY is defined as a weighted cost function of a feature vector. Each taclet can have one or multiple features. The cost of applying a taclet in a proof state is a function (e.g. sum) of the costs associated with its features. When multiple taclets are applicable, the taclet with the least application cost is applied. Features turn syntactic criteria of the current proof state into a cost.

To reason effectively and efficiently within the theory presented in this thesis, several features are defined for guiding the proof search. One of these features that focuses is on reasoning about location sets is covered. Sub-arrays of two-dimensional arrays are described by set operations over the `matrixRange` location set constructor. Several of the provided simplification rules for computing the difference or intersection of these matrix ranges are only applicable under certain conditions, like one range contained in the other or existence of some overlapping area.

Generally, showing these conditions requires performing proof, but often this can also be determined on a purely syntactical basis using features like covered that allows safe guesses on two values expressed as polynomials.
Rule matrixRangeMinusMatrixRangeSpecialCase uses covered feature to measure the existence of such an overlap by comparing the involved indices. The “success” case of the rule is when the condition holds. If the indices are expressed as polynomials, checking the condition can sometimes be done syntactically. Hence, the proof search strategy was equipped with features covered that if they can determine that the indices are in the expected order, the rule is assigned a finite cost, while otherwise the feature will assign infinite cost, effectively preventing the rule to be applied (resulting in avoiding the useless “unsuccessful” branch of the conditional term to be taken).

8. Experimental Results

We have implemented a prototype of our data dependence analysis approach using the KeY deductive verifier and tackled the following challenges:

- extending the core logic to support data dependences by new update types and non-rigid predicates;
- adding several lemmas for dealing with location sets (singletons, array ranges, and matrices); and
- adapting the proof search strategy by defining new features.

Like the majority of HPC applications, we focus on loops over arrays and matrices. This chapter shows the result of our analysis for single loops and nested loops in Sections 8.1 and 8.2 , respectively. Test cases in this are inspired by well-known HPC test suites such as Polybench [59] and NAS[60]. We focus on the computation intensive parts of test cases from theses test suits. The result shows that our approach is highly precise.

We have conducted our experiments on a computer with a Core i7-8750H CPU @ 2.2 GHz, 6 Cores (12 Logical Processors). The number of processors does not affect the performance of KeY as it is not parallelized. 2 GB memory is allocated to KeY. For each proof, the timeout is set on 5 seconds. The maximum number of steps per proof is 5000, unless it is stated otherwise. Proofs include generating abstract predicates and conducting symbolic execution.

8.1. Single Loops

In the following test cases the precondition is:

$$\texttt{noDep}(\underline{\texttt{a[0..a.length-1]}}) \land \texttt{a} \neq \texttt{null} \land \texttt{a.length} > \texttt{N}.$$

Predicate $\texttt{noDep}(\underline{\texttt{a[0..a.length-1]}})$ helps to focus on the isolated loop, since the data dependence history before encountering a loop is usually irrelevant for loop parallelization. Knowing the length of array ($\texttt{a.length} > \texttt{N}$) prevents out of bounds array accesses. The post condition is set to true and expresses reachability.

$$\texttt{noR}(\underline{\texttt{a[i..a.length-1]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge$$
$$\texttt{noRaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge \texttt{noWaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N+1}$$

(a) Loop invariant

$$\texttt{noR}(\underline{\texttt{a[i..a.length-1]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge$$
$$\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\widehat{\texttt{noWaR}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\widehat{\texttt{noWaW}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N+1}$$

(b) Inter-iteration loop invariant

Figure 8.1.: Loop invariants for Listing 8.1

8.1.1. Test Cases

Array Increase

Listing 8.1 increases the value of array elements by one. There is only one type of data dependence present in this loop: an intra-iteration *WaR*.

```
while (i <= N) {
    a[i] = a[i] + 1;
    i = i + 1;
}
```

Listing 8.1: Array increase

Figure 8.1a shows the generated loop invariant. It is fully precise and correctly captures the intra-iteration *WaR* data dependence.

Since the *WaR* data dependence is an intra-iteration data dependence, it is allowed in the inter-iteration loop invariant. Figure 8.1b shows the fully precise inter-iteration loop invariant.

Array Access with a Function Call

In Listing 8.2, function `f` takes an integer as argument. We assume that function `f` is deterministic and pure. Meaning that it only depends on the value of its argument and it

$$\texttt{noR}(\underline{\texttt{a[i..a.length-1]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge$$
$$\texttt{noRaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge \texttt{noWaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N+1}$$

(a) Loop invariant

$$\texttt{noR}(\underline{\texttt{a[i..a.length-1]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge$$
$$\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\widehat{\texttt{noWaR}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\widehat{\texttt{noWaW}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N+1}$$

(b) Inter-iteration loop invariant

Figure 8.2.: Loop invariants for Listing 8.2

does not modify the heap. Therefore, function `f` does not read or write the array elements. Therefore, the generated loop invariants in Figures 8.2 are the same as the invariants generated for Listing 8.1.

```
i = 0;
while (i <= N) {
    a[i] = f(a[i]);
    i=i+1;
}
```

Listing 8.2: Array access with a Function

Intra-iteration Data Dependence

Similar to Listing 8.1, in Listing 8.3 there is a *WaR* data dependence on all elements of array `a`. In addition, there is a read in line 5 following the write in line 4, which together they form a *RaW* data dependence. The result is shown in Figure 8.3a and it is correctly excluding predicates noWaR and noRaW from the loop invariant.

Since both of the data dependence in Listing 8.3 are intra-iteration data dependences, they are both ignored by the inter-iteration loop invariant in Figure 8.3b.

```
1       i = 0;
2       sum = 0;
3       while (i <= N) {
4           a[i] = a[i] + 1;
5           sum = sum + a[i];
6           i = i + 1;
7       }
```

Listing 8.3: Intra-iteration data dependence

$$\texttt{noR}(\underline{\texttt{a[i..a.length-1]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge$$
$$\texttt{noWaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N+1}$$

(a) Loop invariant

$$\texttt{noR}(\underline{\texttt{a[i..a.length-1]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge$$
$$\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\widehat{\texttt{noWaR}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\widehat{\texttt{noWaW}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N+1}$$

(b) Inter-iteration loop invariant

Figure 8.3.: Loop invariants for Listing 8.3

$$\texttt{noR}(\underline{\texttt{a[0]} \cup \texttt{a[i+1..a.length-2]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-2]}}) \wedge$$
$$\texttt{noRaW}(\underline{\texttt{a[0..a.length-2]}}) \wedge \texttt{noWaW}(\underline{\texttt{a[0..a.length-2]}}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N}$$

(a) Loop invariant

$$\texttt{noR}(\underline{\texttt{a[0]} \cup \texttt{a[i+1..a.length-2]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-2]}}) \wedge$$
$$\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..a.length-2]}}, \texttt{empty}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\widehat{\texttt{noWaW}}(\underline{\texttt{a[0..a.length-2]}}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N}$$

(b) Inter-iteration loop invariant

Figure 8.4.: Loop invariants for Listing 8.4

Inter-Iteration Data Dependence

In Listing 8.4 there is *WaR* data dependence on all the elements of array a except $\underline{\texttt{a[0]}}$ as it is never read. The generated loop invariant is shown in Figure 8.4a.

```
i = 0;
while (i <= N-1) {
    a[i] = a[i+1];
    i=i+1;
}
```

Listing 8.4: Inter-iteration data dependence

As this data dependence is an inter-iteration data dependence, it is taken into account by the inter-iteration loop invariant in Figure 8.4b as well.

Inter- and Intra-Iteration Data Dependences

Similar to Listing 8.4, in Listing 8.5 there is a *WaR* data dependence. In addition, there is a read in line 5 after the write in line 4 which together form a *RaW* data dependence. Both data dependences are captured during the loop invariant generation process and the result is shown in Figure 8.5a.

As the *RaW* data dependence is an intra-iteration one, it is ignored by the inter-iteration loop invariant in Figure 8.5b.

```
1       i = 0;
2       sum = 0;
3       while (i <= N-1) {
4           a[i] = a[i+1];
5           sum = sum + a[i];
6           i=i+1;
7       }
```

Listing 8.5: Inter- and intra-iteration data dependences

$$\texttt{noR}(\underline{\texttt{a[i+1..a.length-2]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-2]}}) \wedge$$
$$\texttt{noWaW}(\underline{\texttt{a[0..a.length-2]}})$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N}$$

(a) Loop invariant (b) Inter-iterat

Figure 8.5.: Loop invariants for Listing 8.5

Although both loop invariants capture the data dependences precisely, they miss the predicate $\texttt{noR}(\underline{\texttt{a[0]}})$, due to the heuristic used for generating sub-arrays. Loosing the full precision is undesirable, but it does not harm the data dependence analysis goal in this testcase.

Conditional

In Listing 8.6, as mentioned in Subsection 2.4.3, when the symbolic execution engine reaches the if-then-else, it splits the proof into two branches. To represent both proof branches within a single loop invariant we join them using the symbolic state merging framework [57]. In both proof branches elements of array a are written, but there is no *WaW* data dependence on them. In addition, since elements of array a are not read there can not be any *RaW* or *WaR* data dependence on them. Figure 8.6 includes loop invariants generated for Listing 8.6.

Conditional with Different Number Of Memory Accesses in Different Branches

In Listing 8.7, there are different numbers of memory accesses in the if-then-else branches. While the if-clause causes a *WaR* data dependence, the else-clause only writes on the

```
i = 0;
while (i <= N) {
    if(i > N/2)
        a[i] = 1;
    else
        a[i] = 0;
    i=i+1;
}
```

Listing 8.6: Conditional write on array elements

$$\texttt{noR}(\underline{\texttt{a[0..a.length-1]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge$$
$$\texttt{noWaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N+1}$$

(a) Loop invariant

$$\texttt{noR}(\underline{\texttt{a[0..a.length-1]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge$$
$$\widehat{\texttt{noWaW}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N+1}$$

(b) Inter-iteration loop invariant

Figure 8.6.: Loop invariants for Listing 8.6

$$\texttt{noR}(\underline{\texttt{a[i..a.length-2]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-2]}}) \wedge$$
$$\texttt{noRaW}(\underline{\texttt{a[0..a.length-2]}}) \wedge \texttt{noWaW}(\underline{\texttt{a[0..a.length-2]}}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N}$$

(a) Loop invariant

$$\texttt{noR}(\underline{\texttt{a[0]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..N]}}) \wedge$$
$$\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..a.length-2]}}, \texttt{empty}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\widehat{\texttt{noWaW}}(\underline{\texttt{a[0..a.length-2]}}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N}$$

(b) Inter-iteration loop invariant

Figure 8.7.: Loop invariants for Listing 8.7

array elements. Differences between the two branches do not cause a loss of precision in the generated loop invariants in Figure 8.7a.

```
i = 0;
while (i < N) {
    if(i > N/2)
        a[i] = a[i+1];
    else
        a[i] = 0;
    i=i+1;
}
```

Listing 8.7: Conditional with different number of accesses in different branches

Since the *WaR* data dependence, it is correctly captured by the inter-iteration loop invariant in Figure 8.7b.

Both loop invariants capture the data dependences precisely, but they miss the predicate $\texttt{noR}(\underline{\texttt{a[0]}})$, due to the heuristic used for generating sub-arrays. This loss of precision does not harm the data dependence analysis goal.

Conditional with Similar Data Dependences in Different Branches

Listing 8.8, there is *WaR* data dependence in both branches. The *WaR* data dependence caused by the if-branch is an inter-iteration data dependence (unlike the one in the

$$\texttt{noR}(\underline{\texttt{a[i+1..N-1]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..N-1]}}) \wedge$$
$$\texttt{noRaW}(\underline{\texttt{a[0..N-1]}}) \wedge \texttt{noWaW}(\underline{\texttt{a[0..N-1]}}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq 0 \wedge \texttt{i} \leq \texttt{N}$$

(a) Loop invariant

$$\texttt{noR}(\underline{\texttt{a[i+1..N-1]}}) \wedge \texttt{noW}(\underline{\texttt{a[0]}} \cup \underline{\texttt{a[i..N-1]}}) \wedge$$
$$\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..N-1]}}, \texttt{empty}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\widehat{\texttt{noWaW}}(\underline{\texttt{a[0..N-1]}}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq 0 \wedge \texttt{i} \leq \texttt{N}$$

(b) Inter-iteration loop invariant

Figure 8.8.: Loop invariants for Listing 8.8

else-branch) and it is captured by the inter-iteration loop invariant. Loop invariants for Listing 8.8 are shown in Figure 8.8.

```
i = 0;
while (i <= N-1) {
    if(i > N/2)
        a[i] = a[i+1];
    else
        a[i] = a[i] + 1;
    i=i+1;
}
```

Listing 8.8: Conditional with similar data dependences in different branches

Conditional with Different Kind of Data Dependences in Different Branches

In Listing 8.9, the first half of the array is shifted to the right and the second half is shifted to the left. Therefore, there is *RaW* data dependence on the first half and *WaR* data dependence on the second half. The generated loop invariant is shown in Figure 8.9a over-approximates both of the data dependences over the whole array. This over-approximation is due to the adoption of the merge technique [57]. Since both of the data dependences are inter-iteration they are allowed in the inter-iteration loop invariant in Figure 8.9b.

```
i = 1;
while (i <= N-1) {
    if(i > N)/2)
        a[i] = a[i+1];
    else
        a[i] = a[i-1];
    i=i+1;
}
```

Listing 8.9: Conditional with different kind of data dependences in different branches

$$\texttt{noR}(\underline{\texttt{a[i+1..a.length-2]}}) \wedge \texttt{noW}(\underline{\texttt{a[0]}} \cup \underline{\texttt{a[i..a.length-2]}}) \wedge$$
$$\texttt{noWaW}(\underline{\texttt{a[0..a.length-2]}}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq 1 \wedge \texttt{i} \leq \texttt{N}$$

(a) Loop invariant

$$\texttt{noR}(\underline{\texttt{a[i+1..a.length-2]}}) \wedge \texttt{noW}(\underline{\texttt{a[0]}} \cup \underline{\texttt{a[i..a.length-2]}}) \wedge$$
$$\widehat{\texttt{noWaW}}(\underline{\texttt{a[0..a.length-2]}}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq 1 \wedge \texttt{i} \leq \texttt{N}$$

(b) Inter-iteration loop invariant

Figure 8.9.: Loop invariants for Listing 8.9

$$\texttt{noR}(\underline{\texttt{a[0]} \cup \texttt{a[i+1..a.length-1]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge$$
$$\texttt{noRaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge \texttt{noWaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N+1}$$

(a) Loop invariant

$$\texttt{noR}(\underline{\texttt{a[0]} \cup \texttt{a[i+1..a.length-1]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge$$
$$\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\widehat{\texttt{noWaW}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N+1}$$

(b) Inter-iteration loop invariant

Figure 8.10.: Loop invariants for Listing 8.10

Inter-iteration Data Dependence with Abrupt Termination

In Listing 8.10, array elements are shifted to the left. The generated loop invariants in Figure 8.10 show that our approach can deal with abrupt termination of loops.

```
i = 0;
while (i <= N) {
    if(i == N)
        break;
    else
        a[i] = a[i+1];
    i=i+1;
}
```

Listing 8.10: Array shift program with abrupt termination

Stencil

Stencil programs are a subset of HPC applications that involves repeatedly updating elements of arrays according to fixed patterns which is usually a function of neighboring elements.

Listing 8.11 is an example of a stencil over a one-dimensional array where elements value depend on their neighboring elements.

```
i = 1;
while (i < N) {
    a[i] = a[i-1] + a[i+1];
    i = i + 1;
}
```

Listing 8.11: A stencil over a one-dimensional array

$$\texttt{noR}(\underline{\texttt{a[i+1..a.length-2]}}) \land \texttt{noW}(\underline{\texttt{a[i..a.length-2]}}) \land$$
$$\texttt{noWaW}(\underline{\texttt{a[0..a.length-2]}}) \land$$
$$\texttt{a} \neq \texttt{null} \land \texttt{a.length} > \texttt{N} \land \texttt{i} \geq \texttt{0} \land \texttt{i} \leq \texttt{N}$$

(a) Loop invariant

$$\texttt{noR}(\underline{\texttt{a[i+1..a.length-2]}}) \land$$
$$\texttt{noW}(\underline{\texttt{a[0]}} \cup \underline{\texttt{a[i..a.length-1]}}) \land$$
$$\widehat{\texttt{noWaW}}(\underline{\texttt{a[0..a.length-2]}}, \texttt{empty}, \texttt{empty})$$
$$\texttt{a} \neq \texttt{null} \land \texttt{a.length} > \texttt{N} \land \texttt{i} \geq \texttt{1} \land \texttt{i} \leq \texttt{N}$$

(b) Inter-iteration loop invariant

Figure 8.11.: Loop invariants for Listing 8.11

Non-linear Array Index

This experiment shows that our approach is capable of analyzing loops over arrays with non-linear indices.

In Listing 8.12 there is intra-iteration *WaR* data dependence on array elements `a[0]` and `a[1]`.

```
while (i <= N) {
    a[i] = a[i^2];
    i = i + 1;
}
```

Listing 8.12: Non-affine array access

Figure 8.12a shows the generated loop invariant. Predicate noW(a[i..a.length-1]) stating that from `i` onward no array elements is written, implies that there are no data dependence on a[i..a.length-1]. Although sound, this result is not precise enough. A precise result would also include:

$$\texttt{noWaR}(\underline{\texttt{a[2..a.length-1]}}) \wedge$$
$$\texttt{noRaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge$$
$$\texttt{noWaW}(\underline{\texttt{a[0..a.length-1]}}).$$

The inter-iteration data dependence loop invariant is shown in Figure 8.12b. With covering $\widehat{\texttt{noWaW}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty})$ it is more precise than the loop invariant in Figure 8.12a, but it lacks the following predicate to be fully precise:

$$\widehat{\texttt{noWaR}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty}, \texttt{empty}) \wedge$$
$$\widehat{\texttt{noRaW}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty}, \texttt{empty}).$$

8.1.2. Evaluation

Tables 8.1 and 8.2 show the experimental results for our general loop invariant generation approach and the inter-iteration data dependence loop invariant, respectively. The average time over three runs is shown in their third columns.

There are different factors affecting the time needed for generating the loop invariant. For example, the number of iterations that the loop invariant generation algorithm needs to reach the fixpoint. The fourth columns (No. of Iterations) show this factor. The presence

$$\texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N+1}$$

(a) Loop invariant

$$\texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge$$
$$\widehat{\texttt{noWaW}}(\underline{\texttt{a[0..a.length-1]}}, \texttt{empty}, \texttt{empty})$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N+1}$$

(b) Inter-iteration loop invariant

Figure 8.12.: Loop invariants for Listing 8.12

of conditionals and complicated array access patterns also affect the time needed for data dependence loop invariant generation.

The number of proofs for symbolic execution and generating abstract predicates is shown in the fifth columns (No. of Proofs).

The generated data dependence loop invariant is correct if it does not under-approximate data dependences, and it is precise if it does not over-approximate. All of the generated data dependence loop invariants are correct. In presence of conditionals (Listings 8.6 - 8.9) our approach can lose precision. It is worth noting that the inter-iteration data dependence loop invariant for Listing 8.6 is precise, as the conditional statement is only affecting the intra-iteration data dependences.

Our approach loses precision while encountering non-affine array access Listing 8.12) due to the design of sub-array generation (part of the weakening Algorithm 2). Compared to the state-of-the-art approaches, we have the advantage of being able to analyze such programs at all.

8.2. Nested Loops

Similar to Section 8.1, in the following test cases, except for Multiple Arrays in Section 8.2.1, the precondition is:

$$\texttt{noDep}(\underline{\texttt{a[0..a.length-1]}}) \wedge$$
$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{a[0].length} > \texttt{M}.$$

Table 8.1.: Data dependence loop invariant generation for single loops

Test case	Precise	Time (s)	No. of Iterations	No. of Proofs
Listing 8.1	✓	21.1	3	145
Listing 8.2	✓	21.1	3	145
Listing 8.3	✓	23.4	3	132
Listing 8.4	✓	74.7	4	161
Listing 8.5	✓	32.1	3	136
Listing 8.6	×	20.9	3	118
Listing 8.7	×	130.5	4	215
Listing 8.8	×	26.8	3	130
Listing 8.9	×	36.6	4	143
Listing 8.10	✓	28.2	3	154
Listing 8.11	✓	43.2	3	143
Listing 8.12	×	57.7	3	164

Table 8.2.: Inter-iteration data dependence loop invariant generation for single loops

Test case	Precise	Time (s)	No. of Iterations	No. of Proofs
Listing 8.1	✓	16.7	3	158
Listing 8.2	✓	17.8	3	158
Listing 8.3	✓	21.8	3	164
Listing 8.4	✓	20.5	4	164
Listing 8.5	✓	21.7	3	144
Listing 8.6	✓	14.0	3	112
Listing 8.7	×	44.7	3	154
Listing 8.8	×	23.7	3	144
Listing 8.9	×	24.2	3	182
Listing 8.10	✓	16.7	3	154
Listing 8.11	✓	21.33	3	144
Listing 8.12	×	20.6	4	164

$$\texttt{noR}(\underline{\texttt{a[i..N-1][0..M-1]}}) \land \texttt{noW}(\underline{\texttt{a[0..N-1][0..M-1]}}) \land$$
$$\texttt{a} \neq \texttt{null} \land \texttt{a.length} > \texttt{N} \land \texttt{a[0].length} > \texttt{M} \land$$
$$\texttt{i} \geq \texttt{0} \land \texttt{i} \leq \texttt{N} \land \texttt{j} \geq \texttt{0} \land \texttt{j} \leq \texttt{M}$$

Figure 8.13.: Loop invariants for Listing 8.13

Our approach demands matrices to have a rectangular shape (see Chapter 7), where the length of all rows are equal to `a[0].length`. Predicate $\texttt{a[0].length} > \texttt{M}$ emphasizes that lower bound of the length of row `0` is known. In this section, we include an imperfect nest of loops to show that our approach can deal with such cases in opposition to tools that only handle perfectly nested loops.

At the time of writing this thesis, the implementation of our approach for generating inter-iteration loop invariants for nested loops is not yet finished. Therefore, this section only presents the result of implementing our general loop invariant generation for nested loops.

8.2.1. Test Cases

Read from Matrix

In Listing 8.13, the assumption is that function `f` is deterministic and pure. As a result, the loop is only reading the elements of matrix `a`. Therefore, there is no data dependence on matrix `a`. The generated loop invariant in Figure 8.13 states that there is no write on matrix `a` which implies that there can not be *WaR*, *RaW*, or *WaW* data dependence on this matrix.

```
i = 0;
while (i <= N - 1) {
    j = 0;
    while (j <= M - 1) {
        f(a[i][j]);
        j = j+1;
    }
    i = i+1;
}
```

Listing 8.13: Nested loop reading from a matrix

$$
\begin{aligned}
&\texttt{noR}(\underline{\texttt{a[i..a.length-2][0..a[0].length-2]}}) \wedge \\
&\texttt{noW}(\underline{\texttt{a[0..a.length-2][0..a[0].length-2]}}) \wedge \\
&\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{a[0].length} > \texttt{M} \wedge \\
&\texttt{i} \geq 0 \wedge \texttt{i} \leq \texttt{N} \wedge \texttt{j} \geq 0 \wedge \texttt{j} \leq \texttt{M}
\end{aligned}
$$

Figure 8.14.: Loop invariants for Listing 8.14

Conditional Read

In Listing 8.14, similar to Listing 8.13, the assumption is that the function `f` does not read and write on the elements of matrix `a`. The loop is only reading the elements of matrix `a` when the if-condition holds. The generated loop invariant in Figure 8.14, same as Figure 8.13, states that there is no write on matrix `a`. This implies that there can not be *WaR*, *RaW*, or *WaW* data dependence on matrix `a`.

```
i = 0;
while (i <= N - 1) {
    j = 0;
    while (j <= M - 1) {
        if(j < (M/2))
            f([i][j]);
        j = j+1;
    }
    i = i+1;
}
```

Listing 8.14: Conditional read from matrix

Intra-Iteration Data Dependence in the Inner Loop

Listing 8.15 shows a nested loop with intra-iteration *WaR* data dependence in the inner loop. The outer loop invariant in Figure 8.15 captures this data dependence and shows that there is no *RaW* or *WaW* data dependence on matrix `a`. In addition, it shows that at each iteration `i`, array elements of row `i` and onward are not read and are not written.

```
while (i <= N - 1) {
    j = 0;
    while (j <= M - 1) {
        a[i][j] = a[i][j] + 1;
        j = j+1;
    }
    i = i+1;
}
```

Listing 8.15: Nested loop with intra-iteration *WaR* data dependence

$$
\begin{aligned}
&\texttt{noR}(\underline{\texttt{a[i..a.length-2][0..a[0].length-2]}}) \wedge \\
&\texttt{noW}(\underline{\texttt{a[i..a.length-2][0..a[0].length-2]}}) \wedge \\
&\texttt{noRaW}(\underline{\texttt{a[0..a.length-2][0..a[0].length-2]}}) \wedge \\
&\texttt{noWaW}(\underline{\texttt{a[0..a.length-2][0..a[0].length-2]}}) \wedge \\
&\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{a[0].length} > \texttt{M} \wedge \\
&\texttt{i} \geq 0 \wedge \texttt{i} \leq \texttt{N} \wedge \texttt{j} \geq 0 \wedge \texttt{j} \leq \texttt{M}
\end{aligned}
$$

Figure 8.15.: Loop invariants for Listing 8.15

Inter-Iteration Data Dependence in the Inner Loop

In Listing 8.16 there is inter-iteration *WaR* data dependence in the inner loop. The generated loop invariant for the outer loop in Figure 8.16 shows that this data dependence is correctly detected. Predicates about the matrix not being null and its dimensions lower bounds are preserved by the loop and therefore appear in the loop invariant. In addition, predicates showing the lower and upper bounds of the loop indices are generated precisely. In this experiment, the maximum number of steps per proof is 10000.

```
while (i <= N - 1) {
    j = 0;
    while (j <= M - 2) {
        a[i][j] = a[i][j + 1];
        j = j+1;
    }
    i = i+1;
}
```

Listing 8.16: Shifting row elements to left

$$
\begin{aligned}
&\texttt{noR}(\underline{\texttt{a[i..a.length-2][0..a[0].length-2]}}) \wedge \\
&\texttt{noW}(\underline{\texttt{a[i..a.length-2][0..a[0].length-2]}}) \wedge \\
&\texttt{noRaW}(\underline{\texttt{a[0..a.length-2][0..a[0].length-2]}}) \wedge \\
&\texttt{noWaW}(\underline{\texttt{a[0..a.length-2][0..a[0].length-2]}}) \wedge \\
&\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{a[0].length} > \texttt{M} \wedge \\
&\texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N} \wedge \\
&\texttt{j} \geq \texttt{0} \wedge \texttt{j} \leq \texttt{M-1}
\end{aligned}
$$

Figure 8.16.: Loop invariants for Listing 8.16

Inter-Iteration Data Dependence in the Outer Loop

In Listing 8.17, there is inter-iteration *WaR* data dependence in the outer loop. It is correctly captured by the precise loop invariant generated for the outer loop in Figure 8.17. In this experiment, the maximum number of steps per proof is 10000.

```
while (i <= N - 2) {
    j = 0;
    while (j <= M - 1) {
        a[i][j] = a[i+1][j];
        j = j+1;
    }
    i = i+1;
}
```

Listing 8.17: Shifting matrix rows up

Multiple Arrays

This experiment shows that our approach is capable of dealing with multiple arrays. In this test case the post condition is true as usual, and the precondition is:

$$
\begin{aligned}
&\texttt{noDep}(\underline{\texttt{a[0..a.length-1]}}) \wedge \texttt{noDep}(\underline{\texttt{b[0..b.length-1]}}) \\
&\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{b} \neq \texttt{null} \wedge \texttt{b.length} > \texttt{M} \wedge \texttt{N} \doteq \texttt{M}.
\end{aligned}
$$

Figure 8.18 shows the precise loop invariant generated for Listing 8.18.

$$
\begin{aligned}
&\texttt{noR}(\underline{\texttt{a[i+1..a.length-2][0..a[0].length-2]}}) \wedge \\
&\texttt{noW}(\underline{\texttt{a[i..a.length-2][0..a[0].length-2]}}) \wedge \\
&\texttt{noRaW}(\underline{\texttt{a[0..a.length-2][0..a[0].length-2]}}) \wedge \\
&\texttt{noWaW}(\underline{\texttt{a[0..a.length-2][0..a[0].length-2]}}) \wedge \\
&\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \texttt{a[0].length} > \texttt{M} \wedge \\
&\texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N-1} \wedge \texttt{j} \geq \texttt{0} \wedge \texttt{j} \leq \texttt{M}
\end{aligned}
$$

Figure 8.17.: Loop invariants for Listing 8.17

```
i = 0;
while (i <= N - 1) {
    j = 0;
    while (j <= M - 1) {
        a[i] = b[j];
        j = j+1;
    }
    i = i+1;
}
```

Listing 8.18: Multiple arrays

$$
\begin{aligned}
&\texttt{noW}(\underline{\texttt{a[i..a.length-2]}}) \wedge \texttt{noR}(\underline{\texttt{b[j..b.length-2]}}) \wedge \\
&\texttt{noW}(\underline{\texttt{b[0..b.length-2]}}) \wedge \texttt{noR}(\underline{\texttt{a[0..a.length-2]}}) \wedge \\
&\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > \texttt{N} \wedge \\
&\texttt{b} \neq \texttt{null} \wedge \texttt{b.length} > \texttt{M} \wedge \\
&\texttt{N} \doteq \texttt{M} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{N} \wedge \texttt{j} \geq \texttt{0} \wedge \texttt{j} \leq \texttt{M}
\end{aligned}
$$

Figure 8.18.: Loop invariants for Listing 8.18

Table 8.3.: Data dependence loop invariant generation results for nested loops

Test case	Precise	Time (min)	No. of Iterations	No. of Proofs	Proof Steps
Listing 8.13	✓	14.7	3	2046	5000
Listing 8.14	×	14.8	3	2046	5000
Listing 8.15	✓	37.9	3	2389	10000
Listing 8.16	✓	54.4	3	2452	10000
Listing 8.17	✓	73.1	3	2769	10000
Listing 8.18	✓	0.65	4	333	5000

8.2.2. Evaluation

Table 8.3 shows the result of our experiments with nested loops. All the generated loop invariants are correct. Except for the loop invariant of Listing 8.14, all the results are precise. In the generation of loop invariant for Listing 8.14, our approach loses precision due to the presence of a conditional statement.

Unlike the results for single loops in Table 8.1, which our approach on average takes around a minute to generate results, it takes several minutes to generate data dependence loop invariants for nested loops over matrices. This is due to the presence of matrices rather than nests of loops. As for Listing 8.18, our approach takes less than a minute to analyze a nested loop over one-dimensional arrays.

Number of iterations that the loop invariant generation algorithm needs to reach the fixpoint is shown in the fourth column (No. of Iteration). Column *No. of Proofs* shows the number of proofs during the loop invariant generation process, and *Proof Steps* column shows the maximum number of steps each proof is allowed to take.

8.3. Threats to Validity

Concerning *internal* threats to validity, there is the question whether our results for Java can be transferred to C, the prevalent HPC language. This is possible, because the code of interest in the HPC domain is restricted to loops over arrays with indexed access of an array element being the sole use of pointer arithmetic. This maps precisely to the dependences created by array accesses in Java. In fact, the programs used in our experiments were obtained as a one-to-one model from their C counterparts.

Checking correctness of experimental outcomes and their interpretation is straightforward. Also the outcome, up to minor variations in observed user time, is deterministic.

As data dependence analysis is part of the parallelization process, state-of-the-art tools do not report statistics of their data dependence analysis approaches separately. Tools like [61] that report such statistics report it as the number of data dependences found in the *whole* program. Therefore, there is no fair chance for a one-to-one comparison here.

The main *external* threat is the relatively small number and size of performed experiments. However, this threat is mitigated for *methodological* reasons. The size and number of code samples required for a representative selection is much smaller than for dynamic and even for static approaches, because: (i) symbolic execution and the abstract domain used in dependence invariants are able to detect and to express the desired dependences *by design*; (ii) the symbolic approach can analyze small code fragments containing loops that are the parallelization targets in isolation; (iii) complex functional properties can be ignored.

Hence, we only need to ensure that the examples are representative for the HPC application domain and cover to a sufficient degree the phenomena causing over- or under-approximation in other approaches [10–15, 18, 19, 21, 22, 25, 62, 63]. These are well-known and discussed in the examples above. In particular, our test cases contain loops with conditional statements and we have shown that we can maintain correctness, while dynamic data dependence analysis approaches can under approximate in presence of conditionals and lead to incorrect results. Besides, our approach supports analyzing non-linear array indices while dynamic approaches simply can not.

In addition, we need to ensure that our analysis is indeed *automatic*, as claimed.

We do not claim performance properties of our tool due to the small number of examples and prototype status of the tool. We stress that we do not need to apply our tool to large code bases (of thousands of code lines) but can work on isolated code sequences or loops only. This is sufficient for our application scenario of code parallelization.

9. Related Work

Related work to this thesis are mainly categorized into two groups: (i) data dependence profilers (Section 9.1), and (ii) loop invariant generation (Section 9.2). Section 9.3 covers a few other topics that are investigating similar problems.

9.1. Data Dependence Profilers

Detecting data dependences in loops is often called *data dependence test*, or *data dependence profiling*. As in most cases loops iterate over arrays, *array data dependence test* is also common. Even if dependences are known precisely, which is a special case of reachability and, therefore, undecidable, the problem is NP-hard [64]. It can be approximated statically, dynamically, or in a hybrid fashion.

9.1.1. Static

Static auto-parallelizing tools and compilers use static data dependence analysis techniques. Since they only use information available at compile time they are also called *compiler-based* methods. These techniques conservatively over-approximate data dependences that might not occur at runtime, as often the value of pointers and array indices can not be resolved at compile time. Reporting false positives can result in lost parallelization opportunities. Another shortcoming of these techniques and tools developed based on them is that they are limited to loops with affine linear bounds and array indices [2, 4, 5, 10, 13].

On the other hand, these approaches are sound [8] as they do not miss any data dependences.

Static dependence analyses [1–9] use several techniques that can be categorized as the following: (i) systems of linear inequality constraints solved using integer programming [1, 2, 4, 7]; (ii) graph structures like reference trees [5], program dependence graphs [65, 66]; (iii) renaming [3, 67]; (iv) symbolic value propagation [6].

In [2] data dependence analysis is reduced to finding an integer solution for set of inequality constraints. In this approach array and loop indices must be linear functions.

An integer programming algorithm is suggested in [4] to find data dependences between array elements in nested loops where the loop bounds and array indices are affine functions of loop indices. The Omega test [7] approximates dependence direction and distance vector with integer programming. Describing the relation between iterations in which the conflicting read/write occurs. Using dependence difference parameter was the approach chosen in [1] leading to an exact method for data dependence analysis under very restricted conditions. Static parallelization tools like autoPar [13] use such techniques for data dependence analysis.

Reference tree is a decision tree introduced in [5] to approximate data dependences. It is a relation between two array access which maps one iteration of array access into all corresponding data dependents of another iteration. Other static data dependence analysis approaches, use Program Dependence Graph (PDG) [65, 66]. Each PDG node represents a statement of the program, and edges represent either control or data dependences. JOANA [9] is a tool for software security analysis which uses PDG to find data dependences. Through data dependences, it checks the leak in information flow. But this approach also over-approximates data dependences since it is not value sensitive. LLVM [68] data dependence analysis (used by [11]) uses PDG to extract data dependences but it does not have indices, subscripts or multi-dimensional array accesses.

Starting from process tree, identify inter-process dependences, and then perform selective renaming, is the approach introduced in [3] to find data dependences.

Selective renaming is a version of the renaming technique introduced in [67]. Renaming statically introduces a new name to distinguish independent occurrence of a variable or dynamically allocates separate storage for a given static occurrence [3].

An approximate value-based approach is suggested in [6]. It performs a symbolic value set propagation using a monotone data flow system to compute data dependence.

These approaches are conservative and over-approximate data dependences, i.e. they might report false dependences, but do not miss existing ones [8]. In contrast, our approach is based on a fully precise and faithful program logic using symbolic execution interleaved with first-order reasoning to prove the absence of data dependences.

We are not limited to affine nested loops, and the loop index does not need to change linearly. Our logic is able to represent non-linear integer constraints and the calculus is able to prove some non-linear problems automatically. We have the possibility to export sub-goals to be solved by SMT solvers [69]. Because of this, we can profit from the quickly evolving capabilities in the automated reasoning area.

9.1.2. Dynamic

Dynamic parallelization tools [18–22] use dynamic data dependence analysis techniques which captures data dependences at runtime. These analyses are optimistic (under-approximating) as they rely on actual program runs. Therefore they might miss data dependences that do not belong to the paths traversed in the current execution. In addition, these techniques have an enormous runtime overhead (factor of 100 or more [26]).Increasing the diversity of inputs, enlarges the set of data dependences identified by dynamic analysis tools but also increases the overhead.

SD3 [18] and Prospector [19] divide the memory address space into multiple sections and perform data-dependence analysis on the separately. As most of other parallelization tools Prospector only analyzes loops, but SD3 takes the whole program into account. Kremlin [20] uses hierarchical critical path analysis and profiles data dependences only within specific code regions. Alchemist [21] and Tareador [22] use Valgrind [62] for data dependence analysis. Valgrind is a dynamic binary analysis tool that uses shadow values to track read and write accesses on memory locations and registers. Parwiz [16] and Parceive [17] use a dynamic binary instrumentation tool (Intel Pin [63]) to instrument the binary code of a program and detect data dependences.

Kremlin [20] is a parallelization recommendation system that profiles data dependences only within specific code regions. It compresses memory accesses with stride patterns.

Our approach is static, but it uses symbolic execution and in this way retains some advantages of the dynamic approaches: for example, it can distinguish between different symbolic execution paths. The advantage is that even for a failed proof attempt (caused by insufficient automation or by presence of data dependence), all path conditions from proven symbolic execution paths can be safely used in conditional parallelization by compilers.

9.1.3. Hybrid

There are a few tools that combine static and dynamic data dependence analysis. Sampaio et al. [23] apply static dependence analysis and use the result for parallelization. It then generates tests to validate at runtime whether the can be taken. Rus et al. [24] use static analysis. They formulate conditions and insert them into the source code. These conditions evaluate at runtime whether a loop can be parallelized or not.

DiscoPoP [25, 26] is a parallelization discovery tool that records memory accesses in a signatures data structure. Signature is encoding an approximate representation of an unbound set of elements with a bounded amount of state [70]. Using signatures reduces memory overhead. DiscoPoP runs dynamic analysis only for code sections where the data

dependence detection requires runtime information, the framework has a lower profiling overhead than the other approaches named here. DiscoPoP is focused on enhancing the speed in finding parallelization opportunities. It uses the combination of the following techniques to profile data dependences in a hybrid fashion.

- Using PLUTO to statically extract data dependences from loops. Then, excluding loops and profiling only data dependences outside the loops. In the end, merging static and dynamic dependences. This technique is limited to polyhedral loops.
- Using LLVM, statically identifying data dependences of scalar variables, not including passed-by-reference parameters and pointers. Then, identifying memory instructions that create the dependences and excludes them from parallelization. This method, however, does not skip profiling polyhedral loops.

Compared to the hybrid tools, our approach is not limited to specific shapes of loops or functions to produce sound results. In addition, we support multidimensional arrays and pointers that are left out of analysis by the hybrid tools mentioned above.

9.2. Loop Invariant Generation

Different loop invariant generation techniques are general/generic in a sense that they don't cover data dependences, in contrast we're domain specific. On the other hand, these techniques are not general enough because they come with many limitations (only covering polynomials, no arrays, only int, etc.), in contrast we cover a larger grammar.

Loop Invariant Generation with Predicate Abstraction

There is a vast research corpus on loop invariant generation using a variety of techniques like abstraction interpretation (including shape analysis and predicate abstraction) [53, 71, 72], templates [73–76], recurrence equations [77, 78], learning [79], etc. Our approach is an extension of [56] to loop invariant generation for dependence predicates. All mentioned approaches focus on *functional* properties of a loop. To the best of our knowledge, none of them has been adapted to generate loop invariants that capture (the absence of) data dependences.

Flanagan [80] and Boogie verifier [81] approaches, also abstract a loop with predicates containing free variables. But the logical calculus that quantifies over these variables is different from the logical calculus that uses the invariant in a proof. Using the same logical calculus for generating the loop invariant as the logical calculus that uses the loop invariant in a proof, distinguishes our approach from Flangen [80] and Boogie verifier [81]

approaches. Our approach has the advantages of (i) being able to reuse the framework, and (ii) not needing to translating the syntax of the generated loop invariant.

Loop Invariant Generation for Nested Loops

Loop invariant generation for nested loops with conditionals is rarely addressed in literature. The method in [82] generates loop invariants only when program variables are integers and assignments to program variables are polynomial expressions. In contrast, we allow arrays in addition to integer variables and assignments can include function and procedure calls on program variables.

9.3. Others

Information Flow

There is some resemblance between data dependences and information flow [83]. Indeed, both problems can be approached by deductive verification (our approach and [84, 85], respectively). But whereas information flow is a relational property, data dependence is essentially a reachability problem. Also, the latter is more fine-grained.

Static Analysis of Array Programs

Static analysis of array programs is another line of research close to our work. The approach in [86] formulates the problem as a reasoning about permissions. They analyze the loop body to obtain a permission precondition for a single loop iteration. Permission is expressed as a maximum over the variables changed by the loop execution. Unlike our approach, they only record type of memory accesses and not their order.

Specification Techniques for Capturing Loop Data Dependences

Specification techniques for capturing loop data dependences for correct loop parallelization is another aspect of our work. Blom et. al. [87] suggests to generate iteration contracts to capture data dependences between different loop iterations. Their technique needs user to provide part of the specification that our approach can generate automatically.

10. Conclusion and Future Work

This thesis stems from the long-overlooked idea of solving HPC problems with formal approaches. State-of-the-art data dependence analysis tools suffer from over-approximation and, occasionally, under-approximation. In contrast, our approach provides highly precise and sound results.

In this thesis, we presented a static approach for data dependence analysis based on deductive verification in an expressive program logic. It differs from existing approaches in several important aspects and has the following advantages over state-of-the-art techniques:

- program logic with a formally and mathematically rigorous semantics of data dependences;
- sound calculus to reason about the program logic;
- loop invariant generation for data dependences; and
- highly precise results.

We defined data dependences formally and presented mathematically rigorous semantic definitions. In addition, we provided a calculus for reasoning about absence of data dependences in programs and we can argue formally about its soundness. To enable automated reasoning, we presented a loop invariant generation technique based on predicate abstraction to infer data dependence properties about loops.

Our approach is fully automatic. It proves the absence of dependences with full precision for loop-free and non-recursive programs. Equipped with automatic loop invariant generation, it performs data dependence analysis with high precision in the presence of loops. Our loop invariant generation approach has no under-approximation, meaning no data dependence is missed. This enables a safe parallelization based on our analysis. Although over-approximation can happen, our heuristics are highly precise as shown in Chapter 8. They provide full precision (modulo arithmetic) for a sub-class of loops over arrays with affine array index and linear changes in the loop counter. However, they need

further adjustments to provide the same level of precision for other applications, e.g., non-affine array accesses and nonlinear loop counters.

The inter-iteration data dependence analysis introduced in Chapter 5 makes our analysis an ideal starting point for parallelizing loop iterations (e.g., applying DoAll, reduction, and geometric decomposition patterns). The more assertive approach introduced in Chapter 4 that captures both intra- and inter-iteration data dependences is the more suitable option for parallelizing loop bodies (e.g., applying pipeline pattern).

We feel encouraged that our prototype can replicate and improve results obtained from leading data dependence analysis tools. In the future, we plan to take our approach further in the following aspects.

Using Method Contracts

Formulating data dependence analysis problems as formal verification problems provides an opportunity for using the powerful body of work in formal verification, for example using method contracts.

Currently, we either inline function calls or treat them as a black box. In the latter case we assume that the function does not read and write the data under analysis. For example, in Listing 10.1 that array elements are arguments of function `f`. Assume that the function does not read and write, the following data dependence loop invariant is generated:

$$\begin{aligned}&\texttt{noR}(\underline{\texttt{a[i..a.length-1]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge \\ &\texttt{noRaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge \texttt{noWaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge \\ &\texttt{a} \neq \texttt{null} \wedge \texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{a.length}.\end{aligned}$$

One possible direction for future work is to incorporate contracts of the called methods in data dependence analysis. Assume the contract of method `f` states that it writes on the array elements. This means that the write happening in the loop is happening after the write in function `f`. Therefore there is *WaW* data dependence on location set $\underline{\texttt{a[i]}}$. By taking contract of function `f` into account the following loop invariant should be generated:

$$\begin{aligned}&\texttt{noR}(\underline{\texttt{a[i..a.length-1]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge \\ &\texttt{noRaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge \\ &\texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{a.length}.\end{aligned}$$

```
i = 0;
while (i <= a.length - 1) {
    a[i] = f(a[i]);
    i=i+1;
}
```

Listing 10.1: Array access with a function

Incorporating method contracts will make our approach more modular. Extending our program logic to use and generate data dependence method contracts can eventually make the data dependence analysis of recursive functions possible.

Loop Invariants with Disjunction

Invariants in loops over conditional statements, would be more precise if we take the if-then-else conditions into account. This goal can be achieved by adapting the merge technique [57] differently.

For example, for the loop in Listing 10.2 the current approach produces the following invariant which is mere conjunctions over predicates:

$$\begin{aligned}&\texttt{noR}(\underline{\texttt{a[0]}}) \wedge \texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge \\ &\texttt{noRaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge \texttt{noWaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge \\ &\texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{a.length-1}.\end{aligned}$$

```
i = 0;
while (i < a.length - 1) {
    if(i > (a.length - 1)/2)
        a[i] = a[i+1];
    else
        a[i] = 0;
    i=i+1;
}
```

Listing 10.2: Conditional with Different Memory Accesses

While adapting the merge technique [57] differently results in the following invariant with a disjunction over the if-then-else condition:

$$\begin{aligned}&((\texttt{i > (a.length - 1)/2} \wedge \texttt{noR}(\underline{\texttt{a[0]}}) \wedge \texttt{noRaW}(\underline{\texttt{a[0..a.length-1]}})) \\ &\vee \\ &(\texttt{i} \leq\texttt{(a.length - 1)/2} \wedge \texttt{noR}(\underline{\texttt{a[0..a.length-1]}}))) \wedge \\ &\texttt{noW}(\underline{\texttt{a[i..a.length-1]}}) \wedge \texttt{noWaW}(\underline{\texttt{a[0..a.length-1]}}) \wedge \\ &\texttt{i} \geq \texttt{0} \wedge \texttt{i} \leq \texttt{a.length-1}.\end{aligned}$$

The latter invariant hints at a parallelization tool to split the loop into halves by applying the reduction pattern. Then, parallelize the second half by applying the DoAll pattern, and parallelize the first half only after taking a restructuring step (due to existence of *WaR*).

Improving Precision

Although we offer high precision for a considerable sub-set of HPC applications, there is room for improvement in particular for non-affine and nonlinear array accesses and loop counters.

Merging into Parallelization Tools

Data dependence analysis is a step in the parallelization process. Our approach can be integrated into an auto-parallelizer or a parallelization recommendation system. Parallelization tools can delegate data dependence profiling of loops, and verification of the profiled data dependences of loop-free programs to our tool.

Usage in Information Security

There is an open question that whether our data dependence analysis approach can be used in the context of information flow security. For example, if early on we can show that a confidential information is not read we can show that there is no information leak. Therefore, our approach might be helpful to improve efficiency of information flow analysis.

Generating Method Contracts

Part of each method contract states the memory locations that affect the execution of the method (accessible in JML) and the locations that are affected by its execution (assignable in JML). Both of these memory location sets can be formulated using data dependence predicates, e.g., all memory locations except ls where noR(ls) affect the execution of a method. Our approach can be enhanced to generate these parts of method contracts automatically.

Bibliography

[1] William Pugh and David Wonnacott. “An Exact Method for Analysis of Value-based Array Data Dependences”. In: *Lang. and Compilers for Parallel Computing, 6th Intl. Workshop*. Ed. by Utpal Banerjee et al. Vol. 768. LNCS. Springer, 1993, pp. 546–566.

[2] Utpal Banerjee. “An introduction to a formal theory of dependence analysis”. In: *The Journal of Supercomputing* 2.2 (1988), pp. 133–149. DOI: 10.1007/BF00128174.

[3] Frances E. Allen et al. “A framework for determining useful parallelism”. In: *Proc. 2nd Intl. Conf. on Supercomputing*. Ed. by Jacques Lenfant. ACM, 1988, pp. 207–215. DOI: 10.1145/55364.55385.

[4] Paul Feautrier. “Dataflow analysis of array and scalar references”. In: *Intl. J. of Parallel Programming* 20.1 (1991), pp. 23–53. DOI: 10.1007/BF01407931.

[5] Dror E. Maydan, S. Amarsinghe, and Monica S. Lam. “Data Dependence and Data-Flow Analysis of Arrays”. In: *Lang. and Compilers for Parallel Computing, 5th Intl. Workshop*. Ed. by Utpal Banerjee et al. Vol. 757. LNCS. Springer, 1992, pp. 434–448. DOI: 10.1007/3-540-57502-2_63.

[6] Wolfram Amme et al. “Data Dependence Analysis of Assembly Code”. In: *Intl. J. of Parallel Programming* 28.5 (2000), pp. 431–467. DOI: 10.1023/A:1007588710878.

[7] William Pugh. “The Omega test: a fast and practical integer programming algorithm for dependence analysis”. In: *Proc. Supercomputing ’91*. Ed. by Joanne L. Martin. ACM, 1991, pp. 4–13. DOI: 10.1145/125826.125848.

[8] William Pugh and David Wonnacott. “Static Analysis of Upper and Lower Bounds on Dependences and Parallelism”. In: *ACM Trans. Program. Lang. Syst.* 16.4 (1994), pp. 1248–1278. DOI: 10.1145/183432.183525.

[9] Gregor Snelting et al. “Checking probabilistic noninterference using JOANA”. In: *it - Information Technology* 56.6 (2014), pp. 280–287.

[10] Uday Bondhugula et al. “A practical automatic polyhedral parallelizer and locality optimizer”. In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA*. Ed. by Rajiv Gupta and Saman P. Amarasinghe. New York, NY, USA: ACM, 2008, pp. 101–113. DOI: 10.1145/1375581.1375595.

[11] Tobias Grosser, Armin Größlinger, and Christian Lengauer. “Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation”. In: *Parallel Process. Lett.* 22.4 (2012), p. 1250010. DOI: 10.1142/S0129626412500107.

[12] Sara Royuela et al. “Compiler analysis for OpenMP tasks correctness”. In: *Proceedings of the 12th ACM International Conference on Computing Frontiers, CF’15, Ischia, Italy*. Ed. by Claudia Di Napoli et al. New York City, USA: ACM, 2015, 7:1–7:8. DOI: 10.1145/2742854.2742882.

[13] Chunhua Liao et al. “Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions”. In: *Int. J. Parallel Program.* 38.5-6 (2010), pp. 361–378. DOI: 10.1007/s10766-010-0139-0.

[14] Pedro Ramos et al. “Automatic annotation of tasks in structured code”. In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT 2018, Limassol, Cyprus*. Ed. by Skevos Evripidou, Per Stenström, and Michael F. P. O’Boyle. New York City, USA: ACM, 2018, 31:1–31:13. DOI: 10.1145/3243176.3243200.

[15] Gleison Souza Diniz Mendonca et al. “Automatic Insertion of Copy Annotation in Data-Parallel Programs”. In: *28th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2016, Los Angeles, CA, USA, October 26-28, 2016*. Washington, DC, USA: IEEE Computer Society, 2016, pp. 34–41. DOI: 10.1109/SBAC-PAD.2016.13.

[16] Alain Ketterlin and Philippe Clauss. “Profiling Data-Dependence to Assist Parallelization: Framework, Scope, and Optimization”. In: *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada*. IEEE Computer Society, 2012, pp. 437–448. DOI: 10.1109/MICRO.2012.47.

[17] Andreas Wilhelm et al. “Parceive: Interactive parallelization based on dynamic analysis”. In: *6th IEEE International Workshop on Program Comprehension through Dynamic Analysis, PCODA 2015, Montreal, QC, Canada, March 2, 2015*. IEEE Computer Society, 2015, pp. 1–6. DOI: 10.1109/PCODA.2015.7067176.

[18] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. “SD3: A Scalable Approach to Dynamic Data-Dependence Profiling”. In: *43rd Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 535–546. DOI: 10.1109/MICRO.2010.49.

[19] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. “Prospector: Discovering parallelism via dynamic data-dependence profiling”. In: *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism, HOTPAR*. Vol. 10. Berkeley, USA: USENIX Association, 2010, pp. 395–416. URL: https://faculty.cc.gatech.edu/~hyesoon/prospector_4page.pdf.

[20] Saturnino Garcia et al. “Kremlin: rethinking and rebooting gprof for the multicore age”. In: *32nd ACM SIGPLAN Conf. on Progr. Lang. Design and Implementation*. Ed. by Mary W. Hall and David A. Padua. New York, NY, USA: ACM, 2011, pp. 458–469. DOI: 10.1145/1993498.1993553.

[21] Xiangyu Zhang, Armand Navabi, and Suresh Jagannathan. “Alchemist: A Transparent Dependence Distance Profiling Infrastructure”. In: *The 7th Intl. Symp. on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 47–58. DOI: 10.1109/CGO.2009.15.

[22] Vladimir Subotic et al. “Automatic Exploration of Potential Parallelism in Sequential Applications”. In: *International Supercomputing Conference ISC*. Ed. by Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer. Vol. 8488. LNCS. New York City, USA: Springer, 2014, pp. 156–171. DOI: 10.1007/978-3-319-07518-1_10.

[23] Diogo Nunes Sampaio et al. “POSTER: Hybrid Data Dependence Analysis for Loop Transformations”. In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. 2016, pp. 439–440. DOI: 10.1145/2967938.2974059.

[24] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. “Hybrid analysis: static & dynamic memory reference analysis”. In: *Proceedings of the 16th international conference on Supercomputing*. 2002, pp. 274–284. DOI: 10.1145/514191.514229.

[25] Nicolas Morew et al. “Skipping Non-essential Instructions Makes Data-Dependence Profiling Faster”. In: *European Conference on Parallel Processing*. New York City, USA: Springer, 2020, pp. 3–17. DOI: 10.1007/978-3-030-57675-2_1.

[26] Mohammad Norouzi. “Enhancing the Speed and Automation of Assisted Parallelization”. PhD thesis. Technical University of Darmstadt, Germany, 2022. URL: http://tuprints.ulb.tu-darmstadt.de/22884/.

[27] Reiner Hähnle and Marieke Huisman. “Deductive Verification: from Pen-and-Paper Proofs to Industrial Tools”. In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Vol. 10000. LNCS. Cham, Switzerland: Springer, 2019, pp. 345–373. DOI: 10.1007/978-3-319-91908-9_18.

[28] Susanne Graf and Hassen Saïdi. “Construction of Abstract State Graphs with PVS”. In: *Computer Aided Verification, 9th Intl. Conf., CAV*. Ed. by Orna Grumberg. Vol. 1254. LNCS. New York, NY, USA: Springer, 1997, pp. 72–83. DOI: 10.1007/3-540-63166-6_10.

[29] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. “Predicate Abstraction for Program Verification”. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. New York City, USA: Springer, 2018, pp. 447–491. DOI: 10.1007/978-3-319-10575-8_15.

[30] Elvira Albert et al. “Certified Abstract Cost Analysis”. In: *Fundamental Approaches to Software Engineering, 20th Intl. Conf. FASE*. Ed. by Esther Guerra and Mariëlle Stoelinga. Vol. 12649. LNCS. New York City, USA: Springer, Apr. 2021, pp. 24–45. DOI: 10.1007/978-3-030-71500-7_2.

[31] Dominic Steinhöfel. “REFINITY to Model and Prove Program Transformation Rules”. In: *Proc. 18th Asian Symposium on Programming Languages and Systems (APLAS)*. Ed. by Bruno C. d. S. Oliveira. LNCS. New York City, USA: Springer, 2020, pp. 311–319. DOI: 10.1007/978-3-030-64437-6_16.

[32] Wolfgang Ahrendt et al., eds. *Deductive Software Verification—The KeY Book: From Theory to Practice*. Vol. 10001. LNCS. Cham, Switzerland: Springer, 2016. ISBN: 978-3-319-49811-9. DOI: 10.1007/978-3-319-49812-6.

[33] Richard Bubel, Reiner Hähnle, and Asmae Heydari Tabar. “A Program Logic for Dependence Analysis”. In: *Integrated Formal Methods - 15th Intl. Conf. IFM 2019*. Ed. by Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa. Vol. 11918. LNCS. New York, NY, USA: Springer, 2019, pp. 83–100. ISBN: 978-3-030-34967-7. DOI: 10.1007/978-3-030-34968-4_5.

[34] Reiner Hähnle et al. “Safer Parallelization”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12477. Lecture Notes in Computer Science. Springer, 2020, pp. 117–137. DOI: 10.1007/978-3-030-61470-6_8.

[35] Asmae Heydari Tabar, Richard Bubel, and Reiner Hähnle. “Automatic Loop Invariant Generation for Data Dependence Analysis”. In: *10th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2022, Pittsburgh, PA, USA, May 22-23, 2022*. ACM, 2022, pp. 34–45. DOI: 10.1145/3524482.3527649.

[36] Nathan Wasser, Asmae Heydari Tabar, and Reiner Hähnle. “Modeling Non-deterministic C Code with Active Objects”. In: *Fundamentals of Software Engineering - 8th International Conference, FSEN 2019, Tehran, Iran, May 1-3, 2019, Revised Selected Papers*. Ed. by Hossein Hojjat and Mieke Massink. Vol. 11761. Lecture Notes in Computer Science. Springer, 2019, pp. 213–227. DOI: 10.1007/978-3-030-31517-7_15.

[37] Nathan Wasser, Asmae Heydari Tabar, and Reiner Hähnle. “Automated model extraction: From non-deterministic C code to active objects”. In: *Science of Computer Programming* 204 (2021). Article 102597. ISSN: 0167-6423. DOI: 10.1016/J.SCICO.2020.102597.

[38] Vaughan R. Pratt. “Semantical Considerations on Floyd-Hoare Logic”. In: *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*. IEEE Computer Society, 1976, pp. 109–121. DOI: 10.1109/SFCS.1976.27.

[39] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic logic. Foundations of computing*. 2000.

[40] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[41] James Gosling, William N. Joy, and Guy L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996. ISBN: 0-201-63451-1.

[42] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. “SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution”. In: *ACM SIGPLAN Notices* 10.6 (June 1975), pp. 234–245. DOI: 10.1145/800027.808445.

[43] James C. King. “Symbolic Execution and Program Testing”. In: *Communications of the ACM* 19.7 (July 1976), pp. 385–394. DOI: 10.1145/360248.360252.

[44] J. Mack Adams, James Armstrong, and Melissa Smartt. “Assertional checking and symbolic execution: An effective combination for debugging”. In: *Proceedings of the annual ACM/CSC-ER conference*. ACM Press, 1979, pp. 152–156. DOI: 10.1145/800177.810051.

[45] Roger B. Dannenberg and George W. Ernst. “Formal Program Verification Using Symbolic Execution”. In: *IEEE Trans. Software Eng.* 8.1 (1982), pp. 43–52. DOI: 10.1109/TSE.1982.234773.

[46] Kurt Gödel. “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I”. In: *Monatshefte für Mathematik und Physik* 38 (1931), pp. 173–198.

[47] Stephen A Cook. “Soundness and completeness of an axiom system for program verification”. In: *SIAM Journal on Computing* 7.1 (1978), pp. 70–90. DOI: 10.1137/0207005.

[48] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. “Preliminary design of JML: a behavioral interface specification language for java”. In: *ACM SIGSOFT Softw. Eng. Notes* 31.3 (2006), pp. 1–38. DOI: 10.1145/1127878.1127884.

[49] Benjamin Weiß. “Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction”. PhD thesis. Karlsruhe Institute of Technology, 2011. ISBN: 978-3-86644-623-6. URL: https://d-nb.info/1010034960.

[50] Arthur J. Bernstein. “Analysis of Programs for Parallel Processing”. In: *IEEE Trans. Electron. Comput.* 15.5 (1966), pp. 757–763. DOI: 10.1109/PGEC.1966.264565.

[51] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. “Parallel Programming with a Pattern Language”. In: *Int. J. Softw. Tools Technol. Transf.* 3.2 (2001), pp. 217–234. DOI: 10.1007/s100090100045.

[52] Mohammad Norouzi Arab, Felix Wolf, and Ali Jannesari. “Automatic construct selection and variable classification in OpenMP”. In: *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019*. Ed. by Rudolf Eigenmann, Chen Ding, and Sally A. McKee. ACM, 2019, pp. 330–341. DOI: 10.1145/3330345.3330375.

[53] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Conf. Record of the 4th Symp. on Principles of Programming Languages*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. New York, NY, USA: ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[54] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999. ISBN: 978-3-540-65410-0. DOI: 10.1007/978-3-662-03811-6. URL: https://doi.org/10.1007/978-3-662-03811-6.

[55] Patrick Cousot and Radhia Cousot. “Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation”. In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*. Ed. by John Williams. ACM, 1995, pp. 170–181. DOI: 10.1145/224164.224199. URL: https://doi.org/10.1145/224164.224199.

[56] Benjamin Weiß. “Predicate abstraction in a program logic calculus”. In: *Science of Computer Programming* 76.10 (2011), pp. 861–876. DOI: 10.1016/j.scico.2010.06.008.

[57] Dominic Scheurer, Reiner Hähnle, and Richard Bubel. “A General Lattice Model for Merging Symbolic Execution Branches”. In: *18th International Conference on Formal Engineering Methods ICFEM*. Ed. by Kazuhiro Ogata, Mark Lawford, and Shaoying Liu. Vol. 10009. LNCS. New York City, USA: Springer, 2016, pp. 57–73. DOI: 10.1007/978-3-319-47846-3_5.

[58] Elvira Albert et al. “A formal verification framework for static analysis—As well as its instantiation to the resource analyzer COSTA and formal verification tool KeY”. In: *Software & Systems Modeling* 15.4 (2016), pp. 987–1012. DOI: 10.1007/S10270-015-0476-Y.

[59] Tomofumi Yuki and Louis-Noël Pouchet. *PolyBench/C 4.1*. web.cse.ohio-state.edu/~pouchet.2/software/polybench/, Last accessed on February 26, 2024.

[60] David H Bailey et al. “The NAS parallel benchmarks—summary and preliminary results”. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. 1991, pp. 158–165.

[61] Zhen Li, Ali Jannesari, and Felix Wolf. “An Efficient Data-Dependence Profiler for Sequential and Parallel Programs”. In: *IEEE Intl. Parallel and Distrib. Processing Symp.* IEEE Computer Society, 2015, pp. 484–493.

[62] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June*. Ed. by Jeanne Ferrante and Kathryn S. McKinley. New York, USA: ACM, 2007, pp. 89–100. DOI: 10.1145/1250734.1250746.

[63] Chi-Keung Luk et al. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. Ed. by Vivek Sarkar and Mary W. Hall. New York City, USA: ACM, 2005, pp. 190–200. DOI: 10.1145/1065010.1065034.

[64] Kleanthis Psarris. “Program analysis techniques for transforming programs for parallel execution”. In: *Parallel Comput.* 28.3 (2002), pp. 455–469. DOI: 10.1016/S0167-8191(01)00132-6.

[65] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The Program Dependence Graph and Its Use in Optimization”. In: *ACM Trans. Program. Lang. Syst.* 9.3 (1987), pp. 319–349. DOI: 10.1145/24039.24041.

[66] Feng Li, Antoniu Pop, and Albert Cohen. “Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs”. In: *IEEE Micro* 32.4 (2012), pp. 19–31. DOI: 10.1109/MM.2012.49.

[67] Ron Cytron and Jeanne Ferrante. “What’s In a Name? -or- The Value of Renaming for Parallelism Detection and Storage Allocation”. In: *Intl. Conf. on Parallel Processing*. Pennsylvania State University Press, 1987, pp. 19–27.

[68] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[69] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org, Last accessed on February 26, 2024.

[70] Daniel Sánchez et al. “Implementing Signatures for Transactional Memory”. In: *40th Annual IEEE/ACM Intl. Symp. on Microarchitecture*. IEEE Computer Society, 2007, pp. 123–133. DOI: 10.1109/MICRO.2007.24.

[71] Thomas A. Henzinger et al. “Abstractions from proofs”. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Ed. by Neil D. Jones and Xavier Leroy. New York City, USA: ACM, 2004, pp. 232–244. DOI: 10.1145/964001.964021.

[72] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. “Shape Analysis by Predicate Abstraction”. In: *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*. Ed. by Radhia Cousot. Vol. 3385. Lecture Notes in Computer Science. New York City, USA: Springer, 2005, pp. 164–180. DOI: 10.1007/978-3-540-30579-8_12.

[73] Michael D. Ernst. “Summary of Dynamically Discovering Likely Program Invariants”. In: *2001 International Conference on Software Maintenance, ICSM 2001, Florence, Italy, November 6-10, 2001*. Washington, DC, United States: IEEE Computer Society, 2001, pp. 540–544. DOI: 10.1109/ICSM.2001.972767.

[74] Michael D. Ernst et al. “The Daikon system for dynamic detection of likely invariants”. In: *Sci. Comput. Program.* 69.1-3 (2007), pp. 35–45. DOI: 10.1016/j.scico.2007.01.015.

[75] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. “Linear Invariant Generation Using Non-linear Constraint Solving”. In: *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*. Ed. by Warren A. Hunt Jr. and Fabio Somenzi. Vol. 2725. Lecture Notes in Computer Science. New York City, USA: Springer, 2003, pp. 420–432. DOI: 10.1007/978-3-540-45069-6_39.

[76] ThanhVu Nguyen et al. “Using dynamic analysis to discover polynomial and array invariants”. In: *34th International Conference on Software Engineering, ICSE*. Ed. by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. Washington, DC, United States: IEEE Computer Society, 2012, pp. 683–693. DOI: 10.1109/ICSE.2012.6227149.

[77] Laura Kovács and Andrei Voronkov. “Finding Loop Invariants for Programs over Arrays Using a Theorem Prover”. In: *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Marsha Chechik and Martin Wirsing. Vol. 5503. Lecture Notes in Computer Science. New York City, USA: Springer, 2009, pp. 470–485. DOI: 10.1007/978-3-642-00593-0_33.

[78] Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. “Valigator: A Verification Tool with Bound and Invariant Generation”. In: *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*. Ed. by Iliano Cervesato, Helmut Veith, and Andrei Voronkov. Vol. 5330. Lecture Notes in Computer Science. New York City, USA: Springer, 2008, pp. 333–342. DOI: 10.1007/978-3-540-89439-1_24.

[79] Xujie Si et al. “Learning Loop Invariants for Program Verification”. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems, NeurIPS*. Ed. by Samy Bengio et al. Cambridge, MA, USA: MIT Press, 2018, pp. 7762–7773.

[80] Cormac Flanagan and Shaz Qadeer. “Predicate abstraction for software verification”. In: *The 29th Symp. on Principles of Prog. Lang.* Ed. by John Launchbury and John C. Mitchell. New York, NY, USA: ACM, 2002, pp. 191–202. DOI: 10.1145/503272.503291.

[81] Michael Barnett et al. “Boogie: A Modular Reusable Verifier for Object-Oriented Programs”. In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. Ed. by Frank S. de Boer et al. Vol. 4111. LNCS. Salmon Tower Building New York City, USA: Springer, 2005, pp. 364–387. DOI: 10.1007/11804192_17.

[82] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. “Invariant Generation for Multi-Path Loops with Polynomial Assignments”. In: *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI*. Ed. by Isil Dillig and Jens Palsberg. Vol. 10747. LNCS. Springer, 2018, pp. 226–246. DOI: 10.1007/978-3-319-73721-8_11.

[83] Andrei Sabelfeld and Andrew C. Myers. “Language-Based Information-Flow Security”. In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121.

[84] Ádám Darvas, Reiner Hähnle, and David Sands. “A Theorem Proving Approach to Analysis of Secure Information Flow”. In: *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings*. Ed. by Dieter Hutter and Markus Ullmann. Vol. 3450. Lecture Notes in Computer Science. Springer, 2005, pp. 193–209. DOI: 10.1007/978-3-540-32004-3_20. URL: https://doi.org/10.1007/978-3-540-32004-3%5C_20.

[85] Christoph Scheben and Simon Greiner. “Information Flow Analysis”. In: *Deductive Software Verification—The KeY Book: From Theory to Practice*. Ed. by Wolfgang Ahrendt et al. Vol. 10001. LNCS. New York City, USA: Spinger, 2016. Chap. 13, pp. 453–472. DOI: 10.1007/978-3-319-49812-6_13.

[86] Jérôme Dohrau et al. “Permission Inference for Array Programs”. In: *Computer Aided Verification - 30th International Conference, CAV*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10982. LNCS. Springer, 2018, pp. 55–74. DOI: 10.1007/978-3-319-96142-2_7.

[87] Stefan Blom, Saeed Darabi, and Marieke Huisman. “Verification of Loop Parallelisations”. In: *Fundamental Approaches to Software Engineering - 18th International Conference, FASE*. Ed. by Alexander Egyed and Ina Schaefer. Vol. 9033. LNCS. Springer, 2015, pp. 202–217. DOI: 10.1007/978-3-662-46675-9_14.

A. Semantics

A.1. Semantics of Dependence Predicates

$$s(\texttt{noWaR}) = \{ls \mid s = (\sigma, Acc),\ Acc = \langle M_1(ls_1)\rangle \circ\ \ldots \circ \langle M_n(ls_n)\rangle$$
$$\text{and for all } i, j = 1 \ldots n \text{ for which } i < j$$
$$\text{where } M_i = \textit{Read} \text{ and } M_j = \textit{Write},$$
$$\text{it holds that } ls \cap ls_i \cap ls_j = \emptyset\}$$

$$s(\texttt{noWaW}) = \{ls \mid s = (\sigma, Acc),\ Acc = \langle M_1(ls_1)\rangle \circ\ \ldots \circ \langle M_n(ls_n)\rangle$$
$$\text{and for all } i, j = 1 \ldots n \text{ for which } i < j$$
$$\text{where } M_i = \textit{Write} \text{ and } M_j = \textit{Write},$$
$$\text{it holds that } ls \cap ls_i \cap ls_j = \emptyset\}$$

$$s(\texttt{noW}) = \{ls \mid s = (\sigma, Acc),\ Acc = \langle M_1(ls_1)\rangle \circ\ \ldots \circ \langle M_n(ls_n)\rangle, M_i \in \{\textit{Read}, \textit{Write}\}$$
$$\text{and for all } j = 1 \ldots n \text{ for which } M_j = \textit{Write},\ \text{it holds that } ls \cap ls_j = \emptyset\}$$

A.2. Semantics of History Dependence Predicates

$$s(\texttt{noWHist}) = \{(ls, i) \mid s = (\sigma, Acc),\ 0 \leq i < n,$$
$$Acc = \langle M_1(ls_1)\rangle \circ \ldots \circ \langle M_{n-i}(ls_{n-i})\rangle \circ \ldots \circ \langle M_n(ls_n)\rangle,$$
$$\text{for all } 0 < j \leq n - i \text{ with}$$
$$M_j = \textit{Write},\ \text{it holds that } ls \cap ls_j = \emptyset\}$$

$$s(\texttt{noWaRHist}) = \{(ls, i) \mid s = (\sigma, Acc), 0 \leq i < n,$$
$$Acc = \langle M_1(ls_1)\rangle \circ \ldots \circ \langle M_{n-i}(ls_{n-i})\rangle \circ \ldots \circ \langle M_n(ls_n)\rangle,$$
$$\text{and for all } 0 < j < k \leq n - i \text{ with } M_j = \textit{Read}$$
$$\text{and } M_k = \textit{Write}, \text{ it holds that } ls_j \cap ls_k \cap ls = \emptyset\}$$

$$s(\texttt{noWaWHist}) = \{(ls, i) \mid s = (\sigma, Acc), 0 \leq i < n,$$
$$Acc = \langle M_1(ls_1)\rangle \circ \ldots \circ \langle M_{n-i}(ls_{n-i})\rangle \circ \ldots \circ \langle M_n(ls_n)\rangle,$$
$$\text{and for all } 0 < j < k \leq n - i \text{ with } M_j = \textit{Write}$$
$$\text{and } M_k = \textit{Write}, \text{ it holds that } ls_j \cap ls_k \cap ls = \emptyset\}$$

A.3. Semantics of $\widehat{\text{noWaR}}$

$$s(\widehat{\text{noWaR}}) = \{(ls, wLs, rLs, futWLs) \mid s = (\sigma, \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_n),$$
$$rLs, wLs, futRLs \in \mathcal{D}^{\texttt{LocSet}} \text{ such that } ls \cap (rLs \cup \textstyle\bigcup_R \widehat{acc}_n) \cap futWLs = \emptyset \text{ and}$$
$$\text{if } n > 1: \ (ls, \emptyset, \emptyset, futWLs \cup wLs \cup \textstyle\bigcup_W \widehat{acc}_n) \in s'(\widehat{\text{noWaR}})$$
$$\text{with } s' = (\sigma, \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_{n-1})\}$$

A.4. Semantics of Inter-Iteration History and Access Predicates

$$
\begin{aligned}
s(\widehat{\texttt{noWHist}}) \quad & = \{(loc, label) \mid s = (\sigma, \widehat{Acc}),\ \widehat{Acc} = \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_n, \\
& \quad 0 \leq label < n, \\
& \text{and for all } 0 < i \leq n - label \text{ it holds that } \textstyle\downarrow_W \widehat{acc}_i \cap loc = \emptyset\} \\
s(\widehat{\texttt{noWaRHist}}) \quad & = \{(loc, label) \mid s = (\sigma, \widehat{Acc}),\ \widehat{Acc} = \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_n, \\
& 0 \leq label < n, \\
& \text{and for all } 0 < i < j \leq n - label \\
& \text{it holds that } \textstyle\downarrow_W \widehat{acc}_j \cap \downarrow_R \widehat{acc}_i \cap loc = \emptyset\} \\
s(\widehat{\texttt{noWaWHist}}) \quad & = \{(loc, label) \mid s = (\sigma, \widehat{Acc}),\ \widehat{Acc} = \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_n, \\
& 0 \leq label < n, \\
& \text{and for all } 0 < i < j \leq n - label \\
& \text{it holds that } \textstyle\downarrow_W \widehat{acc}_j \cap \downarrow_W \widehat{acc}_i \cap loc = \emptyset\} \\
s(\widehat{\texttt{wPred}}) \quad & = \{(loc, label) \mid s = (\sigma, \widehat{Acc}),\ \widehat{Acc} = \widehat{acc}_1 \bigcirc \cdots \bigcirc \widehat{acc}_n, \\
& \quad 0 \leq label < n, \\
& loc = \textstyle\downarrow_W \widehat{acc}_{n-label}\}
\end{aligned}
$$

B. Sequent Calculus Rules

B.1. Axiomatization of Data Dependences

In the following su is state update with no memory access update.

B.1.1. Rules for noWaR

The side condition in readAccessAppOnNoWaR and writeAccessAppOnNoWaR is that loc_2 does not contain data dependence predicates.

stateUpdateAppOnNoWaR $\quad \{su\}\mathsf{noWaR}(loc) \rightsquigarrow \mathsf{noWaR}(\{su\}ls)$

readAccessAppOnNoWaR $\quad \{\backslash\mathsf{R}(loc_1)\}\mathsf{noWaR}(loc_2) \rightsquigarrow \mathsf{noWaR}(\{\backslash\mathsf{R}(loc_1)\}loc_2)$

writeAccessAppOnNoWaR

$$\{\backslash\mathsf{W}(loc_1)\}\mathsf{noWaR}(loc_2) \rightsquigarrow \mathsf{noR}((\{\backslash\mathsf{W}(loc_1)\}loc_2) \cap loc_1) \wedge \mathsf{noWaR}((\{\backslash\mathsf{W}(loc_1)\}loc_2) \setminus loc_1)$$

knownNoWaR

$$\frac{\Gamma, \mathsf{noWaR}(loc_1) \Longrightarrow \text{if } (loc_2 \subseteq loc_1) \text{ then } (\text{true}) \text{ else } (\mathsf{noWaR}(loc_2 \setminus loc_1)), \Delta}{\Gamma, \mathsf{noWaR}(loc_1) \Longrightarrow \mathsf{noWaR}(loc_2), \Delta}$$

knownNoR $\dfrac{}{\Gamma, \mathsf{noR}(loc) \Longrightarrow \mathsf{noWaR}(loc), \Delta}$

knownNoW $\dfrac{}{\Gamma, \mathsf{noW}(loc) \Longrightarrow \mathsf{noWaR}(loc), \Delta}$

B.1.2. Rules for noWaW

The side condition in readAccessAppOnNoWaW and writeAccessAppOnNoWaW is that loc_2 does not contain data dependence predicates.

stateUpdateAppOnNoWaW $\quad \{su\}\mathsf{noWaW}(loc) \rightsquigarrow \mathsf{noWaW}(\{su\}loc)$

readAccessAppOnNoWaW $\quad \{\backslash\mathsf{R}(loc_1)\}\mathsf{noWaW}(loc_2) \rightsquigarrow \mathsf{noWaW}(\{\backslash\mathsf{R}(loc_1)\}loc_2)$

writeAccessAppOnNoWaW

$$\{\backslash\mathsf{W}(loc_1)\}\mathsf{noWaW}(loc_2) \rightsquigarrow \mathsf{noW}((\{\backslash\mathsf{W}(loc_1)\}loc_2) \cap loc_1) \wedge \mathsf{noWaW}((\{\backslash\mathsf{W}(loc_1)\}loc_2) \setminus loc_1)$$

knownNoWaW

$$\frac{\Gamma, \mathsf{noWaW}(loc_1) \Longrightarrow \text{if } (loc_2 \subseteq loc_1) \text{ then (true) else } (\mathsf{noWaW}(loc_2 \setminus loc_1)), \Delta}{\Gamma, \mathsf{noWaW}(loc_1) \Longrightarrow \mathsf{noWaW}(loc_2), \Delta}$$

knownNoW $\quad \dfrac{}{\Gamma, \mathsf{noW}(loc) \Longrightarrow \mathsf{noWaW}(loc), \Delta}$

B.1.3. Rules for noR

The side condition in writeAccessAppOnNoR and readAccessAppOnNoR is that loc_2 does not contain data dependence predicates.

stateUpdateAppOnNoR $\quad \{su\}\mathsf{noR}(loc) \rightsquigarrow \mathsf{noR}(\{su\}loc)$

writeAccessAppOnNoR $\quad \{\backslash\mathsf{W}(loc_1)\}\mathsf{noR}(loc_2) \rightsquigarrow \mathsf{noR}(\{\backslash\mathsf{W}(loc_1)\}loc_2)$

readAccessAppOnNoR

$$\frac{\Gamma \Longrightarrow \text{if } (loc_1 \cap \{\backslash\mathsf{R}(loc_1)\}loc_2 \neq \emptyset) \text{ then (false) else}(\mathsf{noR}(\{\backslash\mathsf{R}(loc_1)\}loc_2)), \Delta}{\Gamma \Longrightarrow \{\backslash\mathsf{R}(loc_1)\}\mathsf{noR}(loc_2), \Delta}$$

knownNoR $\quad \dfrac{\Gamma, \mathsf{noR}(loc_1) \Longrightarrow \text{if } (loc_2 \subseteq loc_1) \text{ then (true) else } (\mathsf{noR}(loc_2 \setminus loc_1)), \Delta}{\Gamma, \mathsf{noR}(loc_1) \Longrightarrow \mathsf{noR}(loc_2), \Delta}$

B.1.4. Rules for noW

The side condition in readAccessAppOnNoW and writeAccessAppOnNoW is that loc_2 does not contain data dependence predicates.

stateUpdateAppOnNoW $\quad \{su\}\mathsf{noW}(loc) \rightsquigarrow \mathsf{noW}(\{su\}loc)$

readAccessAppOnNoW $\quad \{\backslash\mathsf{R}(loc_1)\}\mathsf{noW}(loc_2) \rightsquigarrow \mathsf{noW}(\{\backslash\mathsf{R}(loc_1)\}loc_2)loc)$

writeAccessAppOnNoW

$$\frac{\Gamma \Longrightarrow \text{if}(loc_1 \cap \{\backslash\mathsf{W}(loc_1)\}loc_2 \doteq \emptyset)\ \text{then}(\text{false})\ \text{else}(\mathsf{noW}(\{\backslash\mathsf{W}(loc_1)\}loc_2)), \Delta}{\Gamma \Longrightarrow \{\backslash\mathsf{W}(loc_1)\}\mathsf{noW}(loc_2), \Delta}$$

knownNoW $\quad \dfrac{\Gamma, \mathsf{noW}(loc_1) \Longrightarrow \text{if } (loc_2 \subseteq loc_1)\ \text{ then } (\text{true})\ \text{ else } (\mathsf{noW}(loc_2 \setminus loc_1)), \Delta}{\Gamma, \mathsf{noW}(loc_1) \Longrightarrow \mathsf{noW}(loc_2), \Delta}$

B.2. Shift Write Rule

$$\text{shiftWrite}\ \frac{\{\backslash\mathsf{W}'(loc)\}\Gamma, \mathsf{wPred}(loc, 0) \Longrightarrow \phi, \{\backslash\mathsf{W}'(loc)\}\Delta}{\Gamma \Longrightarrow \{\backslash\mathsf{W}(loc)\}\phi, \Delta}\ \text{cond}$$

With the side condition cond that *loc* does not contain data dependence predicates.

B.3. Renamed Memory Access Update Application

renamedReadAppOnNoW

$$\{\backslash \mathsf{R}'(rLoc)\}\mathsf{noW}(loc) \rightsquigarrow \mathtt{noWHist}(\{\backslash \mathsf{R}'(rLoc)\}loc, 1)$$

renamedReadAppOnNoRaW

$$\{\backslash \mathsf{R}'(rLoc)\}\mathsf{noRaW}(loc) \rightsquigarrow \mathtt{noRaWHist}(\{\backslash \mathsf{R}'(rLoc)\}loc, 1)$$

renamedReadAppOnNoWaR

$$\{\backslash \mathsf{R}'(rLoc)\}\mathsf{noWaR}(loc) \rightsquigarrow \mathtt{noWaRHist}(\{\backslash \mathsf{R}'(rLoc)\}loc, 1)$$

renamedReadAppOnNoWaW

$$\{\backslash \mathsf{R}'(rLoc)\}\mathsf{noWaW}(loc) \rightsquigarrow \mathtt{noWaWHist}(\{\backslash \mathsf{R}'(rLoc)\}loc, 1)$$

renamedReadOnNoWAtHistory

$$\{\backslash \mathsf{R}'(rLoc)\}\mathtt{noWHist}(loc, lb) \rightsquigarrow \mathtt{noWHist}(\{\backslash \mathsf{R}'(rLoc)\}loc, \{\backslash \mathsf{R}'(rLoc)\}lb + 1)$$

renamedReadOnNoRaWAtHistory

$$\{\backslash \mathsf{R}'(rLoc)\}\mathtt{noRaWHist}(loc, lb) \rightsquigarrow \mathtt{noRaWHist}(\{\backslash \mathsf{R}'(rLoc)\}loc, \{\backslash \mathsf{R}'(rLoc)\}lb + 1)$$

renamedReadOnNoWaRAtHistory

$$\{\backslash \mathsf{R}'(rLoc)\}\mathtt{noWaRHist}(loc, lb) \rightsquigarrow \mathtt{noWaRHist}(\{\backslash \mathsf{R}'(rLoc)\}loc, \{\backslash \mathsf{R}'(rLoc)\}lb + 1)$$

renamedReadOnNoWaWAtHistory

$$\{\backslash \mathsf{R}'(rLoc)\}\mathtt{noWaWHist}(loc, lb) \rightsquigarrow \mathtt{noWaWHist}(\{\backslash \mathsf{R}'(rLoc)\}loc, \{\backslash \mathsf{R}'(rLoc)\}lb + 1)$$

renamedReadOnWritePred

$$\{\backslash \mathsf{R}'(rLoc)\}\mathtt{wPred}(loc, lb) \rightsquigarrow \mathtt{wPred}(\{\backslash \mathsf{R}'(rLoc)\}loc, \{\backslash \mathsf{R}'(rLoc)\}lb + 1)$$

B.4. Subsumption Rules for History Data Dependence Predicates

noWImpliesNoWHist $\quad \mathsf{noW}(loc) \rightsquigarrow \mathtt{noWHist}(loc, lb)$

noRaWImpliesNoRaWHist $\quad \mathsf{noRaW}(loc) \rightsquigarrow \mathtt{noRaWHist}(loc, lb)$

noWaRImpliesNoWaRHist $\quad \mathsf{noWaR}(loc) \rightsquigarrow \mathtt{noWaRHist}(loc, lb)$

noWaWImpliesNoWaWHist $\quad \mathsf{noWaW}(loc) \rightsquigarrow \mathtt{noWaWHist}(loc, lb)$

noWHistOnBothSides

$$\frac{\Gamma, \mathtt{noWHist}(loc_1, lb_1), lb_1 \leq lb_2 \Longrightarrow \mathtt{noWHist}(loc_2 \setminus loc_1, lb_2), \Delta \quad \Gamma, \mathtt{noWHist}(loc_1, lb_1), lb_1 > lb_2 \Longrightarrow \mathtt{noWHist}(loc_2, lb_2), \Delta}{\Gamma, \mathtt{noWHist}(loc_1, lb_1) \Longrightarrow \mathtt{noWHist}(loc_2, lb_2), \Delta}$$

noRaWHistOnBothSides

$$\frac{\Gamma, \mathtt{noRaWHist}(loc_1, lb_1), lb_1 \leq lb_2 \Longrightarrow \mathtt{noRaWHist}(loc_2 \setminus loc_1, lb_2), \Delta \quad \Gamma, \mathtt{noRaWHist}(loc_1, lb_1), lb_1 > lb_2 \Longrightarrow \mathtt{noRaWHist}(loc_2, lb_2), \Delta}{\Gamma, \mathtt{noRaWHist}(loc_1, lb_1) \Longrightarrow \mathtt{noRaWHist}(loc_2, lb_2), \Delta}$$

noWaRHistOnBothSides

$$\frac{\Gamma, \mathtt{noWaRHist}(loc_1, lb_1), lb_1 \leq lb_2 \Longrightarrow \mathtt{noWaRHist}(loc_2 \setminus loc_1, lb_2), \Delta \quad \Gamma, \mathtt{noWaRHist}(loc_1, lb_1), lb_1 > lb_2 \Longrightarrow \mathtt{noWaRHist}(loc_2, lb_2), \Delta}{\Gamma, \mathtt{noWaRHist}(loc_1, lb_1) \Longrightarrow \mathtt{noWaRHist}(loc_2, lb_2), \Delta}$$

noWaWHistOnBothSides

$$\frac{\Gamma, \mathtt{noWaWHist}(loc_1, lb_1), lb_1 \leq lb_2 \Longrightarrow \mathtt{noWaWHist}(loc_2 \setminus loc_1, lb_2), \Delta \quad \Gamma, \mathtt{noWaWHist}(loc_1, lb_1), lb_1 > lb_2 \Longrightarrow \mathtt{noWaWHist}(loc_2, lb_2), \Delta}{\Gamma, \mathtt{noWaWHist}(loc_1, lb_1) \Longrightarrow \mathtt{noWaWHist}(loc_2, lb_2), \Delta}$$

B.5. Relation of wPred with History Data Dependence Predicates

B.5.1. Relation of wPred and noWHist

writePredANDnoWHistSameLabel

$$\frac{\Gamma, \mathtt{wPred}(loc_1, lb), \mathtt{noWHist}(loc_2, lb), loc_1 \cap loc_2 \doteq \emptyset \Longrightarrow \phi, \Delta}{\Gamma, \mathtt{wPred}(loc_1, lb), \mathtt{noWHist}(loc_2, lb) \Longrightarrow \phi, \Delta}$$

writePredANDnoWHistRightSameLabel

$$\frac{\Gamma, \mathtt{wPred}(loc_1, lb) \Longrightarrow loc_1 \cap loc_2 \doteq \emptyset \wedge \mathtt{noWHist}(loc_2, lb+1), \Delta}{\Gamma, \mathtt{wPred}(loc_1, lb) \Longrightarrow \mathtt{noWHist}(loc_2, lb), \Delta}$$

writePredAFTERnoWHist

$$\frac{\Gamma, \mathtt{wPred}(loc_1, lb), \mathtt{noWHist}(loc_2, lb+1), \mathtt{noWHist}(loc_2 \setminus loc_1, lb) \Longrightarrow \phi, \Delta}{\Gamma, \mathtt{wPred}(loc_1, lb), \mathtt{noWHist}(loc_2, lb+1) \Longrightarrow \phi, \Delta}$$

B.5.2. Relation of wPred and noRHist

writePredANDnoRHistRightSameLabel

$$\frac{\Gamma, \mathtt{wPred}(loc_1, lb) \Longrightarrow \mathtt{noRHist}(loc_2, lb+1), \Delta}{\Gamma, \mathtt{wPred}(loc_1, lb) \Longrightarrow \mathtt{noRHist}(loc_2, lb), \Delta}$$

writePredAFTERnoRHist

$$\frac{\Gamma, \mathtt{wPred}(loc_1, lb), \mathtt{noRHist}(loc_2, lb+1), \mathtt{noRHist}(loc_2, lb) \Longrightarrow \phi, \Delta}{\Gamma, \mathtt{wPred}(loc_1, lb), \mathtt{noRHist}(loc_2, lb+1) \Longrightarrow \phi, \Delta}$$

B.5.3. Relation of wPred and noWaRHist

writePredANDnoWaRHistSameLabel

$$\frac{\Gamma, \texttt{wPred}(loc_1, lb), \texttt{noWaRHist}(loc_2, lb), \texttt{noWHist}(loc_1 \cap loc_2, lb+1) \Longrightarrow \phi, \Delta}{\Gamma, \texttt{wPred}(loc_1, lb), \texttt{noWaRHist}(loc_2, lb) \Longrightarrow \phi, \Delta}$$

writePredANDnoWaRHistRightSameLabel

$$\frac{\Gamma, \texttt{wPred}(loc_1, lb) \Longrightarrow \texttt{noWaRHist}(loc_2 \setminus loc_1, lb+1) \wedge \texttt{noWHist}(loc_1 \cap loc_2, lb+1), \Delta}{\Gamma, \texttt{wPred}(loc_1, lb) \Longrightarrow \texttt{noWaRHist}(loc_2, lb), \Delta}$$

writePredBEFOREnoWaRHist

$$\frac{\Gamma, \texttt{wPred}(loc_1, lb+1), \texttt{noWaRHist}(loc_2, lb), \ \texttt{noWHist}(loc_1 \cap loc_2, lb+2) \Longrightarrow \phi, \Delta}{\Gamma, \texttt{wPred}(loc_1, lb+1), \texttt{noWaRHist}(loc_2, lb) \Longrightarrow \phi, \Delta}$$

writePredBEFOREnoWaRHistRight

$$\frac{\Gamma, \texttt{wPred}(loc_1, lb+1) \Longrightarrow \texttt{noWaRHist}(loc_2 \setminus loc_1, lb) \wedge \texttt{noWHist}(loc_1 \cap loc_2, lb+2), \Delta}{\Gamma, \texttt{wPred}(loc_1, lb+1) \Longrightarrow \texttt{noWaRHist}(loc_2, lb), \Delta}$$

B.5.4. Relation of wPred and noWaWHist

writePredANDnoWaWHistSameLabel

$$\frac{\Gamma, \mathtt{wPred}(loc_1, lb), \mathtt{noWaWHist}(loc_2, lb), \mathtt{noWHist}(loc_1 \cap loc_2, lb+1) \Longrightarrow \phi, \Delta}{\Gamma, \mathtt{wPred}(loc_1, lb), \mathtt{noWaWHist}(loc_2, lb) \Longrightarrow \phi, \Delta}$$

writePredANDnoWaRHistRightSameLabel

$$\frac{\Gamma, \mathtt{wPred}(loc_1, lb) \Longrightarrow \mathtt{noWaWHist}(loc_2 \setminus loc_1, lb+1) \wedge \mathtt{noWHist}(loc_1 \cap loc_2, lb+1), \Delta}{\Gamma, \mathtt{wPred}(loc_1, lb) \Longrightarrow \mathtt{noWaWHist}(loc_2, lb), \Delta}$$

writePredBEFOREnoWaWHist

$$\frac{\Gamma, \mathtt{wPred}(loc_1, lb+1), \mathtt{noWaWHist}(loc_2, lb),\ \mathtt{noWHist}(loc_1 \cap loc_2, lb+2) \Longrightarrow \phi, \Delta}{\Gamma, \mathtt{wPred}(loc_1, lb+1), \mathtt{noWaWHist}(loc_2, lb) \Longrightarrow \phi, \Delta}$$

writePredBEFOREnoWaWHistRight

$$\frac{\Gamma, \mathtt{wPred}(loc_1, lb+1) \Longrightarrow \mathtt{noWaWHist}(loc_2 \setminus loc_1, lb) \wedge \mathtt{noWHist}(loc_1 \cap loc_2, lb+2), \Delta}{\Gamma, \mathtt{wPred}(loc_1, lb+1) \Longrightarrow \mathtt{noWaWHist}(loc_2, lb), \Delta}$$

B.6. Update Application on Inter-Iteration Data Dependence Predicates

writeAccessOnInterIterationNoWaR

$$\{\backslash\mathsf{next};\ \backslash\mathsf{W}(loc_1)\}\widehat{\mathsf{noWaR}}(loc_2, wLs, rLs, futWLs) \rightsquigarrow$$
$$\{\backslash\mathsf{next}\}\widehat{\mathsf{noWaR}}(\{\backslash\mathsf{W}(loc_1)\}loc_2, (\{\backslash\mathsf{W}(loc_1)\}wLs) \cup loc_1, \{\backslash\mathsf{W}(loc_1)\}rLs, \{\backslash\mathsf{W}(loc_1)\}futWLs)$$

readAccessOnInterIterationNoWaR

$$\{\backslash\mathsf{next};\backslash\mathsf{R}(loc_1)\}\widehat{\mathsf{noWaR}}(loc_2, wLs, rLs, futWLs) \rightsquigarrow$$
$$\{\backslash\mathsf{next}\}\widehat{\mathsf{noWaR}}(\{\backslash\mathsf{R}(loc_1)\}loc_2, \{\backslash\mathsf{R}(loc_1)\}wLs, (\{\backslash\mathsf{R}(loc_1)\}rLs) \cup loc_1, \{\backslash\mathsf{R}(loc_1)\}futWLs)$$

checkNoWaRForIteration

$$\{u||\backslash\mathsf{next}\}\widehat{\mathsf{noWaR}}(loc, wLs, rLs, futWLs) \rightsquigarrow (\{u, \backslash\mathsf{next}\}(loc \cap rLs \cap futWLs = \emptyset)) \wedge$$
$$\{u\}\widehat{\mathsf{noWaR}}(\{\backslash\mathsf{next}\}loc, \emptyset, \emptyset, \{\backslash\mathsf{next}\}futWLs \cup \{\backslash\mathsf{next}\}wLs)$$

checkNoWaRForLastIteration

$$\{\backslash\mathsf{next}\}\widehat{\mathsf{noWaR}}(loc, wLs, rLs, futWLs) \rightsquigarrow \{\backslash\mathsf{next}\}(loc \cap rLs \cap futWLs = \emptyset)$$

writeAccessOnInterIterationNoWaW

$$\{\backslash\mathsf{next};\ \backslash\mathsf{W}(loc_1)\}\widehat{\mathsf{noWaW}}(loc_2, wLs, futWLs) \rightsquigarrow$$
$$\{\backslash\mathsf{next}\}\widehat{\mathsf{noWaW}}(\{\backslash\mathsf{W}(loc_1)\}loc_2, (\{\backslash\mathsf{W}(loc_1)\}wLs) \cup loc_1, \{\backslash\mathsf{W}(loc_1)\}futWLs)$$

readAccessOnInterIterationNoWaW

$$\{\backslash\mathsf{next};\backslash\mathsf{R}(loc_1)\}\widehat{\mathsf{noWaW}}(loc_2, wLs, futWLs) \rightsquigarrow$$
$$\{\backslash\mathsf{next}\}\widehat{\mathsf{noWaW}}(\{\backslash\mathsf{R}(loc_1)\}loc_2, \{\backslash\mathsf{R}(loc_1)\}wLs, \{\backslash\mathsf{R}(loc_1)\}futWLs)$$

checkNoWaWForIteration

$$\{u||\backslash\mathsf{next}\}\widehat{\mathsf{noWaW}}(loc, wLs, futWLs) \rightsquigarrow (\{u, \backslash\mathsf{next}\}(loc \cap futWLs = \emptyset)) \wedge$$
$$\{u\}\widehat{\mathsf{noWaW}}(\{\backslash\mathsf{next}\}loc, \emptyset, \{\backslash\mathsf{next}\}futWLs \cup \{\backslash\mathsf{next}\}wLs)$$

checkNoWaWForLastIteration

$$\{\backslash\mathsf{next}\}\widehat{\mathsf{noWaW}}(loc, wLs, futWLs) \rightsquigarrow \{\backslash\mathsf{next}\}(loc \cap futWLs = \emptyset)$$

C. Proof of Sequent Calculus Rules

C.1. Proof of Soundness and Completeness of writeAccessAppOnNoRaW

Proof. We want to show that for all $\mathcal{K}, s, \beta$ with $s = (\sigma, acc_1 \circ \ldots \circ acc_{n-1})$ and side condition that loc_2 does not contain data dependence predicates, the following holds

$$\mathcal{K}, s, \beta \models \{\backslash \mathtt{W}(loc_1)\}\mathsf{noRaW}(loc_2) \iff \mathcal{K}, s, \beta \models \mathsf{noRaW}(\{\backslash \mathtt{W}(loc_1)\}loc_2).$$

Starting from the right side we have

$$\begin{aligned} & \mathcal{K}, s, \beta \models \mathsf{noRaW}(\{\backslash \mathtt{W}(loc_1)\}loc_2) \\ \iff & \text{Fig.2.3} \\ & val_{\mathcal{K},s,\beta}(\{\backslash \mathtt{W}(loc_1)\}loc_2) \in s(\mathsf{noRaW}). \end{aligned} \tag{C.1}$$

From the semantics of JavaDL in Figure 2.3 we know

$$val_{\mathcal{K},s,\beta}(\{\backslash \mathtt{W}(loc_1)\}loc_2) = val_{\mathcal{K},s',\beta}(loc_2) \text{ with } s' = val_{\mathcal{K},s,\beta}(\backslash \mathtt{W}(loc_1)). \tag{C.2}$$

Based on Definition 3.2.1:

$$s' = (\sigma, acc_1 \circ \ldots \circ acc_{n-1} \circ \langle Write(val_{\mathcal{K},s,\beta}(loc_1))\rangle). \tag{C.3}$$

From C.2 together with C.1 we have

$$val_{\mathcal{K},s',\beta}(loc_2) \in s(\mathsf{noRaW}). \tag{C.4}$$

We first show that

$$s(\mathsf{noRaW}) = s'(\mathsf{noRaW}).$$

For an arbitrary location set $ls \in \mathcal{D}^{\mathtt{LocSet}}$

$$\begin{aligned} & ls \in s(\mathsf{noRaW}) \\ \iff & \text{Def. 3.3.1} \\ & \text{for all } 1 \leq i \leq n-1 \text{ and } i < j \leq n-1, \\ & acc_i = \langle Write(ls_i)\rangle \text{ and } acc_j = \langle Read(ls_j)\rangle \text{ it holds that } ls \cap ls_i \cap ls_j = \mathtt{empty}. \end{aligned}$$

As in s'(C.3), $acc_n = \langle Write(val_{\mathcal{K},s,\beta}(loc_1))\rangle$ the following cases have to be considered:

- $1 \leq i \leq n-1$ and $j = n$, or
- $i = n$ and $i < j \leq n$.

Both cases are trivially closed. Therefore

$$\begin{aligned}
&\text{for all } 1 \leq i \leq n \text{ and } i < j \leq n,\\
&\mathit{acc}_i = \langle \mathit{Write}(ls_i) \rangle \text{ and } \mathit{acc}_j = \langle \mathit{Read}(ls_j) \rangle \text{ it holds that } ls \cap ls_i \cap ls_j = \mathsf{empty}\\
\iff &\text{Def. 3.3.1}\\
&ls \in s'(\mathsf{noRaW}).
\end{aligned}$$

Together with C.4 this means

$$\begin{aligned}
&\mathit{val}_{\mathcal{K},s',\beta}(loc_2) \in s'(\mathsf{noRaW})\\
\iff &\text{Def. 3.3.1}\\
&\mathcal{K}, s', \beta \models \mathsf{noRaW}(loc_2)\\
\iff &\text{Def. 3.2.1 and Fig. 2.3}\\
&\mathcal{K}, s, \beta \models \{\backslash \mathsf{W}(loc_1)\}\mathsf{noRaW}(loc_2)
\end{aligned}$$

□

C.2. Proof of Soundness and Completeness of readAccessAppOnNoRaW

Proof. We want to show for all $\mathcal{K}, s, \beta$ with $s = (\sigma, \mathit{acc}_1 \circ \ldots \circ \mathit{acc}_{n-1})$ and side condition that loc_2 does not contain data dependence predicates, the following holds

$$\begin{aligned}
&\mathcal{K}, s, \beta \models \{\backslash \mathsf{R}(loc_1)\}\mathsf{noRaW}(loc_2)\\
\iff &\\
&\mathcal{K}, s, \beta \models \mathsf{noW}((\{\backslash \mathsf{R}(loc_1)\}loc_2) \cap loc_1) \wedge \mathsf{noRaW}((\{\backslash \mathsf{R}(loc_1)\}loc_2) \setminus loc_1).
\end{aligned}$$

Using the semantics of JavaDL (Figure 2.3), now the goal is to show

$$\begin{aligned}
&\mathcal{K}, s, \beta \models \{\backslash \mathsf{R}(loc_1)\}\mathsf{noRaW}(loc_2)\\
\iff &\\
&\mathcal{K}, s, \beta \models \mathsf{noW}((\{\backslash \mathsf{R}(loc_1)\}loc_2) \cap loc_1)\\
&\text{and}\\
&\mathcal{K}, s, \beta \models \mathsf{noRaW}((\{\backslash \mathsf{R}(loc_1)\}loc_2) \setminus loc_1).
\end{aligned}$$

We start from the left side

$$\mathcal{K}, s, \beta \models \{\backslash \mathsf{R}(loc_1)\}\mathsf{noRaW}(loc_2)$$

$\iff$ Def. 2.1.9

$$\mathcal{K}, s', \beta \models \mathsf{noRaW}(loc_2) \text{ with } s' = val_{\mathcal{K},s,\beta}(\backslash \mathsf{R}(loc_1)) \tag{C.5}$$

From Definition 3.2.1 we know

$$s' = (\sigma, acc_1 \circ \ldots \circ acc_{n-1} \circ \langle Read(val_{\mathcal{K},s,\beta}(loc_1))\rangle). \tag{C.6}$$

From the semantics of JavaDL (Figure 2.3) and C.5 we have

$$val_{\mathcal{K},s',\beta}(loc_2) \in s'(\mathsf{noRaW}).$$

Together with the semantics of JavaDL, Definition 3.2.1, and C.6, this means

$$val_{\mathcal{K},s,\beta}(\{\backslash \mathsf{R}(loc_1)\}loc_2) \in s'(\mathsf{noRaW}). \tag{C.7}$$

From Definition 3.3.1 we know that for an arbitrarily chosen $ls \in \mathcal{D}^{\mathsf{LocSet}}$

$ls \in s'(\mathsf{noRaW})$

$\iff$

for all i and j where $1 \leq i < j \leq n$ with $acc_i = \langle Write(ls_i)\rangle, acc_j = \langle Read(ls_j)\rangle$
it holds that $ls \cap ls_i \cap ls_j = \emptyset$.

Together with C.7 this means

$val_{\mathcal{K},s,\beta}(\{\backslash \mathsf{R}(loc_1)\}loc_2) \in s'(\mathsf{noRaW})$

$\iff$

for all i and j where $1 \leq i < j \leq n$ with $acc_i = \langle Write(ls_i)\rangle, acc_j = \langle Read(ls_j)\rangle$
it holds that $val_{\mathcal{K},s,\beta}(\{\backslash \mathsf{R}(loc_1)\}loc_2) \cap ls_i \cap ls_j = \emptyset$.

From the definition of s'(C.6), we know that $acc_n = \langle Read(loc_1)\rangle$. We divide loc_2 int two disjoint location sets $(\{\backslash \mathsf{R}(loc_1)\}loc_2) \cap loc_1$ and $(\{\backslash \mathsf{R}(loc_1)\})loc_2 \setminus loc_1$. Now we need to show

$val_{\mathcal{K},s,\beta}(\{\backslash \mathsf{R}(loc_1)\}loc_2) \in s'(\mathsf{noRaW})$

$\iff$

$\big($for $j = n$ and for all i where $1 \leq i \leq n-1$ with $acc_i = \langle Write(ls_i)\rangle$
it holds that $val_{\mathcal{K},s,\beta}((\{\backslash \mathsf{R}(loc_1)\}loc_2) \cap loc_1) \cap ls_i = \emptyset\big)$
and
$\big($for all i and j where $1 \leq i \leq n-1$ and $i < j \leq n-1$ with $acc_i = \langle Write(ls_i)\rangle$,
$acc_j = \langle Read(ls_j)\rangle$
it holds that $val_{\mathcal{K},s,\beta}((\{\backslash \mathsf{R}(loc_1)\}loc_2) \setminus loc_1) \cap ls_i \cap loc_j = \emptyset\big)$.

Using the Definition 3.3.1 this means that we need to show

$$
\begin{aligned}
& \mathit{val}_{\mathcal{K},s,\beta}(\{\backslash \mathsf{R}(loc_1)\}loc_2) \in s'(\mathsf{noRaW}) \\
\iff & \\
& \mathit{val}_{\mathcal{K},s,\beta}((\{\backslash \mathsf{R}(loc_1)\}loc_2) \cap loc_1) \in s(\mathsf{noW}) \\
& \text{and} \\
& \mathit{val}_{\mathcal{K},s,\beta}((\{\backslash \mathsf{R}(loc_1)\}loc_2) \setminus loc_1) \in s(\mathsf{noRaW}).
\end{aligned}
$$

Together with C.7 this means

$$
\begin{aligned}
& \{\backslash \mathsf{R}(loc_1)\}\mathsf{noRaW}(loc_2) \\
\iff & \\
& \mathit{val}_{\mathcal{K},s,\beta}(\{\backslash \mathsf{R}(loc_1)\}loc_2 \cap loc_1) \in s(\mathsf{noW}) \\
& \text{and} \\
& \mathit{val}_{\mathcal{K},s,\beta}(\{\backslash \mathsf{R}(loc_1)\}loc_2 \setminus loc_1) \in s(\mathsf{noRaW}).
\end{aligned}
$$

From Definition 2.1.9 this is equal to

$$
\begin{aligned}
& \mathcal{K}, s, \beta \models \{\backslash \mathsf{R}(loc_1)\}\mathsf{noRaW}(loc_2) \\
\iff & \\
& \mathcal{K}, s, \beta \models \mathsf{noW}(\{\backslash \mathsf{R}(loc_1)\}loc_2 \cap loc_1) \\
& \text{and} \\
& \mathcal{K}, s, \beta \models \mathsf{noRaW}(\{\backslash \mathsf{R}(loc_1)\}loc_2 \setminus loc_1).
\end{aligned}
$$

□

C.3. Proof of Theorem 4.1.1

Proof. We want to show for all $\mathcal{K}, s, \beta$ with $s = (\sigma, Acc)$ and a side condition that loc does not contain data dependence predicates, the following holds

$$\mathcal{K}, s, \beta \models \{\backslash \mathsf{R}(loc)\}\{\backslash \mathsf{R}'(loc)\}\phi \iff \mathcal{K}, s, \beta \models \phi.$$

Starting from the left side, for all $\mathcal{K}, s, \beta$ we have

$$
\begin{aligned}
& \mathcal{K}, s, \beta \models \{\backslash \mathsf{R}(loc)\}\{\backslash \mathsf{R}'(loc)\}\phi \\
\iff & \text{ Fig. 3.2} \\
& \mathcal{K}, s, \beta \models \{\backslash \mathsf{R}(loc); \backslash \mathsf{R}'(\{\backslash \mathsf{R}(loc)\}loc)\}\phi
\end{aligned}
$$

Together with Definition 2.1.9 this means

$$\mathcal{K}, t, \beta \models \phi \text{ with } \mathit{val}_{\mathcal{K},s,\beta}(\backslash \mathsf{R}(loc); \backslash \mathsf{R}'(\{\backslash \mathsf{R}(loc)\}loc)) = t \tag{C.8}$$

From the side condition we know

$$\{\backslash \mathsf{R}(loc)\}loc = loc.$$

Together with C.8 this means

$$\mathcal{K}, t, \beta \models \phi \text{ with } \mathit{val}_{\mathcal{K},s,\beta}(\backslash \mathsf{R}(loc); \backslash \mathsf{R}'(loc)) = t.$$

Using Definitions 3.2.2 and 3.2.1 we have

$$\mathit{val}_{\mathcal{K},s,\beta}(\backslash \mathsf{R}(loc); \backslash \mathsf{R}'(loc)) = \mathit{val}_{\mathcal{K},s',\beta}(\backslash \mathsf{R}'(loc)) \text{ with } s' = (\sigma, Acc \circ \langle \mathit{Read}(\mathit{val}_{\mathcal{K},s,\beta}(loc))\rangle).$$

From Definition 4.1.2 we end up with

$$\mathit{val}_{\mathcal{K},s',\beta}(\backslash \mathsf{R}'(loc)) = t = (\sigma, Acc).$$

Therefore $s = t$. Together with C.8 we have

$$\mathcal{K}, s, \beta \models \phi.$$

□

C.4. Proof of Theorem 4.1.2

Proof. We want to show for all $\mathcal{K}, s, \beta$ with $s = (\sigma, \mathit{acc}_1 \circ \ldots \circ \mathit{acc}_n)$ and a side condition that loc does not contain data dependence predicates if

$$\mathcal{K}, s, \beta \models \{\backslash \mathsf{R}'(loc)\}\Gamma, \mathtt{rPred}(loc, 0) \Longrightarrow \phi, \{\backslash \mathsf{R}'(loc)\}\Delta. \tag{C.9}$$

then for all $\mathcal{K}, s, \beta$ the following holds

$$\mathcal{K}, s, \beta \models \Gamma \Longrightarrow \{\backslash \mathsf{R}(loc)\}\phi, \Delta.$$

If there is a $\gamma \in \Gamma$ with $\mathit{val}_{\mathcal{K},s,\beta}(\gamma) = false$ or if there is a $\delta \in \Delta$ with $\mathit{val}_{\mathcal{K},s,\beta}(\delta) = true$ then this is trivially true. Therefore, we assume that

$$\mathcal{K}, s, \beta \models \bigwedge(\Gamma \cup \neg\Delta),$$

and aim to show that

$$\mathcal{K}, s, \beta \models \{\backslash \mathsf{R}(loc)\}\phi.$$

From Definition 2.1.9 we know

$$\begin{aligned}\mathcal{K}, s, \beta \models \{\backslash \mathsf{R}(loc)\}\phi \iff \mathcal{K}, s', \beta \models \phi \\ \text{with } s' = \mathit{val}_{\mathcal{K},s,\beta}(\backslash \mathsf{R}(loc)) \\ =_{Def.\ 3.2.1} (\sigma, Acc \circ \langle \mathit{Read}(\mathit{val}_{\mathcal{K},s,\beta}(loc))\rangle).\end{aligned}$$

The new goal is to show

$$\mathcal{K}, s', \beta \models \phi.$$

From Theorem 4.1.1 we know that

$$\mathcal{K}, s, \beta \models \bigwedge(\Gamma \cup \neg\Delta) \iff \mathcal{K}, s, \beta \models \{\backslash \mathsf{R}(loc)\}\{\backslash \mathsf{R}'(loc)\} \bigwedge(\Gamma \cup \neg\Delta).$$

Using Definition 3.2.1 we have

$$\begin{aligned} & \mathcal{K}, s, \beta \models \{\backslash \mathsf{R}(loc)\}\{\backslash \mathsf{R}'(loc)\} \bigwedge(\Gamma \cup \neg\Delta) \\ \iff & \\ & \mathcal{K}, s'', \beta \models \{\backslash \mathsf{R}'(loc)\} \bigwedge(\Gamma \cup \neg\Delta), \\ & \text{with } s'' = (\sigma, Acc \circ \langle \mathit{Read}(\mathit{val}_{\mathcal{K},s,\beta}(loc))\rangle). \end{aligned} \tag{C.10}$$

Using C.9 for s'' yields

$$\mathcal{K}, s'', \beta \models \{\backslash \mathsf{R}'(loc)\}\Gamma, \mathtt{rPred}(loc, 0) \Longrightarrow \phi, \{\backslash \mathsf{R}'(loc)\}\Delta.$$

Together with C.10 this means that

$$\mathcal{K}, s'', \beta \models \mathtt{rPred}(loc, 0) \Longrightarrow \phi. \tag{C.11}$$

From definition of s'' and Definition 4.1.3 we get that

$$\mathcal{K}, s'', \beta \models \mathtt{rPred}(loc, 0),$$

which together with C.11 and $s' = s''$ we conclude

$$\mathcal{K}, s', \beta \models \phi.$$

□

C.5. Proof of Soundness and Completeness of renamedReadAppOnNoRHist

Proof. We want to show for all $\mathcal{K}, s, \beta$ where $s = (\sigma, \mathit{acc}_1 \circ \ldots \circ \mathit{acc}_n)$ and $\mathit{acc}_n = \langle \mathit{Read}(\mathit{val}_{\mathcal{K},s,\beta}(rLoc))\rangle$, with the side condition that loc and lb do not contain data dependence predicates, the following holds

$$\begin{aligned} \mathcal{K}, s, \beta \models \{\backslash \mathsf{R}'(rLoc)\}\mathtt{noRHist}(loc, lb) \iff \\ \mathcal{K}, s, \beta \models \mathtt{noRHist}(\{\backslash \mathsf{R}'(rLoc)\}loc, \{\backslash \mathsf{R}'(rLoc)\}lb + 1). \end{aligned}$$

Considering the side conditions, the goal is to show

$$\mathcal{K}, s, \beta \models \{\backslash \mathsf{R}'(rLoc)\}\mathtt{noRHist}(loc, lb) \iff \mathcal{K}, s, \beta \models \mathtt{noRHist}(loc, lb+1).$$

Starting from the left side we have

$$\begin{aligned}
&\mathcal{K}, s, \beta \models \{\backslash \mathsf{R}'(rLoc)\}\mathtt{noRHist}(loc, lb)\\
\iff &\text{ Def. 2.1.9}\\
&\mathcal{K}, s', \beta \models \mathtt{noRHist}(loc, lb) \text{ with } val_{\mathcal{K},s,\beta}(\{\backslash \mathsf{R}'(rLoc)\}) =_{Def.\ 4.1.2} s' = (\sigma, acc_1 \circ \ldots \circ acc_{n-1})\\
\iff &\text{ Fig. 2.3}\\
&(loc, lb) \in s'(\mathtt{noRHist})\\
\iff &\text{ Def. 4.1.5}\\
&\text{for all } j, 0 < j \leq \overbrace{n-1-lb}^{=n-(lb+1)}, \text{ and } acc_j = \langle Read(ls_j)\rangle\\
&\text{it holds that } loc \cap ls_j = \mathtt{empty}\\
\iff &\text{ Def. 4.1.5}\\
&(loc, lb+1) \in s(\mathtt{noRHist})\\
\iff &\text{ Fig. 2.3}\\
&\mathcal{K}, s, \beta \models \mathtt{noRHist}(loc, lb+1)
\end{aligned}$$

□

C.6. Proof of Theorem 5.3.1

Proof. We want to show for all $\mathcal{K}, s, \beta$ with $s = (\sigma, \widehat{Acc})$ and a side condition that location sets $rls_i, wls_j \in \mathcal{D}^{\mathsf{LocSet}}$ for $i = 1 \ldots n$ and $j = 1 \ldots m$ do not contain data dependence predicates, if the following holds

$$\mathcal{K}, s, \beta \models \overbrace{\{\ldots; \backslash \mathsf{W}'(wls_1); \ldots; \backslash \mathsf{R}'(rls_1); \backslash \mathtt{next}'\}}^{u}\Gamma, \widehat{\mathtt{rPred}}(\bigcup_{i=1\ldots n} rls_i, 0), \widehat{\mathtt{wPred}}(\bigcup_{j=1\ldots m} wls_j, 0)) \Longrightarrow \psi, \{u\}\Delta \tag{C.12}$$

then it holds that

$$\mathcal{K}, s, \beta \models \Gamma \Longrightarrow \{\backslash \mathtt{next}, \backslash \mathsf{R}(rls_1), \ldots, \backslash \mathsf{W}(wls_1), \ldots\}\psi, \Delta. \tag{C.13}$$

If there is a $\gamma \in \Gamma$ with $val_{\mathcal{K},s,\beta}(\gamma) = false$ or if there is a $\delta \in \Delta$ with $val_{\mathcal{K},s,\beta}(\delta) = true$ then this is trivially true. Therefore, we assume

$$\mathcal{K}, s, \beta \models \bigwedge(\Gamma \cup \neg\Delta)$$

and aim to show

$$\mathcal{K}, s, \beta \models \{\backslash\mathtt{next}, \backslash\mathtt{R}(rls_1), \ldots, \backslash\mathtt{W}(wls_1), \ldots\}\psi.$$

We prove the case that $n = m = 1$. For $i, j > 1$ the prove is similar.

Starting from C.13 we have

$$\begin{aligned}
&\mathcal{K}, s, \beta \models \{\backslash\mathtt{next}, \backslash\mathtt{R}(rls_1), \backslash\mathtt{W}(wls_1)\}\psi \\
\iff &\text{Def. 2.1.9} \\
&\mathcal{K}, s_1, \beta \models \{\backslash\mathtt{R}(rls_1), \backslash\mathtt{W}(wls_1)\}\psi \text{ with } s_1 = val_{\mathcal{K},s,\beta}(\backslash\mathtt{next}) \\
&\qquad\qquad = \text{Def. 5.2.4 } (\sigma, \widehat{Acc} \bigcirc \langle\varepsilon\rangle) \\
\iff &\text{Def. 2.1.9} \\
&\mathcal{K}, s_2, \beta \models \{\backslash\mathtt{W}(wls_1)\}\psi \text{ with } s_2 = val_{\mathcal{K},s_1,\beta}(\backslash\mathtt{R}(rls_1)) = \text{Def. 3.2.1} \\
&\qquad\qquad (\sigma, \widehat{Acc} \bigcirc \langle\varepsilon\rangle \circ \langle Read(val_{\mathcal{K},s_1,\beta}(rls_1))\rangle) \\
\iff &\text{Def. 2.1.9} \\
&\mathcal{K}, s_3, \beta \models \psi \text{ with } s_3 = val_{\mathcal{K},s_2,\beta}(\backslash\mathtt{W}(wls_1)) = \text{Def. 3.2.1} \qquad \text{(C.14)} \\
&\quad (\sigma, \widehat{Acc} \bigcirc \langle\varepsilon\rangle \circ \langle Read(val_{\mathcal{K},s_1,\beta}(rls_1))\rangle \circ \langle Write(val_{\mathcal{K},s_2,\beta}(wls_1))\rangle).
\end{aligned}$$

The new goal is to show C.14.

From Theorem 4.1.1 we know that

$$\mathcal{K}, s, \beta \models \bigwedge(\Gamma \cup \neg\Delta)$$

$\iff$ Def.5.2.4

$$\mathcal{K}, s, \beta \models \{\backslash\mathtt{next}; \backslash\mathtt{R}(rls_1); \backslash\mathtt{W}(wls_1)\}\{\backslash\mathtt{W}'(wls_1); \backslash\mathtt{R}'(rls_1); \backslash\mathtt{next}'\} \bigwedge(\Gamma \cup \neg\Delta)$$

$\iff$

$$\mathcal{K}, s', \beta \models \{\backslash\mathtt{R}(rls_1); \backslash\mathtt{W}(wls_1)\}\{\backslash\mathtt{W}'(wls_1); \backslash\mathtt{R}'(rls_1); \backslash\mathtt{next}'\} \bigwedge(\Gamma \cup \neg\Delta),$$
$$\text{with } s' = (\sigma, \widehat{Acc} \bigcirc \langle\varepsilon\rangle),$$

$\iff$ $Def.$ 4.1.2

$$\mathcal{K}, s'', \beta \models \{\backslash\mathtt{W}(wls_1)\}\{\backslash\mathtt{W}'(wls_1); \backslash\mathtt{R}'(rls_1); \backslash\mathtt{next}'\} \bigwedge(\Gamma \cup \neg\Delta),$$
$$\text{with } s'' = (\sigma, \widehat{Acc} \bigcirc \langle\varepsilon\rangle \circ \langle \mathit{Read}(\mathit{val}_{\mathcal{K},s',\beta}(rls_1))\rangle)$$

$\iff$ $Def.$ 4.1.2

$$\mathcal{K}, s''', \beta \models \{\backslash\mathtt{W}'(wls_1); \backslash\mathtt{R}'(rls_1); \backslash\mathtt{next}'\} \bigwedge(\Gamma \cup \neg\Delta), \tag{C.15}$$
$$\text{with } s''' = (\sigma, \widehat{Acc} \bigcirc \langle\varepsilon\rangle \circ \langle \mathit{Read}(\mathit{val}_{\mathcal{K},s',\beta}(rls_1))\rangle \circ \langle \mathit{Write}(\mathit{val}_{\mathcal{K},s'',\beta}(wls_1))\rangle).$$

We see that $s''' = s_3$, $s'' = s_2$, and $s' = s_1$.

Using C.12 for s''' yields

$$\mathcal{K}, s''', \beta \models \overbrace{\{\backslash\mathtt{W}'(wls_1); \backslash\mathtt{R}'(rls_1); \backslash\mathtt{next}'}^{u}\}\Gamma,\ \widehat{\mathtt{rPred}}(rls_1, 0),\ \widehat{\mathtt{wPred}}(wls_1, 0)) \Longrightarrow \psi, \{u\}\Delta.$$

Together with C.15 this means

$$\mathcal{K}, s''', \beta \models \widehat{\mathtt{rPred}}(rls_1, 0),\ \widehat{\mathtt{wPred}}(wls_1, 0) \Longrightarrow \psi. \tag{C.16}$$

In addition, from definition of s''' and Definitions of $\widehat{\mathtt{rPred}}$ in Figure 5.2 and $\widehat{\mathtt{wPred}}$ in Appendix A.4 we have that

$$\mathcal{K}, s''', \beta \models \widehat{\mathtt{rPred}}(rls_1, 0),$$

and

$$\mathcal{K}, s''', \beta \models \widehat{\mathtt{wPred}}(wls_1, 0).$$

which together with C.16 and $s''' = s_3$ we conclude C.14.

□

C.7. Proof of Soundness and Completeness of matrixRangeMinusSingleton

Proof. We assume that $\mathcal{K}, s, \beta \models \texttt{wellFormedMatrix}(matrix, heap)$ holds, and *sRow* is a fresh Skolem constant of type `int` where

$$\mathcal{K}, s, \beta \models \exists k;\ k \geq rowL\ \wedge\ k \leq rowH\ \wedge\ o \doteq \texttt{select}(heap, matrix, \texttt{arr}(k)) \rightarrow$$
$$sRow \geq rowL\ \wedge\ sRow \leq rowH\ \wedge\ o \doteq \texttt{select}(heap, matrix, \texttt{arr}(sRow)). \tag{C.17}$$

We want to show for all $\mathcal{K}, s, \beta$, the following holds

$$\begin{aligned}
\mathcal{K}, s, \beta \models\ & \texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH) \setminus \\
& \texttt{singleton}(o, \texttt{arr}(sCol)) \\
& \doteq \\
& \text{if } (matrix \doteq o\ \vee\ sCol < colL\ \vee\ colH < sCol) \\
& \text{then } (\texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH)) \\
& \text{else } (\text{if } (\exists k;\ k \geq rowL\ \wedge\ k \leq rowH\ \wedge\ o \doteq \texttt{select}(heap, matrix, \texttt{arr}(k))) \\
& \quad \text{then } (\texttt{matrixRange}(heap, matrix, rowL, sRow\ \texttt{-}\ 1, colL, colH) \cup \\
& \qquad \texttt{matrixRange}(heap, matrix, sRow\ \texttt{+}\ 1, rowH, colL, colH) \cup \\
& \qquad \texttt{matrixRange}(heap, matrix, sRow, sRow, colL, sCol\ \texttt{-}\ 1) \cup \\
& \qquad \texttt{matrixRange}(heap, matrix, sRow, sRow, sCol\ \texttt{+}\ 1, colH))) \\
& \quad \text{else } (\texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH))
\end{aligned}$$

From $val_{\mathcal{K},s,\beta}$(C.17) and Definition 7.1.2 we have

$$\begin{aligned}
\mathcal{K}, s, \beta \models\ & \texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH) \\
& \doteq \\
& \texttt{infiniteUnion}\{\texttt{int}\ sRow;\}(\\
& \texttt{arrayRange}(\texttt{select}(heap, matrix, \texttt{arr}(sRow)), colL, colH)).
\end{aligned}$$

Together with Definition 7.1.1 this means

$$\begin{aligned}
\mathcal{K}, s, \beta \models\ & \texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH) \\
& \doteq \\
& \bigcup_{rowL \leq sRow \leq rowH} \texttt{arrayRange}(\texttt{select}(heap, matrix, \texttt{arr}(sRow)), colL, colH).
\end{aligned} \tag{C.18}$$

First case: if $\mathcal{K}, s, \beta \models$ *matrix* $\doteq o$. Together with Definition 7.1.3 this means

$$\mathcal{K}, s, \beta \models \forall\ row;\ o \neq \texttt{select}(heap, matrix, \texttt{arr}(row)).$$

Together with C.18 this means that the singleton and the array ranges can not overlap, therefore

$$\begin{aligned}\mathcal{K}, s, \beta \models\ & matrix \doteq o \rightarrow \\ & \texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH) \setminus \\ & \texttt{singleton}(o, \texttt{arr}(sCol)) \\ & \doteq \\ & \texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH).\end{aligned}$$

Second case: if $\mathcal{K}, s, \beta \models sCol < colL \ \vee \ colH < sCol$.
We know that C.18 holds for all c such that $colL \ \leq \ c \ \leq \ colH$. Therefore $sCol \neq c$, which means the singleton and the array ranges can not overlap. This means

$$\begin{aligned}\mathcal{K}, s, \beta \models\ & sCol < colL \ \vee \ colH < sCol \rightarrow \\ & \texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH) \setminus \\ & \texttt{singleton}(o, \texttt{arr}(sCol)) \\ & \doteq \\ & \texttt{matrixRange}(heap, matrix, rowL, rowH, colL, colH).\end{aligned}$$

Third case: when it holds that

$$\begin{aligned}\mathcal{K}, s, \beta \models\ & matrix \neq o \ \wedge \ sCol \geq colL \ \wedge \ colH \geq sCol \ \wedge \\ & \exists k; \ k \geq rowL \ \wedge \ k \leq rowH \ \wedge \ o \doteq \texttt{select}(heap, matrix, \texttt{arr}(k)).\end{aligned}$$

Together with C.17 this means

$$\mathcal{K}, s, \beta \models o \doteq \texttt{select}(heap, matrix, \texttt{arr}(sRow))$$

Considering C.18, this shows overlap between the singleton and the matrix range which according

to the standard definition of set difference it has to be excluded. Therefore, we have

$$
\begin{aligned}
\mathcal{K}, s, \beta \models\ & matrix \neq o \ \wedge\ sCol \geq colL \ \wedge\ colH \geq sCol \ \wedge \\
& \exists k;\ k \geq rowL \ \wedge\ k \leq rowH \ \wedge\ o \doteq \mathtt{select}(heap, matrix, \mathtt{arr}(k)) \rightarrow \\
& \mathtt{matrixRange}(heap, matrix, rowL, rowH, colL, colH) \setminus \\
& \mathtt{singleton}(o, \mathtt{arr}(sCol)) \\
& \doteq \\
& \bigcup_{rowL \leq r \leq sRow \ \texttt{-}\ 1} \mathtt{arrayRange}(\mathtt{select}(heap, matrix, \mathtt{arr}(r)), colL, colH) \cup \\
& \bigcup_{sRow \ \texttt{+}\ 1 \leq r \leq rowH} \mathtt{arrayRange}(\mathtt{select}(heap, matrix, \mathtt{arr}(r)), colL, colH) \cup \\
& \bigcup_{colL \leq c \leq sCol \ \texttt{-}\ 1} \mathtt{arrayRange}(\mathtt{select}(heap, matrix, \mathtt{arr}(sRow)), colL, c) \cup \\
& \bigcup_{sCol \ \texttt{+}\ 1 \leq c \leq colH} \mathtt{arrayRange}(\mathtt{select}(heap, matrix, \mathtt{arr}(sRow)), c, colH).
\end{aligned}
$$

□